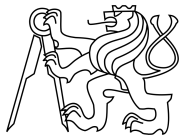
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	GCC a LLVM - Porovnání implementací zadních částí překladačů
Student:	Bc. Pavel Löbl
Vedoucí:	Ing. Radomír Polách
Studijní program:	Informatika
Studijní obor:	Systémové programování
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Nastudujte překladače GCC a LLVM se zaměřením na jejich zadní části (backend).
Popište vstupní rozhraní zadních částí obou překladačů.
Implementujte zadní části pro oba překladače pro architekturu Tiny Machine [1].
Uvedenou architekturu Tiny Machine rozšiřte o instrukce pro volání funkce, případně další instrukce tak, aby bylo možné přeložit jednoduché programy. Interpretr Tiny Machine rozšiřte tak, aby vykonával strojový kód generovaný implementovanými překladači.
Pomocí interpretru otestujte, zda překladače generují validní strojový kód pro cílovou architekturu.
Detailně popište práci s oběma překladači a srovnejte jejich slabiny a silné stránky, zaměřte se na náročnost implementace.

Seznam odborné literatury

[1] Online: http://www.cs.sjsu.edu/ louden/cmptext/

L.S.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 23. listopadu 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Diplomová práce

GCC a LLVM - Porovnání implementací zadních částí překladačů

Bc. Pavel Löbl

Vedoucí práce: Ing. Radomír Polách

17. února 2017

Poděkování

Děkuji vývojářům projektů GCC a LLVM za cenné rady, poskytnuté během vypracovávání této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 17. února 2017 .....

Odkaz na tuto práci

, . *GCC a LLVM - Porovnání implementací zadních částí překladačů.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato diplomová práce se zabývá problematikou překladu programovacích jazyků do strojového kódu. Zaměřuje se na zadní část překladače a vysvětluje základní pojmy a principy fungování generátoru strojového kódu. Dále podrobněji rozebíra implementace generátorů kódu v překladačích GCC a LLVM. Zejména zkoumá způsob, jakým je dosaženo podpory více cílových architektur v jednom kompilátoru.

V praktické části byly projekty GCC a LLVM rozšířeny o generátor kódu pro jednoduchou registrovou architekturu s redukovanou instrukční sadou.

Klíčová slova Závěrečná práce, GCC, LLVM, back-end, implementace, porovnání, generovaní kódu LaTeX.

Abstract

This thesis deals with translation of high-level programming languages into machine code. It focuses mainly on back-end part of the compiler. It explains basic concepts and principles of machine code generator. Further it analyzes in greater detail implementation of code generators in the GCC and LLVM. Particularly examining aspects which enable support of multiple target architectures in a single compiler.

In practical part of this thesis GCC and LLVM projects were extended to support code generation for a simple register architecture with a reduced instruction set.

Keywords Thesis, GCC, LLVM, back-end, implementation, comparison, code generation LaTeX.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

V samých začátcích vývoje výpočetních systémů, bylo zcela běžnou praxí definovat jejich softwarové vybavení přímo v jazyce symbolických adres. Tento přístup je přímočarý, dodává programátorovi plnou kontrolu nad chováním systému a umožňuje mu naplno využít jeho potenciálu. Tento způsob v sobě ale skrývá i jisté nevýhody. Program v jazyce symbolických adres v sobě nutně nese příliš mnoho informací o specifických rysech implementace dané platformy. Díky tomu je obtížnější pouze ze zápisu programu v této formě na první pohled stanovit jeho chování ve větším měřítku. Navíc je takový program napevno svázán s architekturou systému pro kterou byl vytvořen.

Díky těmto skutečnostem začal vývoj vysokoúrovňových programovacích jazyků. Tyto formální jazyky používají abstraktních konstruktů k popisu požadovaného chování počítače. Tím je jednak umožněno mnohem efektivněji vyjádřit strukturu programu a navíc to odstiňuje jeho zápis od jakékoliv hardwarové implementace výpočetního systémů. Zavedením této abstraktní vrstvy však vyvstává otázka jak program v takovém zápisu vykonat pomocí skutečných počítačových architektur.

K tomuto účelu slouží speciální počítačový program nazývaný překladač či kompilátor. Jeho vstupem je zápis programu v abstraktním vysokoúrovňovém jazyce a výstupem program v jazyce symbolických adres nebo strojových instrukcí, které jsou již vykonatelné výpočetní jednotkou počítače. Tento proces překladu by měl splňovat následující kritéria:

1. Překladač by měl upozornit uživatele a ukončit překlad v případě, že vstup není validním programem v daném programovacím jazyce.

2. Pro validní vstup generovat odpovídající výstup. Tedy výsledný program by měl skutečně dělat to, co bylo popsáno vysokoúrovňovým jazykem na vstupu kompilátoru.

3. A dodatečně by měl být výstupní program co možná nejefektivnější. Ať už z pohledu jeho velikosti nebo rychlosti vykonání.

Poslední podmínka je to, co dělá překlad poměrně náročným problémem. A to jak z pohledu teoretického, tak i požadavků na výpočetní výkon. Dnes již je takzvaná *teorie překladačů* v podstatě samostatným podoborem informatiky s přesahem do mnoha dalších oblastí.

Tato diplomová práce se zabývá problematikou překladačů, a to zejména jejich zadní částí (tzv. back-endu), která je zodpovědná za samotné generování výsledného kódu z takzvané vnitřní formy 2.1.1. Cílem práce je seznámit čtenáře s reálnými implementacemi zadních částí dvou pravděpodobně nejznámějších překladačů se svobodnou licencí a to GCC a LLVM. Tyto překladače mají tu vlastnost, že podporují hned několik cílových architektur a dokáží tedy generovat programy pro procesory s různými instrukčními sadami. Tato skutečnost ovlivňuje architekturu jejich zadních částí.

První kapitola se věnuje popisu cílové platformy implementovaných překladačů, která vznikla rozšířením výukové architektury *TinyMachine*. Je popsána instrukční sada a ostatní aspekty platformy, které je potřeba zohlednit při vytváření generátoru kódu. V druhé kapitole jsou představeny základní pojmy z teorie překladačů, zejména ty související s jejich zadní částí. V dalších dvou kapitolách jsou podrobněji rozebrány oba překladače. Především jsou detailněji popsány mechanizmy s jejichž pomocí je možné implementace zadních části překladačů rozšířit za účelem zavedení podpory další cílové platformy.

Cílem praktické části je implementovat generátory kódu v rámci projektu GCC a LLVM pro pomyslnou výukovou architekturu. Tyto upravené překladače by měly být schopny přeložit základní programy v jazyce C. To bude ověřeno pomocí interpretru jazyka symbolických adres cílové architektury.

Kapitola 1

Popis cílové architektury

1.1 TinyMachine

Cílová architektura implementovaných překladačů vznikla rozšířením architektury TinyMachine. Tuto architekturu definoval Kenneth C. Louden ve své knize Compiler Construction [1].

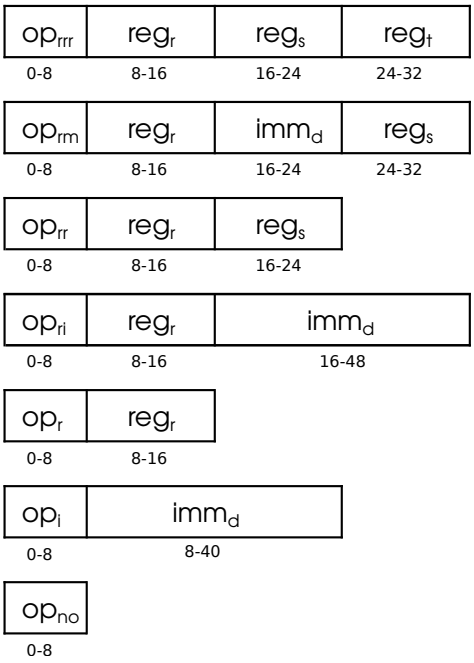
Jedná se o jednoduchou registrovou architekturu typu „load-store“. Její instrukční sada (ISA) obsahuje zvláštní operace pro přístup do paměti. Ostatní instrukce přistupují pouze do registrového pole. Architektura obsahuje celkem pět 32-bitových registrů pro aritmetické operace a tři speciální registry. Programový čítač a registr pro adresovaní aktivačního záznamu funkce a zásobníku.

Paměť je adresovatelná po bajtech a obsahuje jak data, tak i instrukce vykonávaného programu. Všechny instrukce jsou prováděny sekvenčně bez zřetězeného zpracovaní. Součástí instrukční sady nejsou instrukce pro práci s čísly v plovoucí řádové čárce.

1.2 Instrukční sada

Nejvýraznější změnou cílové architektury bylo rozšíření instrukční sady. Byly přidány instrukce pro volaní a návrat z funkce. Bylo by samozřejmě možné implementovat v překladači volaní funkcí přímou manipulací s čítačem instrukcí. Generovaný kód by byl ale poměrně nepřehledný a i na straně implementace překladače by se jednalo nestandardní řešení.

Další skupinou instrukcí, která byla značně rozšířena jsou instrukce pro přístup do paměti. Původní sada obsahovala jen instrukce načtení a uložení registru do paměti. To se ukázalo jako nedostatečné pro implementaci generátoru v reálných překladačích. Byly tedy doplněny instrukce pro ukládaní slabiky, půlky slova a ekvivalentní instrukce pro načítaní, včetně variant se znaménkovým rozšířením načítané hodnoty.



Obrázek 1.1: Formát jednotlivých typů instrukcí v paměti

1.3 Rozložení operační paměti

Architektura je Von Neumannova typu, operační paměť obsahuje jak data tak i instrukce vykonávaného programu. Nejmenší adresovatelnou jednotkou paměti je jeden bajt. Pokud instrukce pracuje s větší jednotkou paměti, pak nejnižší adrese v paměti odpovídá nejméně významný bajt. Jde tedy o systém little endian.

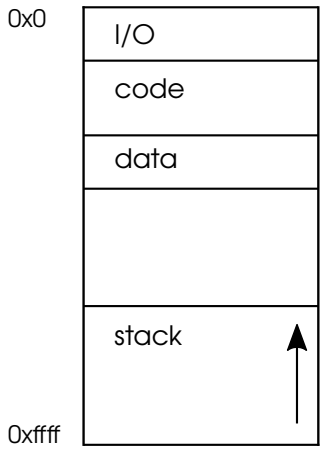
Na nejnižších adresách paměti se nachází prostor mapovaný na vstupně výstupní funkce 1.4.2. Dále následuje kód vykonávaného programu. Nejprve strojový kód a poté data programu. Na konci paměti se nachází zásobník, který roste směrem dolů k nižším adresám.

Tabulka 1.1: Instrukční sada cílové architektury

Instrukce RRR - tři registrové oprandy				
ADD	r	s	t	$r = s + t$
SUB	r	s	t	$r = s + t$
MUL	r	s	t	$r = s * t$
DIV	r	s	t	$r = s \div t$
SHL	r	s	t	$r = s \ll t$ (bitový posuv doleva)
SHR	r	s	t	$r = s \gg t$ (logický posuv doprava)
SHA	r	s	t	$r = s \gg t$ (aritmetický posuv doprava)

XOR	r	s	t	$r = s \oplus t$
AND	r	s	t	$r = s \,\&\, t$
OR	r	s	t	$r = s \mid t$
Instrukce RM - registr, pamět				
LD	r	d	s	načtení slova z adresy $d + s$ do r
LBS	r	d	s	načtení znaménkově rozšířené slabiky z adresy $d + s$ do registru r
LBZ	r	d	s	načtení nulou rozšířené slabiky z adresy $d + s$ do r
LHS	r	d	s	načtení znaménkově rozšířené půlky slova z adresy $d + s$ do registru r
LHZ	r	d	s	načtení nulou rozšířené půlky slova z adresy $d + s$ do registru r
ST	r	d	s	uložní registru r na adresu $d + s$
STB	r	d	s	uložní nejspodnější slabiky registru r na adresu $d + s$
STH	r	d	s	uložní spodní půlky slova registru r na adresu $d + s$
LDA	r	d	s	$r = d + s$
Instrukce RR - registr, registr				
MOV	r	s		$r = s$
INC	r	s		$r = r + s$ check this
DEC	r	s		$r = r - s$
Instrukce R - registr				
IN	r			načtení ASCII kódu stisknuté klávesy
OUT	r			ASCI výstup na obrazovku
CALL	r			$mem[SP] = PC, PC = rSP = SP - 4$
Instrukce RI - registr, hodnota				
JLT	r	d		je-li $r < 0$ pak nastaví čítač instrukcí na hodnotu d
JLE	r	d		je-li $r <= 0$ pak nastaví čítač instrukcí na hodnotu d
JGT	r	d		je-li $r > 0$ pak nastaví čítač instrukcí na hodnotu d
JGE	r	d		je-li $r >= 0$ pak nastaví čítač instrukcí na hodnotu d
JEQ	r	d		je-li $r == 0$ pak nastaví čítač instrukcí na hodnotu d
JNE	r	d		je-li $r! = 0$ pak nastaví čítač instrukcí na hodnotu d
LDC	r	d		$r = d$

Instrukce I - přímý operand				
JMP	d			$PC = d$
CALLI	d			$mem[SP] = PC, PC = d, SP = SP - 4$
Instrukce NO - bez operandu				
RET				$PC = mem[SP], SP = SP + 4$
HALT				ukončí vykonávání programu



Obrázek 1.2: Rozložení fyzické paměti

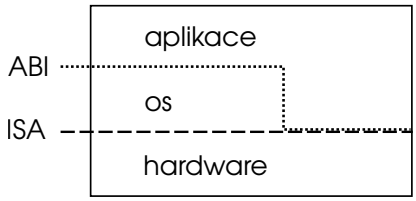
1.4 Aplikační binární rozhraní

Kromě rozšíření a popisu instrukční sady architektury bylo nutné definovat binární aplikační rozhraní (ABI). Jedná se o sadu pravidel na úrovni strojového kódu, jejichž dodržování umožňuje spolupráci mezi jednotlivými funkcemi programu, knihovnami, případně mezi programem a jádrem operačního systému. Pro jednu hardwarovou architekturu může tedy existovat několik různých ABI. Často je toto rozhraní vázáno na operační systém či použitý překladač. Obecně lze tyto nízkoúrovňová pravidla rozdělit do následujících skupin:

- velikost a zarovnání datových typů,
- volací konvence (způsob předávání parametrů do funkcí).

Pokud je výsledný strojový kód spuštěn v prostředí operačního systému, pak lze zahrnout i následující kategorie:

- konvence systémových volání,
- formát objektových souborů a knihoven.



Obrázek 1.3: Vztah mezi aplikací, ISA a ABI

Pro samotné vykonávání programu jsou velmi důležité volací konvence. Ty definují, jakým způsobem jsou předávány formální parametry do funkcí. To zahrnuje zejména v jakém pořadí jsou skalární argumenty ukládány na zásobníku, nebo zda jsou pro předání hodnot použity registry. Obdobně je nutné definovat způsob předání různých typů návratových hodnot. Dále je nutné určit, které z registrů je možné ve funkci použít bez předchozího zálohování jejich hodnot. Typicky se rozdělují registry na následující dvě skupiny:

caller-saved Hodnota registru je zálohována funkcí provádějící volaní a volaná funkce může registr volně používat.

callee-saved Tyto registry nejsou volající funkcí zálohovány. Pokud je volaná funkce potřebuje použít, pak musí jejich hodnotu před prvním použitím zálohovat a před návratem z funkce jí obnovit. Toto je realizováno většinou pomocí zásobníku během prologu a epilogu funkce[1].

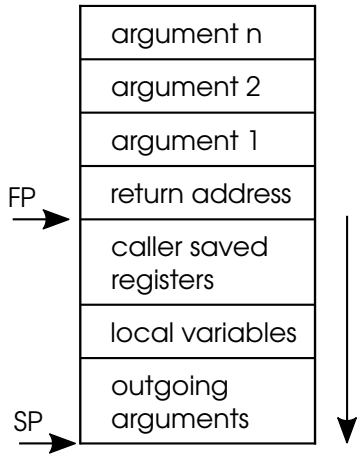
1.4.1 Definice ABI

Pro implementaci reálných překladačů, je nutné definovat pro cílovou architekturu binární rozhraní. Velikost základních datových typů byla zvolena obdobně jako na většině 32-bitových platforem. Tedy 1, 2 a 4 bajty pro typ *char*, *short* respektive *int*.

