VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

BAKALÁŘSKÁ PRÁCE

Brno, 2020 Matej Pončák

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY
DEPARTMENT OF CONTROL AND INSTRUMENTATION

ROZŠÍŘENÍ PŘEKLADAČE JAZYKA C O PODPORU DALŠÍCH EMBEDDED MIKROPROCESORŮ
RETARGETING OF THE C LANGUAGE COMPILER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Matej Pončák**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Petr Petyovský, Ph.D.**
SUPERVISOR

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Automatizační a měřicí technika**

Ústav automatizace a měřicí techniky

Student: Matej Pončák ***ID:*** 203325
Ročník: 3 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Rozšíření překladače jazyka C o podporu dalších embedded mikroprocesorů

POKYNY PRO VYPRACOVÁNÍ:

Cílem práce je navrhnout a realizovat rozšíření překladače jazyka C o podporu dalšího embedded mikroprocesoru:

1. Seznamte se s vlastnostmi a architekturou opensource překladačů jazyka C. (Clang, sdcc)
2. Nastudujte problematiku vytvoření nové backend části překladače Clang a sdcc generující spustitelný kód pro nové procesory.
3. Popište vhodnou rodinu embedded mikroprocesorů, kterou překladač Clang nebo sdcc nepodporuje. Zvolte pro další práci jeden konkrétní mikroprocesor a diskutujte důvody pro jeho volbu. Zvolte pro mikroprocesor referenční překladač a popište jeho vlastnosti.
4. Navrhněte a realizujte vlastní backend část překladače Clang (případně sdcc) generující kód spustitelný pro zvolený mikroprocesor. Zvolte volací konvence funkcí jazyka C tak, aby bylo možné propojit asemblerový výstup backend části překladače s kódem v referenčním vývojovém prostředí.
5. Na vhodném programu otestujte správnost generovaného spustitelného kódu pro zvolený mikroprocesor.
6. Na zvoleném příkladu ověřte a demonstrujte správnou funkci překladače na reálném HW.
7. Zhodnoťte dosažené výsledky a navrhněte další možná rozšíření.

DOPORUČENÁ LITERATURA:

[1] Knuth E. Donald: Umění programování 1.díl - Základní algoritmy, Computer press 2008, ISBN: 978-80-2-1-2025-5.

[2] Dvořák V., Drábek V.: Architektura procesorů, Vutium 1999, ISBN: 80-214-1458-8.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 3.8.2020

Vedoucí práce: Ing. Petr Petyovský, Ph.D.

doc. Ing. Václav Jirsík, CSc.
předseda rady studijního programu

ABSTRAKT

Aby bolo možné programovať určitú cieľovú architektúru v niektorom z vyšších programovacích jazykov, daný prekladač musí túto architektúru podporovať. Práca popisuje štruktúru prekladačov SDCC a LLVM a postup pre vytvorenie podpory pre novú cieľovú architektúru v týchto prekladačoch. Prekladač SDCC je medzi programátormi rozšírený kvôli svojej jednoduchosti a prekladač LLVM zase kvôli svojej veľkej univerzálnosti. Napokon je tento postup implementovaný pre mikroprocesor rady HCS08 v prekladači LLVM.

KĽÚČOVÉ SLOVÁ

jazyk C, prekladač, clang, llvm, sdcc, backend, hcs08

ABSTRACT

In order to program a target architecture in one of the high-level programming languages, the compiler must support that architecture. The thesis describes the structure of SDCC and LLVM compilers and the procedure of retargeting these compilers. The SDCC compiler is widespread among programmers for its simplicity and the LLVM compiler for its great reusability. Finally, this procedure is implemented for the HCS08 series microprocessor in the LLVM compiler.

KEYWORDS

C language, compiler retargetting, clang, llvm, sdcc, backend, hcs08

PONČÁK, Matej. *Rozšíření překladače jazyka C o podporu dalších embedded mikroprocesorů*. Brno, 2020, 73 s. Bakalárska práca. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedúci práce: Ing. Petr Petyovský, Ph.D.

VYHLÁSENIE

Vyhlasujem, že svoju bakalársku prácu na tému „Rozšíření překladače jazyka C o podporu dalších embedded mikroprocesorů“ som vypracoval samostatne pod vedením vedúceho bakalárskej práce, s využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej bakalárskej práce ďalej vyhlasujem, že v súvislosti s vytvorením tejto bakalárskej práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a/alebo majetkových a som si plne vedomý následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona Českej republiky č. 121/2000 Sb., o práve autorskom, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákonníka Českej republiky č. 40/2009 Sb.

Brno 03.08.2020 .....

podpis autora

POĎAKOVANIE

Rád by som sa poďakoval vedúcemu bakalárskej práce pánovi Ing. Petrovi Petyovskému, Ph.D. za odborné vedenie, konzultácie, trpezlivosť a podporu pri tvorbe práce.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam výpisov

Úvod

V posledných rokoch sa čím ďalej tým viac objavujú mikroprocesory v každej oblasti života. Nahrádzajú tak bežne známe logické obvody zložené z diskrétnych súčiastok kvôli svojej jednoduchosti použitia. Vďaka vysokej úrovni integrácie dosahujú veľmi malé rozmery, avšak častokrát s veľkým výpočtovým výkonom. Stretnúť sa s nimi je možné či už ide o rôzne hračky pre deti, kalkulačky, smartfóny alebo rôzne inteligentné zariadenia do domácností. Svoje využitie nachádzajú prakticky kdekoľvek.

Taktiež začínajú byť ľahko dostupné verejnosti kvôli ich nízkej cene, ale aj množstvu návodov. Vďaka nim si aj nezainteresovaný človek vie naprogramovať jednoduché zariadenie, ako napríklad blikanie LED diódou. Väčšinou sú programované v niektorom z vyšších programovacích jazykov (C, C++). Kvôli tomuto zjednodušeniu však málokto vie, ako tieto zariadenia fungujú a ako sa v skutočnosti programujú.

K tomu, aby to fungovalo potrebujeme prekladač, čiže program, ktorý prevádza program zapísaný v zdrojovom jazyku C/C++ na program v strojovom jazyku, ktorý vie mikroprocesor spracovať. Každý druh mikroprocesora však môže mať rôznu architektúru a iné strojové inštrukcie. Preto je nutné, aby bol prekladač prispôsobený k použitiu s daným mikroprocesorom. Práve to je problematika, ktorou sa táto práca zaoberá.

Na začiatok sa čitateľ oboznámi so všeobecnou štruktúrou prekladačov a významom jeho častí. Ďalej sa bude pokračovať zoznámením sa s architektúrou a vlastnosťami prekladača SDCC a prekladového systému LLVM. Následne bude uvedený postup pre pridanie podpory nového mikroprocesora a bude zvolený prekladač, ktorý sa použije. V ďalšej časti bude vybratý a popísaný taký mikroprocesor, ktorý doposiaľ nie je podporovaný vybraným prekladačom.

Hlavnou náplňou práce je upravenie časti prekladača, ktorá je zodpovedná za generovanie výsledného kódu, tak aby bolo možné vybraný prekladač použiť s určitým mikroprocesorom na preklad programov v jazyku C do jazyka symbolických adries. V závere práce je diskutovaný prínos práce a tiež jej nedostatky, či možné vylepšenia.

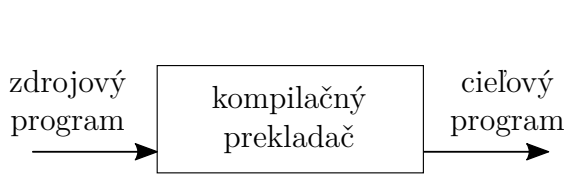
1 Konštrukcia prekladačov

V tejto kapitole je najprv popísaná všeobecná štruktúra prekladačov a následne je čitateľ oboznámený s prekladačmi SDCC a LLVM, ktorými sa táto práca zaoberá. Všeobecné informácie o prekladačoch sú prevzaté najmä z [1], [6] a [2].

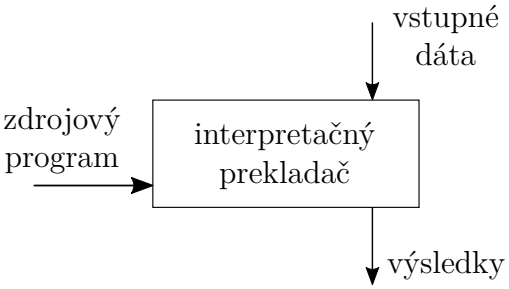
1.1 Základné pojmy

Prekladač je vo všeobecnosti program, ktorý spracúva vstupný zdrojový program napísaný v nejakom zdrojovom jazyku. Prekladače môžeme rozdeliť na kompilačné a interpretačné.

Kompilačný prekladač (obr. 1.1) alebo inak nazývaný *kompilátor* číta zdrojový program a prekladá ho na ekvivalentný cieľový program, ktorého zostavenie je vykonané pred samotným behom programu. Zdrojový a cieľový program musia byť vzájomne ekvivalentné, tzn. že pre všetky možné vstupné údaje musia dávať na výstupe rovnaké výsledky. Ak je vstupným jazykom jazyk symbolických adries a výstupom strojový kód, potom sa takýto prekladač nazýva asembler. V praxi sa týmto názvom často označuje jazyk symbolických adries, avšak toto označenie je nesprávne.



Obr. 1.1: Kompilačný prekladač



Obr. 1.2: Interpretačný prekladač

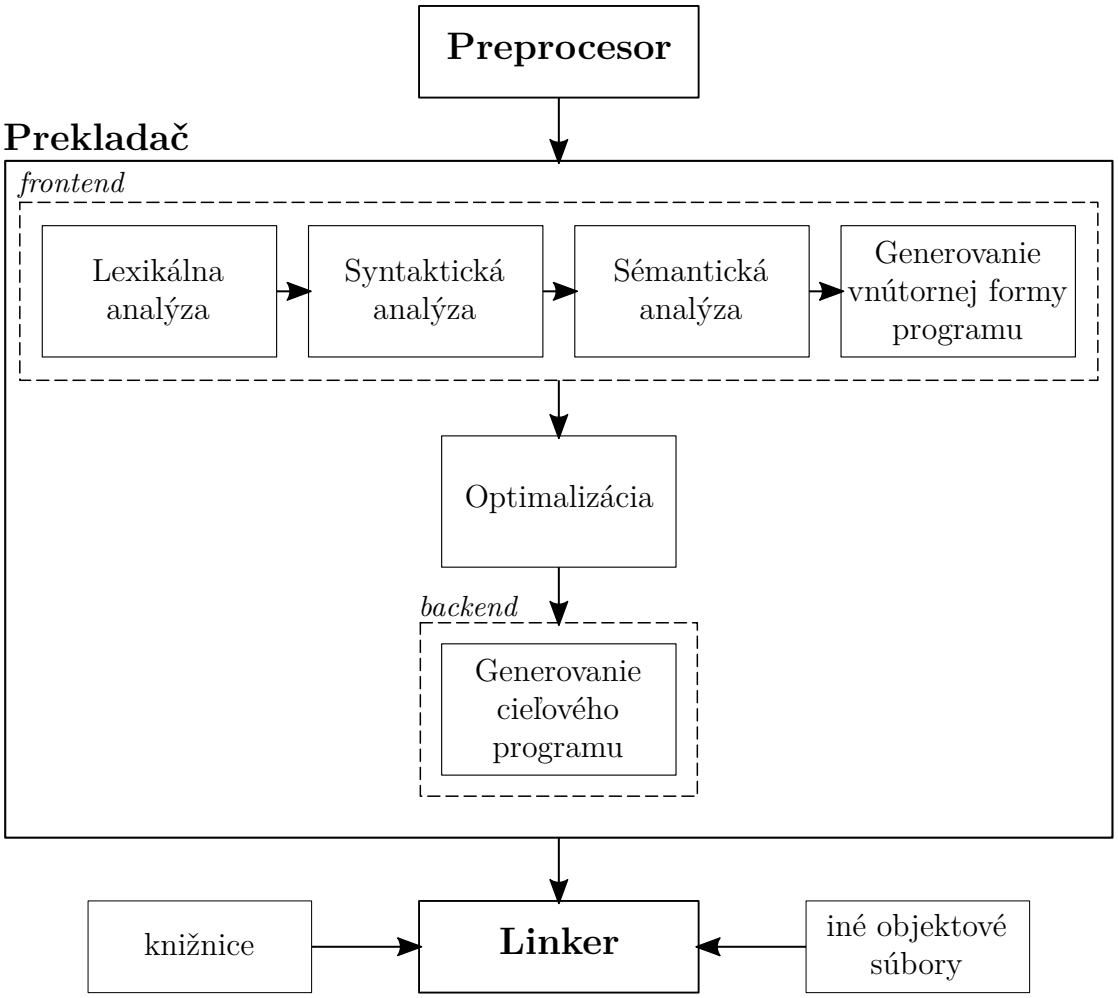
Špeciálnym druhom kompilačného prekladača je tzv. JIT kompilátor, kedy ku kompilácii dochádza v čase behu programu.

Ďalej sa bude práca zaoberať iba prvým typom prekladača – kompilátorom. Pre úplnosť je však doplnený aj stručný popis druhého typu.

Interpretačný prekladač (obr. 1.2) alebo *interpreter* číta zdrojový program po častiach (napríklad riadok po riadku). Tie zanalyzuje, vykoná jednotlivé príkazy a vráti dielčie výsledky. Pri tomto type prekladača nevzniká cieľový program a vykonávanie programu je spravidla niekoľkokrát pomalšie v porovnaní s kompilátorom.

1.1.1 Časti prekladača

Preprocesor je vstupom celého reťazca prekladača. Slúži na predprípravu zdrojového programu. Umožňuje pripojenie externých súborov, podmienený preklad a použitie makier. Samotný *prekladač* je možné rozdeliť na šesť častí ako na obr. 1.3. Jednotlivé časti predstavujú celky, ktoré vykonávajú čiastkové operácie. V praxi je najčastejšie využívaný prekladač ako celok. Je však možné niektoré časti spúšťať aj samostatne. Výstupom prekladača je cieľový program pre určitú architektúru procesora. Pri použití viacerých zdrojových súborov je nutné výsledný program pospájať do jedného celku. To zabezpečuje program nazývaný *linker*.



Obr. 1.3: Štruktúra prekladačov

Pre začiatok je nutné si zadefinovať pojmy frontend a backend, ktoré budú viackrát spomenuté v ďalších kapitolách. Koncept prekladača s vnútorným jazykom dovoľuje rozdelenie prekladača práve na spomenuté dve časti.

Frontend vykonáva lexikálnu, syntaktickú a sémantickú analýzu a generuje vnútornú formu programu. Backend generuje cieľový program. V niektorých literatúrach sa blok optimalizácie označuje aj ako samostatná middle-end časť prekladača. Takéto rozdelenie má niekoľko výhod:

- umožňuje rozklad celého problému na dve jednoduchšie časti
- pre konštrukciu oboch častí sa používajú rôzne metódy
- po definícii vnútorného jazyka je možné implementáciu oboch častí rozdeliť dvom nezávislým programátorom
- frontend je závislý iba na jazyku zdrojového programu, zatiaľ čo **backend je závislý iba na cieľovej architektúre**

Lexikálna analýza

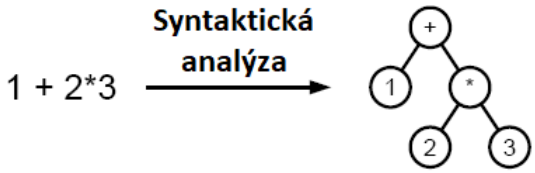
Vstupom lexikálneho analyzátora je zdrojový program vo forme reťazca znakov tak, ako je vo vstupnom súbore. Úlohou je upraviť tento vstupný reťazec do formy vhodnej pre syntaktickú analýzu a spracovanie sémantiky. Výstupom lexikálneho analyzátora je reťazec lexikálnych symbolov. Pod lexikálnymi symbolmi rozumieme identifikátory, kľúčové slová, konštanty, jednoznakové a viacznakové oddeľovače a operátory. Lexikálny analyzátor má tieto funkcie:

- nájdenie a rozpoznanie lexikálneho symbolu, čiže lexémy
- zakódovanie lexikálnych symbolov
- vynechanie nepotrebných znakov a symbolov

Lexikálna analýza môže byť ako samostatná fáza prekladača, ktorá vykoná kompletnú lexikálnu analýzu celého programu. V tomto prípade sa vytvorí medziprodukt, ktorý je ďalej spracovaný syntaktickým analyzátorom. Častejšie však zvykne byť lexikálna a syntaktická analýza súčasťou jednej fázy. Všetky lexikálne symboly prevádza lexikálny analyzátor do celočíselnej reprezentácie. Tým sa zjednoduší ďalšia práca prekladača, pretože nie je potrebné pracovať s reťazcami znakov premenlivej dĺžky, ale len s číslami, ktoré majú rovnakú dĺžku.

Syntaktická analýza

Úlohou syntaktickej analýzy je zistiť, či je zdrojový program napísaný syntakticky správne. To znamená, či reťazec lexikálnych symbolov patrí do zdrojového jazyka, ktorý má určenú svoju gramatiku. Ak to tak skutočne je, výstupom je abstraktný syntaktický strom, ktorý popisuje syntaktickú štruktúru zdrojového programu.



Obr. 1.4: Príklad tvorby abstraktného syntaktického stromu [37]

Sémantická analýza

Sémantický analyzátor zisťuje na základe popisu sémantiky, či vzniknutý abstraktný syntaktický strom má zmysel. Kontroluje napríklad to, či súhlasia dátové typy operátorov, ľavých a pravých strán priradzovacích príkazov, prípadne ak je to možné, dôjde k implicitnému pretypovaniu. Ďalej kontroluje, či použité premenné sú už deklarované a správnosť ich použitia vzhľadom k ich dátovému typu. Pri programovacích jazykoch, ktoré umožňujú preťažovanie funkcií, je nutné určiť, ktorá funkcia bude volaná a či je volaná správne. Výstupom sémantického analyzátora je anotovaný syntaktický strom s prehľadom dátových typov.

Generovanie vnútornej formy programu

V tejto fáze prekladu dochádza ku generovaniu vnútornej formy programu, inak zvanej *medzikód*, z anotovaného AST. Medzikód môže mať rôzne formy. Najčastejšie však ide buď o trojadresový kód, kedy dve adresy reprezentujú operandy a tretia adresa je miesto uloženia výsledku, alebo ďalšou formou je jazyk symbolických adries nezávislý na cieľovej architektúre.

