VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘIZPŮSOBENÍ PLATFORMY LLVM PRO MIKROPROCESOR MOTOROLA 68000

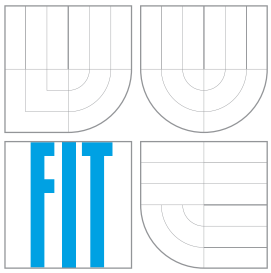
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

VLADIMÍR BLAHOŽ

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘIZPŮSOBENÍ PLATFORMY LLVM PRO MIKROPROCESOR MOTOROLA 68000

RETARGETING OF LLVM PLATFORM FOR MOTOROLA 68000 MICROPROCESSOR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — VLADIMÍR BLAHOŽ
AUTHOR

VEDOUCÍ PRÁCE — prof. Ing. TOMÁŠ HRUŠKA, CSc.
SUPERVISOR

BRNO 2010

Abstrakt

Bakalářská práce popisuje obecnou problematiku překladačů, seznamuje čtenáře s platformou Low-Level Virtual Machine a možnostmi její modifikace. Dále se zabývá principy architektury typu Motorola 68000 a implementací podpory její instrukční sady pro platformu LLVM.

Abstract

This bachelor's thesis deals with general questions of compilers, describes the Low-Level Virtual Machine platform and its modification options. Furthermore it concerns about principles of Motorola 68000 architecture and implementation of its instruction set for the LLVM platform.

Klíčová slova

překladač, back-end, LLVM, Low-Level Virtual Machine, Motorola 68000, M68k.

Keywords

compiler, back-end, LLVM, Low-Level Virtual Machine, Motorola 68000, M68k.

Citace

Vladimír Blahož: Přizpůsobení platformy LLVM pro
mikroprocesor Motorola 68000, bakalářská práce, Brno, FIT VUT v Brně, 2010

Přizpůsobení platformy LLVM pro mikroprocesor Motorola 68000

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. Ing. Tomáše Hrušky CSc.

.....
Vladimír Blahož
18. května 2010

Poděkování

Chtěl bych poděkovat panu Mgr. Miloslavu Trmačovi za cenné rady při psaní této práce a její programové části.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Tato práce se zabývá problematikou překladačů z vyšších programovacích jazyků do jazyka strojového. V dnešní době je k dispozici nepřeberné množství více či méně optimalizovaných překladačů pro různé typy koncových zařízení a přestože vzhledem k dnešním velikostem pamětí již nebývají kladeny tak veliké nároky na optimalizaci kódu co do velikosti, vysoce optimalizované překladače se staly nedílnou součástí programového vybavení většiny programátorů.

Tato bakalářská práce má za úkol přizpůsobit platformu *Low-Level Virtual Machine* (dále jen LLVM), která je takovýmto překladačem z vyšších programovacích jazyků do strojového kódu pro různé architektury, pro mikroprocesor *Motorola 68000*.

V kapitole 2 je stručně probrána problematika překladačů, následuje kapitola 3, která popisuje konstrukci platformy LLVM a možnosti jejího rozšiřování. Další části zahrnují samotné práce na přizpůsobení tohoto překladače. Konkrétně kapitola 4 se zabývá cílovou architekturou *Motorola 68000*, kapitola 5, popisuje implementaci její instrukční sady do systému LLVM, a kapitola 6 shrnuje výsledek a nedostatky práce.

Kapitola 2

Překladače z vyšších programovacích jazyků

Prakticky všechny programy jsou v dnešní době psány v některém z vyšších programovacích jazyků. Jelikož procesory počítačů tyto programy provádějí ve vlastních strojových jazycích, je třeba mít k dispozici program zvaný překladač, nebo také kompilátor. Kompilátor dostane na vstupu program napsaný ve zdrojovém jazyce (v našem případě vyšší programovací jazyk – například C/C++) a přeloží jej na funkčně ekvivalentní program napsaný v jazyce cílovém (tedy nejčastěji strojový kód, nebo jazyk symbolických instrukcí cílové architektury). Překlad kódu je vícestupňový a podle literatury [4] obvykle probíhá v následujících krocích:

1. lexikální analýza,
2. syntaktická analýza,
3. sémantická analýza,
4. generování vnitřního kódu,
5. optimalizace,
6. generování cílového kódu.

2.1 Lexikální analýza

Na program lze v této nejzákladnější fázi překladu nahlížet jako na řetězec znaků. Lexikální analyzátor tento řetěz znaků prochází a rozkládá jej na *lexikální symboly*, což jsou symboly zdrojového programu (klíčová slova, indentifikátory apod.).

2.2 Syntaktická analýza

Syntaktický analyzátor obdrží na vstupu řetěz lexikálních symbolů zdrojového programu a snaží se jej převést na tzv. *derivační strom*. Pokud takovýto strom nalezne, pak je vstupní program syntakticky správně napsaný, v opačném případě nikoliv.

2.3 Sémantická analýza

Sémantický analyzátor kontroluje, zda je v pořádku sémantika programu. Na vstupu obdrží derivační strom ze syntaktické analýzy a na svém výstupu generuje *abstraktní syntaktický strom*. Kontroluje především definice všech proměnných, jejich datové typy a může provádět implicitní typové konverze.

2.4 Generování vnitřního kódu

Generátor vnitřního kódu svůj vstup, kterým je abstraktní syntaktický strom, převádí na vnitřní reprezentaci překladače (nejčastěji tříadresný kód). Přestože v této fázi by teoreticky již bylo možné vygenerovat program ve výstupním jazyce, generování tohoto mezikódu se provádí z několika důvodů:

- přímý překlad do cílového programu je složitý
- vygenerovaný vnitřní kód lze snadněji optimalizovat
- tento mezistupeň umožňuje rozšířit překladač, aby byl schopen pracovat s více vstupními i výstupními jazyky

2.5 Optimalizace

Optimalizátor svůj vstup, kterým je vnitřní kód překladače, převádí na tzv. *optimalizovaný vnitřní kód*. Optimalizačních technik je celá řada a přestože optimalizace vnitřního kódu může v jednoduchém překladači i zcela chybět, u složitých kompilátorů se může jednat o nejsložitější komponentu a fázi celého překladu. Typickými optimalizacemi jsou např. šíření konstanty, odstranění „mrtvého kódu“ apod.

2.6 Generování cílového kódu

Generátor cílového kódu svůj vstup – optimalizovaný vnitřní kód překladače – převádí do výstupního jazyka (tedy jazyka symbolických instrukcí, nebo strojového kódu).

2.7 Front-end a back-end

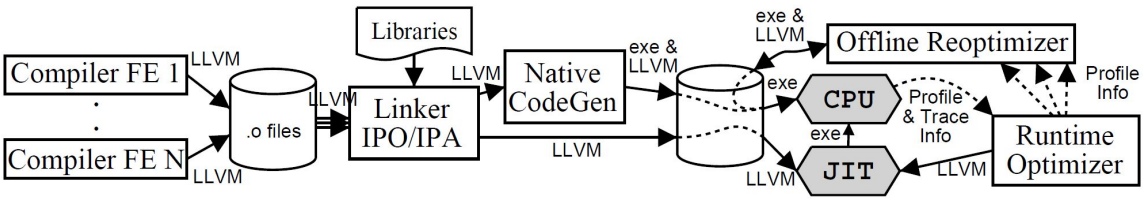
První čtyři fáze překladu souvisí hlavně se vstupním jazykem a jeho převodem do jednotné interní reprezentace kompilátoru. Taková modularita umožňuje nahrazením těchto částí přizpůsobit překladač pro více vstupních jazyků. Podobně jako zbylé dva kroky překladače souvisí s výstupním jazykem a ty je zase možné upravovat podle zvolené cílové architektury. Toto je jeden z důvodů generování vnitřního mezikódu uprostřed překladu. Kompilátor je tak rozdělen logicky na dva relativně samostatné celky – první nazývaný *front-end* pro překlad zdrojového programu do vnitřního kódu překladače a *back-end* překládající tento vnitřní kód na výstupní cílový program.

Kapitola 3

Low-Level Virtual Machine

Low-Level Virtual Machine je rozsáhlý systém vytvořený pro podporu analýzy a transformací libovolných programů. K tomu využívá vysokoúrovňových technik jak v době kompilace a linkování, tak za běhu programu. Celý systém umožňuje velice efektivní optimalizace v průběhu celého života programu. Jedná se o open-source program, jsou tedy k dispozici veškeré zdrojové kódy celého systému. Základními komponentami, z nichž se LLVM skládá, je *C & C++ front-end* založený na překladači GCC, který kromě C a C++ podporuje např. Objective-C, Fortran atd., stále se rozšiřující soubor optimalizátorů a několik back-endů, tedy generátorů kódu pro některé běžné (i méně obvyklé) architektury (např. X86, X86-64, PowerPC, PowerPC-64, ARM, Thumb, SPARC apod.). Kromě toho obsahuje LLVM i front-end podporující jazyky Java a Scheme a další se stále vyvíjejí. LLVM využívá pro své potřeby vniřní kód, v dokumentaci [2] nazývaný *LLVM assembly language*. Přes to všechno, jak už název napovídá, neposkytuje LLVM vysokoúrovňové komponenty jako *garbage collector* nebo generátor kódu za běhu programu.

Obrázek 3.1, převzatý z článku o LLVM[3], zachycuje princip jeho architektury. Front-end kompilátoru vygeneruje kód (resp. kódy) ve vnitřním *LLVM assembly language*, které jsou spojeny *LLVM linker*em, který sám již provádí některé optimalizace. Poté je kód přeložen do nativního kódu cílové architektury. Pro některé je k dispozici i *Just-in-time* překladač (JIT), který umožňuje překlad za běhu programu.