Předávání argumentu do funkcí odpovídá volací konvenci CDECL používané na architekturách x86. Argumenty jsou předávány pomocí zásobníku. Hodnota ukazatele zásobníku i adres jednotlivých argumentů jsou zarovnány na velikost slova. Struktury jsou předávány po jednotlivých položkách jako několik samostatných argumentů.

Registry *r2* a *r3* jsou zálohovány volanou funkcí, registry *r0* a *r1* zálohuje funkce volající. Základní typy jsou z funkce navráceny registrem *r0*. Struktury jsou navráceny uložením hodnot na adresu dodanou skrytým argumentem z volající funkce. Prostor na zásobníku alokovaný pro argumenty funkce je dealokován volající funkcí.

[1] Prolog je sled instrukcí na začátku kódu funkce, jehož úkolem je daným způsobem nastavit zásobník a registry před zahájením vykonávání samotného výpočtu funkce. Konkrétní prováděné operace záleží na použitém ABI. Analogicky epilog se nachází na konci funkce a typicky provádí „uklízecí“ práce nutné pro návrat z funkce.



Obrázek 1.4: Rozložení zásobníku při vstupu do funkce

1.4.2 Vstupně-výstupní funkce

Architektura umožňuje realizovat vstupně-výstupní (I/O) funkce dvěma způsoby. Prvním je přímým použitím instrukcí IN a OUT. Instrukce IN zablokuje vykonávání programu dokud, uživatel nestiskne klávesu, a poté načte odpovídající kód v ASCII do určeného registru. Analogicky instrukce OUT vytiskne hodnotu registru v ASCII reprezentaci na obrazovku.

Jazyk C neobsahuje žádné konstrukty pro vstupně-výstupní operace. Ty jsou běžně realizovány pomocí standardní knihovny jazyka C. Tam jsou tyto funkce implementovány buď pomocí systémových volaní operačního systému, nebo pomocí vkládaného assembleru. Vkládání kódu symbolických adres do zdrojového kódu však vyžaduje podporu ze strany použitého překladače.

Pro zjednodušení implementace překladače dovoluje architektura provádět vstupně-výstupní operace i pomocí běžného přístupu do paměti. Pokud je čten bajt z adresy 0x2 pak se běh programu zastaví dokud uživatel nestiskne klávesu. Její ASCII kód je poté načten do daného registru. Obdobně, je-li zapsán bajt na adresu 0x1 odpovídající znak je vypsán na obrazovku.

Další možností by mohlo být načtení strojového kódu implementující I/O operace pomocí instrukcí IN a OUT do paměti. V podstatě by se jednalo o jednoduchý BIOS. Ten by byl načten ještě před startem programu na předem známou adresu. Ve zdrojovém kódu programu by se pak jednotlivé funkce volali pomocí ukazatele na funkci získaného přetypováním konstanty odpovídající adrese I/O funkce v paměti. Tento strojový kód byl ale musel splňovat stejné volací konvence, jako kód samotného programu generovaný překladačem 1.4.

1.5 Implementace cílové platformy

Implementace cílové platformy vychází z implementace pro původní architekturu TinyMachine. Jedná se tedy o interpret jazyka symbolických adres. Došlo ale v podstatě ke kompletnímu přepsání původní implementace, která byla sice velice jednoduchá, ale kód byl velmi těžce modifikovatelný. Hlavním důvodem byla nutnost doplnit interpret syntaktickým analyzátorem jazyka symbolických adres. V direktivách assembleru se totiž mohou vyskytovat i aritmetické výrazy.

1.5.1 Interpret jazyka symbolických adres

Nová implementace je jednoduchý interpret GNU Assembleru. Což je jazyk generovaný oběma překladači. Podpora rozhodně není úplná. Je podporovaná pouze podmnožina direktiv a u některých pouze zjednodušené varianty bez některých volitelných argumentů. Celá implementace je rozdělena do následujících modulů:

parser Lexikální analyzátor a jednoduchý syntaktický analyzátor implementovaný rekurzivním sestupem.

opcode Definice operačních znaků a tříd instrukcí.

cpool Naivní tabulka symbolů (návěští).

tmi Vlastní implementace interpretru.

1.5.2 Knihovna základních funkcí

Pro zjednodušení ladění a testování přeložených programů byla implementována „knihovna“ základních funkcí. Ta poskytuje základní možnosti vstupu a výstupu pomocí funkcí *in*, *out* a *prints*.

Obsahuje dále funkci *strlen* a *memcpy*. V některých případech se totiž překladač (GCC) rozhodne nahradit delší sekvenci instrukcí přesunující data v paměti voláním funkce *memcpy*. A to i při uvedení volby *-fno-builtin*.

Kapitola 2

Obecné pojmy a struktura překladačů

V této kapitole jsou vysvětleny nejzákladnější pojmy, se kterými se běžně setkáváme jak v oboru zabývající se teorií konstrukce překladačů, tak i v konkrétních dnes běžně používaných implementacích překladačů jako jsou GCC a LLVM. Tyto pojmy jsou dále používány v dalších kapitolách zabývající se blíže těmito projekty.

Překladač je nástroj, který provádí převod mezi programem zapsaným ve vstupním jazyce do formy výstupního jazyka. Tato transformace musí zachovávat význam daného programu, nesmí se tedy měnit jeho funkce. Typicky je vstupním jazykem některý z programovacích jazyků vyšší úrovně jako C,C++ nebo Java.. Výstupním jazykem je platformě závislý strojový kód nebo jazyk symbolických adres pro konkretní architekturu procesoru. Jako například X86, Arm, Sparc a jiné. Často je výstupní strojový kód závislý i na použitém operačním systému.

Zcela běžně jsou na překlad kladeny různé požadavky tak, aby výsledný program splňoval některé parametry. Například co možná nejmenší velikost vygenerovaného strojového kódu nebo jeho rychlejší vykonání cílovou architekturou. Pak mluvíme o optimalizujícím překladači.

Nutno dodat, že tyto požadavky jsou často protichůdné. Obecně je nemožné sestrojit překladač, který by pro všechny programy, ze vstupního jazyka, vytvořil optimální kód dle daných kriterií. Uvažujme libovolný program, který neprodukuje žádný výstup a nikdy se nezastaví. V tomto případě by měl optimální překladač optimalizující na velikost, nahradit takový program prázdnou, nekonečnou smyčkou. Jinými slovy by tento překladač byl schopen vyřešit problém zastavení [2].

2.1 Typická struktura překladače

Jelikož je celý proces transformace ze zdrojového jazyka poměrné komplexní úkol, bývá běžně rozdělen do několika kroků. Takzvaných průchodu. Každý z nich je zodpovědný za konkretní část transformace. Průchody běží postupně po sobě a výstup jednoho je vstupem do následujícího. Ale je zcela běžné, že některý průchod je aplikován opakovaně a nebo je naopak zcela vynechán, pokud například uživatel nechce použít některou z implementovaných optimalizací. V následující části jsou popsány nejdůležitější průchody překladu [3, 2].

lexikální analýza Úkolem lexikálního analyzátoru je načítat vstup a rozdělovat ho na jednotlivé dané symboly definované ve vstupním jazyce, takzvané „tokeny“. Například *if*, *var* atd. Kód lexikálního analyzátoru bývá často generován z definice vstupního jazyka. Hlavní důvod jeho existence je zjednodušení implementace syntaktického analyzátoru.

syntaktická analýza V tomto kroku jsou načítány jednotlivé symboly produkované v předchozím kroku a je z nich konstruována stromová struktura takzvaný abstraktní syntaktický strom (AST). Ten je vlastní reprezentací vstupního programu. Během tohoto procesu je také kontrolováno, že vstupní program je skutečně validním programem v daném jazyku.

sémantická analýza Tento průchod kontroluje, zda AST reprezentující vstupní program splňuje všechny požadavky na práci s typy a proměnnými. Tedy zda deklarace proměnných předchází jejich použití v zápisu programu, pokud to jazyk vyžaduje nebo jestli je daná operace definována pro použité typy proměnných.

generovaní vnitřní formy Tento krok prochází AST a generuje z něho vnitřní formu překladače. Většinou se jedná o nějakou variantu tříadresního kódu nebo SSA formy.

optimalizace Nad vnitřní formou poté dochází ke většině optimalizací. Většinou v mnoha oddělených krocích. Toto je realizováno ve střední části překladače, který není závislá na vstupním jazyku ani cílové platformě.

generovaní platformě závislého kódu Poté dochází k překladu z vnitřní formy do platformě závislého kódu, jazyce symbolických adres. V tomto kroku dochází i k platformě závislým optimalizacím.

strojový kód Nakonec dochází k překladu z jazyka symbolických adres do strojového kódu. Ten je poté pomocí assembleru proložen do strojového kódu a sestavovacím programem („linkerem“) složen do jednoho spustitelného souboru.

V praxi se tyto jednotlivé kroky většinou seskupují do tří celků s jasně definovanými rozhraními. První neboli přední částí (angl. front-end) je lexikální a syntaktický analyzátor a často také generátor vnitřní formy. Právě převod vysokoúrovňového jazyka do vnitřní formy, umožňuje následující části překladače odstínit od specifik zpracovávaného jazyka.

Následuje střední část (middle-end), která se stará o optimalizace vnitřní formy. Je jedinou částí překladače, která není závislá na vstupním jazyku ani na cílové platformě. A poslední, zadní část (back-end), který zajišťuje převod vnitřní formy do strojového kódu cílové platformy.

Tento tří-fázový návrh překladače umožňuje kombinování více front-endů a back-endů v rámci jednoho projektu. Takový překladač dokáže generovat strojový kód pro různé platformy hned z několika programovacích jazyků za použití stejné optimalizační části [2].

2.1.1 Vnitřní forma

Vnitřní forma překladače je vyjádření programu, které není závislé na vstupním ani výstupním jazyku. Navíc má toto alternativní vyjádření takové vlastnosti, které umožňují efektivněji provádět některé optimalizace. Není tedy neobvyklé, že překladač používá i více vnitřních forem, pro různé druhy optimalizací.

2.1.1.1 Tří-adresový kód

Tří-adresový kód je často používaný typ vnitřní formy. Je to zápis podobný jazyky symbolických adres. Výrazy ve vstupním programu jsou rozloženy na jednoduché operace se třemi argumenty. Dva argumenty jsou operandy dané operace a třetí je výsledek. Argumenty často nejsou skutečnými registry či paměťové buňkami, ale jde o symbolické proměnné. Jedná se o linearizovanou formu syntaktického stromu. Například složený výraz `a = 4 * b + 2` může vypadat v tří-adresovém zápisu následovně:

```
t1 = 4 * b
 a = t1 + 2 .
```

K rozložení výrazu musí překladač použít nové dočasné proměnné. Ty jsou ale později eliminovány pří alokovaní skutečných registrů [4].

2.1.1.2 SSA forma

Další velmi častým znakem používané vnitřní formy je takzvaná SSA (static single assignment) forma. Nejedná se ani tak o samostatný typ vnitřní formy,

[2] Některá literatura definuje pouze front-end a back-end. V tomto případě je back-endem myšlena optimalizační části spolu s generátorem strojového kódu. Jiné zdroje definují navíc i middle-end jako samostatnou část, která provádí platformě nezávislé optimalizace nad vnitřní formou. Potom se back-endem myslí generátor kódu.

ale spíše o vlastnost vnitřních forem obecně. Je tedy používaná jak pro tříadresové vnitřní formy tak i pro formy se stromovou strukturou.

Jak název napovídá jde o podobu vnitřní formy, kde do každé proměnné je přiřazeno pouze jednou. Každé další přiřazení ve vstupním programu způsobí vytvoření nové verze dané proměnné ve vnitřní formě. Operace v této formě používají vždy nejnovější verzi dané proměnné.

Tato vlastnost má za následek, že program v tomto tvaru v sobě přímo obsahuje informace o vazbách mezi definicí a použitím dané proměnné. To umožňuje efektivnější provádění optimalizací založených na analýze datových závislostí jako je eliminace stejných podvýrazů nebo propagace konstant [5].

2.2 Zadní část překladače

Úkolem zadní části překladače je transformace vnitřní formy do kódu cílové platformy. Typicky do objektového nebo strojového kódu. A to takovým způsobem, aby co možné nejefektivněji využíval možnosti dané architektury. Tento proces se skládá z několika podproblémů [3]:

- výběr instrukcí,
- alokování registrů,
- plánování instrukcí.

2.2.1 Výběr instrukcí

Nejdůležitějším krokem při generovaní kódu je výběr instrukcí cílové architektury. Účelem tohoto průchodu je převést kód vnitřní formy do platformě závislých instrukcí cílové architektury. Tedy najít takovou množinu instrukcí cílové architektury, která odpovídá danému kódu vnitřní formy a tím pádem i vstupnímu programu. Tomuto přiřazení instrukcí vstupnímu programu se zpravidla říká pokrytí [2].

Je běžné, že instrukční sady obsahují některé pokročilé instrukce pro provádění složitějších operací. Velmi často existuje více možností jak vstupní program pokrýt instrukcemi cílové platformy. Cílem je najít takové pokrytí, které bude co možná nejlepší dle zadaného kritéria. Například pokrytí, které má nejmenší součet velikosti použitých instrukcí nebo minimální celkový počet taktů potřebný k jejich vykonání.

Obecně lze algoritmy řešící tento problém rozdělit do několika kategorií. Ty nejednodušší jsou založeny na jednoduchém nahrazovaní jednotlivých konstruktů vnitřní formy sekvencí instrukcí cílové platformy. Jedná se o nejstarší a nejjednodušší přístup ke generovaní kódu nazývaný jako „macro expansion“. Nevýhodou je velmi nízká kvalita generovaného kódu. Zejména pro architektury s komplexní instrukční sadou. Proto se tento způsob často kombinuje s

peephole optimalizátorem[3], který se snaží krátké úseky více instrukcí zkombinovat do jedné složitější instrukce a tím výsledný kód zkvalitnit.

Další implementace jsou založené na různých algoritmech pro pokrývání grafové reprezentace vnitřní formy. Tato technika často používající heuristiky k nalezení ideálního pokrytí dnes patří pravděpodobně k nejlépe zmapovaným a nejvíce používaným metodám. [7]

2.2.2 Alokování registrů

Funkce vysokoúrovňového jazyka může obsahovat libovolný počet lokálních proměnných. Naopak procesor obsahuje pouze konečný počet registrů. Není tedy možné přiřadit každé proměnné funkce jeden registr na celou dobu běhu dané funkce. Z toho vyplývá, že v některých částech programu je nutné některé proměnné uložit do paměti za účelem uvolnění registru a poté hodnotu opět načíst ve chvíli kdy se s ní znovu pracuje, tzv. „spilling“. Přístup do paměti je ale z pravidla řádově pomalejší než přístup do registrového pole. Alokátor registrů se snaží najít takové přiřazení registrů proměnným, které co možná nejvíce omezí počet přístupů do paměti. [2]

Nejtriviálnějším přístupem, který se ale nedá moc považovat za řešení problému, je přidělování vůbec neprovádět a všechny proměnné držet pouze v paměti. Jak již bylo naznačeno, nevýhodou je nízká kvalita kódu z důvodu jeho pomalosti. Poměrně jednoduchá technika s podstatně lepšími výsledky je takzvané lineární skenovaní. Registry jsou proměnným přiřazovány v pořadí, ve kterém se objevují v programu a až ve chvíli, kdy už není žádný registr volný, dojde k uložení některé proměnné do paměti.

Nejčastěji se ale setkáváme s algoritmy založených na řešení pomocí barvení grafu, na které je možno problém přiřazení registru převést. Tyto se často nazývají „Chaitin-Briggs“ alokátory, podle jmen autorů dvou stěžejních prací zabývající se tímto problémem [8].

2.2.3 Plánování instrukcí

Moderní procesory zpravidla využívají instrukčního paralelismu. Provádění instrukce je rozděleno do několika nezávislých kroků, které na sebe navzájem navazují tzv. „pipelining“. Takový procesor má v jeden okamžik rozpracovaných několik instrukcí v různých stádiích. Tento přístup zvyšuje výrazně výkon, neboť je hardware v čase mnohem lépe využit. Ideálně je v každém taktu zapojen do výpočtu každý článek řetězce. Mezi prováděnými instrukcemi existují závislosti, které při proudovém zpracování mohou způsobit takzvané hazardy. Například není možné načítat operand instrukce, který ještě

[3]Peephole optimalizace je technika, kdy se optimalizace provádí pouze na několika po sobě jdoucích instrukcích. Ty tvoří takzvané okno, které se postupně posunuje nad vstupním programem [6].

nebyl vypočten. Pokud procesor detekuje takový hazard, pak pozdrží vykonaní instrukce vložením prázdné instrukce. Existuje ale možnost, že by místo prázdné instrukce šla vložit bez způsobení hazardu některá z instrukcí, která je součástí výpočtu.