Najväčšia výhoda spočíva v myšlienke, že v prípade potreby vytvorenia prekladača pre m rôznych vstupných jazykov a n rôznych cieľových architektúr by bolo potrebné vytvoriť $m \cdot n$ prekladačov. Použitie prekladača s generovaním medzikódu uľahčí túto úlohu na vytvorenie $m + n$ prekladačov.

Optimalizácia

Optimalizácia je proces zmeny programu za účelom skrátenia jeho dĺžky, zvýšenia rýchlosti, prípadne oboch z uvedených. Je však nutné, aby vzniknutý program bol ekvivalentom pôvodného. Takéto transformácie odstraňujú neefektívne či zbytočné operácie alebo menia ich poradie. Všeobecne tieto zmeny môžu prebiehať nad vnútornou formou, kedy ide o strojovo nezávislé optimalizácie alebo ku zmenám môže dochádzať aj v časti generátora cieľového kódu. Tie sú už však závislé od cieľovej architektúry.

Typicky medzi strojovo nezávislé optimalizácie patrí:

- odstránenie výpočtov s konštantami
- eliminácia spoločných výrazov
- presun invariantných výpočtov pred cyklus
- redukcia ceny operácie
- odstránenie nedosiahnuteľných príkazov

Generovanie cieľového programu

Ako bolo vyššie spomenuté, na cieľovej architektúre je najviac závislý generátor cieľového programu. Práca sa zaoberá najmä úpravou tejto časti prekladača, a preto je detailnejšie popísaná v nasledujúcom texte.

1.1.2 Štruktúra generátora cieľového programu

Vstupom generátora cieľového programu je postupnosť príkazov vo vnútornom jazyku prekladača. Výstupom je konečný produkt prekladu – cieľový program, ktorý je ekvivalentom zdrojového programu. Ten môže mať všeobecne tieto formy:

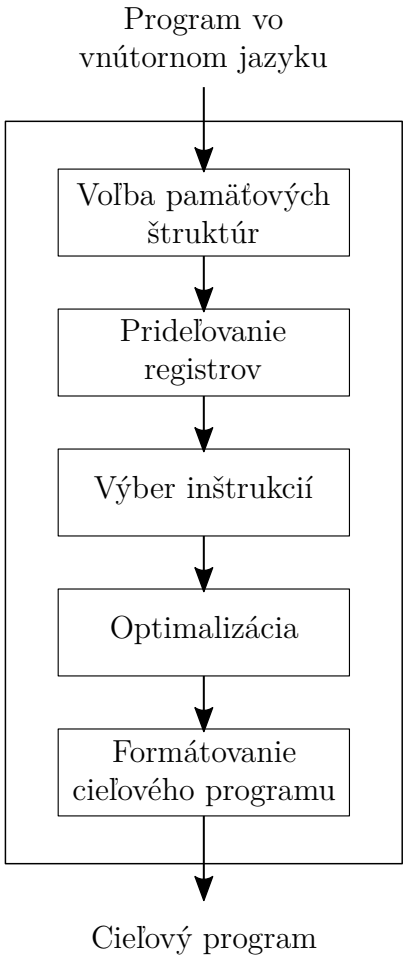
- postupnosť strojových inštrukcií s absolutnými adresami
- postupnosť strojových inštrukcií s relatívnymi adresami, tzn. premiestniteľný program
- postupnosť príkazov jazyka symbolických adries

Proces tvorby cieľového programu môže byť rozdelený na niekoľko častí, ktoré sú zobrazené na obr. 1.5.

Voľba pamäťových štruktúr zahŕňa reprezentáciu údajov zdrojového programu údajmi cieľového programu, ich pridelenie pamäťovým prvkom počítača a určenie spôsobu prístupu k samotným údajom.

Prideľovanie registrov je proces, pri ktorom dochádza k výberu konkrétnych registrov pre prechodné alebo trvalé uschovanie niektorých premenných cieľového programu a pre operácie vykonávané nad nimi. Táto časť zásadne ovplyvňuje efektifitu výsledného programu.

Výber inštrukcií predstavuje zámenu inštrukcií jazyka vnútornej formy za strojové inštrukcie. Prideľovanie registrov a výber inštrukcií sú vzájomne závislé úlohy, pretože voľba inštrukcie môže predpisovať použitie daného registra a naopak dostupnosť údajov v registroch ovplyvňuje výber inštrukcií. Vhodné naplánovanie inštrukcií



Obr. 1.5: Štruktúra generátora cieľového programu

zabezpečí ich zreťazené vykonávanie (tzv. pipelining). Ide najmä o zavedenie oneskorených skokov a riešenie konfliktov pri prístupe k výsledkom predchádzajúcich inštrukcií.

Strojovo závislé optimalizácie alebo inak zvané *peephole* optimalizácie sa vykonávajú už na vygenerovanom cieľovom programe. Medzi nich patrí napríklad:

- využitie špecifických inštrukcií
- odstránenie redundantných presunov údajov
- optimalizácia skokových inštrukcií
- zlučovanie inštrukcií pri kombinácii adresovacích módov

Formátovanie cieľového programu zabezpečí, že výstupom generátora je program formátovaný v tvare, aký je požadovaný pri ďalšom spracovaní. V prípade cie-

ľového programu v jazyku symbolických adries ide o tvar, ktorý vyžaduje asembler. V prípade výstupu v tvare strojového programu sa vytvára súbor so zakódovanými strojovými inštrukciami.

1.2 Prekladač SDCC

SDCC je bezplatný, voľne dostupný prekladač. Samotný názov SDCC je skratka pre Small Device C Compiler, čo v preklade znamená prekladač jazyka C pre malé zariadenia. Ako názov napovedá, prioritne je určený pre 8-bitové mikroprocesory. Tabuľka 1.1 zobrazuje podporované dátové typy prekladača SDCC. Podporované sú aj najnovšie štandardy jazyka C ako napríklad ISO C11, ISO C17 a ISO C2x [13]. SDCC projekt okrem samotného prekladača zahŕňa aj linker, asembler, simulátor a debugger. Používanie je možné na všetkých bežných platformách, ako je Linux, macOS a Windows [12].

Oficiálne sú prekladačom podporované nasledovné cieľové architektúry:

- rodina mikroprocesorov Intel MCS51 (8031, 8032, 8051, 8052, atď.),
- rodina HC08 od NXP (kedysi Motorola, Freescale) (HC08, HCS08),
- Maxim (kedysi Dallas) DS80C390,
- rodina Zilog Z80 (Z80, Z180, gbz80, Rabbit 2000/3000, Rabbit 3000A),
- Toshiba TLCS-90,
- Zilog eZ80 v móde Z80,
- STM8,
- Padauk PDK14,

a podpora pre mikroprocesory Microchip PIC a Padauk PDK15 je stále vo vývoji.

Tab. 1.1: Prehľad podporovaných dátových typov v prekladači SDCC [13]

dátový typ	šírka	štandardne
`_Bool / bool`	8 bitov, 1 bajt	unsigned
`char`	8 bitov, 1 bajt	unsigned
`short`	16 bitov, 2 bajty	signed
`int`	16 bitov, 2 bajty	signed
`long`	32 bitov, 4 bajty	signed
`long long`	64 bitov, 8 bajtov	signed
`float`	4 bajty (IEEE 754)	signed
`pointer`	1, 2, 3 alebo 4 bajty	generic
`__bit`[1]	1 bit	unsigned

Všetok kód projektu SDCC je šíriteľný pod licenciou GPL (GNU Public License). Z toho vyplýva, že verejnosti je vo všeobecnosti zaručená sloboda spúšťať program na akýkoľvek účel, meniť ho a študovať ako funguje, ďalej šíriť kópie, zdokonaľovať program a vylepšenia publikovať a to všetko aj s prípadným spoplatnením [38].

1.2.1 História

História SDCC siaha do roku 1995, kedy samotný autor projektu Sandeep Dutta sa pokúšal naprogramovať vývojovú dosku osadenú mikroprocesorom rady 8051. V tom čase nebol dostupný žiaden prekladač jazyka C pre tento čip, a preto sa rozhodol vytvoriť vlastný prekladač. Prvotná verzia bola veľmi jednoduchá avšak jej zverejnenie malo veľmi dobré ohlasy. Taktiež vzniklo mnoho požiadaviek na rozšírenie funkcií. Preto sa autor rozhodol pokračovať na vylepšení vtedajšej verzie a po troch rokoch sa podarilo vydať plne funkčný prekladač. Spočiatku bol na stránkach služby Geocities, ale po tom ako bol viackrát prekročený mesačný limit sťahovania, projekt bol presunutý na stránky SourceForge [16].

1.2.2 Jazykové rozšírenie

Väčšina 8-bitových mikroprocesorov má inštrukcie a dáta umiestnené v rôznych pamäťových oblastiach a pristupuje sa k ním pomocou rôznych inštrukcií. Napríklad rodina mikrokontrolérov 8051 má až tri pamäťové oblasti. Pre určenie, kde sa bude daný objekt v pamäti nachádzať existujú v jazyku C kľúčové slová `auto, static, extern, register`, avšak to nepostačuje na popis ostatných oblastí. Ak sa vezmú do úvahy aj ukazovatele, problém je omnoho rozsiahlejší. V súvislosti s tým sa naskytajú otázky, v ktorej pamäťovej oblasti bude ukazovateľ umiestnený a do ktorej oblasti ukazuje. Taktiež je potrebné riešiť to, ak argumenty niektorých funkcií štandardných knižníc sú ukazovatele. SDCC rieši tento problém zavedením nových kľučových slov, ktoré explicitne popisujú umiestnenie premennej do pamäťovej oblasti. Rovnakým spôsobom je vyriešený problém s ukazovateľmi. Aby nebolo nutné písať funkcie pre kombináciu všetkých pamäťových oblastí, riešením je použitie generického trojbajtového ukazovateľa, pričom najvyšší bajt obsahuje informácie o tom, v ktorej pamäťovej oblasti je uložený objekt, na ktorý ukazovateľ ukazuje. Pri kompilácii sa určí, do akej oblasti sa pristupuje a zabezpečí sa vloženie inštrukcií pre načítanie a zápis z alebo do príslušnej oblasti. Táto technika však navýši veľkosť programu, ale zaistí možnosť použitia štandardných funkcií ako napríklad `strcmp`.

Často dochádza aj k tomu, že premenná musí byť umiestnená na konkrétnej pozícii v pamäti. Rovnako ani pre toto neposkytuje štandard jazyka C žiadne riešenie.

[1] Podporované iba v niektorých cieľových architektúrach, napr. mcs51, ds390, ds400.

Kvôli tomuto problému SDCC zavádza kľúčové slovo `at` pre špecifikáciu absolútnej adresy premennej. Vďaka tomu je možné pristupovať k pamäťovo mapovaným perifériam pomocou bežnej syntaxe jazyka [11].

Výpis 1.1: Príklad deklarácie premenných v rôznych pamäťových oblastiach [11]

```
/* nasledujuce pole bude umiestnene v pamati programu,
pre pristup k prvkom pola bude pouzita instrukcia MOVC */
code short array_in_code[3] = {0x01, 0x02, 0x03};

/* premenna bude alokovana v internej RAM pamati,
pre pristup k nej bude pouzita instrukcia MOV */
data unsigned char in_internal_ram;

/* premenna bude alokovana v externej RAM pamati,
pre pristup k nej bude pouzita instrukcia MOVX */
xdata char array_in_external_ram[9];

/* tato premenna bude alokovana na adrese 0x8000
v externej pamati RAM */
xdata at 0x8000 ADC_PORTA;

/* ukazovatel ulozeny v internej pamati
ukazujuci na objekt v externej pamati */
xdata char * p;

/* ukazovatel ulozeny v externej pamati
ukazujuci na objekt v pamati programu */
code char * xdata p;

/* ukazovatel ulozeny v pamati programu
ukazujuci na objekt v internej pamati */
data char * code p;
```

1.2.3 Prekrývanie pamäťových miest

Najväčšou prekážkou pri programovaní mikroprocesorov s malou pamäťou je obmedzená veľkosť zásobníka pre lokálne premenné a predávanie argumentov funkciám. Čiastočne to je kompenzované predávaním argumentov pomocou registrov, ale stále je nutné alokovať lokálne premenné. V prekladači SDCC sa to rieši považovaním

premenných za statické. To však na úkor reentrancie[2]. Ďalej je problém s malou pamäťou riešený prekrývaním pamäťových miest. To znamená, že premenné rôznych funkcií, ktoré nevolajú žiadne ďalšie funkcie, sa uložia v pamäti na to isté miesto.

V prípade potreby reentrancie je možné ten istý zdrojový súbor preložiť s možnosťou `--stack-auto`, kedy budú všetky funkcie považované za reentrantné alebo je možné v zdrojovom kóde pri deklarácii funkcie určiť či má byť reentratná [11], [13].

1.2.4 Štruktúra SDCC

Prekladač SDCC je zložený z podobných častí, aké boli popísané v časti 1.1.1 tejto práce [11], [13], [17].

1. preprocesor jazyka C – príkaz `sdcpp`
2. prekladač – príkaz `sdcc`
3. asembler – príkaz `sdas<`*`target`*`>`, kde `<`*`target`*`>` predstavuje názov cieľovej architektúry
4. linker – príkaz `sdld`
5. debugger – príkaz `sdcdb`

Súčasťou frontend časti je generovanie abstraktného syntaktického stromu (AST) a kontrola dátových typov. Ako bolo spomenuté v kapitole 1.1.1, táto časť by nemala byť závislá na cieľovej architektúre. Toto pravidlo je však porušené kvôli zavedeniu nových pamäťových oblastí. Napríklad pre architektúru 8051 je vytvorená pamäťová oblasť `xdata` pre externú RAM, zatiaľ čo architektúra Z80 alebo AVR ňou nedisponujú. SDCC umožňuje, aby slovo `xdata` predstavovalo pre 8051 názov pamäťovej oblasti, ale pre inú architektúru môže byť toto slovo použité ako bežný identifikátor v jazyku C (názov premennej, makra, funkcie).

Generovanie kódu

Počas fázy generovania vnútornej formy je syntaktický strom rozložený na medzikód, tzv. *iCode*, v tvare trojadresového kódu. Vnútorne je táto forma reprezentovaná ako obojsmerne viazaný lineárny zoznam. Pri generovaní medzikódu prekladač predpokladá, že cieľová architektúra má nekonečné množstvo registrov a vytvára mnoho dočasných premenných. Aby sa znížil počet týchto premenných, zisťuje sa rozmedzie ich životnosti. Fáza alokácie registrov určuje pre každý operand typ a počet potrebných registrov. Vo väčšine prípadov mikroprocesorov ich je iba niekoľko, ktoré môžu byť využité pre nepriame adresovanie, a preto úlohou prekladača je vyhradiť (alokovať) vhodný register pre daný ukazovateľ. Prekladač sa snaží rôznymi algoritmami

[2]Reentrantná funkcia je taká, ktorá dovoľuje svoje viacnásobné volanie v jednom čase.

docieliť ukladanie operandov do registrov. Avšak ak už nie je žiaden k dispozícii, kontroluje na základe rozmedzia životnosti operandov, ktorý momentálne nevyužívaný operand je možné prepísať. V tejto fáze je nutné uvažovať aj o ďalších špecifikách jednotlivých cieľových architektúr. Niektoré mikroprocesory majú akumulátor, čo je špeciálny register pre uchovávanie medzivýsledkov operácií. Práca s ním je rýchlejšia a preto je vhodné akumulátor upredňostňovať pred ostatnými registrami.

Po vygenerovaní vnútornej formy je vykonaných viacero štandardných strojovo nezávislých optimalizácii. Po tejto fáze nasleduje generovanie cieľového kódu pre daný typ mikroprocesora. Ide o nahradenie inštrukcií medzikódu inštrukciami cieľovej architektúry. Niektoré inštrukcie sú vygenerované spôsobom charakteristickým pre príslušný mikroprocesor. Napríklad mikroprocesor Z80 nevyužíva registre na predávanie parametrov a tiež nepodporuje tabuľku skokov.

Ďalšou fázou je aplikovanie strojovo závislých inštrukcií na základe popísaných pravidiel. Poslednou fázou je nahradenie kódu zapísaného v jazyku strojových inštrukcií strojovým kódom.

iCode

Použitím možnosti `--dump-i-code` pri kompilácii je možné zobraziť vygenerovaný medzikód. Tabuľka 1.2 uvádza niektoré príkazy tejto vnútornej formy. Nasledujúci príklad zobrazuje textovú reprezentáciu medzikódu.

```
foo.c(11:14:19:1) *(iTemp6 [lr5:16]{_near * int}[r0])
    := iTemp10 [lr13:14]{int}[r2 r3]
```

Každý riadok obsahuje informáciu o názve súboru, čísle riadku, poradie vykonávania, kľúč v hešovacej tabuľke a hĺbku vnorenia. Taktiež každý operand nesie informáciu o rozmedzí svojej životnosti, špecifikáciu dátového typu a register, v ktorom je premenná umiestnená.

1.2.5 Inštalácia a používanie SDCC

Ako bolo spomenuté vyššie, prekladač SDCC je možné používať na všetkých bežne používaných platformách. V tejto časti je uvedený postup pre inštaláciu v operačnom systéme Linux. Návod na inštaláciu a používanie na ostatných systémoch je k dispozícii v manuáli SDCC [13].

Zostavenie

1. Stiahnutie aktuálneho obrazu všetkých súborov projektu a tiež spustiteľných súborov je možné zo stránky [14]. Najaktuálnejšia rozpracovaná verzia zdrojových kódov je dostupná v repozitári v systéme Subversion. Získať ju je možné

Tab. 1.2: Niektoré príkazy vnútornej formy prekladača SDCC [13]

iCode	operandy	popis	ekvivalent v jazyku C
`+`	IC_LEFT, IC_RIGHT, IC_RESULT	sčítanie	`IC_RESULT = IC_LEFT + IC_RIGHT;`
`UNARYMINUS`	IC_LEFT, IC_RESULT	unárny mínus	`IC_RESULT = -IC_LEFT;`
`CALL`	IC_LEFT, IC_RESULT	funkčné volanie	`IC_RESULT = IC_LEFT();`
`JUMPTABLE`	IC_JTCOND, IC_JTLABELS	tabuľka skokov	`príkaz switch`
`CAST`	IC_LEFT, IC_RIGHT, IC_RESULT	pretypovanie	`IC_RESULT = typeof(IC_LEFT) IC_RIGHT;`
`POINTER_SET`	IC_RIGHT, IC_RESULT	nepriame priradenie	`*(IC_RESULT) = IC_RIGHT;`
`IPUSH`	IC_LEFT	vloženie na zásobník	`žiaden`

pomocou príkazu v termináli:
`svn checkout svn://svn.code.sf.net/p/sdcc/code/trunk sdcc`.