Obrázek 3.1: Architektura systému LLVM

3.1 LLVM assembly language

Tento vnitřní kód, který LLVM využívá, je založený na principu *Static single assignment* (SSA), což znamená, že libovolné proměnné může být přiřazena hodnota pouze jednou

a poté již nesmí být přepsána. Tato reprezentace programu poskytuje nízkoúrovňové operace a je možné jí zapsat jakýkoliv program, přeložený z libovolného vyššího programovacího jazyka. Příklad kódu LLVM assembly language:

```
%0 = add i32 %X, %X           ; yields {i32}:%0
%1 = add i32 %0, %0           ; yields {i32}:%1
%result = add i32 %1, %1

%2 = shl i32 %result, i8 3
@Y = mul i32 %2, 5
```

Tabulka 3.1: Ukázka kódu LLVM assembly language

Z předešlého příkladu jsou viditelné některé základní syntaktické prvky jazyka:

- komentáře jsou uvozeny znakem `;`,
- názvy lokálních proměnných jsou uvozeny znakem `%`,
- názvy globálních proměnných jsou uvozeny znakem `@`,
- pokud výsledek operace není přiřazen pojmenované proměnné, jsou vytvořeny dočasné proměnné pojmenované sekvenčně.

Vnitřní kód LLVM podporuje také modularitu programů. Každý modul sestává z globálních proměnných, tabulky symbolů a funkcí. Moduly jsou následně skládány dohromady pomocí *LLVM linker*u. Příklad modulu „Hello world“ převzatý z dokumentace LLVM[2]:

```
; Declare the string constant as a~global constant.
@.LC0 = internal constant [13 x i8] c"hello world\0A\00"    ; [13 x i8]*

; External declaration of the puts function
declare i32 @puts(i8 *)                                     ; i32(i8 *)*

; Definition of main function
define i32 @main() {                                        ; i32()*
  ; Convert [13 x i8]* to i8  *...
  %cast210 = getelementptr [13 x i8]* @.LC0, i64 0, i64 0   ; i8 *

  ; Call puts function to write out the string to stdout.
  call i32 @puts(i8 * %cast210)                             ; i32
  ret i32 0
}

; Named metadata
!1 = metadata !{i32 41}
!foo = !{!1, null}
```

Tabulka 3.2: Příklad modulu „Hello world“

3.2 TableGen

Při tvorbě nového back-endu pro podporu instrukční sady procesoru, která dosud není v LLVM implementovaná, je velice užitečným pomocníkem nástroj *TableGen*. TableGen je součástí LLVM a jeho využití spočívá v záznamu informací specifických pro vybranou architekturu. Využívá jazyk s vlastní syntaxí a na svém výstupu generuje soubory v jazyce C++. Výhodou použití nástroje TableGen je minimalizace duplikací informací, které je třeba o cílové architektuře zaznamenat, a tím i menší šance na chyby při jejich psaní.

Zdrojové soubory pro TableGen obsahují záznamy (*records*), které se dělí na třídy (*classes*) a definice (*definitions*). Každý záznam má své jednoznačné jméno, seznam nadtříd a seznam hodnot, které tvoří jádro informací zaznamenávaných o architektuře. Třídy se používají pro vytváření abstraktních záznamů, které slouží jako vzor pro vytváření definic s některými shodnými hodnotami. Typicky je pak možné vidět u konkrétních back-endů třídy jako `RegisterClass`, nebo `Instruction`, z nichž jsou poté odvozovány další třídy (například `FPInst` reprezentující instrukci v plovoucí řádové čárce u X86 back-endu). Definice jsou konkrétní koncové záznamy, které nemají žádné nedefinované hodnoty. Jsou uvozeny klíčovým slovem `def`.

Komentáře TableGen používá stejné jako např. C/C++, tedy „`//`“ pro zakomentování celého řádku a „`/* */`“ ohraničující libovolně dlouhý komentář.

TableGen je silně typovaný a podporuje řadu datových typů od nejjednodušších, jako je např. `bit`, až po velice složité, jako `dag`:

- `bit` je boolovský datový typ
- `int` je 32-bitová číselná hodnota
- `string` je libovolně dlouhý řetězec znaků
- `bits<n>` reprezentuje řetězec bitů libovolné, ale pevně dané délky **n**
- `dag` je datový typ, který doslova znamená `directed acyclic graph`, tedy orientovaný acyklický graf, který se využívá při výběru instrukcí
- `code` je podobný datový typ jako `string`. Používá se, jak název napovídá, pro zápis větších textů, obvykle kódu, protože nevyžaduje zápis uvozovek a podobných znaků pomocí escape sekvencí.
- `list<ty>` je seznam prvků datovéto typu **ty**
- `Třída (class)` - název třídy lze použít jako datový typ, což umožňuje vytvořit prvek datového typu `list` s položkami, které musí být zvolené třídy, nebo její podtřídy (například `list<Register>`, jako seznam registrů)

Kromě jednorázové definice třídy je možné definovat i takzvanou `multitřídu (multiclass)`, která umožňuje vytvořením jediné její instance vytvořit hned několik definic najednou. Toho bývá s výhodou využíváno pro definice instrukcí, které mohou pracovat s několika různými typy operandů. Řekněme, že procesor podporuje několik instrukcí, které mohou provádět operaci nad dvěma registry, nebo nad registrem a číselnou hodnotou. V tabulce 3.3 je příklad, jak by mohla vypadat multitřída a vytvoření její instance pro tento případ.

```
multiclass AritInst<string OpcStr, SDNode OpNode> {
  def rr    : Inst<(outs GPRegs:$dst0), (ins GPRegs:$src0, GPRegs:$src1),
              !strconcat(OpcStr, " $dst0, $src0, $src1"),
              [(set GPRegs:$dst0, (OpNode GPRegs:$src0, GPRegs:$src1))]>;

  def ri    : Inst<(outs GPRegs:$dst0), (ins GPRegs:$src0, i32imm:$src1),
              !strconcat(OpcStr, " $dst0, $src0, $src1"),
              [(set GPRegs:$dst0, (OpNode GPRegs:$src0, (i32 imm:$src1)))]>;
}

  defm ADD:AritInst<"add", add>;
  defm SUB:AritInst<"sub", sub>;
```

Tabulka 3.3: Multiclass v TableGenu

V multitřídě jsou vytvořeny dvě definice, každá pro daný typ operandů. Multitřída je pak definována parametry, které jednoznačně určí operaci a textovou reprezentaci instrukce. Vytvoření instance multitřídy klíčovým slovem `defm` pak vytvoří hned dvě definice – prvním příkazem tedy definice `ADDrr` a `ADDri`, druhým `SUBrr` a `SUBri`.

Z dokumentace LLVM[2] je také ukázka, která naznačuje, kolik informací back-end potřebuje ohledně dané architektury. Konkrétně je to ukázáno na instrukci pro sečtení dvou 32-bitových registrů procesoru X86. Kód, který TableGen generuje (pro ukázku zkrácený asi na třetinu), je zachycen v tabulce 3.4.

```
def ADD32rr {   // Instruction X86Inst
I~string Namespace = "X86";
  dag OutOperandList = (outs GR32:$dst);
  dag InOperandList = (ins GR32:$src1, GR32:$src2);
  string AsmString = "add{l}\t{$src2, $dst|$dst, $src2}";
  list<dag> Pattern = [(set GR32:$dst, (add GR32:$src1, GR32:$src2))];
  list<Register> Uses = [];
  list<Register> Defs = [EFLAGS];
    .
    .
  bits<3> ImmTypeBits = { 0, 0, 0 };
  bit hasOpSizePrefix = 0;
  bit hasAdSizePrefix = 0;
  bits<4> Prefix = { 0, 0, 0, 0 };
  bit hasREX_WPrefix = 0;
  FPFormat FPForm = ?;
  bits<3> FPFormBits = { 0, 0, 0 };
}
```

Tabulka 3.4: Výstup nástroje TableGen

Je zřejmé, že psát takový kód pro každou instrukci by bylo velice náročné na tvorbu i údržbu a vyžadovalo by spoustu kopírování duplikací. TableGen však všechny tyto infor-

mace sám vygeneroval z kódu 3.5.

```
let Defs = [EFLAGS],
    isCommutable = 1,                   // X = ADD Y,Z --> X = ADD Z,Y
    isConvertibleToThreeAddress = 1 in // Can transform into LEA.
def ADD32rr  : I<0x01, MRMDestReg, (outs GR32:$dst),
                                   (ins GR32:$src1, GR32:$src2),
                "add{l}\t{$src2, $dst|$dst, $src2}",
                [(set GR32:$dst, (add GR32:$src1, GR32:$src2))]>;
```

Tabulka 3.5: Vstup nástroje TableGen

V příkladu 3.5 je definován záznam `ADD32rr`, který jednoznačně určuje konkrétní instrukci, je odvozen od třídy `I` a za jejím identifikátorem následují konkrétní parametry. Z těch jsou nejzásadnější vstupy a výstupy instrukce, řetězec, který je využitý v případě, že je generován výstup čitelný pro člověka, a nakonec vzorec, kterým se daná instrukce mapuje do stromové struktury, která reprezentuje program. Příznaky nad definicí jsou hodnoty specifické pro tuto instrukci, které nelze zobecnit pro celou třídu *I*.