Plánovaní instrukcí je optimalizace, která se snaží najít takové pořadí instrukcí, které nezmění význam programu a zároveň způsobí co možná nejméně pozastavovaní zřetězeného zpracovaní instrukcí. Některé procesory, většinou zástupci VLIW architektury vůbec neobsahují logiku pro detekci hazardu. To snižuje složitost hardwaru procesoru. V tomto případě je ale zcela na překladači (plánovači instrukcí), aby určil správné pořadí instrukcí nejen z pohledu optimality, ale i správnosti generovaného kódu. [9]

2.3 Vytvoření spustitelného programu

Výstupem překladače je program v jazyce symbolických adres. Ten je nutné dále zpracovat aby jsme dostali program ve spustitelné podobě.

2.3.1 Překladač jazyka symbolických adres

Překladač jazyka symbolických adres (assembler) je program, který transformuje program v jazyce symbolických adres do strojového kódu. To zahrnuje jednak kódování instrukcí do binární formy v jaké jsou prováděny cílovou architekturou a také nahrazení všech vyskytujících se symbolů jejich skutečnými adresami. Assembler také produkuje metadata, které umožňují sestavovacímu programu nebo operačnímu systému s kódem dále pracovat.

Tento, o metadata rozšířený strojový kód, se nazývá objektový soubor. V praxi existují různé formáty objektových souborů napříč operačními systémy. Typický objektový soubor obsahuje následující sekce:

hlavička Hlavička obsahuje základní informace o typu souboru.

segment kódu Často označován jako .text, kódový segment obsahuje samotný strojový kód. Tedy instrukce cílové architektury v binární formě.

datový segment Datový segment obsahuje inicializované globální proměnné programu.

relokační tabulka Relokační tabulka obsahuje reference na místa ve strojovém kódu, které je nutné upravit v případě, že je program načítán do paměti na jinou adresu, než bylo uvažováno assemblerem. Typicky jde o absolutní skoky a přístupy k globálním proměnným. Alternativou je pozičně nezávislý kód[4].

[4]Pozičně nezávislý kód, je strojový kód, který lze možné spustit po načtení na jakoukoliv adresu v paměti. Často se používá v prostředí operačních systémů pro sdílené knihovny.

tabulka symbolů Obsahuje adresy symbolů. Zejména těch externích, tedy těch, které jsou přístupné i z jiného objektového souboru.

Mezi nejpoužívanější typu objektových souborů patří ELF, PE či COFF.

2.3.2 Sestavení programu

Často je výsledný program složen z více zdrojových souborů nebo používá externí či systémové knihovny. Každý takový soubor nebo knihovna jsou tvořeny alespoň jedním objektovým souborem. Linker tyto objektové soubory načte, provede nutné relokace a vyřeší nedefinované symboly (externí závislosti). Výstupem je spustitelný program, který je přípraven pro operační systém k načtení do paměti a spuštění.

Kapitola 3

GCC

GCC (GNU Compiler Collection) je soubor překladačů vyvíjených v rámci projektu GNU. Projekt byl v roce 1984 založen Richardem Stallmanem s cílem vytvořit kompletní svobodný operační systém unixového typu. Jelikož v té době neexistoval žádný svobodný překladač z jazyka C, který by mohl být použit pro vývoj projektu GNU, rozhodl se Stallman vyvinout překladač vlastní.

První verze GCC vyšla v roce 1987, tehdy ještě pod názvem GNU C Compiler. Tato událost měla velký dopad na vývoj otevřeného softwaru obecně, neboť dodnes je jazyk C de facto standard ve vývoji operačních systémů s otevřeným zdrojovým kódem. Dostupnost volně dostupného překladače způsobila rozmach otevřeného softwaru i za hranicemi samotného projektu GNU.

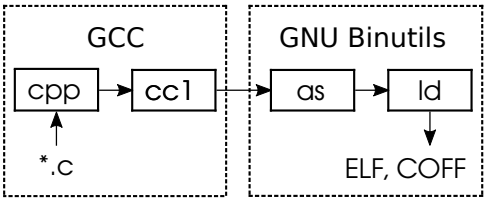
V roce 1997 vznikla odnož projektu nazvané EGCS, která umožňoval provádění lepších optimalizací a obsahovala lepší podporu C++ a jazyka Fortran. Jednalo se o reakci vývojářů na příliš konzervativní přístup oficiálního vedení projektu a velmi pomalé a obtížné začleňování změn. Po dvou letech Free software foundation uznala projekt EGCS jako oficiální překladač projektu GNU a došlo k opětovnému spojení obou projektů.

GCC bylo portováno na mnoho různých architektur od mikrokontrolérů až po nejvýkonnější platformy. Mimo jiné a podporuje několik programovacích jazyků jako C, C++, Fortran, Java či Go a je používáno jako výchozí překladač na většině moderních unixových systému jako jsou různé varianty GNU/Linux či BSD. [10]

3.1 GCC z pohledu uživatele

Nejběžnější případ, kdy se setkáváme s programem GCC, je překlad zdrojového kódu v jazyce C. Většinou je překlad vyvolán uživatelem pomocí zadaní následujícího příkazu:

```
gcc foo.c -o foo .
```



Obrázek 3.1: Překlad C souboru pomocí GCC

Ten způsobí, že je pomocí GCC vytvořen spustitelný soubor `foo` ze zdrojového souboru `foo.c`.

Ve skutečnosti je příkaz gcc pouze prostředník pro interakci s uživatelem a jednotlivé kroky vytváření spustitelného souboru jsou obstarávány samostatnými programy. Příkaz gcc v sobě obsahuje všechny důležité systémové cesty, v nichž se nachází důležité komponenty pro jednotlivé programy. Jako například standardní knihovna či hlavičkové soubory. Tyto cesty jsou do programu gcc napevno uloženy v době překladu překladače.

Prvním programem, který je spuštěn je preprocesor jazyka C, který se nazývá cpp. cpp je součástí projektu GCC a mimo jazyk C podporuje i C++ a Objective-C. Preprocesor načte zdrojový soubor a provede všechny direktivy preprocesoru. To zahrnuje vkládání hlavičkových souborů, expanzi maker, vyhodnocovaní výrazů podmíněné kompilace či odstranění komentářů. Výsledkem je častý C bez direktiv preprocesoru rozšířený o speciální značky přidávající do souboru potřebné informace pro chybové výpisy překladače a případné ladění výsledného programu v debuggeru.

Po odstranění všech direktiv je vstupní soubor předán samotnému překladači. Ten pro jazyk C běžně najdeme pod názvem cc případně cc1. Překladač provádí všechny kroky kompilace jako ověření správnosti syntaxe, různé optimalizace až po generovaní kódu výsledného kódu v jazyce symbolických adres.

Program v tomto zápisu je zpracován překladačem jazyka symbolických adres. Typicky se jedná o GNU Assembler, nazývaný gas nebo jen as. Assembler provádí překlad zápisu symbolických adres do strojových instrukcí. Výstupem je objektový soubor jehož formát se liší v závislosti na použitém operačním systému.

Naposledy je programem gcc zavolán program ld, GNU Linker. Linkeru jsou předány cesty ke knihovnám a tak může všechny objektové soubory spojit do výsledného programu, který je již připraven ke spuštění.

Nutno dodat, že programy ld a as tedy linker a assembler nejsou součástí projektu GCC. Jsou stejně jako GCC součástí GNU, ale samostatného projektu GNU Binutils.

V další části textu bude podrobněji rozebrána vnitřní struktura samotného překladače, tedy programu cc1. Ostatní články řetězce nejsou předmětem této práce.

3.2 Koncept konfigurace GCC

GCC je spíše než samostatný překladač, systém pro sestavení překladačů ze zdrojových kódů. Odtud taky současně používaný anglický název GNU Compiler Colection. V podstatě se jedná o soubor zdrojových kódů a programů, které umožňují v době sestavovaní projektu přeložit překladač pro zadanou cílovou architekturu a s podporou zvolených vysokoúrovňových jazyků.

Zdrojové kódy hlavních průchodů zadní části GCC jsou částečně vygenerovány z definičních souborů cílové platformy 3.5. Takto vytvořená instance GCC je vždy závislá na této cílové platformě. A může tedy generovat kód pouze pro ni. [11]

3.3 Vnitřní formy GCC

GCC používá celkem tři vnitřní formy. GENERIC, GIMPLE, a RTL. Generic slouží jako nezávislá reprezentace vstupního vysokoúrovňového jazyka. Jedná se o vnitřní formu používanou na rozhraní mezi front-endem a optimalizační částí překladače. Všechny platformě nezávislé optimalizace jsou poté realizovány nad podmnožinou GENERICu, která se nazývá GIMPLE. Ten je používán jako jednotná forma mezi všemi optimalizačními průchody v middle-endu. Poslední vnitřní forma je RTL (Register Transfer Language). RTL slouží jako nezávislé rozhraní k back-endům a pro provádění nízkoúrovňový a platformě závislých optimalizací. [12]

3.3.1 GENERIC

Vstupní vysokoúrovňový jazyk je zpracován front-endem a pro každou funkci vstupního jazyka je vytvořen odpovídající abstraktní syntaktický strom. Typy uzlů těchto stromu jsou definovány v $gcc/tree.def$, ty jsou společné pro všechny fron-tendy. Pokud to charakter vstupního jazyka vyžaduje, může front-end definovat i své vlastní typy uzlů. Pokud se strom skládá pouze z předdefinovaných uzlů, pak se jedna reprezentaci v GENERIC formátu.

3.3.2 GIMPLE

GIMPLE je hlavní vnitřní reprezentace používaná pro většinu optimalizací. Jedná se o podmnožinu reprezentace GENERIC. Vzniká rozložením složitějších struktur GENERICU do n-tic s nejvíce třemi operandy. Jde tedy o variantu tříadresního kódu. GIMPLE byl inspirován jazykem SIMPLE IL použitým v McCAT překladači. Pro provádění většiny optimalizací je navíc často převeden do formy splňující vlastnosti SSA. V dokumentaci se tato forma GIMPLE někdy nazývá jako „Tree SSA“.

Část překladače která realizuje převod ze stromové struktury do GIMPLE se nazývá $gimplifier$. Pokud front-end vkládá do abstraktních stromů vlastní

typy uzlů, pak musí implementovat funkci, která je schopna převést tyto uzly do GENERIC nebo přímo do GIMPLE formy.

Během vyváření GIMPLE dochází k převodu řídících struktur na podmíněné skoky. Forma která ještě obsahuje řídící struktury se nazývá „High GIMPLE“ a po konverzi na skoky „Low GIMPLE“.

3.3.3 RTL

RTL (register transfer language) je vnitřní forma používaná zadní části GCC, tedy generátorem kódu. Jedná se o abstraktní, univerzální nízkoúrovňový jazyk, blízký jazyku symbolických adres. Poprvé byla tato reprezentace použita Jackem Davidsonem and Christopherem Fraserem v jejich práci na platformě nezávislém generátoru kódu [6]. Dnes je tato forma používaná v mnoha překladačích.

V GCC slouží ke dvěma hlavním účelům. Jednak jako vnitřní reprezentace používaná k některým nízkoúrovňovým optimalizacím během posledních fází překladu. Tedy během generování kódu. Dále pak jako jazyk pro popis instrukcí cílové platformy. Tyto definice jsou součástí definičních souborů cílové platformy tzv. „machine descriptions“.

V textové podobě se RTL zapisují pomocí S-výrazů. Podobným způsobem jako programovací jazyk Lisp. Jazyk RTL se skládá z celkem čtyř základních konstruktů. Čísel, vektorů, řetězců a RTL výrazů. Řetězce se většinou vyskytují ve výrazech obsahující návěští jako například instrukce skoku nebo volaní funkce. Spolu s vektory se ale vyskytují i v RTL konstruktech pro popis cílové architektury. Nejdůležitější je poslední skupina, RTL výrazy. Zkráceně nazývané RTX. [12]

3.3.3.1 Výrazy RTL

Výrazy jsou základním stavebním blokem jazyka RTL. Jde v podstatě o uspořádanou n-tici, skládající se z identifikátoru výrazu a jeho operandů Operandem může být kterýkoliv ze základních RTL objektů. Samozřejmě i další výraz. Typy jednotlivých argumentů jsou pevně určeny identifikátorem výrazu.

Výrazy v RTL mají poměrně široké využití. Některé slouží k popisu objektů vnitřní formy jako je například registr nebo paměťová buňka. Jiné reprezentují samotné operace s těmito objekty. Existují i výrazy, které nemají přímý obraz v reálné architektuře a jsou spíše strukturami vlastní implementace back-endu. Do této skupiny patří například výraz instrukce nebo výrazy používané v rámci MD souborů.

Definice výrazů se nachází v souboru gcc/rtl.def. Ten mimo jiné definuje i jednotlivé třídy do kterých jsou výrazy rozděleny. Tabulka 3.1 ukazuje některé z tříd RTL výrazů a jejich zástupce.

Tabulka 3.1: Ukázka některých tříd RTL výrazů

třída RTL výrazu	**popis**
RTX_OBJ	Výraz pro vlastní objekt jako (REG), (MEM) nebo (SYMBOL_REF).
RTX_CONST_OBJ	Výraz reprezentuje konstantní objekt.
RTX_COMPARE	Výraz pro nekomutativní porovnání jako (GT), (LE) a jiné..
RTX_COMM_COMPARE	Výraz pro komutativní porovnání například (NE), (EQ).
RTX_UNARY	Výraz pro unární operace jako (NEG), (NOT). Třída obsahuje i výrazy pro znaménkové a bezznaménkové rozšíření.
RTX_COMM_ARITH	Výraz pro komutativní binární operace jako (PLUS), (MULT) či (AND).
RTX_BIN_ARITH	Výraz pro nekomutativní binární operace jako (MINUS), (DIV) nebo (MOD).
RTX_TERNARY	Pro výrazy se třemi operandy , jako např. (IF_THEN_ELSE).
RTX_INSN	Výraz reprezentující celé instrukce. Jako např. (INSN), (JUMP_INSN) nebo (CALL_INSN) viz 3.3.3.3.
RTX_EXTRA	Ostatní výrazy používané v MD souborech 3.5.5. Dále výrazy měnící stav modelu (SET), (USE). A také falešné instrukce jako CODE_LABEL.

3.3.3.2 Datové typy

Jazyk RTL, jako nezávislá vnitřní forma, používá také vlastní platformě nezávislé typy. Každý výraz v RTL, který se odkazuje na nějaká data zároveň obsahuje informaci o jejich typu. V textovém zápisu RTL výrazu se datový typ zapisuje za kód výrazu oddělením dvojtečkou.

Typů v RTL existuje poměrně velké množství. Velikost každého z nich je ale definována jako celistvý násobek jednoho bajtu, jako základní jednotky. Její velikost určuje implementace back-endu pomocí makra BITS_PER_UNIT. Následující tabulka ukazuje základní celočíselné typy a jejich vzájemný vztah.

3.3.3.3 Insns

Pomocí dosud popsaných RTL výrazů je možné abstraktním jazykem popsat sémantiku jednotlivých instrukcí cílové platformy. V jazyce symbolických adres, který je výstupem překladače, ale instrukce nesou (někdy implicitně) i další informace, než je význam jejich vlastní operace. Například, která in-

Tabulka 3.2: Ukázka některých datových typů RTL

typ	**význam a použití**
BImode	Reprezentuje jediný bit.
QImode	Reprezentuje jednobajtovou celočíselnou hodnotu.
HImode	Reprezentuje dvoubajtovou celočíselnou hodnotu.
SImode	Reprezentuje čtyřbajtovou celočíselnou hodnotu.
DImode	Reprezentuje osmibajtovou celočíselnou hodnotu.
PSImode	Reprezentuje typ velikosti čtyři bajty, hodnota ale nevyžívá celý tento prostor. Vhodne například pro některé ukazatele.
PDImode	Reprezentuje typ velikosti osum bajtů, hodnota ale nevyžívá celý tento prostor. Vhodne například pro některé ukazatele.
VOIDmode	Speciální RTL typ pro případy kdy na daném typu nezáleží nebo není specifikován.

strukce následuje či zda není dané instrukci přiřazeno nějaké návěští. Z těchto důvodů RTL definuje speciální skupinu výrazů souhrnně zvaných *insns*. Tyto výrazy slouží jako kontejner pro samotný výraz reprezentujícího význam dané instrukce a tyto dodatečné informace. Každý výraz typu *insn* obsahuje čtyři základní argumenty. Unikátní identifikátor, který ji odlišuje od ostatních *insn* výrazů. Druhým parametrem je identifikátor předchozí instrukce a třetím parametrem je identifikátor instrukce následující. Jedná se tedy o dvojitý spojový seznam. Tato sousednost RTL instrukcí vyjadřuje jejich vzájemnou pozice v programu. Jako identifikátory instrukcí jsou použity celá čísla.