2. Zmenu aktuálneho priečinka vykonáme príkazom `cd sdcc`.
3. Príkazom `./configure` sa spustí skript, ktorý analyzuje systém a nakonfiguruje ho, aby inštalácia prebehla správne.
4. Samotná kompilácia SDCC prekladača sa spustí príkazom `make`.
5. Posledným krokom je príkaz `sudo make install`, ktorý prekopíruje všetky spustiteľné súbory, hlavičkové súbory a dokumentáciu do inštalačného priečinka.

Otestovanie správnosti zostavenia je možné napísaním príkazu `sdcc --version` do terminálu, ktorý by mal vypísať zoznam podporovaných cieľových architektúr a verziu, ako napríklad: `SDCC : mcs51/z80/avr/ds390/pic16/pic14/ds400/hc08/s08 3.9.0 #11195 (May 8 2006)(UNIX)`.

Používanie

Pre skompilovanie programu stačí do terminálu napísať príkaz `sdcc sourcefile.c`, čo vygeneruje niekoľko výstupných súborov. Pre vygenerovanie jediného súboru s programom v jazyku symbolických adries je potrebné vykonať preklad s parametrom `-S`. Výber cieľovej architektúry je možný pomocou parametra `-m<port>`,

kde `<port>` predstavuje jej názov. Pre určenie konkrétneho modelu z celej rodiny mikroprocesorov s názvom `<mcu>` slúži parameter `-p<mcu>`. Zoznam ďalších voliteľných parametrov pre kompiláciu je možné zistiť zadaním príkazu `sdcc --help`.

1.3 Prekladač LLVM

LLVM je projekt, ktorý zastrešuje vývoj úzko spätých nízkoúrovňových nástrojov (napr. kompilátor, asembler, debugger) kompatibilných s existujúcimi nástrojmi. Projekt nielenže poskytuje vynikajúce nástroje ako napríklad Clang, ale vyniká najmä svojou vnútornou architektúrou. Používanie je možné na bežných platformách, ako je Linux, Solaris, FreeBSD, NetBSD, macOS a Windows. Informácie uvedené v tejto časti sú prevzaté najmä z [18] a [19].

História tohto projektu siaha do roku 2000, kedy vznikol na univerzite Illinois v rámci výskumu techník dynamickej kompilácie. Prvotná verzia LLVM bola voľne publikovaná v roku 2003 pod licenciou samotnej univerzity. V roku 2005 si spoločnosť Apple Inc. najala jedného z prvotných tvorcov projektu a vytvorila tím, ktorý mal pracovať na LLVM kvôli jeho použitiu v rámci ich vývojových systémov. Dodnes je LLVM súčasťou najnovších vývojových nástrojov spoločnosti Apple a projekt finančne podporuje.

Názov LLVM bol pôvodne skratkou pre *Low Level Virtual Machine*, čo v preklade znamená nízko-úrovňový virtuálny stroj. Časom sa od tohto názvu upustilo, aby sa predišlo nejasnostiam. LLVM sa totiž rokmi vyvinulo do takej podoby, že má iný charakter ako to, čo väčšina súčasných vývojárov považuje za virtuálny stroj. V súčasnosti je projekt spravovaný nadáciou LLVM a je šírený pod Apache licenciou, ktorá dovoľuje slobodné použitie ako aj úpravu softvéru a jeho redistribúciu [36].

V čase vzniku projektu boli podobné nástroje vytvárané ako celok, pričom nebolo možné využiť len niektorú časť. Napríklad neexistoval žiaden spôsob, ako by sa dal využiť len syntaktický analyzátor prekladača GCC[3] pre statickú analýzu zdrojového kódu. Zatiaľ čo skriptovacie jazyky poskytovali spôsob, pre ich využitie v rozsiahlejších aplikáciách, časti týchto prekladačov sa nedali opakovane využiť ani v rámci samotného prekladača. Prekladače boli buď statické alebo dynamické napríklad vo forme JIT.

Preto od samého začiatku bolo LLVM vytvorené ako súbor znovu použiteľných knižníc s dobre navrhnutým rozhraním. Tieto moduly sa tak dajú jednoducho využiť aj mimo samotného prekladača. Rovnako to uľahčuje prácu pri pridávaní nových súčastí. Okrem toho sa štruktúra používa na statický ale aj dynamický preklad.

[3]GCC je zbierka prekladačov pôvodne napísaná ako systémový prekladač pre operačný systém GNU. Obsahuje frontend časť pre programovacie jazyky C, C++, Objective-C, Fortran, Ada, Go a D [35].

Dokonca LLVM nahradilo mnoho prekladačov, ktoré boli vyvinuté len pre špeciálne účely.
Celý LLVM projekt sa skladá z týchto častí [19]:

1. LLVM jadro – samotná implementácia častí prekladača
2. Clang – frontend prekladača pre C/C++/ObjC (bližšie popísaný nižšie)
3. LLDB – debugger (náhrada za debugger GDB určený pre prekladač GCC)
4. libc++ a libc++ABI – štandardné knižnice s plnou podporou C++11 a C++14
5. compiler-rt – podpora niektorých nízko úrovňových funkcií a niektoré dynamické knižnice pre nástroje dynamického testovania
6. OpenMP – podpora paralelného programovania
7. polly – optimalizácie pre vyrovnávaciu pamäť
8. libclc – implementácia štandardnej knižnice OpenCL
9. klee – implementácia symbolického virtuálneho stroja
10. LLD – linkér, ktorým je možné priamo nahradiť systémový linkér

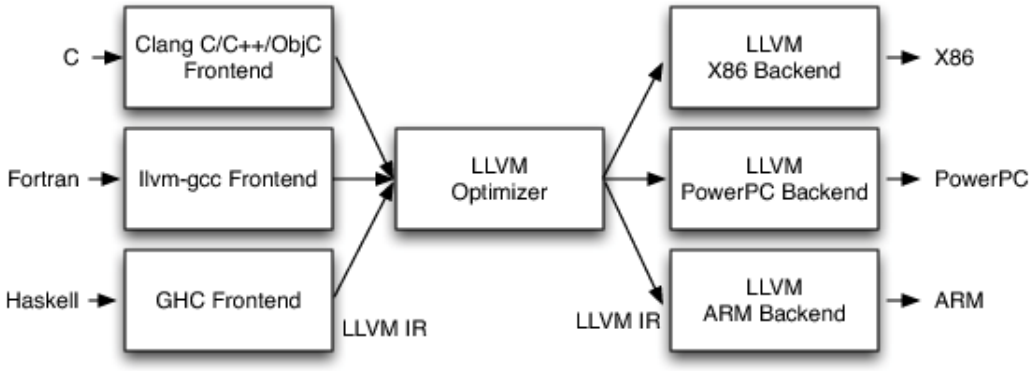
1.3.1 Štruktúra LLVM

Štruktúra systému LLVM vychádza zo všeobecného popisu, ktorý rozdeľuje prekladač na tri hlavné časti (pozri obr. 1.6). Ako bolo spomenuté, najväčšia výhoda tejto štruktúry pramení z jednoduchosti rozšírenia prekladača. Či už pre pridanie nového zdrojového jazyka alebo novej cieľovej architektúry. Vďaka tomu je možné prekladač využiť pre jazyky C, C++, ObjC, Fortran, Haskell a mnohé iné. Okrem toho sú oficiálne podporované cieľové architektúry ako napríklad: x86, PowerPC, ARM, AVR, AArch64, MSP430 a iné. Okrem tejto oficiálnej podpory architektúr je možné na internete nájsť zdrojové kódy pre podporu Z80, TriCore alebo Cpu0.

Vstupom celého prekladača je teda súbor, v ktorom je program zapísaný v zdrojovom jazyku. Výstupom každej frontend časti je program vo vnútornej forme, tzv. LLVM IR. Poslednou časťou prekladača je backend vytvorený pre danú cieľovú architektúru. Výstupom backend časti prekladača je skompilovaný program v jazyku symbolických adries, prípadne spustiteľný súbor.

1.3.2 Clang

Spočiatku využívalo LLVM ako svoj frontend program llvm-gcc. To bol prekladač GCC, ktorý bol upravený tak, aby na výstupe generoval medzikód v jazyku LLVM IR. Spoločnosť Apple v tom čase využívala hlavne jazyk Objective-C, ale ten nebol pre GCC takou prioritou. Preto sa spoločnosť rozhodla pre vývoj vlastného frontendu Clang pod projektom LLVM tak, aby sa dal využiť ako priama náhrada za



Obr. 1.6: Štruktúra LLVM [18]

prekladač GCC. V roku 2007 tak bola vydaná prvotná verzia. Dnes spoločnosť Apple využíva hlavne jazyk Swift, pre ktorý si tiež vyvinula vlastnú frontend časť.

Clang dlhé roky dosahoval podstatne lepšie výsledky v porovnaní s prekladačom GCC v testoch založených na porovnávaní rýchlosti kompilácie, ale aj samotného behu programu. GCC sa však časom zlepšilo a v dnešnej dobe dosahuje porovnateľné výsledky ako Clang.

Najväčšími prednosťami Clangu sú lepšia diagnostika (výpis 1.2), jednoduchšia integrácia naprieč programovacími prostrediami, ľahký vývoj a údržba a v neposlednom rade licencia, ktorá dovoľuje komerčné využitie. Vďaka týmto vlastnostiam je Clang využitý pre aplikácie, ktoré dbajú na svoj výkon, ako napríklad prehliadač Chrome alebo Firefox [25].

Výpis 1.2: Výstup syntaktickej analýzy

```
$ clang -fsyntax-only t.c
t.c:7:39:
error: invalid operands to binary expression ('int' and 'struct␣A')
  return ((SomeA.X + 40) + SomeA) / 42 + SomeA.X;
          ~~~~~~~~~~~~~~ ^ ~~~~~
```

1.3.3 LLVM IR

Medzikód projektu LLVM pripomína sadu inštrukcií architektúry RISC[4]. Je vo forme trojadresového kódu a je založený na SSA (Static Single Assignment) forme, čo znamená že, do každej premennej je za celý čas zapísaná hodnota iba jedenkrát. Vďaka tomu je zaručená typová bezpečnosť, operácie na nízkej úrovni, flexibilita

[4]Architektúra s obmedzenou sadou inštrukcií

a schopnosť reprezentovať všetky vysoko úrovňové jazyky. Taktiež tento zápis výrazne zjednodušuje operácie vykonávané na medzikóde, a preto sa medzikód používa v rámci všetkých operácií, ktoré nie sú závislé na zdrojovom jazyku alebo cieľovej architektúre. Keďže forma SSA povoľuje jediné zapísanie do premennej, v tejto fáze je vytváraných mnoho dočasných premenných [20].
LLVM medzikód môže byť v troch rôznych formách, pričom všetky sú rovnocenné:

- medzikód v operačnej pamäti – pri behu systému LLVM
- medzikód v binárnej podobe (prípona `.bc`) – vhodné pre rýchle načítavanie (napr. JIT)
- medzikód v textovej podobe (prípona `.ll`)

Vďaka tejto vlastnosti je možné medzikód uložiť na disk a neskôr s ním plnohodnotne pracovať.

Výpis 1.3: Zdrojový program v jazyku C

```
unsigned add1(unsigned a, unsigned b)
  {
  return a+b;
  }

unsigned add2(unsigned a, unsigned b)
  {
  if (a == 0) return b;
  return add2(a-1, b+1);
  }
```

Výpis 1.3 prestavuje zápis funkcií sčítavania dvoma spôsobmi. V prvom prípade ide o klasické sčítanie dvoch premenných, zatiaľ čo v druhom prípade je využitá rekurzia, kedy jedna premenná sa inkrementuje a druhá dekrementuje. Výsledok je dosiahnutý ak dekrementovaná premenná dosiahne nulovú hodnotu. Výsledkom je potom premenná, ktorá bola inkrementovaná.

Výpis 1.4 zobrazuje ten istý program ale zapísaný vo vnútornej forme LLVM. Prvá funkcia je nahradená veľmi podobným ekvivalentom. Rozdiel je iba v tom, že vzniká jedna dočasná premenná, do ktorej sa ukladá výsledok a ten je návratovou hodnotou funkcie. Druhá funkcia je na prvý pohľad zapísaná zložitejšie. Je pridaných viacero dočasných premenných, aby bola zachovaná forma SSA. Tiež je použitých niekoľko návestí, aby boli jednoznačne definované ciele skokov. Na začiatku funkcie sa porovnáva dekrementovaná premenná s nulou. Ak je hodnota tejto premennej nula, funkcia vráti výsledok v podobe inkrementovanej premennej. Ak je ale hodnota tejto premennej väčšia ako nula, dekrementuje sa a druhá premenná sa inkrementuje. Následne sa zavolá táto funkcia zase, ale už s novými premennými.

Výpis 1.4: Ekvivalentný program v medzikóde prekladača LLVM

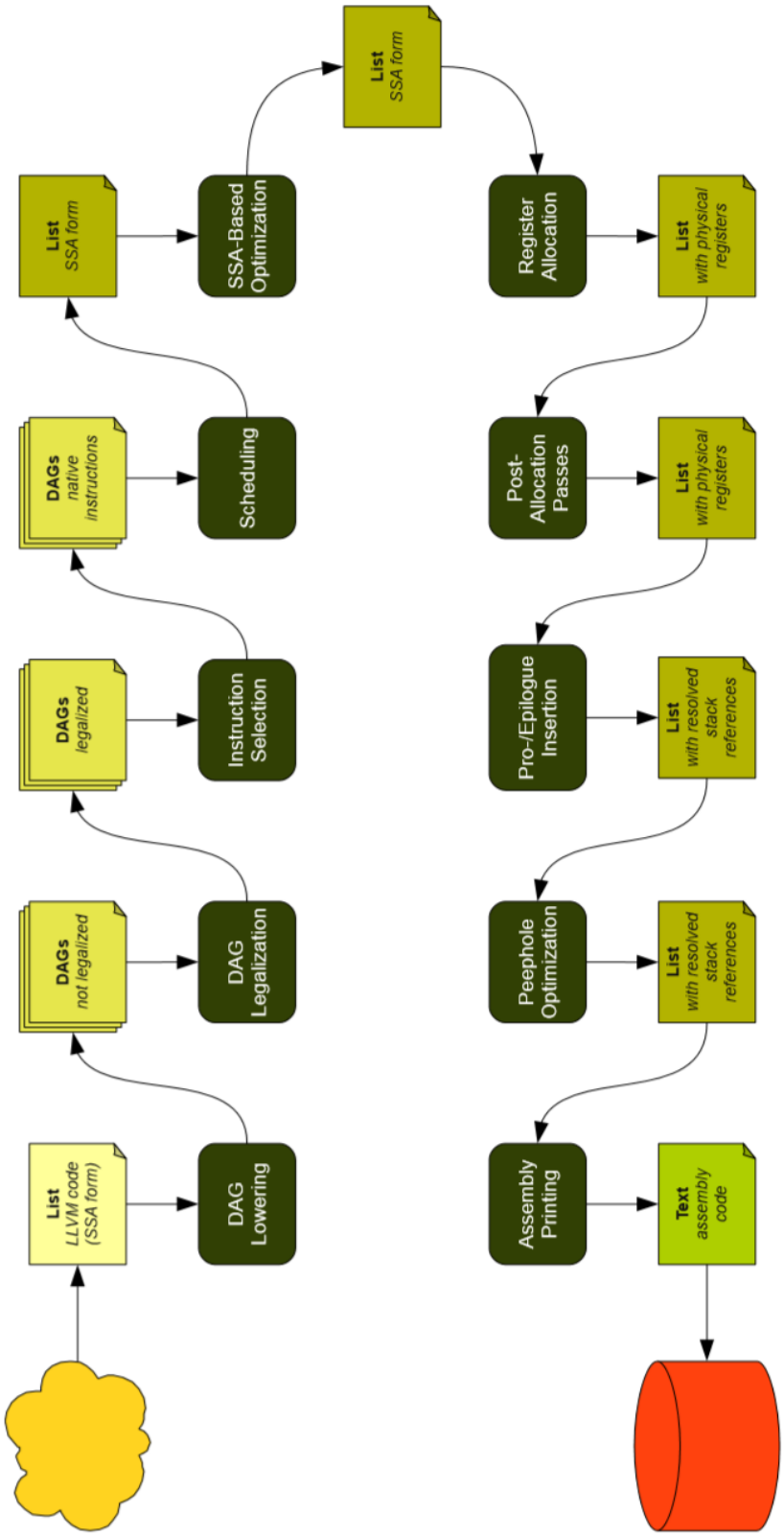
```
define i32 @add1(i32 %a, i32 %b)
  {
  entry:
    %tmp1 = add i32 %a, %b
    ret i32 %tmp1
  }

define i32 @add2(i32 %a, i32 %b)
  {
  entry:
    %tmp1 = icmp eq i32 %a, 0
    br i1 %tmp1, label %done, label %recurse

  recurse:
    %tmp2 = sub i32 %a, 1
    %tmp3 = add i32 %b, 1
    %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
    ret i32 %tmp4

  done:
    ret i32 %b
  }
```

Vnútorná forma LLVM okrem samotného programu nesie aj informácie o cieľovej architektúre, ako napríklad jej názov a konkrétny typ použitého procesora alebo platformy, ďalej veľkosť jednotlivých dátových typov a ich spôsob zarovnania v pamäti, atribúty programu a v neposlednom rade je zahrnutá aj informácia o verzii prekladového systému.

1.3.4 Generátor cieľového programu

Generátor cieľového programu projektu LLVM je univerzálne navrhnutý tak, aby generoval efektívny a kvalitný kód pre štandardné mikroprocesory, ktoré sú založené na práci s registrami. Generovanie programu je typicky rozdelené do niekoľkých prechodov celého programu, okrem nich môže byť v prípade potreby pridaný špecifický prechod, napríklad na spracovanie čísel s pohyblivou desatinnou čiarkou, pre ktoré môže byť vytvorený osobitný zásobník.

Aby bolo možné pochopiť štruktúru backend časti prekladača, je potrebné porozumieť ako generátor kódu funguje. Pre preklad vnútornej formy LLVM do jazyka symbolických adries je nutné vykonať niekoľko krokov, ktoré zobrazuje obrázok 1.7. Informácie v tejto časti sú čerpané z [4], [7], [8] a [21].