TableGen je velice silný nástroj pro záznam informací o cílové architektuře, přesto však zatím není možné jím zachytit všechno potřebné a množství kódu je třeba dopsat „ručně“ pomocí `C++`. Řeč je například o práci se zásobníkem, volání a návraty z podprogramů, přesuny mezi registry apod. Podle dokumentace LLVM[2] se do budoucna počítá se schopností TableGenu zaznamenat veškeré informace, které back-end potřebuje, a úplně tak eliminovat nutnost ošetřovat cokoliv mimo něj.

Kapitola 4

Motorola 68000

Mikroprocesor Motorola 68000 (zkráceně M68k) vznikl z projektu *Motorola Advanced Computer System on Silicon*, který započal v roce 1976. Jedná se o 32-bitový procesor, který v počátcích pracoval na frekvenci 4,6 a 8 Mhz, později až na frekvencích 10 a 16,67 MHz. Jádro je navrženo firmou *Freescale Semiconductor*. M68k je architektura ze skupiny CISC procesorů (Complex Instruction Set Computer), což znamená, že pracuje s rozsáhlou a komplexní sadou instrukcí a relativně malým počtem registrů. Do rodiny M68k patří například mikroprocesory MC68000 (16-/32-bitový mikroprocesor), MC68EC00 (16-/32-bitový vestavěný mikrokontrolér), MC68HC000 (nízkonapěťový 16-/32-bitový mikroprocesor), nebo koprocesory MC68881 (koprocesor pro výpočty v plovoucí řádové čárce) a MC68882 (rozšířený koprocesor pro výpočty v plovoucí řádové čárce) a jiné.

4.1 Registrová sada

Mikroprocesory řady M68k obsahují dvě skupiny registrů - *user* registry a *supervisor* registry (mohlo by být volně přeloženo jako *uživatelská* a *správcovská* sada registrů). Obyčejné programy spouštěné v uživatelském módu mohou pracovat pouze s uživatelskou sadou registrů, *supervisor* registry tedy pro tuto práci nejsou podstatné.

Registrová sada mikroprocesorů M68k pro práci s celými čísly sestává z 16-ti 32-bitových obecných registrů (`D0` – `D7` a `A0` – `A7`), 32-bitový programový čítač `PC` a 8-bitový příznakový registr `CCR` [1].

4.1.1 Datové registry (D0 – D7)

Datové registry se používají pro operace s bity, bitovými poli (1-32 bitů), byty (8 bitů), slovy (16 bitů), dvojslovy (32 bitů) a čtyřslovy (64 bitů). Také je lze použít jako indexové registry.

4.1.2 Adresové registry (A0 – A7)

Adresové registry se používají jako indexové registry, registry pro uložení bázové adresy, nebo ukazatelé na zásobník. Registr A7 (synonymum pro něj je SP) funguje jako ukazatel na vrchol hardwarového zásobníku při volání podprogramů a při práci s výjimkami.

4.1.3 Programový čítač (PC)

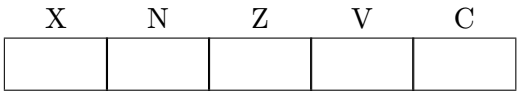
Programový čítač obsahuje uloženou adresu právě provádění instrukce. Procesor jeho hodnotu automaticky zvyšuje.

4.1.4 Příznakový registr (CCR)

V uživatelském módu je k dispozici pouze prvních pět bitů nižšího bytu příznakového registru. Bity příznakového registru jsou nastavovány většinou aritmetických instrukcí. Tyto nastavené příznaky pak ovlivňují chování instrukcí systémových a programových. Práce s příznakovým registrem je podmíněna dvěma zásadami - konzistence mezi instrukcemi, použitími a instancemi a smysluplné výsledky.

Konzistence mezi instrukcemi znamená, že všechny instrukce, které jsou speciálním případem nějaké nadřazené instrukce, ovlivňují příznakový registr stejným způsobem, jako tato nadřazená instrukce. Konzistence mezi použitími znamená, že podmíněné instrukce se budou chovat stejně, ať už příznaky nastavila instrukce přesunu, porovnání, nebo testovací instrukce. A končeně konzistence mezi instancemi zaručuje, že všechny instance stejné instrukce ovlivňují příznakové bity stejným způsobem.

První čtyři bity vyjadřují výsledek provedené instrukce, pátý bit, zvaný `extend` (X-bit), se používá při výpočtech s rozšířenou přesností. V podstatě duplikuje bit `carry` a je od něj oddělen jen pro usnadnění programovacích technik.



Obrázek 4.1: Příznakový registr

- `X (Extend)` je nastavován na stejnou hodnotu, jako `carry` bit.
- `N (Negative)` je nastaven, pokud je nastaven nejvýznamější bit výsledku. V opačném případě vynulován.
- `Z (Zero)` je nastaven v případě, že výsledek je nulový, jinak nulován.
- `V (Overflow)` je nastaven do jedničky v případě přetečení aritmetické operace. Jinak nastaven na nulu.
- `C (Carry)` je nastavován pokud dochází k přenosu nejvýznamějšího bitu při sčítání, nebo výpujčce při odečítání. V opačných případěch nulován.

4.1.5 Registy pro výpočty v plovoucí řádové čárce

Cílem této práce je implementovat instrukční sadu pro práci v pevné řádové čárce, registry pro *floating-point* jednotku se tedy nebude zabývat.

4.2 Organizace dat v registrech

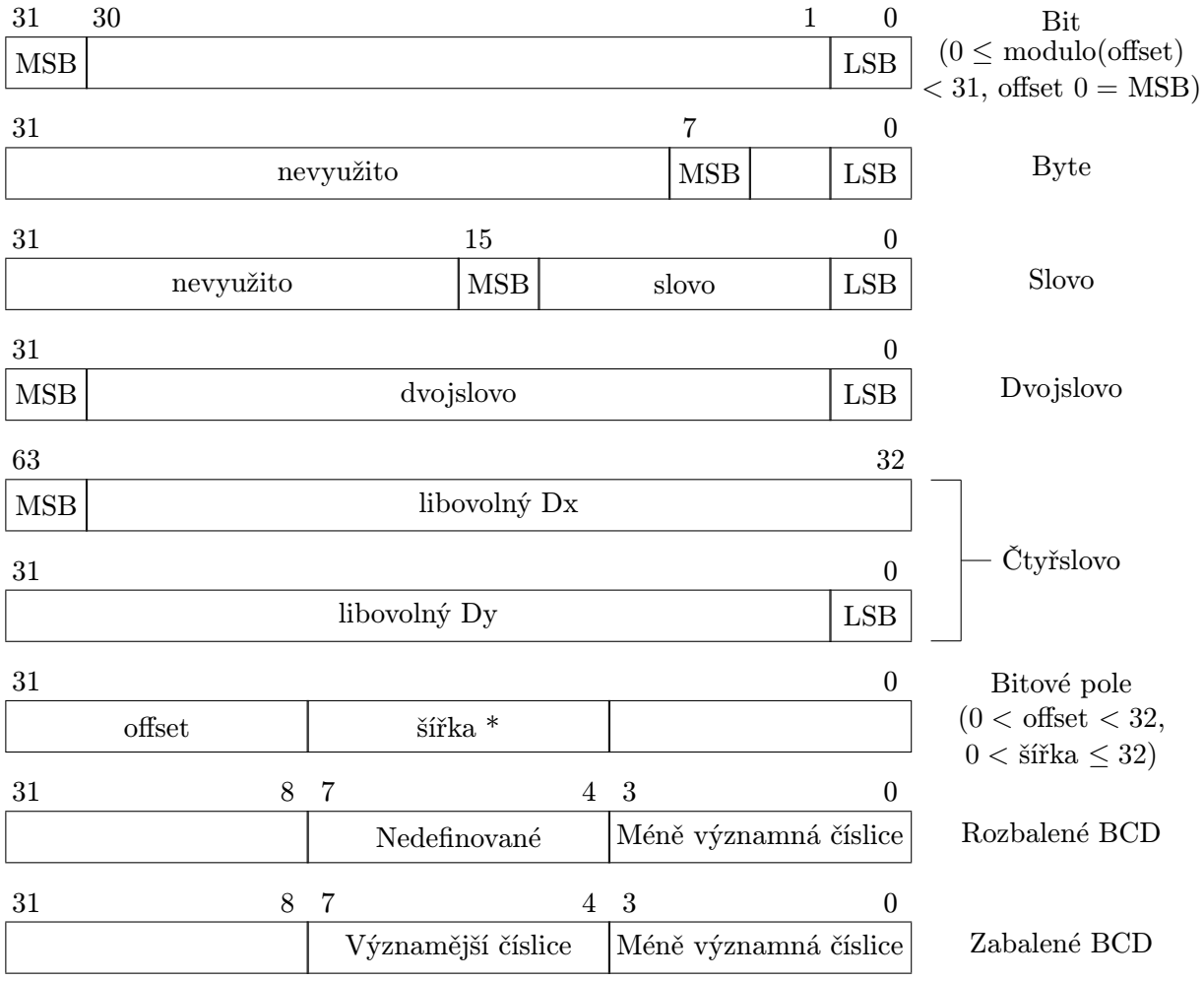
4.2.1 Datové registry

Všechny registry pro práci v pevné řádové čárce mají 32 bitů. Operandy velikosti byte nebo slovo zabírají spodních osm respektive šestnáct bitů registru. Dvojslovo zabírá celý rozsah registru. Instrukce pracující s datovými registry, ať už jako se zdrojovým, nebo cílovým operandem, čte, respektive modifikuje vždy jen příslušnou část registru podle velikosti operandu instrukce. Takže například bytová instrukce zapisující výsledek do datového registru modifikuje jen nejnižších osm bitů registru, ostatní nechá nezměněné. Je-li uloženo v registru dvojslovo, adresa jeho nejméně významného bitu (LSB - *least significant bit*) je 0, adresa nejvýznamějšího bitu (MSB - *most significant bit*) je 31. Jestliže je v datovém registru uloženo bitové pole, adresa jeho MSB je 0 a adresa LSB je velikost registru pole mínus jedna (*offset*).