Existuje několik druhů RTL výrazů reprezentující instrukce:

insn Základní výraz pro instrukci, která nerealizuje skok v programu ani volaní funkce. Obsahuje tři základních argumenty definující sousednost instrukcí v programu. A další čtyři, které jsou vysvětleny v tabulce 3.6.

jump_insn Výraz reprezentující instrukce, které jsou skokem v programu. Kromě stejných argumentů jako *insn*, obsahuje navíc argument JUMP_-LABEL, který obsahuje identifikátor určující speciální instrukci zvanou code_label, která je cílem skoku.

call_insn Výraz pro instrukce, které realizují volaní funkce. Obsahuje všechny argumenty jako *insn* a speciální argument CALL_INSN_FUNCTION_-USAGE. Jedná se o seznam výrazů *use*, *clobber*, *set* a *mem* které definují registry a pozice na zásobníku používané volanou funkcí. Například výraz *clobber* definuje registry, které jsou přepisovány volanou funkci a mem většinou popisuje argumenty na zásobníku.

code_label Speciální výraz reprezentující návěští. Kromě základních argumentů obsahuje navíc argument CODE_LABEL_NUMBER který jednoznačně identifikuje dané návěští v rámci celého vstupního programu.

Tedy i v rámci všech překládaných funkcí. Druhým je LABEL_KIND kterým se odlišují návěští, které jsou alternativním vstupním bodem do funkce.

note Výraz obsahující dva zvláštní argumenty. Slouží jako nositel informace o pozici ve zdrojovém souboru a jako značka vymezující rozsah platnosti poměrných nebo obsluhy výjimek. Někdy se lze ve výpisu setkat s hodnotou NOTE_INSN_DELETED, která vznikne po smazaní původní instrukce některým z optimalizačních průchodů.

Výpis 3.1: RTL reptezentace krátké funkce

```
(note 4 1 16 2 NOTE_INSN_BASIC_BLOCK)
(note 16 4 3 2 NOTE_INSN_PROLOGUE_END)
(note 3 16 15 2 NOTE_INSN_FUNCTION_BEG)
(insn 15 3 12 2
  (set (reg:SI 0)
    (plus:SI
      (reg/v:SI 5)
      (const_int 4)))
  214 {*leasi}
  (expr_list:REG_DEAD
    (reg/v:SI 5])
    (nil)))
(insn 12 15 18 2 (use (reg/i:SI 0))
  (nil))
(jump_insn 18 12 17 2 (simple_return)
  (nil))
```

Tabulka 3.3: Tabulka argumentů *insn* výrazů

PATTERN	Vlastní výraz popisující sémantiku instrukce. Musí jít o výraz mající vedlejší efekt. Jako například *set*, *call*, *use*, *clobber*, *return* a jiné.
INSN_CODE	Číslo instrukčního vzoru v souboru vzorů 3.5.5, které odpovídá této *insn*. Pokud ještě nedošlo k porovnaní, pak obsahuje hodnotu -1. V textovém výpisu je doplněn i symbolický název vzoru.
LOG_LINKS	Seznam výrazů definující závislosti instrukci uvnitř základního bloku. V současné době je tato položka považována za zastaralou.
REG_NOTES	Seznam výrazů poskytující různé přidružené informace o instrukci. Nejčastěji jde o informace o použitých registrech.

3.4 Back-end GCC

Zadní část překladače GCC je generátor kódu operující nad vnitřní formou RTL. Všechny transformace tedy probíhají nad touto formou. Definiční soubory pro danou cílovou platformu pak poskytují back-endu potřebné informace a funkce k transformování programu v RTL do tvaru, kdy ho lze jednoduše přepsat z RTL reprezentace do jazyka symbolických adres. [12]

3.4.1 Hlavní průchody back-endu

Celý proces převodu vnitřní formy RTL do platformě závislých instrukcí se skládá z několik samostatných kroků [12]:

generovaní RTL Tento průchod, někdy také nazývaný „expand“. je prvním průchodem zadní části GCC. Je zodpovědný za převod programu z GIMPLE reprezentace do RTL. To je realizováno pomocí standardních instrukčních vzorů. Dokumentace uvádí, které ze standardních jmen instrukční vzorů musí být povinně v back-endu definovány. U nepovinných je schopen generátor RTL zvolit alternativní způsob jak převést daný GIMPLE konstrukt do RTL. V případě však že není k dispozici požadovaný vzor, končí překlad interní chybou.

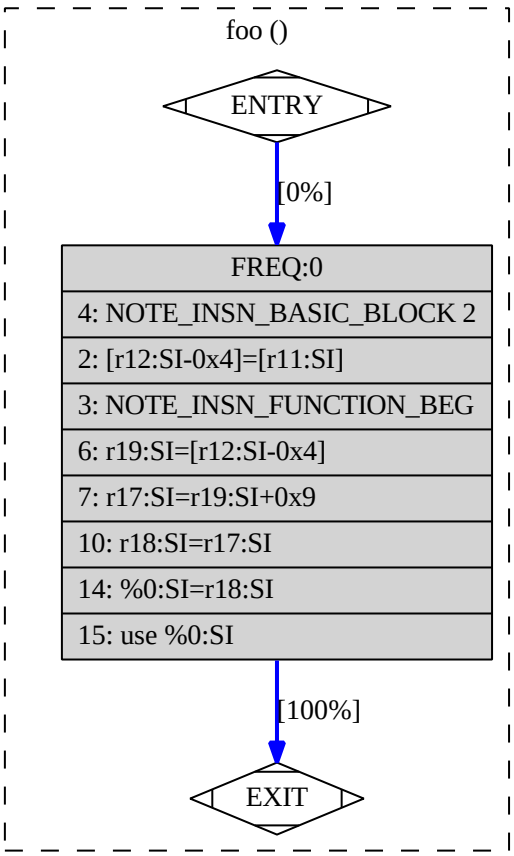
optimalizace Po vygenerovaní vnitřní formy RTL následuje několik optimalizačních průchodů. Jako eliminace společných podvýrazů nebo konverze podmínek.

kombinování instrukcí Tento průchod taky někde nazývaný „combine“ se snaží sloučit dvě nebo tři RTL instrukce s datovou závislostí do jedné. Jedná se o tedy o jednoduchý *peephole* optimalizátor. Spolu s prvním průchodem generující RTL jsou nejvíce tyto průchody odpovědny za výběr výsledných instrukcí.

alokování registrů Jedna z nejkomplikovanějších částí nejen back-endu ale i celého GCC. Ve skutečnosti jde o několik průchodů s cílem nahradit nepřípustné argumenty instrukčních vzorů validními alternativami, a všechny pseudoregistry nahradit registry pevnými.

Přiřazení pevných registrů na interprocedurální úrovni provádí průchod IRA (integrated register allocator). Následuje průchod LRA (local register allocator). Ten nahradí pseudo registry těmi alokovanými a zbylé výskyty nahradí přístupy do paměti. Dále musí nahradit všechny operandy RTL výrazů, která nesplňují omezení dané instrukce. Dále je LRA také zodpovědný za generovaní prologu a epilogu funkce. Tedy zálohovaní a obnovení registrů.

výstup Posledním stupněm je průchod nazývaný „final“. Ten zapíše do výstupního souboru pro každou RTL instrukci, její odpovídající reprezentaci v jazyce symbolických adres. Tento průchod využívá definice výstupních šablon z instrukčních vzorů.



Obrázek 3.2: Funkce v RTL ihned po vygenerovaní z GIMPLE

3.5 Definiční soubory cílové platformy

Pro rozšíření GCC o další cílovou architekturu, je třeba cílovou platformu a to zejména instrukční sadu procesoru popsat. To je realizováno pomocí dvou mechanizmů. Obecné vlastnosti architektury jsou definovány souborem C maker, formát jednotlivých instrukcí a jejich význam se popisuje jazykem RTL. [12]

Celkem se implementace skládá ze tří hlavních souborů:

target.h Obsahuje C makra definující základní charakteristiky cílové architektury a vlastnosti specifické pro použité ABI.

target.md Definice jednotlivých instrukcí cílové architektury pomocí speciálních konstrukcí jazyku RTL. Tento soubor slouží jako vstup programu, který v době překladu GCC vytvoří dle těchto definic kód generátoru kódu cílové platformy.

target.c Obsahuje implementace funkcí, které předefinovávají výchozí chování generátoru pro konkretní platformu v různých situacích. Takzvané „target hooks“. V podstatě plní podobnou úlohu jako soubor *target.h*.

3.5.1 Model registrů

GCC interně rozlišuje dva základní typy registrů. Pevné registry, které skutečně existují v daném procesoru. Často nazývané „hard“ registry. A poté registry dočasné, nazývané „pseudoregistry“. Těch je na rozdíl od pevných neomezený počet a jsou používány vnitřní formou RTL.

Během generování výsledného kódu jsou pseudo registry nahrazeny alokátorem registry pevnými. Pevné registry mohou být navíc fixní. Tento příznak se nastavuje registrům se speciálním významem, které nemůže alokátor registrů použít k běžným výpočtům. Typicky se jedná například o ukazatel zásobníku a ukazatel rámce zásobníku.

Je běžné že procesor obsahuje více typů registrů. Například speciální registry pro výpočty v plovoucí řádce. Nebo speciální indexovací registry, které je možné použít jen v určitých adresovacích módech. Proto také back-end definuje makra popisující registrové skupiny a příslušnost registrů do těchto skupin dle možného použití v různých typech instrukcí.

3.5.2 Popis maker target.h

Většina základních charakteristik cílové architektury jako je velikost datových typů, registrové třídy, registry nebo chování zásobníku se v GCC popisuje pomocí soustavy C maker. V tabulce 3.4 jsou vysvětlena základní často používaná makra, které back-end definuje v soubor *target.h*

Tabulka 3.4: Tabulka maker GCC back-endu

Schéma paměti	
BITS_BIG_ENDIAN	Pokud je nejvyšší bit bajtu bitem s největším významem, pak musí být nastaveno na 1.
BYTES_BIG_ENDIAN	Pro systémy typu big-endian nastaveno na hodnotu 1.
WORDS_BIG_ENDIAN	Toto makro určuje pořadí víceslovných objektu v paměti. Hodnota 1 znamená nejvýznamnější slovo na nižší adrese.

UNITS_PER_WORD	Počet slabik (nejmenší adresovatelných jednotek) v jednom slově.
STRICT_ALIGNMENT	Definuje zda je nutné vždy dodržovat požadované zarovnaní typů v paměti. Nastaveno na 1 pokud instrukce nepracují s nezarovnanými daty.
FUNCTION_BOUNDARY	Hodnota v bitech definující požadované zarovnání první instrukce funkce.
BIGGEST_ALIGNMENT	Nejvyšší možné zarovnaní datového typu v bitech.
PARM_BOUNDARY	Běžné zarovnaní parametrů funkce na zásobníku v bitech. Argumenty všech typů budou zarovnaný na alespoň tuto velikost.
STACK_BOUNDARY	Definuje nejmenší možné zarovnaní ukazatele zásobníku na daném procesoru.
Datové typy	
SHORT_TYPE_SIZE	Velikost C typu short v bitech.
INT_TYPE_SIZE	Velikost C typu int v bitech.
LONG_TYPE_SIZE	Velikost C typu long v bitech.
DEFAULT_SIGNED_CHAR	Definuje, zda je typ char interně uložen jako znaménkový čí nikoliv.
Rozložení a popis zásobníku	
STACK_GROWS_DOWNWARD	Nadefinováno, pokud přidaní dat na zásobník sníží adresu ukazatele zásobníku.
FRAME_GROWS_DOWNWARD	Je nutné nadefinovat, pokud jsou lokální proměnné uloženy na negativních adresách vzhledem k ukazateli rámci zásobníku.
STACK_POINTER_OFFSET	Relativní adresa prvního z odchozích argumentů (argumenty volaných funkcí) vzhledem k ukazateli zásobníku.
FIRST_PARM_OFFSET()	Relativní adresa prvního argumentu vzhledem k ukazateli argumentů.
Popis registrů	
REGISTER_NAMES	Pole konstantních C řetězců definující názvy jednotlivých registrů.

REG_CLASS_NAMES	Pole konstantních C řetězců obsahující názvy registrových skupin (vhodné pro ladící výpisy).
REG_CLASS_CONTENTS	Pole celých čísel určující příslušnost registrů do registrových skupin. Interpretace je následující. Pokud register r je obsažen v $n-té$ registrové skupině, pak $r-tý$ nejvýznamnější bit čísla na $n-té$ pozici v poli musí být nastaven na 1.
FIXED_REGISTERS	Pole obsahující číslo 1 na pozici n pokud $n-tý$ pevný registr je fixní. Tedy má speciální význam a nebude použit alokátorem. Pro alokovatelné registry obsahuje 0.
CALL_USED_REGISTERS	Obdobně jako FIXED_REGISTERS nastavuje 1 pro registry, které je možné použít ve funkci bez předchozího zálohovaní jejich hodnoty. Včetně těch nastavených ve FIXED_REGISTER. Ostatní s hodnotou 0 budou uloženy a obnoveny na začátku respektive konci funkce.
Registry adresující zásobník	
STACK_POINTER_REGNUM	Číslo registru používaný jako ukazatel zásobníku. Musí být zároveň nastaven jako fixní pomoci makra FIXED_REGISTERS.
FRAME_POINTER_REGNUM	Číslo registru, který je použit jako ukazatel rámce zásobníku.
ARG_POINTER_REGNUM	Číslo registru který je použit jako ukazatel argumentů funkce. Pokud není schodný s ukazatelem rámce zásobníku, pak musí být nastaven jako fixní pomocí FIXED_REGISTERS případně nastavit jeho eliminaci pomocí ELIMINABLE_REGS.

HARD_FRAME_POINTER_ REGNUM	Pokud není relativní pozice lokálních proměnných známá do doby než proběhne alokace registrů (např. kvůli neznámému počtu zálohovaných registrů), pak toto makro definuje speciální fixní registr, který je interně použit dokud neproběhne alokace.
REG_ALLOC_ORDER	Pole čísel pevných registrů v pořadí, ve kterém je mé alokátor používat. Nejnižší index bude použit jako první.
Ostatní	
HARD_REGNO_MODE_OK	Výraz určující, zda zadaný RTL datový typ může být uložen v daném registru.
ELIMINABLE_REGS	Pole dvojic registrů adresující zásobník. Přítomnost dvojice *(a,b)* v poli značí, že se může překladač pokusit nahradit register *a* registrem *b*. Je-li navíc umožněno platformní funkcí TARGET_CAN_ELIMINATE.
INITIAL_ELIMINATION_OFFSET	Toto makro určuje vzájemnou vzdálenost dvou ukazatelů zásobníků na konci prologu. Toto makro je povinné pokud je definováno ELIMINABLE_REGS

3.5.3 Platformě specifické funkce target.c

Soubor *target.c* implementuje platformě specifické funkce. Umožňuje pro cílovou architekturu předefinovat některou z funkcí generátoru funkcí s vlastním chováním. Back-end GCC volá tyto funkce během generovaní kódu ve chvíli, kdy je potřeba s kódem provést nějakou platformě závislou akci.

Jde o modernější náhradu mechanizmu maker ze souboru *target.h*. Definice těchto maker jsou totiž v době překladu GCC dosazena do na různá místa ve zdrojovém kódu vnitřních algoritmů back-endu. To v podstatě znemožňuje sestavit generátor s podporou několika cílových architektur. Naproti tomu soubor *target.c* definuje C strukturu s názvem *targetm* která obsahuje ukazatele na implementace platformě specifických funkcí. Díky této nepřímé vazbě je dosaženo platformní nezávislosti algoritmů generátoru i po sestavení projektu.