Obr. 1.7: Štruktúra backend časti prekladača LLVM [7]

Zostavenie grafu

Prvým krokom je prevod vnútornej formy v tvare trojadresového kódu na orientovaný acyklický graf (DAG). Každý základný blok[5] programu reprezentuje práve jeden DAG. Hrany grafu kódujú tok dát a uzly predstavujú inštrukcie, ktoré sú reprezentované objektom triedy `SDNode` a ten obsahuje:

- číslo (`OpCode`) – identifikátor inštrukcie
- výsledky (`Defs`) – objekt uchováva informáciu o dátovom type pre všetky výsledky. Nie všetky inštrukcie produkujú výsledok, ako napríklad inštrukcie skoku. Taktiež niektoré kombinované inštrukcie, napr. delenie/zvyšok po delení, môžu generovať viacero výsledkov.
- operandy (`Uses`) – predstavujú zoznam všetkých uzlov, na ktorých závisí. Môže ísť o dátovú závislosť, kedy uzol používa hodnotu definovanú iným uzlom, alebo závislosť riadiacu, ktorá určuje poradie vykonaných operácií.

Legalizácia

Pretože cieľová architektúra nemusí podporovať všetky dátové typy, je ich nutné konvertovať na také, ktoré sú podporované. Existujú dve hlavné možnosti, ako je to možné docieliť. Malé dátové typy je možné previesť na väčšie (napr. `i1` na `i8`), ide o tzv. *povýšenie*. Alebo druhou možnosťou je tzv. *rozšírenie* veľkých dátových typov na malé (napr. `i16` na pár `i8` hodnôt). Tieto zmeny môžu ku hodnote premennej pridávať znamienko alebo nulu, aby mal výsledný program rovnaké správanie ako vstupný.

Cieľová architektúra tiež nemusí podporovať všetky inštrukcie vnútornej formy. Preto je potrebné previesť uzly grafu na také, ktoré sú natívne podporované. Docieliť to je možné napríklad zámenou nepodporovanej inštrukcie za sekvenciu iných podporovaných inštrukcií.

Výber inštrukcií

Behom tejto fázy sú inštrukcie vnútornej formy legalizovaného grafu zamieňané za strojové inštrukcie cieľovej architektúry. Pre túto transformáciu sa využíva technika *pattern matching*, ktorá spočíva vo výhľadávaní definovaných vzorov v grafe a náhradou za príslušný vzor, ktorý reprezentuje strojovú inštrukciu. Týmto je vygenerovaný nový DAG, ktorý obsahuje už iba natívne inštrukcie.

[5]Základný blok predstavuje lineárnu sekvenciu programu bez skokov a teda iba s jedným vstupom a s jedným výstupom.

Cieľová architektúra môže poskytovať inštrukcie, ktoré sú presným ekvivalentom inštrukcií vnútornej formy. Vyhľadávaný vzor však môže tvoriť aj sekvencia viacerých inštrukcií alebo naopak, jedna inštrukcia môže byť nahradená postupnosťou niekoľkých strojových inštrukcií.

Plánovanie

Novovzniknutý DAG graf je rozložený na sekvenčnú postupnosť inštrukcií. Každá funkcia je reprezentovaná ako objekt triedy `MachineFunction`, ktorá obsahuje postupnosť základných blokov, čiže inštancií triedy `MachineBasicBlock`. Napokon základné bloky sú zložené zo samotných strojových inštrukcií, čiže objektov triedy `MachineInstr`. Plánovanie určí poradie inštrukcií, v akom majú byť vykonávané a to na základe viacerých kritérií, ako napríklad využitie čo najmenšieho počtu registrov.

Postupnosť inštrukcií je stále vo forme SSA, ako tomu bolo pri vnútornej forme LLVM. Kvôli tomu tento kôd ešte nie je úplne validný. Všetky inštrukcie stále využívajú nekonečné množstvo virtuálnych registrov.

Alokácia registrov

Táto fáza spočíva v nahradení nekonečného množstva virtuálnych registrov na fyzické, ktorých počet je limitovaný. Ak počet fyzických registrov nie je postačujúci, niektoré virtuálne registre sú namapované do pamäte, to znamená, že je potrebné vygenerovať tzv. *spill* kód na premiestnenie hodnoty z registra do pamäte. V tejto fáze kód prestáva byť naďalej v SSA forme.

LLVM poskytuje štyri rôzne implementácie alokácie registrov. Výber je možné uskutočniť parametrom `--regalloc=<regalloc_name>` pre statický kompilátor `llc`, kde `<regalloc_name>` predstavuje jednu z nasledujúcich možností:

- `greedy` – **predvolený** alokátor a zároveň veľmi dobre odladený, snaží sa o minimalizáciu *spill* kódu
- `basic` – využíva veľmi jednoduchý alokátor a poskytuje rozhranie pre jeho jednoduché rozšírenie a vývoj nového alokátora registrov
- `fast` – pracuje so základným blokom programu a snaží sa o uchovanie hodnôt v registroch a ich znovuvyužitie, čo najviac ako to je možné
- `pbqp` – alokátor je založený na riešení PBQP[6] problému [22]

[6]Partitioned Boolean Quadratic Programming

Vkladanie prológu a epilógu

Po tom ako boli fyzické registre alokované a bol vygenerovaný potrebný *spill* kód, je možné určiť, koľko miesta je potrebné vyhradiť pre každú funkciu na zásobníku.

Doposiaľ boli všetky referencie na zásobník abstraktné. V dôsledku toho, že už je známe koľko miesta potrebuje funkcia na zásobníku, je možné abstraktné referencie nahradiť konkrétnymi posuvmi voči ukazovateľu zásobníka.

Výpis kódu

Behom poslednej fázy je vygenerovaný výsledný kód. Pre statický preklad je výsledkom program zapísaný v jazyku symbolických adries alebo objektový kód. Pri využití JIT kompilácie je strojový kód zapísaný priamo do pamäte.

Optimalizácie

V rámci generovania kódu je vykonávaných niekoľko prechodov, kedy dochádza k optimalizáciam programu. Jedná sa predovšetkým o strojovo závislé optimalizácie. Niektoré z nich sú však vykonávané ešte na vnútornej forme LLVM, iné už na výslednom kóde.

1.3.5 Inštalácia a používanie LLVM

V tejto časti je uvedený postup pre inštaláciu v operačnom systéme Linux. Informácie sú prevzaté z [23], kde je možné zistiť aj ďalšie podrobnosti, alebo aj to ako postupovať pri inštalácií na iných platformách. Návod pre používanie je rovnaký pre všetky podporované platformy.

Zostavenie

1. `git clone https://github.com/llvm/llvm-project.git` – Stiahnutie aktuálneho obrazu všetkých súborov projektu z verejne dostupného GitHub repozitára. Iné vydané verzie je možné získať na stránke `http://releases.llvm.org/`.
2. Nasleduje konfigurácia pred zostavením:
   - `cd llvm-project`
   - `mkdir build`
   - `cd build`
   - `cmake -G "Unix Makefiles" -DLLVM_ENABLE_PROJECTS="clang" -DCMAKE_BUILD_TYPE="Release" ../llvm` – Štandardne sa zostavuje iba LLVM jadro, zostavenie iných súčastí (napr. frontend časti Clang) je

nutné zvlášť povoliť. Existuje niekoľko spôsobov ako je možné projekt zostaviť. Možnosť `Release` je určená pre vytvorenie finálnej verzie, zatiaľ čo možnosť `Debug` je určená pre vývojárov.

3. `make -j N` – Samotné zostavenie jadra LLVM a ďalších zvolených súčastí. Vykonávanie tejto fázy je veľmi pomalé z dôvodu využitia len jedného jadra procesora. Proces je možné zrýchliť paralelným behom programu a to zadaním parametra `-j N`, kde namiesto `N` sa napíše počet jadier procesora, ktoré sa majú využiť.
4. `sudo make install` – Prekopírovanie všetkých spustiteľných súborov do systémového inštalačného priečinka.

Projekt je kompilovaný pomocou systémového prekladača, napr. GCC. Zaujímavosťou je možnosť využitia dvojkrokového zostavenia, kedy sa najprv skompiluje prekladač Clang pomocou systémového prekladača (napr. GCC), a následne sa pomocou neho skompiluje znova prekladač Clang a ostatné súčasti LLVM jadra. Docieliť to je možné za pomoci parametra `-DCLANG_ENABLE_BOOTSTRAP=On` pre príkaz `cmake` [24].

Používanie

Pre začiatok je vhodné si vytvoriť jednoduchý testovací program v jazyku C.

Výpis 1.5: Testovací program Hello World

```
#include <stdio.h>

int main()
  {
  printf("Hello␣World\n");
  return 0;
  }
```

Preklad tohto programu do spustiteľného súboru pre práve používanú cieľovú architektúru je možný pomocou príkazu `clang hello.c -o hello`. Na prevod zdrojového kódu do medzikódu LLVM slúži príkaz `clang -emit-llvm hello.c -c`. Výstupom tak bude medzikód v binárnej forme. Ak je potrebný výstup v textovej forme, namiesto parametra `-c` je nutné zadať parameter `-S`. Prevod medzi týmito dvoma formami je tiež možné uskutočniť programom `llvm-as` resp. `llvm-dis`. Príkazom `lli hello.bc` sa spúšťa program za pomoci JIT kompilátora. Na skompilovanie LLVM medzikódu do jazyka symbolických adries sa dá využiť program `llc`.

Kompilovanie programu pre inú architektúru, ako pre tú, na ktorej prekladač beží je možné spustením prekladača s parametrom `-triple`. Parameter sa udáva v tvare

`<architektúra>-<predajca>-<systém>-<abi>`[7]. Z týchto parametrov je povinný iba prvý, ak na ostatných nezáleží, nahradia sa slovom `unknown`.

Použitie parametra `-O4` pri kompilácii zabezpečí, že program bude najprv spracovaný linkérom a až tak budú vykonané všetky optimalizácie. Táto technika sa označuje ako LTO (Link-Time Optimization) a prekladaču umožňuje program oveľa lepšie optimalizovať, pretože zdrojový program vidí ako celok a nie po častiach.

[7] ABI je skratka pre aplikačné binárne rozhranie, čo predstavuje nízko úrovňové rozhranie medzi aplikáciami, knižnicami alebo časťami aplikácií.

2 Vytvorenie novej backend časti

Aby bolo možné prekladač používať pre generovanie kódu spustiteľného na cieľovej architektúre, ktorá nie je prekladačom podporovaná, je nutné upraviť jeho backend časť. V nasledujúcej kapitole bude čitateľ oboznámený s postupom, pomocou ktorého je možné vyhovieť tejto požiadavke. Postup riešenia tohto problému bude popísaný pre voľne dostupné prekladače SDCC a LLVM popísané v predošlej kapitole v časti 1.2 a 1.3 a vychádza z ich architektúry.

Na konci kapitoly budú zhrnuté informácie o oboch prekladačoch a vyberie sa jeden z nich, v ktorom sa nová backend časť implementuje.

Najprv je potrebné ozrejmiť niektoré pojmy. Slovom *port* sa v prekladači SDCC označuje cieľová architektúra, zatiaľ čo v prekladači LLVM je označovaná slovom *target*.

2.1 Prekladač SDCC

Postup pridania podpory novej cieľovej architektúry pre prekladač SDCC je v oficiálnej dokumentácii projektu [13] popísaný veľmi málo. Študent Jakub Horník však vo svojej diplomovej práci [10] uvádza veľmi detailný popis. Informácie sú preto prevažne čerpané z jeho práce a tiež zo zdrojových kódov prekladača [12]. Postup bude vysvetlený na existujúcom porte `mcs51`, čo predstavuje architektúru 8051.

2.1.1 Popis cieľovej architektúry

Prvým krokom je vytvorenie priečinka s názvom nového portu v priečinku `src`. Základom popisu je štruktúra PORT v súbore `src/port.h`, ktorá popisuje všetky vlastnosti portu. V súbore `mcs51/main.c` je táto štruktúra inicializovaná. Medzi tieto vlastnosti patrí napríklad:

- unikátne id nového portu
- názov portu (použité pre parameter `-m`)
- ukazovateľ na funkciu, ktorá prevádza medzikód na cieľový program
- povolené parametre príkazového riadku
- príkazy a argumenty pre spustenie asemblera a linkera
- prípony výstupných súborov asemblera a linkera
- cesta k štandardným hlavičkovým súborom
- ukazovatele na funkcie pre optimalizáciu
- cesta k súboru s pravidlami pre peephole optimalizáciu
- veľkosti dátových typov
- definovanie špeciálnych pamäťových oblastí

- špecifické kľúčové slová (pozri 1.2.2)
- určenie vlastností zásobníka
- definovanie vlastností debuggera
- ukazovateľ na inicializačnú funkciu
- ukazovateľ na funkciu pre alokáciu registrov
- ukazovateľ na funkciu pre vygenerovanie inicializačného programu
- existencia inštrukcií pre násobenie a delenie
- povolenie transformácie logických podmienok
- určenie endianity (little alebo big endian)

Ďalej v súboroch `mcs51/main.h` a `mcs51/main.c` je potrebné napísať funkcie pre spracovanie parametrov zadaných v termináli pri preklade. V týchto súboroch je zvykom napísať aj inicializačné funkcie a zabezpečiť generovanie tabuľky prerušení.

2.1.2 Alokácia registrov

Úlohou tejto časti je priradenie registrov operandom inštrukcií cieľového jazyka. Funkcie pre túto fázu sa píšu v súboroch `mcs51/ralloc.h` a `mcs51/ralloc.c`. Aby bolo možné zistiť stav jednotlivých registrov, každý z nich je popísaný v tabuľke registrov. V nej je uvedený typ a označenie registra, aký operand sa v ňom nachádza, offset operandu alebo aj to či je register voľný alebo vyhradený pre iný účel. Podobná tabuľka je použitá aj pre dátovú pamäť.

Pri alokácii registrov môže všeobecne dôjsť k trom stavom, ktoré je nevyhnutné riešiť.

Operand sa už nachádza v registri

Táto varianta je najlepšia, nie je potrebné alokovať žiaden register a operand môže byť priamo použitý. Ak je operand zložený z viacerých bajtov, je nutné popísaný postup aplikovať pre každý jeden bajt.

Operand v registri ešte nie je, ale niektorý z registrov je voľný

V tomto prípade sa vyberie prvý voľný register a premiestni sa do neho operand uložený v pamäti, v inom registre alebo sa do neho vloží konštanta.

Operand v registri ešte nie je, žiaden register nie je voľný

Táto možnosť je najzložitejšia. Vyžaduje vykonanie analýzy rozsahu životnosti premenných uložených v registroch. Nasleduje rozhodnutie, ktorý register bude uvoľnený. Predovšetkým sa vyberá taký, v ktorom je uložená premenná, ktorá už nie je

potrebná. Ak taký register nie je, vyberie sa register s hodnotou, ktorá bola najdlhšie nepoužitá a dočasne sa odloží do pamäte pre dáta. Napokon sa do uvoľneného registra vloží požadovaný operand.

2.1.3 Generátor kódu

V tejto časti prekladu dochádza k prevodu vnútornej formy prekladača iCode do jazyka symbolických adries daného procesora. Predpokladá sa, že každý operand je už umiestnený v niektorom z registrov. Program vykonávaný počas generovania kódu sa píše do súborov `mcs51/gen.h` a `mcs51/gen.c`. Vstupom je teda postupnosť iCode inštrukcií. Tie sa postupne prechádzajú a pre každú inštrukciu sa podľa iCode identifikátora zavolá príslušný podprogram na vygenerovanie cieľového kódu. Preto je nevyhnutné, aby každý príkaz vnútornej formy bol popísaný ekvivalentom v jazyku symbolických adries. Príkazy môžu byť nahradené jedinou inštrukciou, ale aj celou postupnosťou inštrukcií. Princíp generovania ilustruje nasledujúci príklad.

Výpis 2.1: Prevod vnútornej formy na symbolické inštrukcie

```
// prehladavanie linearneho zoznamu
for (ic = lic; ic; ic = ic->next)
  {
  // volanie podprogramu podla iCode identifikatora
  switch (ic->op)
    {
    case '!':
      genNot(ic);
      break;

    case '~':
      genCpl(ic);
      break;

    case UNARYMINUS:
      genUminus(ic);
      break;
    }
  }
```

Výpis 2.2: Podprogram pre vygenerovanie inštrukcie

```
static void genCpl(iCode *ic)
  {
  // zistenie velkosti operandu
  int size = AOP_SIZE (IC_RESULT (ic));
  // vygenerovanie pre kazdy jeden bajt
  while (size--)
    {
    // premiestnenie operandu do akumulátora
    MOVA (aopGet (IC_LEFT (ic), offset, FALSE, FALSE));
    // vygenerovanie kodu v jazyku symbolickych adries
    emitcode ("cpl", "a");
    // ulozenie vysledku
    aopPut (IC_RESULT (ic), "a", offset++);
    }
  }
```

2.1.4 Peephole optimalizácia

Prekladač SDCC využíva pre peephole optimalizácie mechanizmus, ktorý vyhľadáva určité úseky kódov, ktoré sa zhodujú s popísanými vzormi a tieto úseky nahrádza optimálnejším kódom. Dochádza tak ku eliminácii redundantných operácií. Keďže tieto optimalizácie sú strojovo závislé, pravidlá pre ich mechanizmus sú popísané zvlášť pre každú cieľovú architektúru. Sada pravidiel, ktoré sú štandardne použité je súčasťou skompilovaného prekladača. Je však možné aplikovať aj ďalšie optimalizácie, a to pri kompilácii s parametrom `--peep-file <filename>`, kde `<filename>` predstavuje názov súboru s popísanými pravidlami. Popis pravidiel pre peephole optimalizácie je ukázaný na nasledujúcom príklade, kde je eliminovaný zbytočný presun.

Výpis 2.3: Príklad popisu pravidiel peephole optimalizácií

```
replace
  {
  mov %1, a
  mov a, %1
  }
by
  {
  mov %1, a
  }
```

Toto pravidlo zabezpečí nahradenie inštrukcií:

```
mov r1, a
mov a, r1
```

jedinou inštrukciou:

```
mov r1, a
```

Pri vykonávaní optimalizácií budú v poradí aplikované všetky pravidlá. Ak sa však vykoná optimalizácia, ktorá má za slovom `replace` uvedené slovo `restart`, optimalizácie začnú prebiehať od začiatku. Táto funkcia by sa mala z dôvodu zachovania rýchlosti prekladu využívať iba v prípadoch, kedy by vykonaná optimalizácia mohla umožniť vykonanie ďalšej, prípadne rovnakej optimalizácie [13].