Některé instrukce mohou vracet výsledek velikosti čtyřslova. Takovýto výsledek může být uložen do libovolných dvou datových registrů bez omezení na jejich párování či pořadí.

Procesory M68k podporují i práci s binárně kódovanými desítkovými čísly (dále jen BCD - *binary-coded decimal*). Pracují se dvěma základními kódy pro BCD operace, a to *packed* a *unpacked* (zabalený a rozbalený). V rozbaleném formátu definuje každý byte jedno desítkové číslo reprezentované binárně. Jelikož lze všechny desítkové číslice definovat pouze čtyřmi bity, je tato číslice zapsána v nižších čtyřech bitech, zbylé čtyři jsou nedefinované. Při použití zabaleného formátu BCD mohou každý byte obsadit až dvě desítkové číslice - méně významná obsazuje méně významné čtyři bity, významější číslice je zapsána na významější bity.

Organizace celočíselných dat v datových registrech je znázorněna na obrázku 4.2 podle [1].



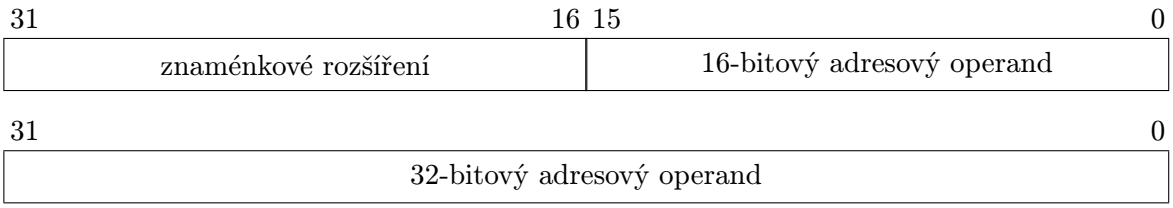
* pokud šířka + offset > 32, bitové pole se v registru "stočí dokola"

Obrázek 4.2: Organizace dat v datových registrech

4.2.2 Adresové registry

Adresové registry a ukazatel na zásobník jsou také 32 bitů veliké, jejich použití je ale omezeno pouze jako operandy velikosti slova, nebo dvojslova. Pro bytové operace je není možné adresovat. Pokud je adresový registr zdrojovým operandem, je načteno jeho méně významné slovo, nebo celé dvojslovo, v závislosti na velikosti operandu. Je-li adresový registr operandem cílovým, je výsledkem ovlivněna celá šířka registru. Pokud je výsledek pouze slovo, je znaménkově rozšířen na 32 bitů a uložen do celého registru.

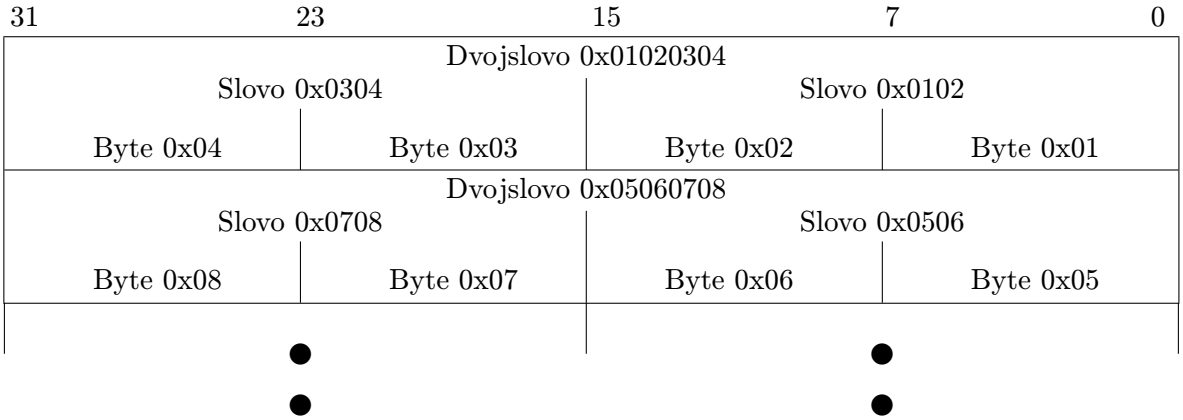
Organizace dat v adresových registrech je znázorněna na obrázku 4.3 podle [1].



Obrázek 4.3: Organizace dat v adresových registrech

4.3 Organizace dat v paměti

Mikroprocesory rodiny M68k patří do skupiny procesorů typu *big-endian*. To znamená, že při zápisu do paměti zapisují významější byty na nižší adresy (oproti procesorům typu little-endian, u kterých je to přesně naopak). Paměť je adresovatelná po bytech. Adresa N zvoleného dvojslova v paměti tak odpovídá adrese nejvýznamějšího bytu nejvýznamějšího slova, méně významné slovo je tedy na adrese $N + 2$ a nejméně významný byte na $N + 3$ (viz obrázek 4.4 podle [1]).



Obrázek 4.4: Organizace dat v paměti

4.4 Adresovací módy

Většina instrukcí pro Motoroly řady 68000 je tzv. dvouadresných. To prakticky znamená, že operace obdrží jeden cílový a jeden zdrojový operand, provede nad nimi požadovanou operaci a výsledek uloží na pozici cílového operandu. Cílový operand je tak vlastně zároveň druhým zdrojovým operandem. Unární operace tedy dostávají pouze jeden operand, nad kterým je výpočet proveden a uložen zpět na pozici.

Pro CISC architektury je typické, že používají relativně velké množství adresovacích módů a pro přístup do paměti se neomezují pouze na instrukce typu `load` a `store`. Stejná instrukce tak může být vyvolána se spoustou kombinací různých operandů. Lokace operandů nebo způsob výpočtu efektivní adresy může být určen více způsoby. Může být určen přímo operačním kódem instrukce. V instrukčním slově může být určen registr, který je operandem, nebo v instrukci v poli pro efektivní adresu může být zakódovaný režim pro výpočet efektivní adresy operandu. Následující podkapitoly popisují jednotlivé adresovací režimy a jsou doplněny obrázky inspirovanými manuálem k Motorolám M68k [1].

4.4.1 Datový registr - přímá adresace

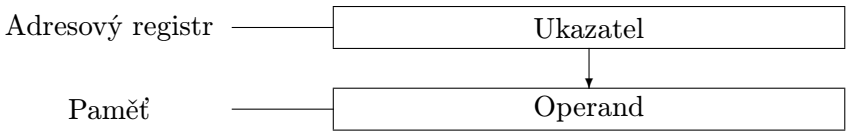
Operand instrukce je přímo uložen v datovém registru. Syntaxe tohoto adresovacího režimu v jazyce symbolických instrukcí je `Dn`, kde n je číslo datového registru.

4.4.2 Adresový registr - přímá adresace

Operand instrukce je přímo uložen v adresovém registru. Syntaxe tohoto adresovacího režimu v jazyce symbolických instrukcí je `An`, kde n je číslo adresového registru.

4.4.3 Adresový registr - nepřímá adresace

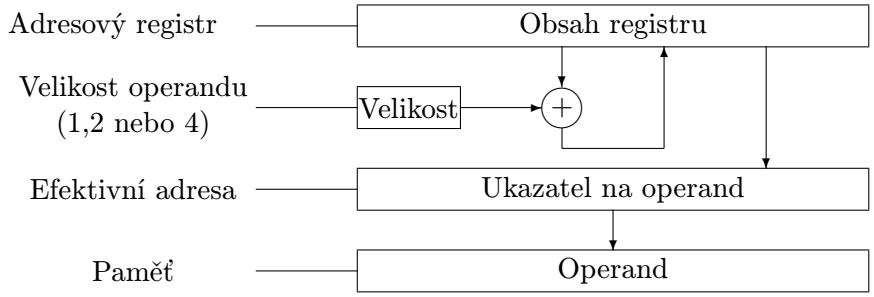
Operand této instrukce je uložen v paměti. Adresa operandu je v adresovém registru. Syntaxe tohoto adresovacího režimu je `(An)`.



Obrázek 4.5: Adresový registr - nepřímá adresace

4.4.4 Adresový registr - nepřímá adresace s postinkrementací

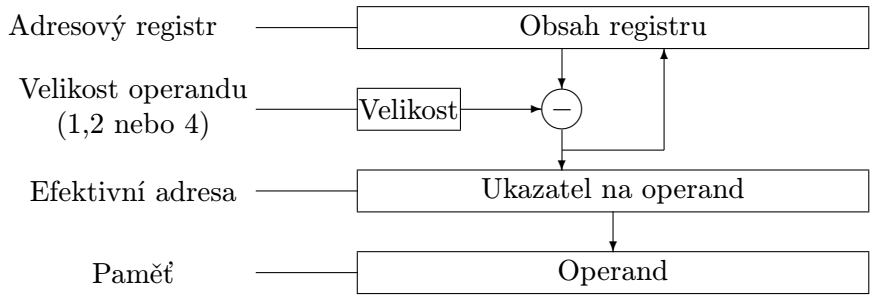
Operand této instrukce je uložen v paměti. Adresa operandu je v adresovém registru. Po provedení operace se adresa v adresovém registru zvýší o hodnotu jedna, dvě nebo čtyři, v závislosti na velikosti operace (byte, slovo nebo dvojslovo) Syntaxe tohoto adresovacího režimu je `(An)+`.