Tabulka 3.5: Některé platformě závislé funkce

název funkce	**význam**
TARGET_CAN_ELIMINATE(A,B)	Tato funkce vrací *true* pokud je v aktuální funkci možné nahradit registr A registrem B. Registry A, B je dvojice definovaná makrem ELIMINABLE_REGS.
TARGET_FRAME_POINTER_REQUIRED	Vrací *true* pokud aktuální funkce vyžaduje ukazatel rámce zásobníku.
TARGET_FUNCTION_VALUE_REGNO_P(R)	Funkce vrací *true* pokud registr R je použit pro navracení funkce.
TARGET_RETURN_IN_MEMORY(T,F)	Vrací *true* pokud návratová hodnota typy T ma být z funkce typu F vrácena pomocí zásobníku.

3.5.4 Kontext funkce a vlastní data

Back-end definuje C strukturu *function*, která obsahuje data vztahující se k dané funkci. Tato struktura je v platformě specifických funkcí dostupná pod proměnou *cfun*.

V některých případech jako jsou funkce generující prolog nebo počítající relativní adresy pro eliminaci registrů, je potřeba některá vypočtená data asociovat s danou funkcí pro pozdější použití. Z toho důvodu *function* obsahuje proměnnou *machine*, což je C struktura, kterou je možné vlastím způsobem definovat v souboru *target.c* a používat k ukládání vlastních dat. Použití globálních proměnných pro tento účel není bezpečné a naprosto selže v případě vnořených funkcí.

3.5.5 Instrukční vzory target.md

Popis instrukční sady cílové architektury se provádí pomocí takzvaných instrukčních vzorů („instruction patterns“). Instrukční vzory jsou speciální RTL výrazy, které nejsou součástí vnitřní formy. Nejsou tedy přímo používaný v RTL reprezentaci programu v době jeho překladu. Místo toho slouží jako vstup při vytváření back-endu v době překladu GCC.

Instrukční vzory slouží jednak k převodu z GIMPLE do RTL pomocí takzvaných standardních vzorů a zároveň k převodu RTL do instrukcí cílové architektury. Definují jak vztahy mezi konstrukty formy GIMPLE a RTL, tak i mezi RTL a instrukcemi cílové architektury.

3.5.6 Definování instrukčních vzorů

Nejzákladnějším možnost definování instrukčního vzoru je pomocí RTL výrazu *define_insn*. Tento výraz obsahuje alespoň první čtyři z následujících

argumentů:

jméno vzoru Volitelný argument pro označení vzoru jako standardního Uvedení jména vzoru způsobí, že bude daný vzor využit při generování RTL. Pokud instrukce cílové architektury neodpovídá žádnému z standardizačních vzorů, jako jméno se uvádí prázdný řetězec Případně lze uvést libovolný název začínající znakem *. Takové názvy jsou algoritmy GCC ignorovány, mohou ale usnadnit ladění během vývoje back-endu.

šablona Šablona je RTL výraz definující, které RTL instrukce odpovídají danému instrukčnímu vzoru 3.5.6.1.

podmínka Podmínka je řetězec obsahující kód v jazyce C. Pokud se generátor pokusí pokrýt RTL kód tímto vzorem pak je tento kód spuštěn. To nastane teprve pokud pokrývanému kódu vyhovuje šablona. Ve chvíli, kdy je podmínka vyhodnocována, je již možné z tohoto kódu k argumentům přistupovat. Vyhodnotí-li se podmínka negativně, pak pokrytí není provedeno.

výstupní šablona Výstupní šablona je textový řetězec, který určuje reprezentaci instrukčního vzoru v jazyce symbolických adres cílové platformy. Jedná se přímo o text, který je vytištěn na výstup překladače. Znak % slouží k uvození čísel argumentů na místech jejich výskytů ve výstupním řetězci Někdy je za znakem procenta navíc speciální znak určující formát argumentu. Například %l slouží k vytisknutí názvu návěští.

V některých případech jsou však různé varianty syntaxe instrukce natolik pestré, že je nejde vyjádřit pomocí standardní výstupní šablony. Pak je možné místo textu šablony zapsat C code uvozený znakem hvězdičky. V některých případech může šablona generovat několik instrukcí. V takovém případě se řetězce jednotlivých instrukcí oddělují znaky '\;'.

atributy instrukce Nepovinný argument obsahující pole dodatečných atributu asociovaných s instrukčním vzorem. Ty slouží především pro pokročilé optimalizace v back-endu zaměřující se na proudové zpracovaní instrukcí.

3.5.6.1 RTL šablony

Šablona je RTL výraz, který určuje jaké instrukce v jazyce RTL je možné pokrýt daným instrukčním vzorem, tedy i instrukcí cílové platformy. Pro standardní instrukční vzory navíc šablona určuje jakým způsobem jsou dosazením operandů vytvářeny RTL instrukce z vnitřní formy GIMPLE.

Tohoto je dosaženo pomocí speciálních zástupných RTL výrazů, které určují jaké druhy výrazů se mohou vyskytnout na dané pozici v RTL instrukci. Nejdůležitější z nich je výraz *match_operand*, který má následující formu:

`(match_operand:m n predikát omezení)` .

V případě, že jde o standardní instrukční vzor, pak v čase generovaní RTL bude tento výraz nahrazen n-tým argumentem z GIMPLE uzlu. To znamená, že během generovaní RTL se operandy nekontrolují a jsou pouze na své pozice dosazeny. Pokud dochází k pokrývaní RTL reprezentace instrukčními vzory, výraz vyskytující se pozici výrazu *match_operand* bude použit jako n-tý operand instrukce pokud splňuje predikát:

predikát Predikát je název C funkce, která přijímá dva argumenty. Prvním je RTL výraz, který se snaží být dosazen na tuto pozici v šabloně. Druhým argumentem je datový typ m uvedený za dvojtečkou ve výrazu *match_operand*. Tato funkce vrací logickou hodnotu podle toho zda daný výraz splňuje definované predikátem a může být použit na této pozici. GCC definuje několik platformě nezávislých predikátů jako například register_operand, memory_operand nebo immediate_operand a další. Existuje i možnost nadefinovat vlastní specifické predikáty.

omezení Omezení kladou kromě predikátu další požadavky na druh argumentu. Na rozdíl od predikátu nemají žádný vliv na to, zda šablona pokrývá daný kus RTL kód či nikoliv. Smyslem omezení je přesněji určit jakého typu může daný argument být. Například, zda je argumentem instrukce registr nebo adresa v paměti. Omezení mají význam u instrukčních vzorů s více výstupními šablonami 3.5.6.2. Lze je však použít i pro instrukční vzory bez alternativních verzí. Pokud v tomto případě je omezení více restriktivní než predikát, pak musí LRA průchod zajistit, že je hodnota k dispozici na přípustném místě. Například hodnota bude předem načtena do registru, pokud instrukce nepracuje s hodnotami v paměti. Omezení jsou zapisovány jako řetězce, v nichž každý znak reprezentuje jeden druh operandu který je pro instrukci validní. Tento řetězec navíc muže být uvozen dalším speciálním znakem, takzvaným modifikátorem. Ten například definuje, zda je argument pouze čten nebo zapisován. Slouží k poskytnutí dodatečných informací pro LRA průchod.

Tabulka 3.6: Tabulka základních omezení a modifikátorů

omezení	**význam**
m	Povoluje jakýkoli operand reprezentující hodnotu v paměti.
r	Operand je obecný registr.
i	Přímá hodnota, tedy konstanta. Povoluje i hodnoty, které budou známy až v době překladu do strojového kódu nebo sestavovaní koncového programu. Jako například návěští.

n	Přímý operand s hodnotou známou v době překladu.
o	Argument je adresa v paměti, která je posouvatelná. Tedy, že po přičtení menšího čísla (ne většího než je počet bajtů velikosti typu argumentu) vznikne opět validní adresa.

modifikátor	**význam**
=	Značí, že hodnota operandu je danou instrukcí měněna.
%	Znamená, že tento a následující operand jsou komutativní a překladač může argumenty vzájemně zaměnit.

Výpis 3.2: Ukázka šablony instrukčního vzoru

```
(set (match_operand:SI 0 "register_operand" "=r")
  (minus:SI
    (match_operand:SI 1 "register_operand" "r")
    (match_operand:SI 2 "register_operand" "r")))
```

3.5.6.2 Instrukční vzory s více výstupními šablonami

V některých případech má instrukce více možných variant podle typů jejich argumentů. Například standardní jméno instrukčního vzoru *movsi* reprezentuje vzor, který přesouvá hodnotu mezi dvěma místy. Většina procesorů typu RISC nedovoluje přesun mezi dvěma adresami v operační paměti. Navíc mají často tyto operace různé operační znaky v závislosti na typu argumentu. Z tohoto důvodu je umožněno v RTL a výstupní šabloně uvést více alternativ daného vzoru.

Výpis 3.3: Ukázka instručního vzoru s alternativami

```
(define_insn "movsi"
  [(set (match_operand:SI 0 "general_operand" "=r,m,r")
     (match_operand:SI 1 "general_operand" "r,r,m"))]
  ""
  "@
  MOV %0, %1
  ST %1, %0
  LD %0, %1"
)
```

V příkladu 3.3 je vidět zápis instrukčního vzoru s alternativami.

3.5.7 Definice pro generátor RTL

V některých případech neexistuje v instrukční sadě cílové platformy instrukce odpovídající některému z povinných standardních instrukčních vzorů. Je možné ale takovou operaci realizovat na cílové platformě pomocí sekvence více instrukcí. K definování tento sekvence instrukcí slouží RTL výraz *define_expand*. Tento výraz je podobný výrazu *define_insn* až na některé odlišnosti:

jméno Jméno standardního vzoru. V tomto případě povinné, neboť tento instrukční vzor nemá jiné využití než během generovaní RTL.

šablona Šablona RTL. Vektor jednotlivých RTL instrukcí realizující požadovanou operaci, která není dostupná na cílové platformě ve formě jedné instrukce.

podmínka Stejný výraz jako u definice instrukčního vzoru pomocí *define_insn*. Většinou se používá k ovlivnění dostupnosti daného vzoru v čase běhu překladače. V případě, že některé varianty cílového procesoru neobsahují některé z instrukcí instrukční sady, může uživatel pomocí parametru pří spuštění GCC některé vzory tímto povolit.

přípravný příkaz Řetězec obsahující C výrazy, které jsou vykonány před tím než je vygenerována instrukce dle RTL šablony. Ve většině případů tyto příkazy nastavují dočasné hodnoty použité jako argumenty výrazu v šabloně. Mohou generovat i další RTL instrukce pomocí volaní emit_insn. V tomto řetězci lze navíc použít dvě speciální makra *DONE* a *FAIL*. První způsobí, že se generování zastaví, šablona nebude použita pro generovaní výstupu a ten obsahuje pouze instrukce vygenerované pomocí volaní *emit_insn*. Druhé je podporováno jen pro některé standardní vzory a způsobí odmítnutí vzoru a vyhledání jiného.

atributy Obdobně jako u *define_insn* vektor obsahující atributy instrukčního vzoru.

3.5.8 Definice vzorů pro pokročílé optimalizace

Kromě základního výrazu *define_insn* pro definici instrukčních vzorů je možné v souboru *target.md* definovat i další výrazy, které umožňují použití pokročilejších optimalizací:

define_split Tento výraz popisuje jak rozdělit komplexní RTL výraz na několik RTL instrukcí. Často lze komplexní instrukci nahradit posloupností více jednodušších instrukcí. Toho lze využít v několika případech. Pokud instrukční vzor produkuje hned několik instrukcí, pak nelze umístit tuto RTL instrukci do delayslotu. Jindy může rozklad komplexní instrukce

způsobit lepší využití proudového zpracovaní oproti jedné instrukci složitější. Dalším případem využití je „combine“ 3.4.1 průchodu, který se pokouší rozdělit komplexní instrukce vzniklé zkombinováním jednodušších, ale nelze je pokrýt žádným existujícím vzorem. Tento výraz má celkem čtyři argumenty. Prvním je RTL výraz, který má být rozdělen. Druhým je podmínka, kterou musí splňovat, aby mohl být nahrazen. Třetí je seznam RTL instrukcí nahrazující původní výraz. A posledním argumentem je přípravný příkaz obdobně jako ve výrazu *define_expand*.

define_peephole2 Definice určující jak nahradit posloupnost RTL instrukcí jinou posloupností. V podstatě se jedná o totožný výraz jako je *define_split*, avšak s tím rozdílem, že nenahrazuje jednu RTL instrukci, ale několik po sobě jdoucích instrukcí.

define_peephole Umožňuje nahradit několik po sobě jdoucích RTL instrukcí rovnou instrukcemi cílové architektury pomocí výstupní šablony. K těmto pokusům o nahrazení dochází až v poslední fázi generování kódu, těsně před vyhodnocováním výstupním šablon. Navíc jen pokud jsou zapnuté optimalizace. Z tohoto důvodu může být každá z RTL instrukcí v nahrazované sekvenci pokrytelná nějakým z instrukčních vzorů deklarovaných pomocí *define_insn*. Jinak překlad programu s vypnutými optimalizacemi skončí interní chybou.

Kapitola 4

LLVM

LLVM je soubor vzájemně spolupracujících programů a aplikačních knihoven pro vývoj překladačů a optimalizátorů.

Projekt byl založen v roce 2000 v rámci výzkumné skupiny vedené Vikramem Advem a Chrisem Lattnerem na Univerzitě Illinois v Urbana Champaign. Prvotní záměr projektu byl výzkum technik dynamického překladu pro statické a dynamické jazyky. Od té doby se ale LLVM rozrostlo a dnes je využíváno v mnoha projektech různého typu [13]. Z tohoto důvodu také upustilo od původního názvu Low level virtual machine, který odkazoval na počáteční záměr projektu a dnes je původní zkratka LLVM jeho samotným jménem.

Nejhlavnějšími částmi projektu je platformě nezávislý optimalizátor a generátor kódu LLVM Core a překladač jazyků C, C++ a Objective-C zvaný Clang. Kromě těchto jazyků je podporovaný i Fortran a Ada v rámci projektu DragonEgg, který používá přední části překladače z projektu GCC. Navíc LLVM v sobě zahrnuje i ostatní programy navazující na samotný překladač jako třeba debugger, assembler či standardní knihovna jazyka C++. Díky tomu se stále více setkáváme s nasazováním LLVM na různých derivátech systému UNIX jako alternativa k dnes již tradiční dvojici GCC a GNU Binutils. [14]

4.1 Popis částí LLVM

Projekt LLVM je celá infrastruktura pro vývoj překladačů, optimalizátorů, křížových překladačů, překladačů jazyků symbolických adres a ostatních programů souvisejících s překladem jazyků vyšší úrovně do strojového kódu. Ať již pro reálné nebo i virtuální stroje. Jedná se o soubor modulů, knihoven a přidružených programů pro vývojáře i pro koncové uživatele. [14]

4.1.1 LLVM Core

LLVM Core je středem a hlavní částí LLVM. Většinou se pojmem LLVM míní právě tato část projektu. Jde o kolekci knihoven a programů vystavěných kolem vnitřní formy nazývané LLVM IR.

Jednotlivé funkce LLVM infrastruktury jsou přístupné pomocí následující sady programů pracujících s touto vnitřní formou. Nejedná se o samostatné projekty postavené nad LLVM, ale spíše o programy umožňující přístup k funkcionalitám infrastruktury bez nutnosti implementovat vlastní nástroje. Slouží vývojářům LLVM případně projektů na něm postavených. Cílový uživatel by se s nimi však neměl běžně potkat [14].

llvm-as Tento program je assemblerem vnitřní formy LLVM. Čte tedy textovou reprezentaci vnitřní formy a produkuje totožný program v binární podobě. Takzvaný bitcode. Jde ale o binární instrukce LLVM IR, nikoliv nativní kód platformy.

llvm-dis Program provádějící inverzní operaci než llvm-as. Tedy disassembler načítající vnitřní formu v binární podobě a převádějící ji do čitelné textové podoby.

llvm-link Linker LLVM bitkódu. Načítá několik souborů vnitřní formy v binární podobě a sestavuje je do jednoho výstupního souboru.

lli Program lli je interpretrem LLVM bitkódu. Na architekturách x86, Sparc a PowerPC je podporován i JIT překlad. [5].

llc Tento program v podstatě reprezentuje back-end LLVM. Je to překladač LLVM bitkódu (tedy vnitřní formy) do instrukcí cílové platformy. Výstupem může být jazyk symbolických adres, i objektový soubor. Zajímavostí je, že back-end LLVM je přirozený křížový překladač. To znamená že jedna instance llc dokáže generovat kód pro všechny podporované cílové architektury. Bez nutnosti znovu sestavit LLVM ze zdrojových kódů.

opt Program opt je optimalizátor vnitřní formy. Ze vstupu načte program v bitkódu nebo textové podobě vnitřní formy. Na načtený program aplikuje dostupné optimalizace implementované v LLVM. A optimalizovaný program uloží na výstup opět ve vnitřní formě LLVM IR.