2.1.5 Registrácia vytvoreného portu

Napokon, aby prekladač vedel na globálnej úrovni, že došlo k vytvoreniu novej cieľovej architektúry, je nutné v príslušných súboroch port zaregistrovať. Najjednoduchším spôsobom ako zistiť, ktoré súbory je potrebné upraviť, je vyhľadanie názvu už podporovaného portu naprieč všetkými súbormi. Jedná sa však predovšetkým o súbory `src/port.h`, `src/SDCCmain.c`, `src/configure` a `src/configure.in`.

Vykonaním týchto úprav je pri kompilácii vytvorený port štandardne povolený a pri konfigurácii je automaticky vytvorený `Makefile` súbor. Následne je možné tento port používať pri kompilácii s parametrom `-mmcs51`.

2.2 Prekladač LLVM

K tomu, aby prekladač vedel vygenerovať kód pre konkrétnu cieľovú architektúru z vnútornej formy LLVM, je potrebné vykonať niekoľko krokov, ktoré budú popísané v nasledujúcom texte. Informácie v tejto časti sú prevažne čerpané z [5], [8] a [28].

Na začiatok je potrebné vytvoriť podpriečinok, ktorý bude obsahovať všetky zdrojové kódy backend časti prekladača. Aby bol nový priečinok a teda nová cieľová architektúra zahrnutá do celého prekladového systému LLVM, je nutné upraviť zostavovacie súboru programu `CMake`.

Najdôležitejším rozhraním medzi cieľovou architektúrou a celým prekladačom sú triedy `TargetMachine` a `DataLayout`. Tie musia byť pre správnu funkciu definované. Trieda `TargetMachine` poskytuje virtuálne metódy, ktoré sa používajú na prístup k implementáciam rôznych tried popisu vlastností cieľovej architektúry (napr. `getInstrInfo, getRegisterInfo, getFrameInfo` atď.). Trieda `DataLayout` určuje aké dátové typy sú natívne podporované a ako sú usporiadané v pamäti. Taktiež udáva typ endianity a veľkosť ukazovateľov. Je to jediná trieda ktorá musí byť

implementovaná, avšak pre plnohodnotné využitie, je potrebné implementovať aj ďalšie rozhrania.

Pretože cieľová architektúra pracuje väčšinou s hodnotami uloženými v registroch, generátoru kódu je potrebné poskutnúť čo najviac informácií o registrovej sade, aby registre boli využité čo najviac efektívne. Medzi najdôležitejšie informácie patrí názov registra a dátový typ. Skupiny registrov, ako napríklad registre pre celočíselné hodnoty alebo hodnoty s pohyblivou desatinnou čiarkou, s ktorými sa pracuje rovnako sa združujú do spoločných tried. Týmto sa umožní prekladaču vybrať hociktorý register z danej skupiny počas fázy alokácie registrov. Aby bolo možné presúvať hodnoty medzi registrami je potrebné popísať ako sa tieto operácie majú vykonávať.

Inštrukčná sada procesora sa líši podľa rôznych funkcií prítomných v architektúre. Pre každú inštrukciu musia byť definované operandy, vzor, podľa ktorého sa má nahradiť inštrukcia vnútornej formy LLVM. V prípade, že výstupom má byť program v jazyku symbolických adries, je potrebné definovať aj reťazec pre výpis. Jedna inštrukcia môže pracovať s rôznymi druhmi operandov a môže využívať rôzne adresovacie módy. Inštrukciu je však nutné popísať zvlášť pre každý typ operandov. Pre zaručenie čo najlepšej kvality výstupného kódu je dobré popísať aj rôzne ďalšie informácie o inštrukciách ako napríklad, či je možné zameniť poradie operandov (komutatívnosť), či sa implicitne používa nejaký register, či sa jedná o operáciu skoku a mnohé ďalšie. Ak má byť výstupom generátora kódu objektový kód, je potrebné definovať aj binárne kódovanie inštrukcie.

Volacie konvencie definujú pravidlá pre volanie funkcií a predávanie parametrov a návratových hodnôt. Popisujú teda spôsob vzájomnej komunikácie volajúcej funkcie s funkciou volanou. Medzi tieto pravidlá patrí:

- poradie a spôsob umiestnenia jednotlivých argumentov (do registrov alebo zásobníka)
- zoznam registrov, ktorých obsah je volaná funkcia povinná zachovať
- spôsob uloženia návratovej hodnoty
- ktorá z funkcií má na starosť odstránenie argumentov zo zásobníka

Popis volacích konvencií spočíva v uvedení sekvencie podmienok a pravidiel, ktoré sú následne aplikované v uvedenom poradí.

Aby mohol byť výsledný kód v jazyku symbolických adries, je nutné implementovať triedu `AsmPrinter`. Pre každý zakladný blok programu je vypísaná návesť a jednotlivé inštrukcie. Na výpis inštrukcií slúži funkcia `printInstruction`, ktorá je automaticky vytvorená z popisu inštrukčnej sady. Táto funkcia následne volá funkciu `printOperand` pre výpis jednotlivých operandov podľa ich typu.

Pretože u mnohých architektúr existuje veľké množstvo ich variánt a taktiež niektoré hardvérové prvky môžu byť voliteľné ako napríklad jednotka pre prácu

s číslami s pohyblivou desatinnou čiarkou, je možné voliteľne pridať informácie o podtypoch architektúry. Výsledkom je, že napríklad niektoré inštrukcie budú v niektorých podtypoch použiteľné a v iných nie.

2.2.1 Nástroj TableGen

Veľmi nápomocnou súčasťou prekladového systému LLVM je nástroj TableGen. Pretože popis cieľovej architektúry je vo všeobecnosti veľmi rozsiahly, úlohou nástroja TableGen je zjednodušenie tohto popisu. Je špecificky navrhnutý tak, aby bolo možné jednotlivé záznamy jednoducho upravovať, združovať a spoločné vlastnosti, aby bolo možné dediť. To všetko redukuje množstvo duplicitných popisov a znižuje sa pravdepodobnosť chyby.

S nástrojom sa pracuje nasledovne. V zdrojových súboroch s príponou `.td` sú popísané základné informácie o cieľovej architektúre, registrová sada, inštrukčná sada, volacie konvencie a ďalšie. Samotné TableGen súbory nemajú žiaden význam, preto sú z nich programom `llvm-tablegen` vytvorené príslušné hlavičkové a zdrojové súbory v jazyku C++. Pre použitie vygenerovaných súborov ich je potrebné zahrnúť pri popise architektúry.

TableGen má syntax, ktorá je založená na C++ šablónach. Okrem toho TableGen zavádza niektoré koncepty automatizácie, ako sú napríklad multitriedy, či príkaz `foreach` alebo `let`.

Súbory TableGen pozostávajú z dvoch hlavných častí: z tried a definícií, pričom obe sú považované za záznamy.

Záznamy majú unikátny názov, zoznam hodnôt a zoznam rodičovských tried. Zoznam hodnôt je hlavný prvok, ktorý uchováva informácie. Ich interpretácia je ponechaná na danú cieľovú architektúru.

Definície predstavujú konkrétnu inštanciu vopred definovaných tried. Spravidla nemajú žiadne nedefinované hodnoty a sú označené kľúčovým slovom `def`. Výpis 2.4 zobrazuje príklad vytvorenia definície s inicializáciou niektorých hodnôt.

Výpis 2.4: Príklad zápisu definície pomocou nástroja TableGen

```
def GPR8: RegisterClass<"HCS08", [i8], 8, (add A)>;
```

Triedy sú abstraktné záznamy, ktoré sa používajú na vytváranie a popis iných záznamov. Triedy je možné dediť a predávať im parametre. Atribúty triedy je možné ponechať nedefinované alebo môžu byť definované až v niektorých ďalších podtriedach. Ak bol niektorý atribút definovaný, nasledujúcou definíciou sa to prepíše [8].

Výpis 2.5: Príklad zápisu triedy pomocou nástroja TableGen

```
class HCS08Reg<string name,
               list<Register> subregs = [],
               list<string> altNames = []>
  : RegisterWithSubRegs<name, subregs>
{
  let Namespace = "HCS08";
  let SubRegs = subregs;
  let AltNames = altNames;
}
```

Multitriedy sú skupiny abstraktných záznamov, z ktorých sa vytvorí viacero inštancií naraz. Napríklad jednou definíciou je možné vytvoriť hneď niekoľko variánt jednej inštrukcie.

Výpis 2.6: Príklad zápisu multitriedy pomocou nástroja TableGen

```
let Defs = [A], Uses = [A] in
multiclass Arith<string OpcStr, SDPatternOperator OpNode> {
  def imm : InstHCS08<2,
                      (outs),
                      (ins i8imm:$src),
                      !strconcat(OpcStr, "␣#$src"),
                      [(set A, (OpNode A, imm:$src))]>;
  def dir : InstHCS08<2,
                      (outs),
                      (ins i8imm:$addr),
                      !strconcat(OpcStr, "$$src"),
                      [(set A, (OpNode A, (load imm:$addr)))]>;
}
```

Výpis 2.7: Príklad vytvorenia inštancií multitriedy

```
defm ADD    : Arith<"add", add>;
defm ORA    : Arith<"ora", or>;
```

Podrobnejšie informácie je možné nájsť v [27].

2.3 Voľba prekladača

SDCC je pomerne jednoduchý prekladač jazyka C navrhnutý najmä pre malé 8-bitové mikroprocesory. Ku štandardu jazyka navyše pridáva rozšírenie v podobe

nových pamäťových oblastí. Spočiatku mal prekladač veľmi významný prínos, avšak v súčasnej dobe sa už nepracuje na jeho vývoji tak intenzívne ako v jeho začiatkoch. Dokumentácia SDCC poskytuje len minimum informácií o tvorbe novej backend časti. Preto pre implementáciu je nutné študovať zdrojové kódy iných cieľových architektúr.

LLVM je veľmi rozsiahly projekt pozostávajúci z množstva modulov, ktoré sa dajú viackrát využiť. Ako frontend používa prekladač Clang, ktorý nepodporuje len jazyk C, ale aj ďalšie jazyky ako C++, Objective-C. Viacero veľkých firiem ako napríklad Apple alebo Microsoft používa tento prekladový systém a investuje do jeho vývoja. Vo svete si získava čím ďalej tým väčšiu popularitu a to nielen kvôli vysokej kvalite generovaného kódu ale aj vďaka rýchlosti prekladu. Celý projekt je veľmi dobre zdokumentovaný a poskytuje viacero návodov pre jeho úpravu a rozširovanie funkcionality.

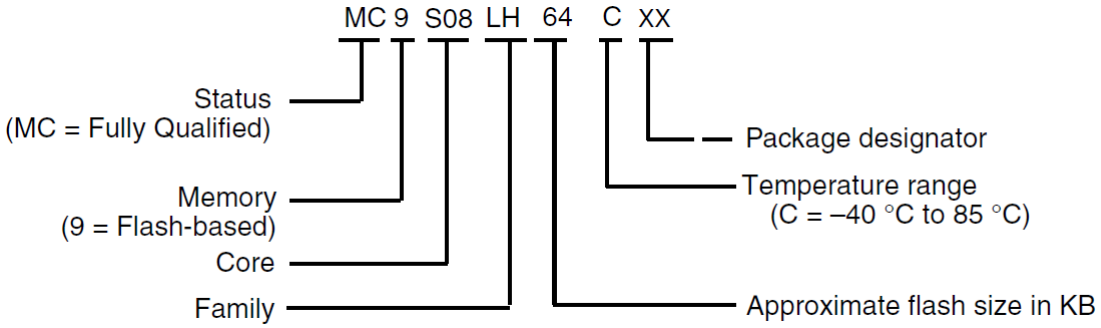
V ďalšej časti práce a teda v samotnej implementácii bude využitý prekladač LLVM práve kvôli uvedeným výhodam.

3 Výber cieľovej architektúry

V tejto práci bude ako cieľová architektúra využitý mikrokontrolér z rodiny HCS08 od výrobcu NXP (kedysi Motorola, Freescale), konkrétne model MC9S08LH64. K tomuto rozhodnutiu došlo z dôvodu viacerých skúseností nadobudnutých v rámci bakalárskeho predmetu Mikroprocesory. V tejto kapitole sú uvedené informácie z [3], [29], [30] a [31].

3.1 Mikrokontrolér HCS08

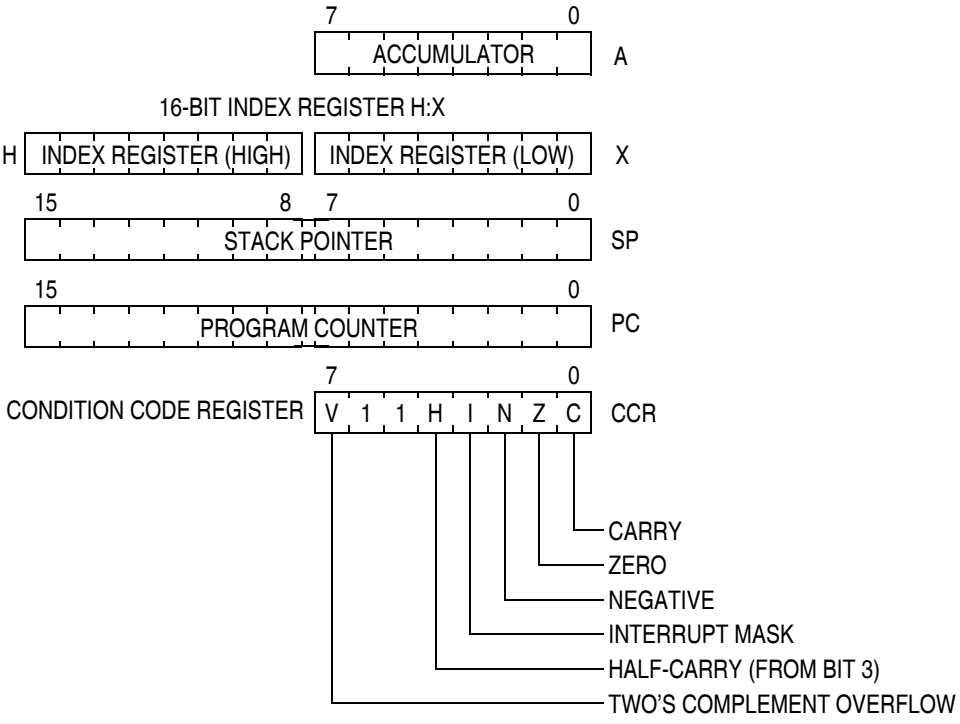
HCS08 je 8-bitový mikrokontrolér, ktorý je novšou generáciou mikrokontroléra HC08. Medzi hlavné vylepšenia patria rýchlejšie inštrukcie debuggera. S pôvodným HC08 je však spätne kompatibilný, čiže každý program bude rovnako funkčný aj na HCS08.



Obr. 3.1: Značenie mikrokontrolérov HCS08 [30]

Mikrokontroléry z tejto rodiny sa vyznačujú rýchlym prístupom do pamäte, malým počtom registrov a komplexnou inštrukčnou sadou CISC. Skladajú sa z jadra HCS08, rozhrania BDC (Background Debug Controller), ktoré slúži na debuggovanie a nahrávanie firmvéru. Nechýba ani podpora až 32 možných zdrojov prerušenia, dekódovanie adresy na úrovni čipu, či hardvérová násobička a delička. Maximálna frekvencia je 40MHz a rýchlosť zbernice najviac 20MHz.

Programátorský model na obr. 3.2 zobrazuje špeciálne registre. Medzi ne patrí akumulátor A používaný pre uschovanie operandov niektorých inštrukcií a tiež ich výsledkov. Registrový pár H:X slúži predovšetkým na indexové adresovanie, ale okrem toho, ho využívajú aj niektoré inštrukcie. Register SP je využitý pre prácu so zásobníkom a ukazuje na jeho vrchol, čiže na prvú voľnú pamäťovú bunku. Po resete je nastavený na vrchol zásobníka a to na hodnotu `0x00ff`. Programový čítač PC ukazuje na adresu ďalšej inštrukcie, ktorá sa bude vykonávať. Po resete obsahuje adresu, ktorá je uložená v resetovacom vektore. Stavový register CCR uchováva informáciu o globálnom povolení prerušení a výsledku porovnávanaia.



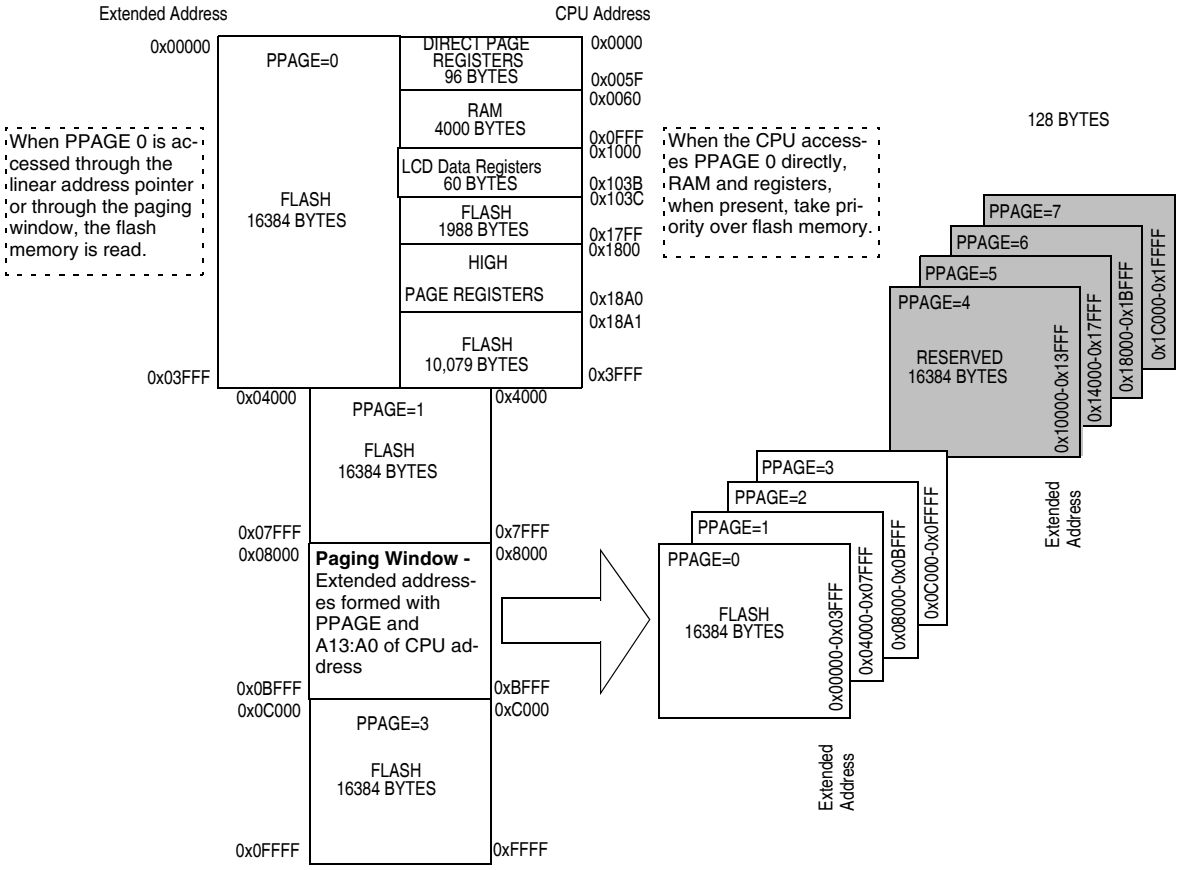
Obr. 3.2: Programovací model HCS08 [31]

Vybraný mikrokontrolér poskytuje 4KB operačnej pamäte RAM a 64KB programovej pamäte FLASH. Pre prístup k pamäťovým miestam v operačnej pamäti s jednobajtovými adresami je možné použiť rýchlejšie inštrukcie. Preto je vhodné do tejto oblasti umiestňovať dáta, s ktorými sa pracuje veľmi často. Pamäťový model na obr. 3.3 zobrazuje mapu pamäte.