Obrázek 4.6: Adresový registr - nepřímá adresace s postinkrementací

4.4.5 Adresový registr - nepřímá adresace s predekrementací

Podobně jako u předešlého adresovacího módu je i zde uložen operand v paměti a jeho adresa je vypočítávána z obsahu adresového registru. Tentokrát je ale obsah registru nejprve snížen o hodnotu jedna, dvě nebo čtyři a teprve výsledek této operace je ukazatelem do paměti. Výsledná upravená efektivní adresa je pak uložena zpět do adresového registru. Syntaxe tohoto adresovacího režimu je `-(An)`.

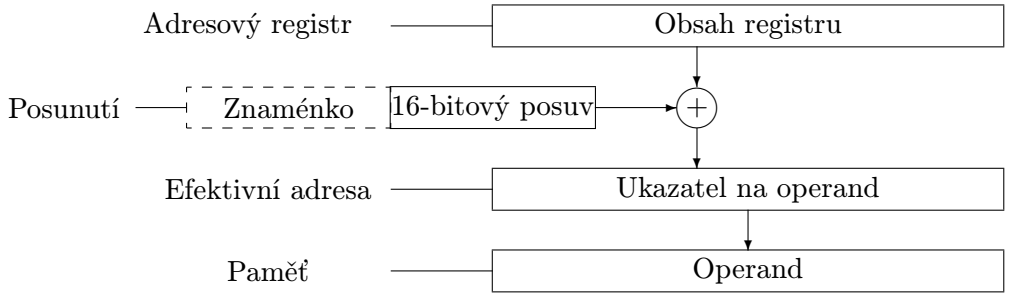


Obrázek 4.7: Adresový registr - nepřímá adresace s predekrementací

4.4.6 Adresový registr - nepřímá adresace s posunutím

V tomto případě je adresa operandu v paměti vypočtena jako součet hodnoty v adresovém registru a znaménkově rozšířené 16-bitové hodnoty, která je parametrem této adresace spolu s identifikátorem adresového registru. Syntaxe tohoto adresovacího režimu je `(d16,An)`, kde `d16` je 16-bitový posun adresy.

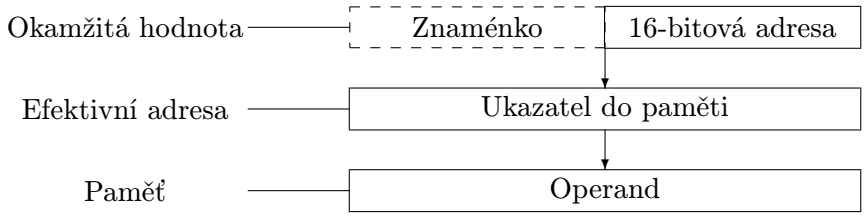
V tomto adresovacím režimu je možné místo adresového registru použít i programový čítač (PC).



Obrázek 4.8: Adresový registr - nepřímá adresace s posunutím

4.4.7 Absolutní adresace - krátká

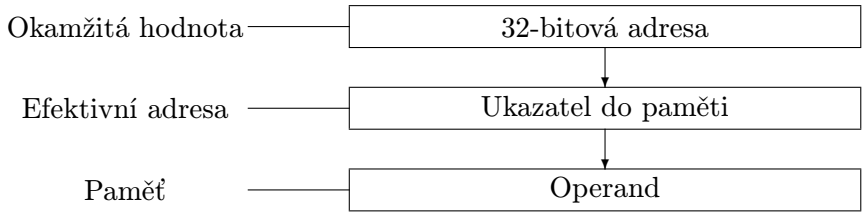
Při tomto způsobu adresace je operand opět v paměti a jeho adresa je určena 16-bitovou hodnotou, která znaménkově rozšířená na 32 bitů. V jazyce symbolických instrukcí se tato adresace značí následující syntaxí: `(xxx).W`, kde `xxx` je šestnáctibitová adresa paměti.



Obrázek 4.9: Absolutní adresace - krátká

4.4.8 Absolutní adresace - dlouhá

Tento adresovací režim je obdobou předešlého s tím rozdílem, že adresa je zadána jako 32-bitová hodnota. Syntaxe vypadá `(xxx).L`.



Obrázek 4.10: Absolutní adresace - dlouhá

Dalšími adresovacími režimy mikroprocesorů M68k, jako indexované adresy, se tato práce zatím nezabývá.

4.5 Systémový zásobník

Ukazatelem na systémový zásobník je osmý adresový registr `A7`. Motoroly M68k nedisponují standardními instrukcemi typu *push* a *pop*. Pro ukládání a načítání ze/do zásobníku se používají přesuny v kombinaci s jedním z nepřímých adresovacích módů – postinkrementačním, nebo předekrementačním (viz sekce 4.4.4 a 4.4.5). Při použití zásobníkového ukazatele v tomto adresovacím módu je adresa vypočítávána stejně jako u kteréhokoliv jiného adresového registru s tou vyjímkou, že i při práci s operandem velikosti byte je adresa zvýšena nebo snížena o hodnotu dvě (nikoliv o jedna, jak by tomu mělo být). Toto opatření udržuje zásobníková data zarovnaná na slova, což zvyšuje efektivitu jeho použití.

Zásobník lze sice používat díky této adresovací volnosti tak, aby rostl směrem k vyšším adresám, stejně jako k adresám nižším. Některé instrukce však pracují samy se zásobníkem (například `RTS` - návrat z podprogramu odebírá ze zásobníku ukazatel návratové adresy), je tedy konvencí, aby zásobník rostl směrem k nižším adresám.

4.6 Příznakový registr

Spousta instrukcí ovlivňuje podle výsledku provedené operace příznakový registr. Pro příklad logické instrukce jako `AND` nebo `OR` ovlivňují bity `Z` (`Zero`) a `N` (`Negative`) podle svého výsledku a nulují bity `V` (`Overflow`) a `C` (`Carry`). Oproti tomu aritmetické instrukce jako `ADD`, `SUB` ovlivňují všech pět bitů příznakového registru podle svého výsledku.

Podle akutálního stavu jednotlivých bitů příznakového registru je řízena funkce některých instrukcí, především podmíněných skoků. U těchto instrukcí je podmínka určena dvěma písmeny (v případě podmínky `True` a `False` jedním písmenem) přímo v názvu symbolické instrukce. Pravdivost podmínky je vypočtena podle tabulky 4.1.

Zkratka	Podmínka	Testování
T[1]	True	1
F[1]	False	0
HI	High	$\overline{C} \wedge \overline{Z}$
LS	Lower or Same	$C \vee Z$
CC (HI)	Carry Clear	$\overline{C}$
CS (LO)	Carry Set	C
NE	Not Equal	$\overline{Z}$
EQ	Equal	Z
VC	Overflow Clear	$\overline{V}$
VS	Overflow Set	V
PL	Plus	$\overline{N}$
MI	Minus	N
GE	Greater or Equal	$N \wedge V \vee \overline{N} \wedge \overline{V}$
LT	Less Than	$N \wedge \overline{V} \vee \overline{N} \wedge V$
GT	Greater Than	$N \wedge V \wedge \overline{Z} \vee \overline{N} \wedge \overline{V} \wedge \overline{Z}$
LE	Less or Equal	$Z \vee N \wedge \overline{V} \vee \overline{N} \wedge V$

Tabulka 4.1: Testování podmínek z CCR

[1] Není možné použít pro instrukci podmíněného skoku `Bcc`

Kapitola 5

Přizpůsobení LLVM pro instrukční sadu M68k

LLVM je dostatečně modulární nástroj k tomu, aby bylo možné doplnit jej o informace o architektuře, kterou doposud nepodporuje. Výstupním formátem překladu se zabývá část back-end (viz 2.7), která se stará o správný překlad z LLVM mezikódu na kód vybrané architektury. Tento cílový kód může být jak ve formátu čitelném pro lidi (symbolické instrukce), tak binární. Tato práce se však zabývá pouze překladem na symbolické instrukce. Back-endy již napsané a podporované LLVM se nachází v adresářové struktuře LLVM ve složce `lib/Target/`. Prvním krokem při tvorbě nového back-endu je tedy vytvoření podsložky v tomto umístění, která bude obsahovat všechny soubory back-endu.

Základním prvkem back-endu je třída odvozená od třídy `TargetMachine`, která poskytuje základní charakteristiku architektury. Třída by měla být definována v souboru nazvaném `xxxTargetMachine.cpp`, kde `xxx` je indentifikátor architektury (v tomto konkrétním případě je to `M68kTargetMachine.cpp`). K tomuto modulu samozřejmě patří i příslušný hlavičkový soubor.

Dalším krokem je definice registrové sady. Zásadní část kódu týkajícího se registrů architektury je možné vygenerovat pomocí nástroje TableGen 3.2 z .td souboru, zde je to `M68kRegisterInfo.td`. Poté je třeba vytvořit podtřídu z `TargetRegisterInfo` (`M68kTargetRegisterInfo.cpp` a `.h`), která umožňuje dodefinovat potřebné součásti, které dosud nelze zpracovat TableGenem, jako interreakce mezi registry, definice rezervovaných registru a podobně.

Podobně jako u registrové sady také instrukční sada je primárně generovaná pomocí TableGenu, a to ze souboru `M68kInstructionInfo.td`, který obsahuje definice jednotlivých instrukcí, jejich operandy, prováděnou operaci i vzor, podle nějž je instrukce mapována při selekci instrukcí. Tento soubor ještě dále spolupracuje s `M68kInstructionFormats.td`, který popisuje možné formáty instrukcí. Stejně jako v případě registrů není možné vše vystihnout TableGenem, chybějící informace, jako práce se zásobníkem nebo přesuny mezi registry, se vyskytují v souboru `M68kInstructionInfo.cpp` (resp. `M68kInstructionInfo.h`).