4.1.2 Clang

Clang je asi nejznámější projekt postavený na LLVM. Jedná se o překladač rodiny jazyků C, tedy C, C++ a Objective-C, využívající právě optimalizací

[5] JIT z anglického just-in-time je označení pro techniky překladu programu, až během jejich běhu. Často se tyto metody, někdy nazývané taký dynamický překlad, používají pro překlad interpretovaného kódu do nativních strojových instrukcí dané architektury [15].

a generátoru kódu LLVM. V podstatě se jedná o frontend překládající vysokoúrovňové jazyky do vnitřní formy LLVM.

Původně bylo záměrem projektu LLVM integrovat optimalizační část a generátor kódu do GCC. Nakonec kvůli zastaralosti a náročně udržovatelnosti kódu GCC došlo ke vzniku samostatného front-endu. Dnes je Clang plnohodnotný překladač a ve světě otevřeného softwaru pravděpodobně nejvážnější konkurent ekosystému kolem GCC. Jediná část celého řetězce vytváření spustitelného souboru, která není doposud plně vyřešena v rámci projektu LLVM je linker.

4.2 Vnitřní forma LLVM IR

LLVM IR je vnitřní forma, která je stěžejní částí projektu LLVM. Jde o nízkoúrovňovou, univerzální, SSA formu v podobě tří-adresového kódu. Zajišťuje propojení front-endu, kde je generována s back-endem kde je transformována do kódu cílové platformy. Nejvíce je samozřejmě využívána ve střední části překladače, kde v podstatě definuje rozhraní jednotlivých optimalizačních průchodů.

Velkou sílou LLVM IR je její plná soběstačnost. Jinak řečeno tato forma kompletně popisuje překládaný program. Je například možné si v průběhu překladu daný program uložit v této vnitřní formě do souboru a později v překladu pokračovat s naprosto stejným výsledkem.

LLVM IR může existovat celkem ve třech formách. Jako textový zápis v podobě assembleru nebo binární reprezentace takzvaný „bitkód“. Ten je vhodný jako vstup interpretru (nebo JIT kompilátoru), případně muže být více takových souborů zpracováno programem llvm-link. Poslední formou je reprezentace v paměti používaná pro efektivní provádění transformací. Všechny tyto formy jsou ekvivalentní a v podstatě všechny programy llvm-core umí pracovat s oběma vnějšími formáty. [16]

4.2.1 Struktura vnitřní formy

Vnitřní forma LLVM je jazyk symbolických adres pro virtuální RISC architekturu. Přitom si ale zachovává některé z informací běžných u vysokoúrovňových jazyků. Jako například hlavičky funkcí. Navíc je tato forma silně typovaná, takže každá proměnná i operace má jasně definovaný typ.

Programy zapsané ve vnitřní formě LLVM se skládají z modulů. Každý modul odpovídá jednomu překládanému souboru. Moduly obsahují funkce, globální proměnné a tabulku symbolů. Jedná se o analogii objektových souborů.

V tělech funkcí se nachází jednotlivé instrukce. Instrukce se rozdělují do několika skupin podle jejich funkce:

terminující instrukce Instrukce, které neprodukují žádnou hodnotu. Jejich návratový typ je typicky *void*. Místo toho zajišťují změnu toku výpočtu ve vnitřní formě. Patří sem instrukce jako *br*, *inidrectbr*, *ret* a další.

binární operace Binární instrukce realizují aritmetické operace. Jejich argumentem jsou dvě hodnoty stejného typu a produkují jednu hodnotu. Ta je totožného typu jako argumenty. Zástupci jsou např. *div*, *add* či *mull*.

bitové binární operace Instrukce realizující bitové operace. Typ výstupní hodnoty je opět stejný jako typ argumentu. Příkladem jsou *shl*, *ashr*, *and* nebo *xor*.

paměťové instrukce Tato skupina obsahuje instrukce pro práci s pamětí. Nejpoužívanější jsou určitě instrukce *load*, *store*. Další je instrukce *alloca*, která alokuje místo na zásobníku[6] nebo instrukce *cmpxchg* a *atomicrmw* pro atomické paměťové operace.

ostatní Ostatní instrukce jako *icmp* porovnávající argumenty. Instrukce podmíněného přesunu *select* nebo instrukce *call*.

Typový systém obsahuje typ funkce, který určuje parametry a návratovou hodnotu funkce. Dále jednoduché celočíselné typy a typy pro reprezentaci čísel v plovoucí řádové čárce. A dále speciální typy pro reprezentaci složených datových typů jako jsou pole a struktury.

Výpis 4.1: Ukázka LLVM IR

```
@a = global i32 3, align 4

define i32 @main() #0 {
  %1 = load i32, i32* @a, align 4, !tbaa !1
  %2 = icmp slt i32 %1, 98
  %. = select i1 %2, i32 7, i32 3
  store i32 %., i32* @a, align 4, !tbaa !1
  ret i32 %.
}
```

Argumenty instrukcí jsou identifikátory. Pro pojmenované proměnné (ty které jsou deklarovány ve vstupním jazyce) se jako identifikátor použije jejich původní název. Pro dočasné proměnné založené překladačem se používají čísla. Navíc jsou všechny identifikátory uvozeny speciálním znakem udávající působnost dané proměnné. Pokud identifikátor začíná znakem %, pak jde o lokální proměnou, globální proměnné jsou uvozeny znakem '@'.

[6]Generátor vnitřní formy typicky vygeneruje tuto instrukci pokud jsou ve zdrojovém kódu použity pole proměnlivé délky (VLA), nebo přímo volaní knihovní funkce alloca.

4.2.2 Univerzálnost versus přenositelnost

Vzhledem k univerzálnosti a soběstačnosti vnitřní formy LLVM by se mohlo zdát, že program reprezentovaný v LLVM IR je platformě nezávislý. Vnitřní forma je sice natolik všeobecná, aby dokázala dostatečně vyjádřit program pro různé platformy. V obecném případě je program reprezentovaný v této formě platformě závislý.

Například v jazyce C jsou velikosti běžných typu stanoveny cílovou platformou. Použití operátoru *sizeof* ve zdrojovém kódu se projeví ve vnitřní formě jako konstanta, jejíž hodnota se liší dle cílové platformy. Vnitřní forma v sobě reflektuje vlastnosti cílového ABI. tudíž obecně není platformě nezávislá.

4.3 Generátor kódu LLVM

Základem zadní části překladače LLVM je takzvaný platformě nezávislý generátor kódu. Jde o soubor abstrakcí a algoritmů, která zajišťují překlad vnitřní formy do strojového kódu cílové platformy.

Celý systém generátoru zahrnuje následující komponenty [17]:

platformě nezávislé algoritmy Jádrem platformě nezávislého generátoru jsou platformě nezávislé průchody, které implementují jednotlivé fáze generovaní kódu jako je výběr a plánování instrukcí nebo alokace registrů. Přesné chování generátoru pro konkrétní platformu lze ovlivnit pomocí ostatních komponent. Zejména implementací tříd popisující cílovou architekturu. Implementace se nachází v adresáři *lib/CodeGen*.

třídy pro popis architektury Soubor C++ tříd, které abstraktním způsobem umožňují popsat vlastnosti cílové architektury. Slouží jako rozhraní pro jádro generátoru k přístupu k informacím o cílové platformě. Hlavičky těchto tříd se nacházejí v *include/llvm/CodeGen*.

imeplementace tříd architektury Konkrétní implementace tříd popisující charakteristiky cílové platformy. Pro každou podporovanou platformu existuje sada těchto tříd v adresáři *lib/Target/<platform>/*.

abstrakce instrukcí Vnitřní forma pro reprezentaci instrukcí cílové platformy. Jde o skupinu tříd umožňující abstraktní přístup ke generovanému programu. Patří sem třídy *MachineFunction*, *MachineBasicBlock* a *MachineInstr*. Tato rozhraní mohou být použita k provádění dodatečných optimalizací. A jsou používány průchody generátoru po proběhnutí výběru instrukcí.

strojový kód Takzvaná *MC* („machine code“) vrstva. Tyto třídy jsou používány posledním průchodem generátoru k převodu vnitřní reprezentace do jazyka symbolických instrukcí nebo rovnou do strojového kódu. Umožňuje generátoru vytvářet i přímo objektové soubory.

4.4 Průchody generátoru LLVM

Proces překladu vnitřní formy do strojového kódu je rozdělen na několik navzájem navazujících průchodů [18]:

1. Výběr a plánovaní instrukcí. Tato fáze načítá program ve vnitřní formě LLVM a snaží se najít ideální pokrytí instrukcemi cílové platformy. K tomu používá vlastní reprezentaci pomocí orientovaného acyklického grafu, tzv. *SelectionDAG*. Tato fáze se skládá z několik samostatných kroků a je podrobněji vysvětlena v samostatné sekci 4.4.2. Výstupem je program vyjádřený reálnými instrukcemi v SSA formě.

2. SSA optimalizace. Volitelný průchod, který může provádět různé optimalizace nad linearizovanou posloupností instrukcí.

3. Alokace registrů. V tomto průchodu dochází k přiřazení virtuálních registrů registry cílové platformy. Tento průchod také generuje dodatečný kód pro odkládání registrů do paměti v případě jejich nedostatku. Po tomto kroku jsou všechny virtuální registry nahrazeny skutečnými a program již není v SSA formě.

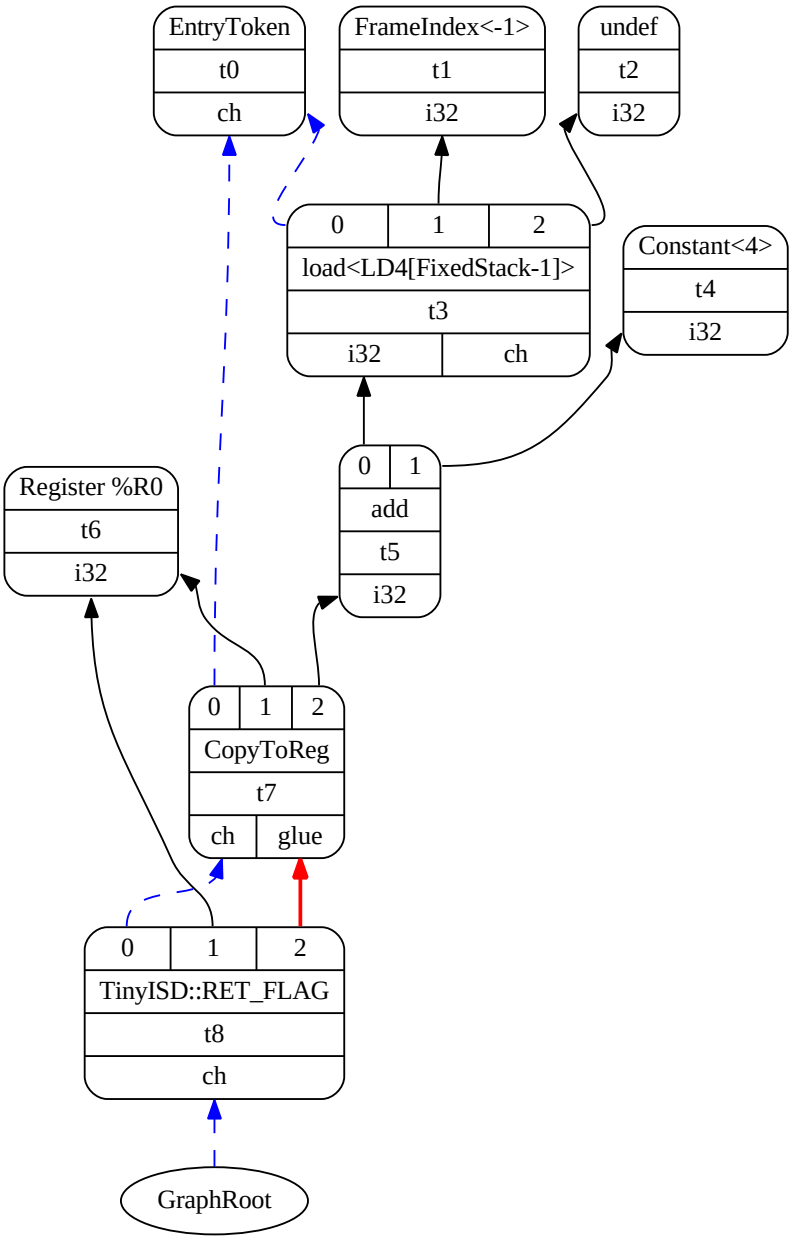
4. Vkládání prologu a epilogu. Po alokaci registrů již je známa potřebná velikost zásobníku. Může tedy dojít k vložení prologu a epilogu funkce. Dále jsou všechny instrukce používající abstraktní index zásobníku nahrazeny instrukcemi s registrem a relativní adresou. Tento průchod je také zodpovědný za případnou eliminaci ukazatele rámce zásobníku, pokud se funkce bez něj obejde.

5. Poslední průchodem je emitovaní kódu. Ten konvertuje abstraktní reprezentaci instrukcí do reprezentace na úrovni vrstvy MC. Pak jsou instrukce již připraveny k zápisu do souboru.

4.4.1 SelectionDAG

SelectionDAG je datová struktura reprezentující překládaný program, na které LLVM provádí výběr a plánování instrukcí. Jde o acyklický orientovaný graf jehož uzly jsou různých typů. Některé reprezentují instrukce, registry, pozice na zásobníku nebo speciální uzly určující začátek a konec daného úseku programu. Atributy každého uzlu jsou argumenty a operační znak, který určuje jakou operaci daný uzel reprezentuje. Většina uzlů grafu definuje jednu hodnotu. Existují ale i uzly s více hodnotami. Například pokud cílová architektura implementuje kombinovanou instrukci dělení se zbytkem.

Argumenty instrukcí jsou v grafu reprezentovány hranou, která vede k uzlu definující danou hodnotu. Navíc každý uzel, který reprezentuje instrukci s vedlejším efektem (například přiřazení do registru) obsahuje jeden speciální



Obrázek 4.1: Příklad struktury SelectionDAG před výběrem instrukcí

argument. Tato hrana, typicky umístěna v uzlu na první pozici určuje vzájemné pořadí mezi instrukcemi. Míří tedy k instrukci (uzlu), která se musí vyskytovat ve výsledném kódu před instrukcí (uzlem), ze kterého vychází.

Další typem je takzvaná *glue* hrana. Ta zajišťuje, že mezi instrukce které spojuje, ve výsledném kódu nebude žádná jiná instrukce. Jde o dodatečnou informaci pro plánovač instrukcí. [19, 17]

4.4.2 Výběr instrukcí

V této části je podrobněji vysvětlen první krok generátoru a to výběr instrukcí. Jak již bylo zmíněno zadní část překladače LLVM k tomuto úkonu používá grafovou reprezentaci programu zvanou *SelectionDAG*, dále zkráceně jen graf. Tento proces dělí na dvě hlavní části. První je legalizace grafu. To jsou transformace grafu do podoby, ve které se skládá pouze z operací a typů, které jsou přímo realizovatelné instrukční sadou cílové architektury. Druhou důležitou částí je samotný výběr instrukcí, který transformuje obecný graf do grafu skládající ho se z cílových instrukcí.

Celý proces výběru se skládá z následujících průchodů:

1. Sestavení grafu. V tomto kroku je graf sestaven naivní expanzí z vnitřní formy LLVM IR. Pro běžné instrukce vnitřní formy jde jen o nahrazení dané operace odpovídajícím uzlem. Je nutné expandovat také vysokoúrovňové prvky vnitřní formy, jako je volaní funkcí nebo formální argumenty. To zajišťuje implementace platformní třídy *TargetLowering*.

2. Legalizace datových typů. Pokud cílová architektura nativně nepodporuje některý z typů použitých ve vnitřní formě pak jsou v tomto kroku nahrazeny alternativní reprezentací. Mohou nastat dva případy. Nahrazení příliš malých typů typy většími tzv. „promoting“ a rozdělení velkého typu na více menších „expanding“. Tyto konverze můžou způsobit vložení rozšiřujících instrukcí pro zajištění korektního chování kódu.