3.1.1 Periférie

Obr. 3.4 zobrazuje blokovú schému mikrokontroléra MC9S08LH64. Medzi jeho hlavné moduly patrí:

- **LCD** – rozhranie pre LCD displej o veľkosti 8x36 alebo 4x40 s možnosťou regulácie kontrastu
- **ADC** – AD prevodník so 16-bitovým rozlíšením, jedným diferenciálnym vstupom a ôsmimi obyčajnými vstupmi, s časom prevodu 2,5 µs, hardvérovým priemerovaním a teplotným senzorom
- **IIC** – modul komunikácie I^2C s rýchlosťou až 100 kbps, multi-master komunikácia, 10-bitové adresovanie
- **ACMP** – analógový komparátor s konfigurovateľným prerušením pre nábežnú alebo závernú hranu (prípadne oboje), možnosť porovnania so vstavaným re-



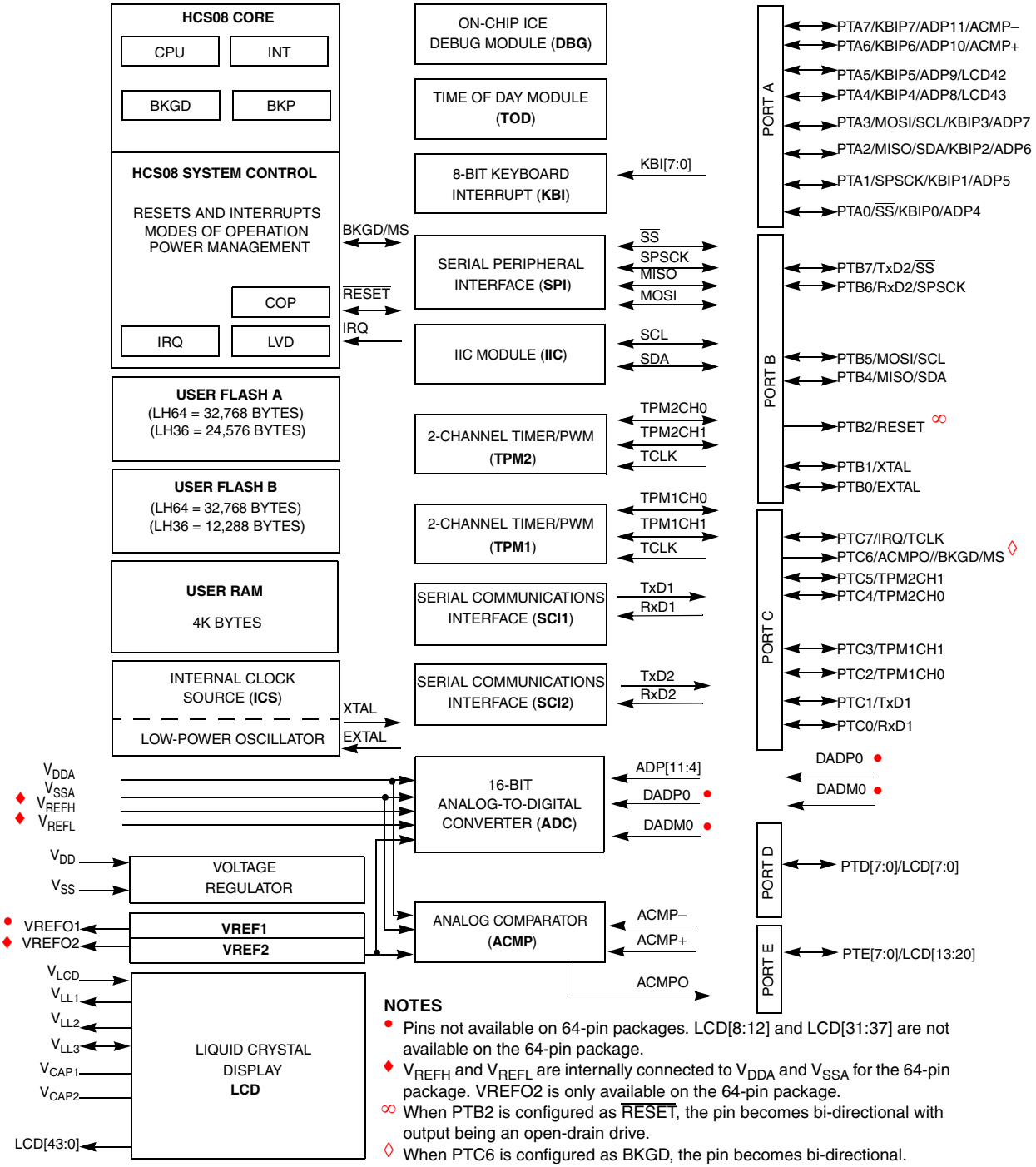
Obr. 3.3: Pamäťový model MC9S08LH64 [31]

ferenčným napätím

- **SCIx** – dva moduly pre plne duplexnú asynchrónnu sériovú komunikáciu UART, možnosť ukončenia režimu spánku na aktívnu hranu
- **SPI** – modul sériového periférneho rozhrania s plne duplexnou komunikáciou
- **TPMx** – dva dvojkanálové moduly časovača s funkciami input capture, output compare a možnosťou PWM výstupu
- **TOD** – modul denného času, štvrťsekundové počítadlo, externý zdroj hodín
- **VREFx** – dvojica vstupov referenčného napätia
- **KBI** – modul prerušenia od klávesnice, poskytuje až osem maskovateľných zdrojov prerušenia
- **PORTA až PORTE** – univerzálne vstupno-výstupné rozhranie

3.1.2 Adresovacie módy

Periférie sú pamäťovo mapované a preto pre prístup k nim sa používajú rovnaké inštrukcie ako pre prístup k dátam uloženým v pamäti. Adresovacie módy určujú



Obr. 3.4: Bloková schéma mikrokontroléra zo série MC9S08LH64 [30]

spôsob, akým procesor pristupuje k operandom a dátam. Niektoré inštrukcie poskytujú využitie niekoľkých rôznych adresovacích módov, zatiaľ čo iné umožňujú použiť len jeden spôsob. HCS08 podporuje tieto adresovacie módy:

- **INH** – registrové adresovanie (Inherent Addressing Mode) – operand sa nachádza v registri, pričom samotná inštrukcia určuje, ktorý register sa použije
- **REL** – relatívne adresovanie (Relative Addressing Mode) – operand inštrukcie predstavuje znamienkový 8-bitový posun pre inštrukcie skokov
- **IMM** – (Immediate Addressing Mode) – hodnota operandu je priamo súčasťou inštrukcie v podobe konštanty
- **DIR** – (Direct Addressing Mode) – hodnotou operandu je adresa, na ktorej sa nachádza samotný operand, používa sa pre rýchlejšie adresovanie pamäťového priestoru s jednobajtovou adresou (`0x0000 - 0x00ff`)
- **EXT** – (Extended Addressing Mode) – hodnotou operandu je adresa tak ako pri **DIR**, umožňuje však adresovať celý adresový priestor (`0x0000 - 0xffff`)
- **Indexové adresovanie** – má niekoľko variant, päť z nich využíva registrový pár H:X a dve z nich využívajú register SP
  - **IX** – (Indexed, No Offset) – adresa operandu je v registrovom páre H:X
  - **IX+** – (Indexed, No Offset with Post Increment) – adresa operandu je v registrovom páre H:X, po načítaní operandu je H:X inkrementované
  - **IX1** – (Indexed, 8-Bit Offset) – efektívna adresa operandu je vypočítaná ako súčet hodnoty v H:X a 8-bitového posunu, ktorý je súčasťou inštrukcie
  - **IX1+** – (Indexed, 8-Bit Offset with Post Increment) – rovnako ako **IX1**, ale po načítaní operandu, je registrový pár H:X inkrementovaný
  - **IX2** – (Indexed, 16-Bit Offset) – rovnako ako **IX1**, ale umožňuje až 16-bitový posun voči adrese v H:X
  - **SP1** – (SP-Relative, 8-Bit Offset) – efektívna adresa operandu je vypočítaná ako súčet hodnoty v registri SP a 8-bitového posunu
  - **SP2** – (SP-Relative, 16-Bit Offset) – rovnako ako **SP1**, ale umožňuje až 16-bitový posun voči adrese v SP

3.2 Voľba referenčného prekladača

Výrobca NXP poskytuje pre vývoj softvéru program CodeWarrior s vlastným prekladačom pre jazyk C, ktorý bude v rámci testov použitý ako referenčný. Túto rodinu mikrokontrolérov taktiež podporuje vyššie spomenutý prekladač SDCC. Ten bude tiež použitý na porovnanie kvality výsledného kódu.

3.2.1 Volacie konvencie prekladača CodeWarrior

Informácie v tejto časti sú prevzaté z [33].

Predávanie argumentov

HCS08 využíva volacie konvencie jazyka C pre všetky funkcie. Volajúca funkcia vkladá argumenty zľava doprava. Po ukončení funkcie, volajúca funkcia odstráni parametre zo zásobníka. Ak funkcia obsahuje pevný počet argumentov a veľkosť posledného parametra sú dva bajty, posledný parameter je funkcii odovzdaný v registrovom páre H:X.

Ak funkcia obsahuje pevný počet argumentov a veľkosť posledného parametra je jeden bajt, posledný parameter je funkcii odovzdaný v registri A. Ak je veľkosť nasledujúceho parametra tiež jeden bajt, tento parameter je odovzdaný funkcii v registri X. Ak je ale veľkosť tohto nasledujúceho parametra dva bajty, parameter je odovzdaný v registrovom páre H:X.

Predávanie návratových hodnôt

HCS08 navracia výsledky funkcií v registroch. Použitý register závisí od konkrétneho návratového typu, ako to je uvedené v tabuľke 3.1. Ak funkcia vracia objekt vačší ako dva bajty, adresa na tento objekt je uložená v registrovom páre H:X.

Tab. 3.1: Predávanie návratových hodnôt [33]

Návratový typ	**Register**
char (signed/unsigned)	A
int (signed/unsigned)	H:X
ukazovatele alebo polia	H:X
ukazovatele na funkcie	H:X

Prológ funkcie

Prológ funkcie má za úlohu rezervovať miesto pre lokálne premenné.

Výpis 3.1: Vkladanie prológu funkcie

```
PSHA      ; iba ak sa v registri nachádza argument
PSHX      ; iba ak sa v registri nachádza argument
AIS #(-s) ; rezervované miesto pre lokálne premenné a spill kód
```

Epilóg funkcie

Epilóg funkcie má za úlohu odstránenie lokálnych premenných zo zásobníka a návrat do volajúcej funkcie.

Výpis 3.2: Vkladanie epilógu funkcie

```
AIS #(t) ; odstránenie miesta pre lokálne premenné,
         ; vrátane pripadných argumentov v registroch
RTS      ; návrat do volajúcej funkcie
```

4 Implementácia vlastnej backend časti

Táto časť práce popisuje postup samotnej implementácie backend časti prekladača pre vybraný mikrokontrolér z rodiny HCS08. Implementácia vychádza z naštudovania viacerých návodov a to najmä [7], [28], [8] ako aj zo zdrojových kódov už implementovaných backend častí. Všetky zdrojové kódy sú uvedené v prílohe B.

Aby vôbec bolo možné backend časť skompilovať, je nutné implementovať niekoľko tried, ktorých kód je príliš rozsiahly. Preto bol na začiatok skopírovaný všetok zdrojový kód pre procesor `ARC` a postupne bol upravovaný, tak aby popisoval cieľovú architektúru HCS08 a generoval spustiteľný kód.

4.1 Registrácia cieľovej architektúry

Zaregistrovanie cieľovej architektúry slúži k tomu, aby sa stala súčasťou celého prekladača a bolo ju možné využívať naprieč všetkými nástrojmi LLVM. Docieliť sa to dá inicializáciou v súbore `TargetInfo/HCS08TargetInfo.cpp`:

Výpis 4.1: Registrácia cieľovej architektúry

```
extern "C" void LLVMInitializeHCS08TargetInfo() {
    RegisterTarget<Triple::hcs08> X(getTheHCS08Target(),
        "hcs08", "HCS08", "HCS08");
}
```

4.2 Popis cieľovej architektúry

Popis pozostáva najmä zo všeobecných informácií o cieľovej architektúre, popisu registrovej a inštrukčnej sady a z popisu volacích konvencií. Väčšinu informácii je možné vyjadriť za pomoci nástroja TableGen, ktorý bol popísaný v časti 2.2.1.

4.2.1 Všeobecné informácie o procesore

Trieda `HCS08TargetMachine` tvorí hlavné rozhranie medzi prekladačom LLVM a vytvorenou backend časťou. Združuje pod sebou všetky časti generátora kódu. Trieda `DataLayout` vo výpise 4.2 popisuje aké dátové typy sú architektúrou podporované a ako sú uložené v pamäti.

Pomlčky oddeľujú jednotlivé časti reťazca. Veľké písmeno `"E"` značí, že dáta sú ukladané spôsobom big-endian. Pokračuje sa určením veľkosti ukazovateľa a jeho

zarovnanie. Následne sú definované veľkosti jednotlivých dátových typov a ich zarovnanie v pamäti. Boolovská premenná je tzv. povýšená na 8-bitovú premennú. Dátový typ i16 a i32 sú rozdelené na jednotlivé bajty a tak sú uložené do pamäte.

Výpis 4.2: Popis umiestenia dát v pamäti

```
DataLayout("E-p:16:8-i1:8-i8:8-i16:8-i32:8");
```

4.2.2 Popis registrovej sady

Aby bolo možné pracovať s registrami programovacieho modelu, je ich potrebné najprv popísať pomocou triedy `HCS08RegisterInfo`. Najdôležitejšou informáciou je názov registra a určenie s akým dátovým typom vie register pracovať.

Registre sú popísané ako inštancie triedy HCS08Reg pomocou nástroja TableGen v súbore `HCS08RegisterInfo.td`. Definícia triedy je uvedená vo výpise 4.3. Príklad definície registra zobrazuje výpis 4.4. Premenná `DwarfRegNum` určuje pod akým číslom je register používaný v ostatných zdrojových súboroch.

Výpis 4.3: Trieda HCS08Reg popisujúca registre

```
class HCS08Reg<string name,
               list<Register> subregs = [],
               list<string> altNames = []>
  : RegisterWithSubRegs<name, subregs>
{
  let Namespace = "HCS08";
  let SubRegs = subregs;
  let AltNames = altNames;
}
```

Výpis 4.4: Definície registra pomocou nástroja TableGen

```
def A : HCS08Reg<"A">, DwarfRegNum<[0]>;
```

Registre s rovnakými vlastnosťami, predovšetkým ich rovnakou veľkosťou, sa zlučujú do tried, aby sa zjednodušila práca s nimi. Pre takúto triedu sa spoločne určí veľkosť registrov, prislúchajúci dátový typ a zoznam registrov patriacich do tejto triedy. Príklad zlučovania registrov do tried zobrazuje výpis 4.5.

Pomocou triedy HCS08Reg je tiež možné popísať aj skutočnosť, že mikroprocesor HCS08 používa pre niektoré inštrukcie dvojicu registrov H:X, hoci ide o dve rôzne

Výpis 4.5: Zlučovanie registrov do tried

```
def GPR8: RegisterClass<"HCS08", [i8], 8,
  (add A, H, X, PCH, PCL, SPH, SPL)>;
```

registre. Príklad je uvedený vo výpise 4.6. V tomto prípade voliteľná premenná `subregs` popisuje, z ktorých registrov je pár zložený a ich poradie určuje, do ktorého sa ukladá vrchný a do ktorého spodný bajt premennej.

Výpis 4.6: Definícia registrových párov

```
let SubRegIndices = [sub_lo, sub_hi],
CoveredBySubRegs = 1 in
{
def HX : HCS08Reg<"H:X", [X, H]>, DwarfRegNum<[1]>;
def PC : HCS08Reg<"PC", [PCL, PCH]>, DwarfRegNum<[3]>;
def SP : HCS08Reg<"SP", [SPL, SPH]>, DwarfRegNum<[5]>;
def XA : HCS08Reg<"X:A", [A, X]>, DwarfRegNum<[0]>;
}
```

4.2.3 Popis inštrukčnej sady

Základná trieda, ktorá všeobecne popisuje všetky inštrukcie sa definuje v súbore `HCS08InstrFormats.td`. Rovnako sú v tomto súbore vytvorené triedy pre rôzne typy operandov, ako napríklad pre okamžité hodnoty alebo pre načítanie operandu z adresy, či použitie rôznych ďalších adresovacích módov. Samotné inštrukcie sa popisujú v súbore `HCS08InstrInfo.td`.