LLVM provádí překlad do cílového jazyka tak, že svůj mezikód nejprve převede na tzv. `DAG` (`Directed Acyclic Graph`), jehož uzly jsou základní prováděné operace a do nehož se mapují instrukce cílové architektury, v nejjednodušším případě ze vzoru, který má instrukce definovaný v souboru `xxxInstructionInfo.td`. Které operace architketura podporuje, případně jak se má back-end vypořádat s těmi, které podporovány nejsou, je popsáno v souboru `M68kISelLowering.cpp` (a jeho hlavičkovém souboru). Soubor `M68kISelDAGToDAG.cpp` po-

pisuje především jak zvolit instrukce, jejichž vzor nelze popsat ve formátu TableGenu (například adresace s posunutím).

Konverze z LLVM reprezentace přeloženého kódu do cílového jazyka symbolických instrukcí [5] je prováděna pomocí metod třídy `M68kAsmPrinter`, která je definována v souboru `AsmPrinter/M68kAsmPrinter.cpp` a využívá textové řetězce z definic instrukční sady z `M68kInstructionInfo.td`.

Volitelně pak může být implementována podpora překladu za běhu programu JIT (`Just-In-Time` kompilace) jako podtřída `TargetJITInfo`. Taktéž může být definována podpora různých typů procesoru dané architektury, tzv. *Subtarget*ů.

Tento bakalářský projekt JIT nepodporuje a stejně tak neimplementuje různé typy procesorů, pouze se omezuje na vlastnosti shodné a použitelné pro všechny procesory rodiny Motorola 68000. Nicméně back-end pro M68k definuje podtřídu `M68kSubtarget`, takže do budoucna je možné jej snadno rozšířit o podporu různých procesorů. Aby však bylo možné zkompilovat back-end s prázdným řetězcem podporovaných *subtarget*ů, bylo třeba upravit soubor `utils/TableGen/SubtargetEmitter.cpp`, aby touto „chybou“ nepřerušoval překlad.

5.1 Překlad s podporou nového back-endu

LLVM se překládá pomocí `Makefile`, který je generován konfiguračním skriptem `configure`. V tomto konfiguračním skriptu jsou vypsány back-endy, kterých se překlad týká. Proto pro správný překlad nového back-endu se nabízejí dvě možnosti. Jednou z nich je ručně upravit přímo konfigurační skript a do proměnné `TARGETS_TO_BUILD` přidat nový cíl. Druhá možnost je upravit autokonfigurační skript, který je zkonstruovaný k vygenerování souboru `configure`. Soubor potřebný k úpravě je `autoconf/configure.ac`. V něm je opět nutné přidat identifikátor nového back-endu všude, kde se vyskytují ostatní.

Po úspěšné modifikaci autokonfiguračního skriptu nebo konfiguračního skriptu je třeba vygenerovat `Makefile`. Aby nový `Makefile` přeložil přidaný back-end, je třeba spouštět konfiguraci s přepínačem `--enable-targets=BACK_END`, kde `BACK_END` je identifikátor nové architektury.

V případě, že byl modifikován autokonfigurační skript, pro M68k lze nový back-end přeložit následující kombinací příkazů z kořenového adresáře LLVM:

```
cd autoconf
./AutoRegen.sh
cd ..
./configure --enable-targets=m68k
make
```

Tabulka 5.1: Překlad LLVM

5.2 Registrace back-endu

Aby bylo možné nový cíl používat jinými nástroji LLVM, je třeba jeho identifikátor zaregistrovat v `TargetRegistry`.

Nejjednodušší způsob, jak toho docílit, je vytvořit modul, v tomto případě pojmenovaný `M68kTargetInfo.cpp`, v něm definovat globální objekt typu `Target` a využít makro `RegisterTarget` pro registraci nového cíle do `TargetRegistry`. Registrace Motoroly 68000:

```
Target llvm::TheM68kTarget;

extern "C" void LLVMInitializeM68kTargetInfo() {
  RegisterTarget<> X(TheM68kTarget, "m68k", "M68k");
}
```

Tabulka 5.2: Registrace M68k

5.3 Target Machine

LLVMTargetMachine je rodičovská třída, z níž jsou odvozeny všechny ostatní třídy reprezentující jednotlivé architektury. Je tomu tak samozřejmě i v případě M68kTargetMachine. Tato třída zapouzdřuje informace o registrové sadě, instrukční sadě a jiné nezbytné údaje o architektuře. Proto implementuje abstraktní metody, které vrací tyto informace o instrukcích, registrech, zásobníku apod., jako `GetInstrInfo()`, `GetFrameInfo()` nebo `GetRegisterInfo()`.

Mimo tyto metody je v objektu TargetMachine definován i tzv. *TargetDescription string*, který specifikuje informace jako velikost ukazatelů, zarovnání paměti, nebo zda je architektura *little-endian*, nebo *big-endian*. Pro Motoroly vypadá konstruktor `TargetMachine` následovně:

```
M68kTargetMachine::M68kTargetMachine(const Target &T, const std::string &TT,
                                     const std::string &FS)
  : LLVMTargetMachine(T, TT),

  Subtarget(TT, FS),
  DataLayout("E-p32:32"),
  InstrInfo(*this),
  FrameInfo(TargetFrameInfo::StackGrowsDown, 4, 0),
  TLInfo(*this),
  InstrItins(Subtarget.getInstrItineraryData())
  {}
```

Tabulka 5.3: M68k TargetMachine konstruktor

Parametrem `DataLayout` je výše zmíněný *TargetDescriptor string*. Písmeno velké **E** zde vyjadřuje, že architektura je typu *big-endian* (v opačném případě by šlo o písmeno malé **e**), za pomlčkou písmeno **p** naznačuje, že budou následovat informace o ukazatelích. První číslo je velikost ukazatele, druhé je zarovnání v paměti.

5.4 Registrová sada

Aby byla podporována registrová sada Motorol, je třeba definovat třídu `M68kRegisterInfo`. V hlavičkovém souboru této třídy je vložen modul vygenerovaný TableGenem ze souboru `M68kRegisterInfo.td`. Vstupní soubor TableGenu obsahuje definice jednotlivých registrů, jejich assemblerovou textovou reprezentaci. Dále jsou zde definovány třídy registrů, jejich velikosti, případně subregistry a omezení alokace rezervovaných registrů. V případě Motoroly jsou TableGenem definovány datové a adresové registry, jako `DataRegs` a `AddrRegs`, programový čítač jako jediný registr třídy `ProgCounter` a příznakový registr, jako registr třídy `CCR`. Z těchto tříd TableGen vygeneruje třídy `DataRegsRegClass`, `AddrRegsRegClass`, `ProgCounterRegClass` a `CCRRegClass`.

Jelikož je vlice špatně sehnatelné ABI (*Application binary interface*) pro Motoroly, které by obyčejně popisovalo konvence, jako konvence pro volání funkcí, *frame pointer* (ukazatel na zásobníkový rámec) nebo *callee-/caller-saved registry* (konvence, které registry ukládá volající procedura, a které volaná), některé tyto informace je možné odvodit z jiných překladačů, nebo je možné je vhodně zvolit. Ukazatel na zásobník je jasně daný, a to registr `A7`, za *frame pointer* byl zvolen registr `A5`, podle zdrojových souborů překladače GCC jsou *caller-saved* registry `D0`, `D1`, `A0`, `A1` a `A7`, ostatní jsou *callee-saved*. Zvolená volací konvence je více popsána v sekci 5.6. Jelikož tyto informace nelze zapsat pro TableGen, aby je měl back-end k dispozici, implementuje třída `M68kRegisterInfo` metody, které tyto údaje vrací. Konkrétně jde o tyto metody:

- `getCalleeSavedRegs` – vrací seznam registrů, které jsou ukládány volanou funkcí,
- `getCalleeSavedRegClasses` – vrací seznam tříd registrů, ukládaných volanou funkcí, prvky seznamu korespondují s prvky vrácenými předešlou metodou,
- `getReservedRegs` – vrací seznam rezervovaných registrů (ukazatel na zásobník, registr pro návratovou adresu atd.),
- `getFrameRegister` – návratovou hodnotou je číslo registru, který se používá, jako ukazatel na rámec zásobníku pro danou funkci,
- `getRARegister` – vrací číslo *Return-address registru*, to jest registru, který uchovává návratovou adresu z funkce,
- `getDwarfRegNum` – navrací tzv. `DWARF` očíslování registru, `DWARF standard` jehož použití je rovněž definováno v ABI, v back-endu pro Motorolu proto tato metoda pouze vrací nazpátek pořadí registru.

Mimo tyto metody třída `M68kRegisterInfo` definuje i funkce, které upravují generovaný DAG a tím i cílový kód. Tyto funkce se starají například o ukládání *calee-saved* registrů, nebo nahrazování abstraktních *frame pointerů*, se kterými se v průběhu kompilace pracuje, konkrétními registry. Jde o metody jako:

- `emitPrologue`
- `emitEpilogue`
- `eliminateFrameIndex`

⋮

5.5 Instrukční sada

V jisté fázi překladu je zdrojový kód v podobě *SelectionDAG*u – graf, který se použije na výběr instrukcí podle jejich vzorů, jeho uzly jsou tak zvané *SDNodes* (od *SelectionDAG Nodes*), které definují nějakou operaci, její operandy, datové typy a další vlastnosti operace (například práce s pamětí). Přehled všech typů *SDNodes* je v souboru `include/llvm/CodeGen/SelectionDAGNodes.h`.