3. Legalizace operandů instrukcí. Předchozí průchod řeší obecně nepodporované operandy na dané platformě. Někdy ale nejsou některé typy podporované pouze některými instrukcemi. Konverzi těchto případů na instrukce s nativní podporou zajišťuje tato legalizační fáze. O konkrétní chovaní se pro dané operace stará opět třída *TargetLowering*. Po tomto kroku se graf stále skládá z obecných platformě nezávislých operací, ale používá již pouze ty, které lze jednoduše vyjádřit cílovými instrukcemi.

4. Výběr instrukcí. V tuto chvíli se již graf skládá pouze z operací, které mají přímé vyjádření v instrukční sadě. Selektor prochází graf a podle dané priority se snaží graf pokrýt instrukcemi cílové platformy. Jaké operace odpovídají kterým instrukcím určují instrukční vzory 4.5.3. Ty jsou hlavním zdrojem informací pro algoritmus pokrývání grafu. Po této transformací jsou uzly grafu již reprezentují instrukce cílové platformy.

5. Plánovaní instrukcí. Posledním krokem je linearizace grafu do posloupnosti instrukcí. Zde probíhají optimalizace proudového zpracovaní instrukcí. Logicky se sice jedná o oddělený krok od samotného pokrývaní, ale je sním spjat, neboť oba algoritmy probíhají nad grafem instrukcí. Výstup tohoto průchodu je lineární reprezentace programu pomocí instrukcí cílové platformy. Stále zůstává v SSA formě. Používá neomezený počet virtuálních registrů a zásobník adresuje pomocích abstraktních indexů.

Za první tří fáze je navíc vždy vložen optimalizační průchod nazývaný *DAGCombiner*. Ten se snaží části grafu zjednodušit pomocí množiny pevných vzorů. Jeho hlavním smyslem je umožnit naivní implementaci legalizačních průchodů.

4.5 Implementace back-endu

V této části je podrobněji vysvětlen postup implementace generátoru kódu pro cílovou platformu. Základním mechanizmem pro vytvoření generátoru je dědění tříd v jazyce C++. Platformě nezávislý generátor LLVM poskytuje sadu tříd, které jsou pro každou podporovanou cílovou platformu odvozeny. Tak vzniknou platformě závislé třídy, které poskytují algoritmům generátoru potřebné informace o cílové platformě.

V průběhu generování kódu generátor volá pomocí virtuálních metod konkrétní implementaci, za účelem získaní informací nebo provedení některého úkonu nad generovaným kódem.

Popisovat do detailu komplexní instrukční sady nebo registrové třídy pomocí imperativního jazyka typu C++ by bylo poměrně zdlouhavé, náchylné k chybám a náročné na údržbu kódu. Z těchto důvodů se tyto části implementace generují z deklarativních předpisů pomocí nástroje TableGen 4.5.2.

4.5.1 Platformní třídy

Základní rozhraním, které používá platformě nezávislý generátor k přístupu platformě specifickým informacím a operacím je třída *LLVMTargetMachine*. Implementace této třídy v sobě obsahuje instance jednotlivých platformě závislých tříd, každá poskytující danou skupinu parametrů. Jakou jsou specifikace registrových tříd, informace o instrukční sadě nebo volacích konvencích [19].

4.5.1.1 TargetLowering

Tato třída hraje důležitou roli během legalizace grafu instrukcí. V konstruktoru této třídy se nastavuje několik parametrů rozhodujících pro převod grafu do instrukcí cílové platformy:

- přiřazení registrových tříd typům vnitřní formy,

- nastavení akcí *promote*/*expand* pro nativně nepodporované kombinace instrukcí a datových typů.

Pokud pro danou kombinaci operace a datového typu nevyhovuje interní řešení generátoru pomocí akcí *promote* a *expand*, pak lze definovat vlastní (*custom*) akci. V tomto případě musí třída implementovat metodu *LowerOperation*, která zajistí nahrazení dané části grafu legálním ekvivalentem.

Dále tato třída typicky implementuje metody *LowerFormalArguments* a *LowerCall*. Ty jsou zavolány během vytváření grafové reprezentace a produkují operace (uzly grafu) zajišťující přístup k argumentům a navracení hodnot z funkce.

4.5.1.2 TargetFrameLowering

Implementace třídy *TargetFrameLowering* poskytuje následující informace o rozložení zásobníku a jeho adresování:

- směr růstu zásobníku,
- zarovnání zásobníku při vstupu do funkce,
- pozice lokálních proměnných relativně vzhledem k ukazateli zásobníku, pří vstupu do funkce.

Třída také implementuje metody zodpovědné za vkládání prologu a epilogu funkce. Sem patří *emitPrologue*/*emitEpilogue* a *spillCalleeSavedRegisters*/*restoreCalleeSavedRegisters*. Ke generovaní prologu a epilogu dochází až po alokaci registrů. Tyto metody proto produkují již konkretní instrukce (typu *MachineInstr*) nikoliv reprezentaci v podobě uzlů grafu.

Další metody jako *hasFP* nebo *hasReservedCallFrame* slouží k optimalizaci práce se zásobníkem. První říká generátoru, zda je možné funkci adresovat bez ukazatele rámce. Druhá indikuje, zda lze provést alokaci místa pro odchozí argumenty již v prologu. Překladač pak alokuje tolik prostoru kolik spotřebuje volaní s největším objemem argumentů a tím se ušetří alokace a dealokace zásobníku při každém odchozím volaní funkce.

4.5.1.3 TargetRegisterInfo

Třída *TargetRegisterInfo* obsahuje informace o registrech a registrových třídách cílové platformy. Tyto informace jsou generovány ze souboru popisující instrukční sadu pomocí *TableGen*.

Tato třída typicky implementuje následující metody:

getCalleeSavedRegs Vrátí seznam registrů, které je ve funkci nutné před použitím zálohovat.

getReservedRegs Seznam registrů, který mají ve funkci speciální význam a nelze je použít za žádných okoností. Například registry adresující zásobník.

getFrameRegister Vrací registr, který bude použit k adresovaní rámce zásobníku.

eliminateFrameIndex Tato metoda je používána průchodem vkládající prolog a epilog. Je zavolána pro každou instrukci adresující zásobník. Jejím úkolem je nahradit abstraktní index zásobníku skutečným ukazatelem zásobníku a relativní adresou.

4.5.1.4 TargetInstrInfo

Tato platformní třída nese popis podporovaných instrukcí cílové platformy. Obsahuje informace jako mnemotechnický zápis instrukce, počet operandů nebo některé z příznaků určující přesněji chování instrukce. Například zda má instrukce komutativní operandy či zda jde o instrukci volaní funkce nebo skoku. Tyto informace jsou generovaný pomocí *TableGen* ze souboru definující instrukční sadu 4.5.3.

Kromě toho tato třída implementuje metody produkující instrukce pro zálohu a obnovu registrů:

copyPhysReg Metoda produkující instrukce přesunu hodnoty registru. Implementace by měla ošetřovat i případy přesunu mezi registry různých registrových tříd.

storeRegToStackSlot Emituje instrukce pro uložení daného registru na abstraktní pozici v zásobníku.

loadRegFromStackSlot Vytváří instrukce pro načtení virtuální pozice v zásobníku do požadovaného registru.

Tyto metody jsou volány z průchodu, který provádí alokací registrů a průchodem vkládající prolog a epilog

4.5.1.5 SelectionDAGISel

Implementace této třídy reprezentuje selektor instrukcí. Je tedy zodpovědná za pokrytí platformě nezávislého ale legálního grafu instrukcemi cílové platformy. Konkrétně to zajišťuje metoda *Select*, která realizuje pokrývaní stromu instrukcemi. Většina platformních implementací v této metodě pomocí ručně psaného kódu pokrývá instrukční vzory, které není možné efektivně pokrýt pomocí selektoru generovaného z instrukčních vzorů 4.5.3. Sem patří například instrukce definující více hodnot nebo složitější adresní módy.

Pro všechny ostatní běžné případy metoda jednoduše předá řízení generovanému selektoru zavoláním metody *SelectCode*[20].

4.5.2 TableGen

TableGen je důležitá komponenta LLVM používaná k popisu cílové architektury. Jde o jazyk s objektovými aspekty, který slouží k deklarativnímu vytváření záznamů. Důvodem pro vznik takového nástroje je fakt, že typicky existuje velké množství záznamů se společnými vlastnostmi. Síla TableGen je v schopnosti jednoduše popsat vlastnosti architektury bez nutnosti zbytečného opakovaní některých informací a tak celý úkol usnadnit a zpřehlednit.

Další výhodou TableGen jsou takzvané doménově specifické back-endy. Samotné jádro programu pouze načítá zdrojový soubor (.td), interpretuje a vytvoří sadu záznamů. Poté jsou spuštěny back-endy, které z poskytnutých záznamu vytváří vlastní výstup. TableGen back-endy používané v rámci generátoru kódu LLVM většinou produkují C++ kód, který je vložen do příslušné platformní třídy. Pro generátor kódu existují back-endy jako *register-info*, *instr-info* nebo *asm-writer*. Tímto mechanizmem je navíc dosaženo, že může být vše deklarováno pouze jednou a na jednom místě. Například definice instrukce na úrovni .td souboru může nést informace pro algoritmus výběru instrukcí, výstup v jazyce symbolických adres, assembler, disassembler nebo ladící výpisy[21].

Syntaxe souboru TableGen je podobná šablonám z jazyka C++. Základem jazyka jsou konstrukty záznam, třída a definice:

záznamy Záznamy jsou základní entitou v jazyka TableGen. Každý záznam má seznam položek, ty nesou užitečné informace o popisované entitě a seznam nadtříd, ze kterých je záznam odvozen.

třídy Třídy jsou částečné záznamy, které definují jen některé (nebo žádné) položky a dovolují být odvozeny běžnými definicemi. Tím je umožněno vyhnout se zbytečně se opakujícím hodnotám v definicích záznamů podobného charakteru. Jako například registry stejné třídy nebo instrukce stejného formátu. Zároveň tento mechanizmus slouží jako rozhraní k platformě nezávislému generátoru, který definuje základní třídy jako *Register*, *RegisterClass* a *Instruction*, které by měla každá implementace (často nepřímo) odvozovat.

definice Definice jsou konkrétní instance záznamů. Typicky nemají žádné nedefinované hodnoty a deklarují se pomocí klíčového slova *def* následovaným jménem záznamu.

TableGen dále poskytuje direktivy jako *foreach* a *multiclass*, které dále usnadňují a automatizují psaní definic. Ve výsledku je ale na výstupu z TableGen pouze seznam záznamu, který je dále zpracován konkrétním back-endem.

Výpis 4.2: Ukázka použití *def* a *class* k deklaraci záznamu registrů R0 a R1

```
class TinyGPRReg<bits<8> num, string n, list<int> dwarf>
  : Register<n> {
  field bits<8> Num = num;
  let Namespace = "Tiny";
  let DwarfNumbers = dwarf;
}

def R0 : TinyGPRReg<0, "0", [0]>;
def R1 : TinyGPRReg<1, "1", [1]>;
```

4.5.3 Instrukční vzory TableGen

Jak již bylo vysvětleno proces převodu vnitřní formy do platformně závislého kódu je prováděn na struktuře acyklického orientovaného grafu (tzv. *Selection-DAG*). Aby mohl platformě nezávislý generátor provést tuto transformaci, je nutné mu poskytnout informace o vztahu mezi platformě nezávislými operaci a konkrétními instrukcemi dané platformy.

Tato vazba je součástí definic instrukcí v souboru popisující instrukční sadu dané platformy. Z tohoto souboru je pomocí *TableGen* v době překladu překladače jednak generován seznam záznamů o vlastnostech instrukcí používaný třídou *TergetRegistrInfo* a také speciální „bajtkód“. Tento speciální „program“ je v době generovaní kódu interpretován průchodem provádějící výběr instrukcí a je jím řízeno pokrývání platformě nezávislého grafu instrukcemi.

Základem pro popis instrukční sady je *TableGen* třída *Instruction*, kterou definuje platformě nezávislý generátor. Tato třída obsahuje následující položky, které naprosto zásadní pro onu vazbu mezi obecným a platformě závislým grafem:

OutOperandList Seznam platformě závislých grafů reprezentující výstupní operandy instrukce.

InOperandList Seznam platformě závislých grafů reprezentující vstupní operandy instrukce.

AsmString Šablona instrukce pro výstup v jazyce symbolických adres.

Pattern Vzor je platformě nezávislý graf, který pokrývá tato instrukce.

Pro názornější ilustraci poslouží příklad instrukce provádějící operaci *or* nad registry ve výpisu 4.3. Pro větší názornost je záznam zobrazen v základní podobě. V praxi se instrukce definují mnohem kratším zápisem pomocí odvození třídního záznamu.

Výpis 4.3: Příklad TableGen záznamu instrukce *or*

```
def ORrr {
  dag OutOperandList = (outs IntRegs:$rd);
  dag InOperandList = (ins IntRegs:$rs1, IntRegs:$rs2);
  string AsmString = "or␣$rs1,␣$rs2,␣$rd";
  list<dag> Pattern = [(set i32:$rd,
  (or i32:$rs1, i32:$rs2))];
}
```

Z povšimnutí stojí fakt, že argumenty operace *or* v proměnné *Pattern* používají datové typy vnitřní formy LLVM. Zatímco vstupní a výstupní argumenty instrukce *ORrr* používají platformě závislou registrovou třídu *IntRegs*.

Tato definice způsobí, že pokud je v platformě nezávislém grafu objevena operace *or* s argumenty typu *i32* je tato část grafu nahrazena grafem složeným z instrukce *ORrr* a registrů ze třídy *InstRegs* na pozici argumentů. Alespoň za předpokladu, že není definován jiný vzor, který by pokrýval stejnou část grafu a byl vyzkoušen dříve. Hlavním kritériem, které ovlivňuje pořadí vzorů během pokrývaní je jejich velikost. Jde tedy o hladový algoritmus. Pro zvýšení priority některé instrukce, lze velikost vzoru uměle zvýšit definováním konstanty *AddedComplexity*.

Generátor dále poskytuje následující *TableGen* třídy pro pokročilejší pokrývání.

Pat/Pattern Umožňuje definovat vzor, který bude nahrazen jednou nebo několika již definovanými instrukcemi.

ComplexPattern Záznam, který není klasickým vzorem, ale definuje metodu (implementovanou v platformní třídě *SelectionDAGISel*), která se pokusí provést pokrytí a v případe schody vrátí ekvivalent pro danou platformu.

A další odvozené varianty. V principu je výsledek vždy stejný. Část grafu je zaměněna za graf skládající se pouze z nativních instrukcí a pseudoinstrukcí.

4.5.4 Pseudoinstrukce

V předešlém textu bylo naznačeno jak lze během legalizace grafu podle specifikovaných akcí nahradit nepodporovanou kombinaci operace a datového typu legální alternativou. V některých případech ale neexistuje v instrukční sadě instrukce odpovídající dané operaci vnitřní formy a je nutné tuto operaci realizovat sekvencí instrukcí.

K tomu to účelu slouží mechanizmus pseudoinstrukcí. Platformní implementace v podstatě předstírá průchodu provádějící výběr instrukcí, že daná operace na cílové platformě nativně existuje. V souboru instrukčních vzorů definuje instrukci standardním způsobem, pouze nastaví příznak, že se nejedná

o instrukcí skutečnou. Tato virtuální instrukce je později nahrazena sekvencí více instrukcí. To může nastat ve dvou místech v průběhu generovaní kódu:

po linearizaci grafu Pro každou instrukci, která má v definici nastaven příznak *usesCustomInserter* je těsně po linearizaci grafu do SSA formy zavolána metoda *EmitInstrWithCustomInserter* třídy *TargetLowering*. V této metodě je možné pomocí vlastního kódu vložit do překládané funkce instrukce realizující danou operaci.

po alokaci registrů Pozdějším bodem, kdy dochází k expanzi pseudoinstrukcí je po alokaci registrů. Pokud má instrukce nastaven příznak *isPseudo* a je v kódu stále přítomna i po alokaci registrů, pak je zavolána metoda *expandPostRAPseudo* třídy *TargetInstrInfo*, která musí danou instrukci eliminovat a nahradit skutečnými instrukcemi.

Závěr

V předchozích kapitolách byla podrobněji rozebrána architektura zadních částí překladačů GCC a LLVM. V obou případech byly popsány nejdůležitější algoritmy a struktury, které provází překládaný program na jeho cestě generátorem kódu. Dále text popisoval konkrétní mechanizmy, kterými je v zadních částech obou překladačů dosaženo podpory více cílových architektur.