Všetky inštrukcie sú inštanciami triedy `InstHCS08`, ktorá je odvodená z hlavnej triedy `Instruction`. Výpis 4.7 zobrazuje definovanie inštrukcie sčítavania, v adresovacom móde IMM, takže úlohou inštrukcie je sčítať hodnotu operandu s hodnotou v registri A, čiže v akumulátore, a výsledok uložiť do akumulátora. Vzor tejto funkcie je `[(set A, (add A, imm:$src))]`. Zoznam vstupných operandov `ins` obsahuje jediný operand, čiže číslo, ktoré sa má pripočítať k akumulátoru. Zoznam výsledkov `outs` je prázdny, pretože je pevne určené, že výsledok sa nachádza v akumulátore a nie je možné ho namapovať na inú premennú. Po nájdení zadaného vzoru v programe reprezentovanom DAG grafom je nahradená za strojovú inštrukciu, ktorá je v jazyku symbolických adries reprezentovaná príkazom `"add #$src"`, kde za `$src` je dosadená konkrétna hodnota operandu. Parametre `Defs` a `Uses` určujú, že inštrukcia implicitne ukladá výsledok do registra `A` a tiež používa ako operand tento register.

Pre zjednodušenie zápisu inštrukcií sú použité multitriedy, ktoré boli popísané v časti 2.2.1.

Výpis 4.7: Príklad definovania inštrukcie

```
let Defs = [A], Uses = [A] in
def ADDimm : InstHCS08<
    (outs),
    (ins i8imm:$src),
    "add␣#$src",
    [(set A, (add A, imm:$src))]>;
```

Zoznam všetkých doposiaľ podporovaných inštrukcií je možné nájsť v prílohe A.

4.2.4 Popis volacích konvencií

Volacie konvencie sa popisujú v súbore `HCS08CallingConv.td`. Výpis 4.8 zobrazuje časť kódu, kde je popísané predávanie argumentov funkciám. Pretože mikroprocesory z rodiny HCS08 sú 8-bitové, je zaručené, že 1-bitové hodnoty budú prevedené na 8-bitové. Ak veľkosť argumentu je 16 bitov, je vložený do registrového páru `H:X`. Ak sú argumenty 8-bitové vložia sa postupne do registrov `A`, `X`. Ak je argumentov viac, sú vložené na zásobník.

Volacie konvencie sú implementované, tak ako boli popísané v časti 3.2.1, aby bolo možné vygenerovaný kód prepojiť s kódom v prekladači CodeWarrior, ktorý bol zvolený ako referenčný.

Výpis 4.8: Trieda popisujúca volacie konvencie

```
def CC_HCS08 : CallingConv<[
  CCIfType<[i1], CCPromoteToType<i8>>,

  CCIfType<[i16], CCAssignToReg<[HX]>>,
  CCIfType<[i8], CCAssignToReg<[A, X]>>,

  CCIfType<[i8], CCAssignToStack<1, 1>>,
  CCIfType<[i16], CCAssignToStack<2, 1>>
]>;
```

4.3 Výpis kódu

Časť prekladača, ktorá je zodpovedná za výpis kódu sa programuje ako samotný modul `MCTargetDesc`, pretože nie je povinná. Je teda potrebné tento modul v prekladači zaregistrovať podobne ako celú cieľovú architektúru a napísať triedu `HCS08InstPrinter`, ktorá je rozhraním medzi prekladačom a modulom.

Pre výpis kódu v jazyku symbolických adries sú používané reťazce definované v súbore `HCS08InstrInfo.td`. V súbore `HCS08InstPrinter.cpp` sú uvedené funkcie, ktoré určujú ako majú byť vypísané jednotlivé inštrukcie, rôzne druhy operandov, náveste, či celé bloky programu.

Ak by bolo potrebné, aby bol výstupný kód generovaný v tvare objektového kódu, bolo by potrebné pri popise inštrukcií uviesť ich kódovanie. Ďalej by bolo potrebné napísať funkcie, ktoré by kódovali operandy operácii. Veľmi dôležité by bolo taktiež napísať funkcie na počítanie absolútnych cieľov skokov, čo pri generovaní kódu v jazyku symbolických adries potrebné nie je.

4.4 Preklad s novovytvorenou backend časťou

Aby bolo možné celý prekladový systém skompilovať spolu s backend časťou HCS08, boli upravené príslušné súbory `CMakeLists.txt` a `LLVMBuild.txt` podľa príkladu ostatných cieľových architektúr. Celý projekt bol nakonfigurovaný príkazom `cmake` spolu s použitím parametra `-DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD="HCS08"`.

Navyše je potrebné upraviť súbory `llvm/include/llvm/ADT/Triple.h` a `llvm/lib/Support/Triple.cpp` na všetkých miestach, kde sú spomenuté ostatné architektúry.

V rámci tejto práce bola upravená iba backend časť prekladača, preto využitie frontend časti Clang s architektúrou HCS08 nie je možné.

5 Testovanie a demonštrácia backend časti

V tejto kapitole sa nachádza niekoľko testovacích programov, ktoré je možné v súčasnej dobe preložiť pomocou backend časti prekladača, ktorého implementácia bola hlavným cieľom tejto práce. Všetky zdrojové kódy sú uvedené v prílohe C. Každý program je uvedený v zdrojovom jazyku C a následne je uvedený preklad do jazyka symbolických adries pomocou:

- referenčného prekladača vo vývojovom prostredí CodeWarrior
- novovytvorenej backend časti v prekladovom systéme
- a pomocou prekladača

Jednotlivé preklady sú prehľadne zobrazené v tabuľkách, pričom spodný riadok udáva pamäťové nároky daného programu.

Vytvorený backend pre mikrokontrolér z rodiny HCS08 je zaregistrovaný iba v backend časti prekladača. Preto pre skompilovanie programu v jazyku C alebo C++ za použitia frontend časti Clang, je potrebné tento program najprv previesť do vnútornej formy, ktorá nie je závislá od cieľovej architektúry, príkazom:

Výpis 5.1: Príkaz na prevod programu do vnútornej formy

```
clang -cc1 -S -emit-llvm -o filename.ll filename.c
```

Aby bol program optimálny je vhodné použiť parameter `-O3`. Z vnútornej formy je možné previesť program do jazyka symbolických adries príkazom 5.2, pričom parameter `-march` určuje, ktorý backend sa má použiť na generovanie kódu.

Výpis 5.2: Príkaz na prevod vnútornej formy do jazyka symbolických adries

```
llc -march hcs08 -o filename.s -filetype=asm filename.ll
```

Pre kompiláciu pomocou prekladača SDCC bol využitý príkaz 5.3. Prekladač nepodporuje parameter `-O3`, a preto nemohli byť na programe vykonané podobné optimalizácie ako v prekladači LLVM. Prekladač CodeWarrior taktiež nepodporuje parameter `-O3`, ale predvolene vykonáva optimalizácie pre využitie čo najmenšieho miesta v pamäti s parametrom `-Os`.

Výpis 5.3: Príkaz pre kompiláciu prekladačom SDCC

```
sdcc filename.c -S -ms08
```

5.1 Testovací program č. 1

Program 5.4 testuje správnosť vygenerovania jednoduchej funkcie, do ktorej je vkladaný jeden argument. Taktiež sa týmto testuje správnosť voľby registov, ktoré sú určené volacími konvenciami. Aritmetická operácia sčítavania je využitá v adresovacom móde IMM.

Z výsledku prekladu v tabuľke 5.1 je jasné, že takúto jednoduchú funkciu preložili všetky tri prekladače rovnako.

Výpis 5.4: Testovací program č. 1

```
char f1(char x)
  {
  return x + 65;
  }
```

Tab. 5.1: Výsledný preklad programu č. 1

Prekladač CodeWarrior	**Prekladač LLVM**	**Prekladač SDCC**
f1_cw: ADD #65 RTS	f1_llvm: ADD #65 RTS	f1_sdcc: ADD #65 RTS
3 bajty	3 bajty	3 bajty

5.2 Testovací program č. 2

Testovací program 5.5 testuje správnosť generovania nepodmienených skokov. Ten je v tomto príklade docielený použitím nekonečnej slučky. Do programu je schválne pridaná premenná, ktorej ale hodnota nebude nikdy navrátená. To preto, aby bolo možné porovnať vlastnosti použitých prekladačov, čo sa týka optimalizácie.

Z výsledku uvedenom v tabuľke 5.2 je patrné, že s týmto testom si najlepšie poradil prekladač LLVM. Ten však vygeneroval jednu návesť navyše, čo však nie je podstatné, pretože nezaberá žiadnu pamäť programu navyše. Prekladač SDCC navyše vygeneroval inštrukciu `RTS`, ktorá je ale zbytočná, keďže program sa z tejto slučky nemôže dostať von. Prekladač CodeWarrior vygeneroval inštrukcie pre inkrementovanie premennej, čo je však zbytočné, keďže táto premenná nebude nikdy použitá.

Mechanizmus použitej optimalizácie sa po anglicky nazýva *Induction Variable Elimination*, kedy hodnota premennej je indukovaná z vnútra cyklu s cieľom zníženia počtu vykonaných operácií [34].

Výpis 5.5: Testovací program č. 2

```
char f2(char x)
  {
  while(1)
    {
    ++x;
    }

  return x;
  }
```

Tab. 5.2: Výsledný preklad programu č. 2

Prekladač CodeWarrior	**Prekladač LLVM**	**Prekladač SDCC**
f2_cw: PSHA TSX INC ,X BRA f2_cw+1	f2_llvm: .LBB0_1: BRA .LBB0_1	f2_sdcc: BRA f2_sdcc RTS
5 bajtov	2 bajty	3 bajty

5.3 Testovací program č. 3

Posledný testovací program 5.6 slúži na otestovanie správnosti a efektivity výberu inštrukcií cieľovej architektúry. V programe sú použité takmer všetky jednoduché aritmetické a logické operácie. Testovací program je zámerne navrhnutý tak, aby bola docielená čo najjednoduchšia zámena inštrukcií a vďaka tomu aby bolo možné, tak ako pri predošlom teste, sledovať vlastnosti týchto prekladačov.

Z výsledných programov v tabuľke 5.3 nie je možné na prvý pohľad určiť, či všetky preklady generujú rovnaké výsledky. Preto boli tieto programy otestované vo vývojovom prostedí a bolo zistené, že všetky tri preklady generujú správne výsledky.

Už na prvý pohľad je zrejmé, že prekladač LLVM použil polovičné množstvo inštrukcií oproti referenčnému prekladaču vývojového prostredia CodeWarrior. Taktiež program zaberá takmer dvakrát menej pamäte a program je takmer trikrát rýchlejší. Prekladač LLVM priamo zamenil každý príkaz jazyka C za jeho ekvivalent v jazyku symbolických adries. Naproti tomu prekladač CodeWarrior využíva viacero presunov medzi registrami a vo veľkej miere využíva indexovací adresovací mód IX, pričom v tomto prípade je oveľa rýchlejší priamy adresovací mód IMM. Taktiež prekladač CodeWarrior zbytočne odkladá hodnotu v registri `H` na zásobník, pretože v ňom nie je uschovaná žiadna hodnota. Prekladač SDCC generuje tiež viacero inštrukcií, ale prevažne pracuje s adresovacím módom IMM a INH, taktiež využíva aj zásobník. Výsledky sú zobrazené garficky na obrázku 5.1.

Výpis 5.6: Testovací program č. 3

```
char f3(char x)
  {
  char result = x-- << 1;
  result += 42;
  result ^= 13;
  result += 8;
  result &= 254;
  result = -result;
  result = result >> 1;
  ++result;
  result |= 4;
  --result;

  return result;
  }
```

5.4 Demonštračný program

Testovanie bolo vykonané vo vývojovom prostredí CodeWarrior na simulátore daného mikrokontroléra z rodiny HCS08. Demonštračný program je možné nájsť v prílohe D, v priečinku `testovanie_c_asm`.

Hlavičky jednotlivých funkcií sa nachádzajú v priečinku `Project_Headers` v `.h` súboroch a výsledky prekladov zo všetkých troch prekladačov v jazyku symbolických adries sú vložené do `.asm` súborov v priečinku `Sources`.

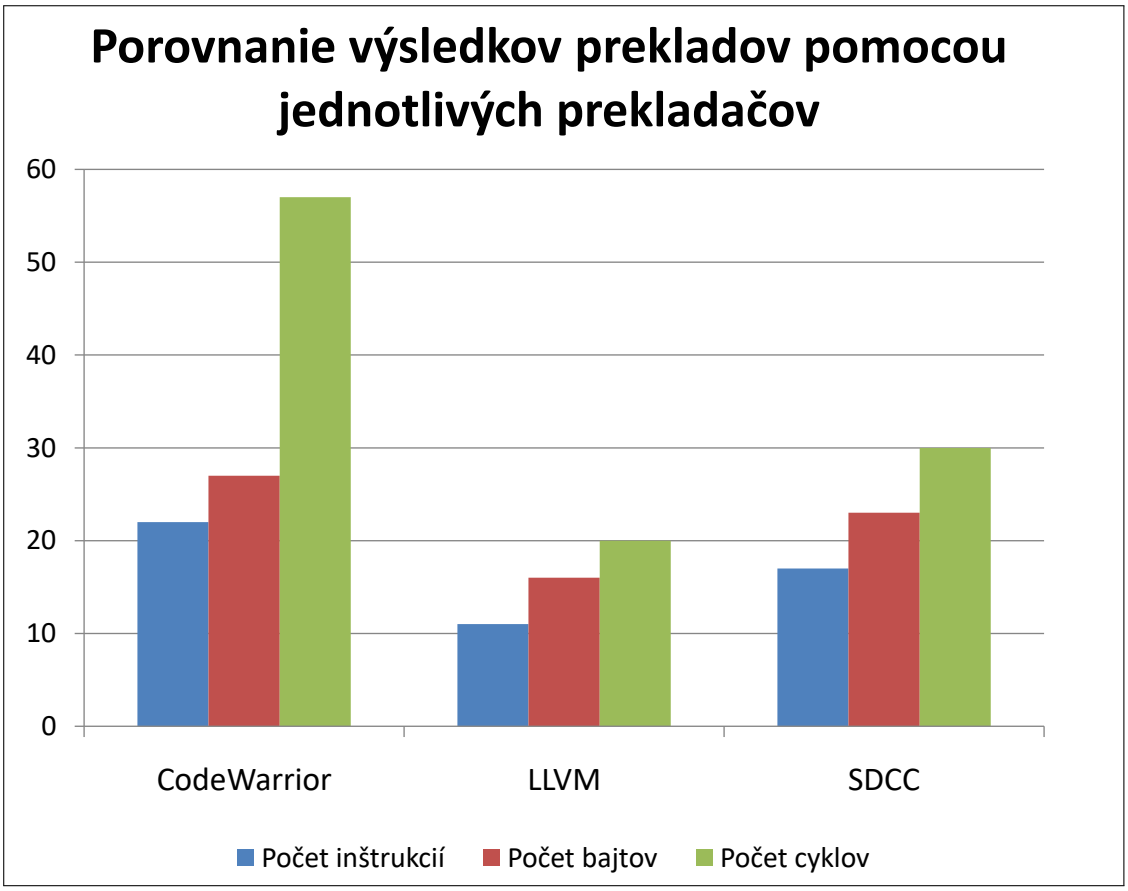
Tab. 5.3: Výsledný preklad programu č. 3

Prekladač CodeWarrior	**Prekladač LLVM**	**Prekladač SDCC**
f3_cw: PSHH LSLA TSX STA ,X ADD #42 STA ,X EOR #13 STA ,X ADD #8 STA ,X AND #-2 STA ,X NEG ,X ASR ,X INC ,X LDA ,X ORA #4 STA ,X DEC ,X LDA ,X PULH RTS	f3_llvm: LSLA ADD #42 EOR #13 ADD #8 AND #-2 NEGA ASRA INCA ORA #4 DECA RTS	f3_sdcc: LSLA ADD #42 CLRX EOR #13 PSHA TXA EOR #0 TAX PULA ADD #8 AND #-2 NEGA LSRA INCA ORA #4 DECA RTS
22 inštrukcií, 27 bajtov, 57 cyklov	11 inštrukcií, 16 bajtov, 20 cyklov	17 inštrukcií, 23 bajtov, 30 cyklov

Zaobalenie funkcií v hlavičkových súboroch príkazom `extern "C"` zabezpečí, že volanie funkcií bude používať volacie konvencie jazyka C. Takto bude zaručená kompatibilita s kódom, ktorý bol vygenerovaný prekladačom LLVM.

V hlavnom programe sú postupne vykonané jednotlivé testy. Najprv je volaná pôvodná funkcia v jazyku C a tak funkcie v jazyku symbolických adries vygenerované všetkými tromi prekladačmi.

Keďže úlohou programu 5.5 je zacyklenie, pre otestovanie ďalších funkcií je potrebné zakomentovať volanie funkcií tohto testu.



Obr. 5.1: Porovnanie výsledku prekladov

6 Zhodnotenie dosiahnutých výsledkov

Výsledkom tejto práce je upravený LLVM prekladač verzie 10.0.0, ktorým je možné preložiť jednoduché zdrojové kódy napísané v programovacom jazyku C do jazyka symbolických adries. Funkčnosť implementácie bola preukázaná v predošlej kapitole 5 na niekoľkých testoch. Z výsledkov prekladov pomocou jednotlivých prekladačov je možné povedať, že LLVM prekladač generuje najlepší kód, nasleduje SDCC a najhoršie výsledky dosiahol referenčný prekladač vývojového prostredia CodeWarrior. Toto tvrdenie však nie je možné kvôli ďalej spomenutým nedostatkom aplikovať globálne na celý prekladač LLVM, ale iba na dané, prípadne podobné testy.

Ako bolo spomenuté v časti 1.3.1, veľkou výhodou prekladača LLVM je jeho štruktúra, ktorá delí implementáciu frontend a backend časti na dve rôzne úlohy. Tieto úlohy však spája spoločná vnútorná forma prekladača. Vďaka tomu je možné upraveným prekladačom skompilovať aj program, ktorého zdrojový kód bol napísaný aj v inom programovacom jazyku, ako je jazyk C. Je však nutné zachovať volacie konvencie jazyka C, aby výsledný kód bol kompatiblilný s referenčným prekladačom, v ktorom by sa vygenerovaný kód využil.

Medzi hlavné nedostatky patrí to, že vytvorená backend časť prekladača podporuje iba určitú časť inštrukcií. Konkrétne ide o 26 inštrukcií z celkového počtu 273. Jedná sa predovšetkým o aritmetické a logické inštrukcie v registrovom adresovacom mode INH a v adresovacom móde IMM, ktorý pracuje s priamou hodnotou. Presný zoznam podporovaných inštrukcií je možné nájsť v prílohe A.1.