Potřebné informace o instrukcích, které cílová architektura používá, nutné k namapování těchto instrukcí do *SelectionDAG*u, jsou definovány v souborech `M68kInstrFormats.td` a `M68kInstrInfo.td`.

- `M68kInstrFormats.td` popisuje formáty instrukcí využívaných následujícím *Target descriptorem* (.td).
- `M68kInstrInfo.td` definuje jednotlivé instrukce, jejich operandy, datové typy, prováděné operace a podobně.

5.5.1 Formáty instrukcí

`xxxInstructionFormats.td` má význam především při tvorbě back-endu, který dokáže generovat instrukce i v binární podobě pro cílovou architekturu. Tehdy je možné s výhodou využít různých formátů unstrukcí například pro různé typy binárních kódování instrukcí. Ve výše zmíněném souboru jsou popsány šablony, podle nichž TableGen později generuje cílové instrukce. Výsledkem této práce je generátor kódu, jehož výstupem je pouze člověkem čitelný kód. Proto přestože v souboru `M68kInstrFormats.td` je předpřipravených několik základních formátů instrukcí, v souboru `M68kInstrInfo.td` je používán pouze jeden – `M68kInst`, který je odvozený přímo z třídy `Instruction`. Tento formát instrukce definuje pouze jmenný prostor a parametry jsou vstupní a výstupní operandy, textový řetězec pro generovaný výstup a vzor pro mapování do `SelectionDAG`u.

5.5.2 Popis instrukcí

Soubor `M68kInstrInfo.td` obsahuje popis jednotlivých instrukcí s využitím šablony v podobě instrukčního formátu `M68kInst` popsaného v sekci 5.5.1. Příkladem definované instrukce je následující instrukce typu *load*, která načítá dvojslovo z adresy uložené v adresovém registru do datového registru.

```
// 32 from AddrReg indirect to DataReg
def MOVE32ArinDr:M68kInst<(outs DataRegs:$dst0),
       (ins AddrRegs:$src0),
"move.l ($src0), $dst0",
[(set DataRegs:$dst0, (load AddrRegs:$src0))]>;
```

Tabulka 5.4: Instrukce MOVE

Klíčové slovo `def` uvozuje začátek definice instrukce, následuje jednoznačný identifikátor, formát instrukce (definovaný v soboru `M68kInstructionFormats.td`) a mezi znaky `<` a `>` jsou parametry instrukce.

Prvním parametrem jsou výstupní argumenty, v tomto případě jde o datový registr, do něhož se bude načtená hodnota ukládat. Druhý parametr jsou výstupní parametry,

tedy adresový registr, který se využije pro získání efektivní adresy operandu. Za nimi je řetězec, který se využije pro generování instrukce v textové podobě. Proměnné uvozené znakem `$` budou nahrazeny konkrétními operandy. Posledním parametrem je vzor, který jasně definuje, co daná instrukce provádí za operace. Pro instrukci `MOVE32ArinDr` tedy označuje přednastavenými operacemi `set` a `load`, že bude nejprve načítáno z adresy uložené v adresovém registru a následně ukládáno do datového registru.

Kromě takto jednoznačně určených instrukcí je možné v souboru `M68kInstrInfo.td` nalézt i definice, které neodpovídají žádné konkrétní instrukci. Tyto většinou plní nějakou pomocnou funkci. Příkladem může být definice `SELECT_CC_Int_ICC`.

```
let usesCustomDAGSchedInserter = 1 in {   // Expanded by the scheduler.
  def SELECT_CC_Int_ICC
   : M68kInst<(outs DataRegs:$dst), (ins DataRegs:$T, DataRegs:$F,
                                     i32imm:$Cond),
            "; SELECT_CC_Int_ICC PSEUDO!",
            [(set DataRegs:$dst, (m68kselecticc DataRegs:$T, DataRegs:$F,
                                  imm:$Cond))]>;
}
```

Tabulka 5.5: Pseudoinstrukce

Na tomto příkladu je zajímavé, že tato instrukce (přesněji její textová reprezentace) by se neměla nikdy objevit ve finálně vygenerovaném kódu. V řetězci, který tuto textovou reprezentaci definuje, je pouze informace, že jde o tzv. *pesudoinstrukci* a je zde pouze aby bylo zamezeno chybovým hlášením. Aby byla tato pseudoinstrukce na konci překladu nahrazena korektní posloupností jiných instrukcí, je zařízeno nastavením příznaku `usesCustomDAGSchedInserter`. Ten způsobí, že těsně před tím, než jsou instrukce převedeny z `DAG`u do textové podoby, je pro tuto instrukci zavolána metoda `EmitInstrWithCustomInserter` z modulu `M68kTargetLowering`. Tato technika byla převzata z back-endu pro procesor `SPARC` a způsobí zde, že operace `select_CC`, která vybírá na svůj výstup ze dvou hodnot podle zadané podmínky (a kterou architektura `SPARC` ani `M68k` nepodporují), bude rozložena na porovnávací instrukci `CMP`, následovanou podmíněným skokem.

5.5.3 Transformační funkce

Kromě instrukcí samotných jsou v souboru `M68kInstrInfo.td` definovány i některé pomocné „funkce“, které jsou využívány při definici instrukcí. Jde například o transformační funkci, která vrací pouze méně nebo více významné slovo dvojslova, predikát, který vrací boolovskou pravdu, pokud testované celé číslo odpovídá 16-bitové hodnotě a jiné:

```
// Transformation Function - get the lower 8 bits.
def LO8 : SDNodeXForm<imm, [{
  return getI32Imm((unsigned)N->getZExtValue() & 0xFF);
}]>;

// Node immediate fits as zero extended number (1-8) on target immediate.
def immZExt1_8  : PatLeaf<(imm), [{
    return ((N->getZExtValue() > 0) &&
    (N->getZExtValue() < 9));
}]>;
```

Tabulka 5.6: Transformační funkce

Transformační funkce `LO8` vrací pouze nejnižších osm bitů z celočíselné hodnoty, predikát immZExt1_8 vrací boolovskou pravdu, pokud testovaná hodnota je v rozmezí 1-8 (využíváno pro instrukce rotací `ROL` a `ROR`).

5.5.4 Adresový operand

Další možností usnadnění implementace některých instrukcí je definice vlastního operandu. U M68k je využita tato možnost k definování adresového operandu pro nepřímý adresovací mód s posunutím (viz 4.4.6).

```
// Address operands
def memri : Operand<iPTR> {
  let PrintMethod = "printMemRegImm";
  let MIOperandInfo = (ops i32imm:$imm, ptr_rc:$reg);
}

// M68k specific addressing mode.
def iaddr  : ComplexPattern<iPTR, 2, "SelectAddrImm",    [], []>;

// 32 from AddrReg with displacement to DataReg
def MOVE32ArdisDr : M68kInst<(outs DataRegs:$dst0), (ins memri:$src0),
"move.l $src0, $dst0",
[(set DataRegs:$dst0, (load iaddr:$src0))]>;
```

Tabulka 5.7: Adresový operand

Na příkladu je nejprve definován adresový operand `memri`, který se skládá ze dvou částí – registru a 32-bitové hodnoty. V případě, že je zvolena instrukce s tímto operandem, k jeho výstupu do textové podoby se použije funkce `printMemRegImm` z modulu `M68kAsmPrinter`. Další definice je `iaddr`, která reprezentuje tento adresový mód ve *vzoru* instrukce a říká, že k rozpoznání, zda je možné instrukci s výhodou použít v adresovacím režimu s posunutím, bude použita funkce `SelectAddrImm` z modulu `M68kISelDAGToDAG`.

5.5.5 Multitřídy

Aby se minimalizovala nutnost kopírování stejného kódu u instrukcí s podobnými nebo stejnými typy operandů, jen jinou operací, definuje *Target descriptor* `M68kInstrInfo.td` několik multitříd (viz sekce 3.2 a kód 3.3). Je celkem pět typů těchto multitříd. Které kombinace operandů zastupují, vyjadřují tabulky 5.8, 5.9, 5.10, 5.11 a 5.12. V těchto tabulkách symbol `Op` reprezentuje libovolnou operaci (aritmetickou, logickou a podobně). Jak je vidět, i v těchto tabulkách se kombinace operandů hodně opakují, ale i přesto je jejich užitím ušetřeno mnoho řádků stejného kódu. Komplikace při užití multitřídy nastává ve chvíli, kdy je třeba zahrnout do ní jak instrukce dvouadresné, tak i jiné (přesněji instrukce, které mají vynulovaný příznak isTwoAddress). Uvnitř definice multitřídy totiž nelze měnit hodnoty těchto příznaků a je třeba je definovat při instancování multitřídy, tedy pro všechny instrukce, které multitřída vygeneruje. Přesně to je případ multitřídy `M68kInst_1`, `M68kInst_2` a `Q32Inst`. Problém je způsoben zápisem některých výsledků operací do paměti. Takovéto instrukce jsou považovány za instrukce typu `store` a není možné je chápat jako dvouadresné. V tabulkách jsou tyto instrukce odděleny prázdným řádkem a v samotném kódu je takováto třída definována dvěma různými multitřídami (tedy například `M68kInst_1` je ve skutečnosti dvě třídy `M68kInst_1TwoAdd` a `M68kInst_1Others`).