Při pohledu na způsoby, kterými oba překladače umožňují definovat platformě závislé části generátoru kódu, je vidět jistý rozdíl. Zřejmě by se dal přisoudit rozdílnému staří obou projektů a okolnostem, které vedli k jejich vzniku. Zatímco GCC používá makra jazyka C k definovaní některých platformních informací, LLVM používá výhradně dědičnosti tříd a virtuálních metod. To LLVM umožňuje pomocí jedné instance generovat kód hned pro několik cílových architektur, zatímco překladač GCC je vždy závislý na konkrétní platformě pro kterou byl sestaven. Pokud se ale oprostíme od samotné implementace mechanizmů umožňující rozšíření generátoru pro danou cílovou architekturu, jsou si tyto mechanizmy velice podobné a v práci s oběma generátory není až tak velký rozdíl.

Například i způsob popisu instrukční sady cílové architektury, který slouží jako hlavní zdroj informací pro průchod provádějící výběr instrukcí v době překladu, vykazuje u obou překladačů jisté analogie. V obou případech se sémantika dané instrukce deklaruje pomocí vzorů vnitřní formy, zápisem velice podobným jazyka Lisp. Přestože vnitřní implementace je zcela odlišná.

Tím se dostáváme k pravděpodobně největšímu rozdílu mezi zkoumanými generátory a to je samotný proces výběru instrukcí. V případě LLVM jde o klasické pokrývaní acyklického orientovaného grafu cílovými instrukcemi dle poměrně jasně daného algoritmu. Na druhé straně GCC spoléhá na svůj *peephole* průchod, který se snaží naivně vytvořený kód zjednodušit. Tento méně sofistikovaný způsob „výběru“ instrukcí je i důvodem proč GCC provádí znovu některé optimalizace, které již byly provedeny na úrovni vnitřní formy [22].

S pomocí získaných znalostí byly v praktické části back-endy obou překladačů rozšířeny o podporu nové cílové architektury. Obě implementace byly vy-

tvořeny postupnými úpravami již existujících generátorů pro procesor MSP430. Ten byl zvolen kvůli jednoduchosti jeho implementace v obou překladačích. Další inspirací byly implementace generátorů pro RISC procesory MIPS a SPARC. Výsledné generátory jsou schopné pro všechny základní úrovně optimalizací, korektně přeložit jednodušší C programy poskytnuté v základní testovací sadě.

Avšak implementace generátoru rozhodně není kompletní a bylo by možné ji dále rozšířit. Například o podporu výpočtů v plovoucí desetinné čárce (softwarovou emulací), polí proměnlivé velikosti (VLA v C99) nebo o podporu funkcí s variabilním počtem argumentů. Obzvlášť náročná je korektní implementace vnořených funkci (v současnosti podporovaných pouze GCC rozšířením).

Literatura

[1] Louden, K. C.: *Compiler Construction: Principles and Practice*. PWS Publishing Company, 1997.

[2] Appel, A. W.: *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[3] Ægidius Mogensen, T.: *Basics of Compiler Design*. lulu.com, 2010.

[4] Aho, A. V.; Lam, M. S.; Sethi, R.; aj.: *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 1986.

[5] Cytron, R.; Ferrante, J.; Rosen, B. K.; aj.: *An Efficient Method of Computing Static Single Assignment Form*. Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1989.

[6] Davidson; Fraser: *The design and application of a retargetable peephole optimizer*. ACM Transactions on Programming Languages and Systems (TOPLAS), 1980.

[7] Blindel, G. S. H.: *Survey on Instruction Selection*. KTH Royal Institute of Technology.

[8] Briggs, P.: *Register Allocation via Graph Coloring*. Rice University, 1992.

[9] Scott, M. L.: *Programming Language Pragmatics*. Morgan Kaufmann, 2006.

[10] A Brief History of GCC. Dostupné z: `https://gcc.gnu.org/wiki/History`

[11] The Conceptual Structure of GCC. Dostupné z: `https://www.cse.iitb.ac.in/grc/intdocs/gcc-conceptual-structure.html`

[12] GNU Compiler Collection (GCC) Internals. Dostupné z: https://gcc.gnu.org/onlinedocs/gccint/

[13] Projects built with LLVM. Dostupné z: http://llvm.org/ProjectsWithLLVM/

[14] Getting Started with the LLVM System. Dostupné z: http://llvm.org/docs/GettingStarted.html

[15] Aycock, J.: *A Brief History of Just-In-Time*. University of Calgary, 2003.

[16] LLVM Language Reference Manual. Dostupné z: http://llvm.org/docs/LangRef.html

[17] The LLVM Target-Independent Code Generator¶. Dostupné z: http://llvm.org/docs/CodeGenerator.html

[18] Erhardt, C.: *Design and Implementation of a TriCore Backend for the LLVM Compiler Framework*.

[19] Writing an LLVM Backend. Dostupné z: http://llvm.org/docs/WritingAnLLVMBackend.html

[20] Bendersky, E.: Life of an instruction in LLVM. Dostupné z: http://eli.thegreenplace.net/2012/11/24/life-of-an-instruction-in-llvm

[21] TableGen. Dostupné z: http://llvm.org/docs/TableGen/index.html

[22] Improve instruction selection. Dostupné z: https://gcc.gnu.org/wiki/Improve_instruction_selection

Příloha A

Seznam použitých zkratek

ASCII American Standard Code for Information Interchange

AST Abstract syntax tree

ABI Application binary interface

COFF Common object file format

BIOS Basic input-output system

ELF Executable and linkable format

FSF Free software foundation

I/O Input/Output

ISA Instruction set architecture

IRA Integrated register allocator

LRA Local register allocator

JIT Just-in-time compiler

PE Portable executable

RTL Register transfer language

RISC Reduced instruction set

SSA Static single assignment

VLA Variable length array

VLIW Very large instruction word

Příloha B

Seznam změněných souborů

B.1 GCC seznam změněných a nových souborů

Výpis B.1: Změněné soubory v GCC

```
config.sub
gcc/config/tiny/constraints.md
gcc/config/tiny/predicates.md
gcc/config/tiny/tiny.c
gcc/config/tiny/tiny-c.c
gcc/config/tiny/tiny.h
gcc/config/tiny/tiny.md
gcc/config/tiny/tiny-modes.def
gcc/config/tiny/tiny.opt
gcc/config/tiny/tiny-opts.h
gcc/config/tiny/tiny-protos.h
gcc/config/tiny/t-tiny
gcc/config.gcc
```

B.2 LLVM seznam změněných a nových souborů

Výpis B.2: Změněné soubory v LLVM

```
CMakeLists.txt
include/llvm/ADT/Triple.h
lib/CodeGen/LiveVariables.cpp
lib/Support/Triple.cpp
lib/Target/Tiny/CMakeLists.txt
lib/Target/Tiny/InstPrinter/TinyInstPrinter.cpp
lib/Target/Tiny/InstPrinter/TinyInstPrinter.h
lib/Target/Tiny/MCTargetDesc/CMakeLists.txt
lib/Target/Tiny/MCTargetDesc/TinyMCAsmInfo.cpp
```

```
lib/Target/Tiny/MCTargetDesc/TinyMCAsmInfo.h
lib/Target/Tiny/MCTargetDesc/TinyMCTargetDesc.cpp
lib/Target/Tiny/MCTargetDesc/TinyMCTargetDesc.h
lib/Target/Tiny/README.txt
lib/Target/Tiny/TargetInfo/CMakeLists.txt
lib/Target/Tiny/TargetInfo/TinyTargetInfo.cpp
lib/Target/Tiny/TinyAsmPrinter.cpp
lib/Target/Tiny/TinyBranchSelector.cpp
lib/Target/Tiny/TinyCallingConv.td
lib/Target/Tiny/TinyFrameLowering.cpp
lib/Target/Tiny/TinyFrameLowering.h
lib/Target/Tiny/Tiny.h
lib/Target/Tiny/TinyInstrFormats.td
lib/Target/Tiny/TinyInstrInfo.cpp
lib/Target/Tiny/TinyInstrInfo.h
lib/Target/Tiny/TinyInstrInfo.td
lib/Target/Tiny/TinyISelDAGToDAG.cpp
lib/Target/Tiny/TinyISelLowering.cpp
lib/Target/Tiny/TinyISelLowering.h
lib/Target/Tiny/TinyMachineFunctionInfo.cpp
lib/Target/Tiny/TinyMachineFunctionInfo.h
lib/Target/Tiny/TinyMCInstLower.cpp
lib/Target/Tiny/TinyMCInstLower.h
lib/Target/Tiny/TinyRegisterInfo.cpp
lib/Target/Tiny/TinyRegisterInfo.h
lib/Target/Tiny/TinyRegisterInfo.td
lib/Target/Tiny/TinySubtarget.cpp
lib/Target/Tiny/TinySubtarget.h
lib/Target/Tiny/TinyTargetMachine.cpp
lib/Target/Tiny/TinyTargetMachine.h
lib/Target/Tiny/Tiny.td
```

Příloha C

Sestavení projektů ze zdrojových kódů

C.1 Sestavení GCC ze zdrojových kódů

V této sekci jsou popsány jednotlivé kroky k sestavení přiloženého projektu GCC ze zdrojových kódů. K překladu projektu je potřeba C++ překladač podporující alespoň ISO C++98. Dále standardní knihovna jazyka C včetně hlaviček, GNU make, GNU binutils, knihovna pro prací s velkými čísly MPFR a knihovna pro práci s čísly v plovoucí desetinné čárce MPC. Dále běžné UNIXové programy jako gzip, tar nebo awk.

Na většině moderních Linux systémů by mělo dostačovat nainstalovaní hlavičkových souborů ke zmíněným knihovnám. Případné chybějící závislosti by měly být odhaleny konfiguračním skriptem v kroku 3. Postup pro sestavení GCC je následující:

1. Nejprve rozbalíme archiv se zdrojovými soubory GCC.

   ```
   tar xzvf gcc-6.2.0.tar.gz
   ```

2. Vytvoříme si v hlavní složce projektu adresář ve kterém budeme projekt sestavovat a přepneme so do něho.

   ```
   mkdir gcc-6.2.0/build
   ```

   ```
   cd gcc-6.2.0/build
   ```

3. Spustíme konfigurační skript s požadovanými parametry.

   ```
   ../configure --target=tiny --enable-languages="c"
   ```

4. A konečně spustíme samotný překlad GCC.

   ```
   make
   ```

Příkaz spuštěný v posledním kroku skončí chybou. To je způsobeno tím, že sestavovací skript automaticky spouští interní testy a některým implementovaný back-end nevyhovuje díky jeho neúplné funkcionalitě.

C.2 Sestavení LLVM ze zdrojových kódů

Pro sestavení projektu LLVM ze zdrojových kódů je potřeba C++ překladač, cmake, make a běžné UNIXové programy. Kroky k sestavení LLVM jsou následující:

1. Nejprve rozbalíme archiv se zdrojovými soubory LLVM.

   ```
   tar xzvf llvm-3.9.0.src.tar.gz
   ```

2. Vytvoříme si v hlavní složce projektu adresář ve kterém budeme projekt sestavovat a přepneme so do něho.

   ```
   mkdir llvm-3.9.0.src/build
   ```

   ```
   cd llvm-3.9.0.src/build
   ```

3. Spustíme programe cmake. který vygeneruje skripty pro program make

   ```
   cmake ..
   ```

4. A konečně spustíme samotný překlad LLVM.

   ```
   make
   ```

Po skončení překladu, je výsledný překladač k dispozici v cestě *llvm-3.9.0.src/build/bin/llc*. Jde však pouze o překladač z vnitřní formy LLVM pro cílovou architekturu. Pro překlad z jazyka C je potřeba mít k dispozici navíc ještě clang, ideálně stejné verze.

Pak lze provést kompilaci programu v jazyce C následujícím příkazem.

```
clang -o - -S -target mips-unknown-linux-gnu -emit-llvm \
 soubor.c | llc -march=tiny
```

C.3 Sestavení interpretru TMI

K sestavení interpretru nejsou potřeba žádné externí knihovny ani nástroje. Měl by stačit jakýkoliv C překladač ISO C99 a program make. Interpret sestavíme následujícím příkazem:

```
cd tmi
make .
```

Tím vznikne spustitelný soubor tmi v aktuálním adresáři.

Příloha D

Testování

Vzhledem ke komplexnosti zdrojových kódů obou projektů je pro efektivní vývoj nutné mít aspoň minimální regresní sadu testů.

Fakt že pro cílovou architekturu existuje pouze interpret assembleru a funkcionalita obou překladačů není úplná, by značně komplikoval použití testovacích infrastruktur v LLVM a GCC. Proto byla vytvořena sada kratších jednoduchých programů testující pouze dostupnou funkcionalitu.

D.1 Psaní testů

Testy jsou krátké jednoduché C programy, většinou testující konkretní funkcionalitu. Pro ověření, že vygenerovaný kód skutečně odpovídá vstupnímu programu slouží dvě speciální direktivy. A to @STATUS, @INPUT a @OUTPUT. První definuje návratovou hodnotu programu. Druhá řetězec, který je podán testu ja standardní vstup a poslední specifikuje jaký řetězec očekávat na výstupu pří skončení testu. Všechny direktivy by měly být uvedeny v ANSI C komentáři na samém začátku souboru.

Výpis D.1: Ukázka zápisu testu

```
/*
@INPUT="foo bar"
@OUTPUT="foo bar"
@STATUS=13
*/

#include "../lib/tiny_std.c"

int main(void) {
int c;
while ((c = in()) != EOF)
  out(c);
```

```
  return 13;
}
```

Všechny testy se nacházejí v adresáři *tests/suite*.

D.2 Spouštění testů

Pro jednoduší spouštění a ověření výsledku testu slouží shell skript *tests/run.sh*. Tento skript je schopen spustit zadanou množinu testů s různým volbami. Jako který překladač ma být použit a s jakými úrovněmi optimalizací má být test přeložen a spuštěn. Skript zároveň načítá direktivy z testů a kontroluje, zda výstup testu opravdu odpovídá zadaným hodnotám.

Pokud uživatel neuvede žádné parametry, pak skript překládá a kontroluje testy na obou překladačích a pro všechny základní úrovně optimalizací (-00 až -O3).

Před spuštěním skriptu je však třeba nastavit potřebné cesty k překladačům a interpretru. To se provádí v konfiguračním souboru *tests/run.rc*.

Výpis D.2: Ukázka výstupu skriptu

```
___llvm ___gcc
0 1 2 3
[LLVM][O0] suite/04-cat.c                    t t t   OK
[LLVM][O1] suite/04-cat.c                    t t t   OK
[LLVM][O2] suite/04-cat.c                    t t t   OK
[LLVM][O3] suite/04-cat.c                    t t t   OK
[ GCC][O0] suite/04-cat.c                    t t t   OK
[ GCC][O1] suite/04-cat.c                    t t t   OK
[ GCC][O2] suite/04-cat.c                    t t t   OK
[ GCC][O3] suite/04-cat.c                    t t t   OK
```

Ve výpisu D.2 je pak vidět výstup příkazu `./run.sh suite/04-cat.c`. První hranatá závorka znázorňuje použitý překladač. Následuje nastavená optimalizační úroveň a poté název testu. Poté následují celkem tři sloupečky obsahující znak t nebo f. Znak t znamená true tedy úspěch daného kroku a f selhání. Prní z písmen značí zda během překladu testu nedošlo k chybě. A to jak z důvody syntaktické chyby v testu, tak i interní chyby překladače.Druhé písmeno ukazuje zda doběhla interpretace vygenerovaného kódu. Opět může dojít k chybě na vstupu (neznámá či nevalidní instrukce) nebo předčasnému ukončení interpretace například z důvodu přístupu za hranici paměti. Poslední z písmen pak značí zda výstup interpretovaného programu odpovídá hodnotám direktiv v zápisu testu.

Pro ověření všech testů s oběma překladači a všemi úrovněmi optimalizací stačí zadat příkaz `./run.sh suite/*`.

Příloha E

Obsah přiloženého CD

Výpis E.1: Obsah přiloženého média

```
+- gcc-6.2.0.tar.gz
+- llvm-3.9.0.src.tar.gz
+- tests/
+- lib -> ../tmi/lib/
|   +- run.rc
|   +- run.sh
|   +- suite/
|
+- text/+
+- tmi
+- cpool.c
+- cpool.h
+- debug.h
+- GRAMMER
+- lib
|   +- alloc.h
|   +- alloc_test.c
|   +- tiny_std.c
|
+- Makefile
+- opcode.c
+- opcode.h
+- parser.c
+- parser.h
+- tmi.c
+- tmii
```