Dôvodom prečo prekladač nepodporuje iné inštrukcie a najmä iné adresovacie módy je to, že aj napriek dlhodobej snahe a štúdiu implementácií v iných cieľových architektúrach sa nepodarilo popísať prevod DAG grafu na strojové inštrukcie v takej miere, aby ich prekladač vedel použiť pri generovaní kódu.

V dôsledku tohto problému nie je možné používať operácie načítavania a ukladania z a do pamäte. To so sebou prináša ďalšie obmedzenia ako napríklad použitie viacerých premenných v programe hoci aj s podporovanými inštrukciami. Taktiež kvôli tomu nebolo možné pokračovať v rozširovaní ďalšej funkcionality prekladača.

Riešením tohto nedostatku by bolo ďalšie štúdium funkcií prekladača LLVM spojené s konzultáciami s niekým, kto má väčšie skúsenosti s prekladovým systémom LLVM. To by však už bolo nad časový rámec zodpovedajúci bakalárskej práci.

Po úspešnej implementácii zvyšku inštrukčnej sady by sa teda mal ďalší vývoj prekladača venovať implementovaniu podmienok, práci so zásobníkom, či podpore ďalších dátových typov.

Záver

Cieľom tejto práce bolo upraviť prekladač tak, aby generoval kód pre doposiaľ nepodporovanú cieľovú architektúru. Najprv však bolo nutné oboznámiť čitateľa so základnými pojmami týkajúcimi sa prekladačov.

Preto bol začiatok práce venovaný zoznámeniu sa so všeobecnou architektúrou prekladačov a významom jednotlivých častí. Ak je prekladač funkčne rozdelený na frontend a backend časť, pre pridanie podpory novej cieľovej architektúry je potrebné upraviť iba backend časť, ktorá je zodpovedná za generovanie výstupného kódu pre danú cieľovú architektúru. Preto bola táto časť detailne popísaná.

Následne bola popísaná architektúra už konkrétnych, voľne dostupných prekladačov SDCC a LLVM. SDCC je pomerne jednoduchý prekladač navrhnutý tak, aby pokryl špecifické požiadavky malých 8-bitových mikroprocesorov. Tento prekladač je ale veľmi slabo zdokumentovaný a preto bolo potrebné detailné naštudovanie kódu pre pochopenie toho, ako funguje. Naproti tomu, LLVM je komplexný, veľmi dobre zdokumentovaný prekladový systém, vyskladaný z množstva modulov, ktoré sa vyznačujú svojou znovu použiteľnosťou.

Pre vytvorenie backend časti podporujúcej novú cieľovú architektúru požadujú oba prekladače detailný popis architektúry mikroprocesora, vytvorenie popisu registrov, strojových inštrukcií či spôsobu volania funkcií. Rovnaká úloha sa však v oboch prekladačoch kvôli ich odlišnej architektúre implementuje rôzne. V SDCC je potrebné všetok kód napísať ručne v jazyku C. Keďže prekladač LLVM je napísaný v jazyku C++, ktorý podporuje polymorfizmus a mechanizmy písania generického kódu, je možné znovu použiť veľkú časť funkcií, ktoré sú už implementované. Popis postupu pridania novej cieľovej architektúry bol náplňou ďalšieho textu práce. Napokon bol prekladač LLVM vybraný kvôli možnosti využitia už naprogramovaných modulov a v neposlednom rade aj kvôli stále sa zvyšujúcej popularite vo svete, ktorá ho predurčuje pre široké využitie v budúcnosti.

Ako cieľová architektúra bol zvolený mikrokontrolér MC9S08LH64 z rodiny HCS08 od výrobcu NXP, kvôli viacerým skúsenostiam nadobudnutým v predošlom štúdiu na škole. Tento mikrokontrolér obsahuje viacero periférii, vďaka ktorým je možné ho použiť v mnohých aplikáciach. Jeho nevýhodou ale je veľmi malý počet registrov, čo výrazne sťažuje jeho programovanie.

Hlavným cieľom práce bolo zrealizovať backend časť prekladača pre zvolený mikroprocesor, ktorý doposiaľ nebol prekladačom podporovaný. To sa úspešne podarilo v kapitole 4. Ďalej nasledovalo vytvorenie vhodných testovacích programov, ktoré preukázali funkčnosť implementácie. Testovacie programy boli preložené upraveným prekladačom LLVM a výsledok bol porovnaný s prekladom získaným referenčným prekladačom vo vývojovom prostredí CodeWarrior a prekladačom SDCC, ktorý ro-

dinu mikroprocerov HCS08 podporuje. Z výsledku testov by sa dalo povedať, že prekladač LLVM vygeneroval najlepší kód, čo sa týka rýchlosti vykonávania programu ale aj jeho veľkosti. Avšak toto tvrdenie nemusí platiť všeobecne, pretože prekladač má viacero nedostatkov. Najväčším problémom je nepodporovanie kompletnej inštrukčnej sady. Zo všetkých 273 inštrukcií podporuje 26, čo je približne iba 10%. Aj napriek veľkej snahe sa nepodarilo implementovať všetky adresovacie módy inštrukcií.

Upravený prekladač LLVM je plne kompatibilný s referenčným prekladačom, avšak je nutné zabezpečiť, aby zdrojový kód mal volacie konvencie jazyka C. To znamená, že prekladač je možné použiť aj pre zdrojové kódy napísané v jazyku C++ pri dodržaní volacích konvencií.

Demonštrácia funkčnosti na reálnom hardvére nebola možná v dôsledku koronavírusovej pandémie. V programovacom prostredí CodeWarrior však bol vytvorený demonštračný program, ktorý je možné otestovať v simulátore mikroprocesora.

Významným prínosom tejto práce sú cenné znalosti získané v oblasti prekladačov a najmä prehľad v prekladovom systéme LLVM, ktorý je čím ďalej, tým viac populárnejší. Študovanie týchto poznatkov zabralo veľmi veľa času, ale bola to nevyhnutná súčasť práce z dôvodu porozumenia celej problematiky a následnej implementácie. Dokopy bolo venované celej práci niečo vyše 320 hodín.

Pretože prekladač rodinu mikroprocesorov HCS08 oficiálne nepodporuje, táto práca by mohla nájsť v tejto oblasti uplatnenie.

Aj napriek tomu, že prekladač nepodporuje kompletnú inštrukčnú sadu, bolo preukázané, že dosahuje významne lepšie výsledky ako referenčný prekladač CodeWarrior. Preto je možné ho použiť na zrýchlenie aritmeticko-logických častí kódu, pôvodne implementovaných v prekladači CodeWarrior, ktoré sú z hľadiska behu programu kritické (napr. výpočet akčného zásahu regulátora) a potrebujú byť vykonávané čo najrýchlejšie.

Literatúra

[1] MELICHAR, B.; JEŽEK, K.; RICHTA, K.; ČEŠKA, M., *Konstrukce překladačů*, Praha: Vydavatelství ČVUT, 1999, 367 s. ISBN 80-01-02028-2.

[2] LEUPERS, R.; MARWEDEL, P., *Retargetable Compiler Technology for Embedded Systems*, Springer, 2001, 176 s. ISBN 978-0-7923-7578-5.

[3] VÁŇA, V., *Začínáme pracovat s mikrokontroléry HC08 NITRON: příručka pro naprosté začátečníky*, Praha: BEN, 2003, 95 s. ISBN 80-7300-124-1.

[4] LOPES, B. C.; AULER, R., *Getting Started with LLVM Core Libraries*, Packt Publishing, 2014, 314 s. ISBN 978-1-78216-692-4.

[5] PANDEY, M.; SARDA, S., *LLVM Cookbook*, Packt Publishing, 2015, 284 s. ISBN 978-1-78528-598-1.

[6] MEDUNA, A.; LUKÁŠ R., *Formálni jazyky a překladače – Studijní opora*, 2006.

[7] ERHARDT, CH., *Design and Implementation of a TriCore Backend for the LLVM Compiler Framework*, FAU Erlangen, 2009 [cit. 02. 01. 2019]. Dostupné z URL: <`https://opus4.kobv.de/opus4-fau/files/1108/tricore_llvm.pdf`>.

[8] NAGY, M., *Překladač jazyka C pro mikroprocesor AVR32*, VUT FIT Brno, 2010 [cit. 05. 05. 2020]. Dostupné z URL: <`http://hdl.handle.net/11012/56018`>.

[9] BLAHOŽ, V., *Přizpůsobení platformy LLVM pro mikroprocesor Motorola 68000*, VUT FIT Brno, 2010 [cit. 07. 05. 2020]. Dostupné z URL: <`http://hdl.handle.net/11012/55919`>.

[10] HORNÍK, J., *Zadní část překladače podmnožiny jazyka C pro 8-bitový procesor*, VUT FIT Brno, 2011 [cit. 20. 11. 2019]. Dostupné z URL: <`http://hdl.handle.net/11012/54208`>.

[11] DUTTA, S., *Anatomy of a Compiler - A Retargetable ANSI-C Compiler*, Circuit Cellar, August 2000, s. 35.

[12] *Small Device C Compiler*, [cit. 14. 11. 2019]. Dostupné z URL: <`http://sdcc.sourceforge.net`>.

[13] *SDCC Compiler User Guide*, [cit. 15. 11. 2019]. Dostupné z URL: <`http://sdcc.sourceforge.net/doc/sdccman.pdf`>.

[14] *SDCC - Snapshot Builds*, [cit. 30. 06. 2020]. Dostupné z URL: <http://sdcc.sourceforge.net/snap.php>.

[15] *SDCC wikipedia*, [cit. 15. 11. 2019]. Dostupné z URL: <https://github.com/roybaer/sdcc-wiki/wiki>.

[16] *SDCC history*, [cit. 16. 11. 2019]. Dostupné z URL: <https://github.com/roybaer/sdcc-wiki/wiki/SDCC-history>.

[17] *Flow of compilation in SDCC*, [cit. 16. 11. 2019]. Dostupné z URL: <https://github.com/roybaer/sdcc-wiki/wiki/Flow>.

[18] LATTNER, CH., *The Architecture of Open Source Applications*, [cit. 30. 11. 2019]. Dostupné z URL: <http://www.aosabook.org/en/llvm.html>.

[19] *Oficiálne stránky projektu LLVM*, [cit. 01. 12. 2019]. Dostupné z URL: <http://llvm.org/>.

[20] *LLVM Language Reference Manual*, [cit. 01. 12. 2019]. Dostupné z URL: <https://llvm.org/docs/LangRef.html>.

[21] *The LLVM Target-Independent Code Generator*, [cit. 06. 12. 2019]. Dostupné z URL: <https://llvm.org/docs/CodeGenerator.html>.

[22] *Partitioned Boolean Quadratic Programming (PBQP)*, [cit. 12. 05. 2020]. Dostupné z URL: <http://www.complang.tuwien.ac.at/scholz/pbqp.html>.

[23] *Getting Started with the LLVM System*, [cit. 30. 12. 2019]. Dostupné z URL: <https://llvm.org/docs/GettingStarted.html>.

[24] *Advanced Build Configurations*, [cit. 30. 12. 2019]. Dostupné z URL: <https://llvm.org/docs/AdvancedBuilds.html>.

[25] *Clang - Features and Goals*, [cit. 07. 12. 2019]. Dostupné z URL: <https://clang.llvm.org/features.html>.

[26] *Clang vs Other Open Source Compilers*, [cit. 07. 12. 2019]. Dostupné z URL: <https://clang.llvm.org/comparison.html>.

[27] *TableGen*, [cit. 31. 12. 2019]. Dostupné z URL: <https://llvm.org/docs/TableGen/>.

[28] *Writing an LLVM Backend*, [cit. 02. 01. 2020]. Dostupné z URL: <https://llvm.org/docs/WritingAnLLVMBackend.html>.

[29] NXP, [online katalogový list], *HCS08 Family: Reference Manual*, Rev. 2, 05/2007, [cit. 03. 01. 2020]. Dostupné z URL: <https://www.nxp.com/docs/en/reference-manual/HCS08RMV1.pdf>.

[30] NXP, [online katalogový list], *MC9S08LH64 Reference Manual*. Rev. 5.1, 05/2012, [cit. 03. 01. 2020]. Dostupné z URL: <https://www.nxp.com/docs/en/reference-manual/MC9S08LH64RM.pdf>.

[31] NXP, [online katalogový list], *MC9S08LH64 Series Data Sheet*. Rev. 6.1, 08/2012, [cit. 04. 01. 2020]. Dostupné z URL: <https://www.nxp.com/docs/en/data-sheet/MC9S08LH64.pdf>.

[32] NXP, [online katalogový list], *MC9S08SE8, MC9S08SE4 Reference Manual*. Rev. 3, 04/2009, [cit. 02. 08. 2020]. Dostupné z URL: <https://www.nxp.com/docs/en/reference-manual/MC9S08SE8RM.pdf>.

[33] NXP, [online katalogový list], *CodeWarrior Development Studio for Microcontrollers V10.x HC(S)08 Build Tools Reference Manual*. Rev. 10.6, 01/2014, [cit. 10. 05. 2020]. Dostupné z URL: <https://www.nxp.com/docs/en/reference-manual/CWMCUS08CMPREF.pdf>.

[34] *Compiler Optimizations: Induction Variable Elimination*, [cit. 02. 08. 2020]. Dostupné z URL: <http://compileroptimizations.com/category/ive.htm>.

[35] *GCC, the GNU Compiler Collection*, [cit. 02. 08. 2020]. Dostupné z URL: <https://gcc.gnu.org/>.

[36] Wikipedia, The free encyclopedia, *SDCC história*, [cit. 05. 12. 2019]. Dostupné z URL: <https://en.wikipedia.org/wiki/LLVM#History>.

[37] Wikipedia, The free encyclopedia, *Syntaktická analýza*, [cit. 31. 10. 2019]. Dostupné z URL: <https://cs.wikipedia.org/wiki/Syntaktická_analýza>.

[38] Wikipedia, The free encyclopedia, *GNU General Public License*, [cit. 15. 11. 2019]. Dostupné z URL: <https://sk.wikipedia.org/wiki/GNU_General_Public_License>.

Zoznam symbolov, veličín a skratiek

frontend	Časť prekladača zodpovedná za generovanie vnútornej formy prekladača zo zdrojového súboru
backend	Časť prekladača zodpovedná za generovanie cieľového kódu z vnútornej formy
AST	Abstract Syntax Tree – v preklade abstraktný syntaktický strom
MCU	Mikrokontrolér
SDCC	Small Device C Compiler
iCode	Názov vnútornej formy prekladača SDCC
debugger	Nástroj na odladzovanie chýb programu
LLVM	Low Level Virtual Machine
LLVM IR	Názov vnútornej formy prekladača LLVM
Clang	Frontend prekladača LLVM
TableGen	Nástroj pre popis cieľovej architektúry
JIT	Just in Time
SSA	Single State Assignment – každej premennej je hodnota priradená iba jedenkrát
DAG	Directed acyclic graph – v preklade orientovaný acyklický graf

Zoznam príloh

A Podporované inštrukcie HCS08

Implementovaná backend časť prekladača podporuje v dobe písania tejto práce 26 z celkových 273 inštrukcií, ich konkrétne vymenovanie sa nachádza v tabuľke A.1. Formát `#opr8i` udáva, že sa jedná o okamžitú 8-bitovú hodnotu. Informácie sú prevzaté z [32].

Tab. A.1: Zoznam doposiaľ podporovaných inštrukcií

Formát inštrukcie	**Adresovací mód**	**Počet bajtov**	**Počet cyklov**
ADD #opr8i	IMM	2	2
SUB #opr8i	IMM	2	2
INCA	INH	1	1
INCX	INH	1	1
DECA	INH	1	1
DECX	INH	1	1
CLRA	INH	1	1
CLRX	INH	1	1
CLRH	INH	1	1
NEGA	INH	1	1
NEGX	INH	1	1
AND #opr8i	IMM	2	2
ORA #opr8i	IMM	2	2
EOR #opr8i	IMM	2	2
COMA	INH	1	1
COMX	INH	1	1
LSLA	INH	1	1
LSLX	INH	1	1
LSRA	INH	1	1
LSRX	INH	1	1
ASLA	INH	1	1
ASLX	INH	1	1
ASRA	INH	1	1
ASRX	INH	1	1
BRA	REL	2	3
RTS	INH	1	5

B Zdrojový kód

Zdrojový kód implementácie novej backend časti prekladača LLVM sa nachádza v priečinku `zdrojovy_kod`. Celý prekladač pozostáva z niekoľkých desiatok tisíc súborov, ktoré nebolo možné z kapacitných dôvodov priložiť k tejto práci. Preto sú v prílohe pripojené iba zdrojové kódy, ktoré boli vytvorené, prípadne upravené. Získať zvyšné zdrojové súbory je možné z oficiálnych stránok prekladového systému LLVM [19]. Pre kompatibilitu však treba stiahnúť verziu 10.0.0.

Pre úspešný preklad s vytvorenou cieľovou architektúrou, je potrebné použiť parameter `-DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD="HCS08"` pre príkaz `cmake`.

Výpis súborov v priečinku `zdrojovy_kod`:

```
zdrojovy_kod/llvm/
├── include/llvm/ADT/
│   └── Triple.h ..... registrácia backend časti
└── lib/
    ├── Support/
    │   └── Triple.cpp ..... registrácia backend časti
    └── Target/
        ├── HCS08/ ..... hlavný priečinok s implementáciou
        ├── CMakeLists.txt
        └── LLVMBuild.txt
```

C Testovacie programy

Popis súborov v priečinku `testovacie_programy`:

```
testovacie_programy/
├── pr1/ ..... testovací program č. 1
│   ├── pr1.c ..... zdrojový program
│   ├── pr1.ll ..... vnútorná forma llvm
│   ├── pr1.asm ..... preklad prekladačom SDCC
│   └── pr1.s ..... preklad prekladačom LLVM
├── pr2/ ..... testovací program č. 2
│   ├── pr2.c ..... zdrojový program
│   ├── pr2.ll ..... vnútorná forma llvm
│   ├── pr2.asm ..... preklad prekladačom SDCC
│   └── pr2.s ..... preklad prekladačom LLVM
└── pr3/ ..... testovací program č. 3
    ├── pr3.c ..... zdrojový program
    ├── pr3.ll ..... vnútorná forma llvm
    ├── pr3.asm ..... preklad prekladačom SDCC
    └── pr3.s ..... preklad prekladačom LLVM
```

D Demonštračný program

Demonštračný program novej backend časti sa nachádza v priečinku `testovanie_c_asm`. Zdrojové programy sa nachádzajú v priečinku `Sources` a hlavičky funkcií sa nachádzajú v jednotlivých hlavičkových súboroch v priečinku `Project_Headers`.