Adresovací mód	**Operandy**
Dn	$Dx = Dx\ Op\ Dy$
An	$Dx = Dx\ Op\ Ax$
(An)	$Dx = Dx\ Op\ (Ax)$
$(d16, An)$	$Dx = Dx\ Op\ (d16, Ax)$
$\#imm$	$Dx = Dx\ Op\ \#imm$
$(xxx).L$	$Dx = Dx\ Op\ (xxx).L$
$(xxx).W$	$Dx = Dx\ Op\ (xxx).W$
(An)	$(Ax) = Dx\ Op\ (Ax)$
$(d16, An)$	$(d16, Ax) = Dx\ Op\ (d16, Ax)$
$(xxx).L$	$(xxx).L = Dx\ Op\ (xxx).L$
$(xxx).W$	$(xxx).W = Dx\ Op\ (xxx).W$

Tabulka 5.8: Operandy multitřídy M68kInst_1

Z této třídy jsou odvozeny například aritmetické instrukce jako `ADD` nebo `SUB`

Adresovací mód	Operandy
Dn	$Dx = Dx\ Op\ Dy$
(An)	$Dx = Dx\ Op\ (Ax)$
$(d16, An)$	$Dx = Dx\ Op\ (d16, Ax)$
$\#imm$	$Dx = Dx\ Op\ \#imm$
$(xxx).L$	$Dx = Dx\ Op\ (xxx).L$
$(xxx).W$	$Dx = Dx\ Op\ (xxx).W$
(An)	$(Ax) = Dx\ Op\ (Ax)$
$(d16, An)$	$(d16, Ax) = Dx\ Op\ (d16, Ax)$
$(xxx).L$	$(xxx).L = Dx\ Op\ (xxx).L$
$(xxx).W$	$(xxx).W = Dx\ Op\ (xxx).W$

Tabulka 5.9: Operandy multitřídy M68kInst_2

Multitřída určena primárně pro definici logických instrukcí, jako `AND`, `OR`.

Adresovací mód	Operandy
Dn	$Dx = Dx\ Op\ Dy$
An	$Dx = Dx\ Op\ Ax$
(An)	$Dx = Dx\ Op\ (Ax)$
$(d16, An)$	$Dx = Dx\ Op\ (d16, Ax)$
$\#imm$	$Dx = Dx\ Op\ \#imm$

Tabulka 5.10: Operandy multitřídy M68kInst_3

Multitřída společná pro instrukce, jejichž cílový operand patří do třídy adresových registrů – `ADDA`, `SUBA` atd.

Adresovací mód	Operandy
(An)	$(Ax) = (Ax)\ Op\ \#imm$
$(d16, An)$	$(d16, Ax) = (d16, Ax)\ Op\ \#imm$
$(xxx).L$	$(xxx).L = (xxx).L\ Op\ \#imm$
$(xxx).W$	$(xxx).W = (xxx).W\ Op\ \#imm$

Tabulka 5.11: Operandy multitřídy M68kInst_4

Ze třídy `M68kInst_4` jsou generovány instrukce, které přijímají druhý operand jen okamžitnou celočíselnou hodnotu, jako `ADDI` nebo `SUBI`.

Adresovací mód	Operandy
Dn	$Dx = Dx\ Op\ \#imm(1-8)$
An	$Ax = Ax\ Op\ \#imm(1-8)$
(An)	$(Ax) = (Ax)\ Op\ \#imm(1-8)$
$(d16, An)$	$(d16, Ax) = (d16, Ax)\ Op\ \#imm(1-8)$
$(xxx).L$	$(xxx).L = (xxx).L\ Op\ \#imm(1-8)$
$(xxx).W$	$(xxx).W = (xxx).W\ Op\ \#imm(1-8)$

Tabulka 5.12: Operandy multitřídy Q32Inst

Druhý operand instrukcí, odvozených od této multitřídy, je okamžitá hodnota v rozmezí 1-8. Jde o tzv. *quick* instrukce, jako `ADDQ` nebo `SUBQ`.

Poslední, co je třeba pro instrukční sadu napsat, je modul `M68kInstrInfo.cpp`. Ten definuje několik důležitých funkcí:

- `copyRegToReg` říká, jakou instrukci použít, pokud chce kompilátor zkopírovat obsah jednoho registru do jiného. Tato operace nelze popsat žádným vzorem v TableGenu, proto jsou tyto instrukce v Target descriptoru definovány bez vzoru,
- `isMoveInstr` vrací boolovskou hodnotu, zda je daná operace přesunem a identifikuje registry, mezi nimiž k přesunu dochází. Používá se pro eliminaci pomocných instrukcí typu „`MOVE D1, D1`“,
- `storeRegToStackSlot` a s ní související,
- `loadRegFromStackSlot` přidává instrukci, nebo instrukce, které slouží k uložení/načtení hodnoty do/ze zásobníku z/do registru.

5.6 Volací konvence

Konvence volání podporocedur a návratu z nich by měla být definována v ABI architektury. Toto je pro architekturu *Motorola 68000* velice špatně sehnatelné. Byla proto konvence zvolená. Parametry vstupující do volané funkce jsou předávány v datových registrech `D0` až `D5`. Pokud je parametrů více, ostatní jsou předány na systémovém zásobníku. Návratová hodnota je předávána v registru `D0`, případně ve dvojici registrů `D0` a `D1`. Tento princip volání funkcí je popsán v souboru `M68kCallingConv.td`, který je využíván modulem `M68kISelLowering.cpp`, který mimo jiné implementuje funkce pro obsluhu volání funkcí a návratu z nich. Konkrétně jde o funkce `LowerFormalArguments` a `LowerReturn`.

5.7 Výběr instrukcí

V době překladu, kdy je zdrojový kód převeden na `SelectionDAG`, obsahuje tento graf uzly, které reprezentují operace, které cílová architektura nemusí podporovat. O převod těchto uzlů na podporované operace se stará modul `M68kISelDAGToDAG.cpp`. Při tomto převodu `SelectionDAG`u na graf obsahující jen nativní operace je volána funkce `Select`, které je předán zkoumaný uzel. Funkce obvykle obsahuje konstrukci typu `switch`, kde prochází jednotlivé typy uzlů, které je třeba potenciálně upravit. V případě M68k je například

připravena větev pro kontrolu uzlů typu `load` pro podporu načítání dat z paměti v predekrementačním a postinkrementačním adresovacím módu. Tento mód zatím bohužel nefunguje, protože nejspíš ještě není LLVM podporován.

5.8 Legalizace *SelectionDAG*u

Tato fáze překladu úzce souvisí se selekcí instrukcí a stará se o to, aby *SelectionDAG* obsahoval pouze legální operace pro danou architekturu. Které to jsou, je popsáno v souboru `M68kISelLowering.cpp`, který definuje třídu `M68kTargetLowering`. V jejím konstruktoru jsou nejprve vypsány třídy registrů a datové typy s nimi spojené:

```
addRegisterClass(MVT::i32, M68k::AddrRegsRegisterClass);
addRegisterClass(MVT::i32, M68k::DataRegsRegisterClass);
```

Tabulka 5.13: Legalizace *SelectionDAG*u, třídy registrů

Následuje výpis akcí, které se mají provést v případě, že selektor instrukcí narazí na uzel s nepodporovanou operací, nebo nepodporovaným typem operandu. Pro tento účel je k dispozici několik funkcí pro jednotlivé typy uzlů, jako `setOperationAction` pro obecnou operaci, nebo `setTruncStoreAction` pro ukládání do paměti s ořízntutím apod.

Všechny tyto funkce se volají se třemi parametry. První je identifikátor operace daného uzlu, druhý je datový typ operace a třetí je akce, která se pro tuto kombinaci (operace - datový typ) má provést. Je možné vybrat jednu z následujících akcí:

- `Promote` – daný datový typ není podporován a měl by být rozšířen na větší datový typ,
- `Expand` – Operace není podporována a měla by být rozložena na posloupnost elementárnějších operací,
- `Custom` – Pokud nestačí rozložit operaci na jednodušší, nebo povýšit datový typ, může být operace zpracována vlastním kódem. Pro takovéto uzly je volána funkce `LowerOperation`,
- `Legal` – Tato akce znamená, že daná operace je pro daný datový typ podporována architekturou a je možné přistoupit k selekci instrukce.

Následující kód legalizuje načítání z paměti a ukládání do paměti čtyřbytový operand s predekrementačním a postinkrementačním módem.

```
setIndexedLoadAction(ISD::POST_INC, MVT::i32, Legal);
setIndexedStoreAction(ISD::POST_INC, MVT::i32, Legal);
setIndexedLoadAction(ISD::PRE_DEC, MVT::i32, Legal);
setIndexedStoreAction(ISD::PRE_DEC, MVT::i32, Legal);
```

Tabulka 5.14: Legalizace *SelectionDAG*u, podporované instrukce

Druhý příklad ukazuje vyřešení porovnávání a podmíněných skoků:

```
setOperationAction(ISD::SELECT, MVT::i32, Expand);
setOperationAction(ISD::SETCC, MVT::i32, Expand);
setOperationAction(ISD::BRCOND, MVT::Other, Expand);
setOperationAction(ISD::BRIND, MVT::Other, Expand);
setOperationAction(ISD::BR_JT, MVT::Other, Expand);
setOperationAction(ISD::BR_CC, MVT::i32, Custom);
setOperationAction(ISD::SELECT_CC, MVT::i32, Custom);
```

Tabulka 5.15: Legalizace *SelectionDAG*u, nepodporované instrukce

LLVM výše uvedenými operacemi, jako SELECT, SETCC aj., řeší problematiku větvení programu, podmíněných skoků, porovnávání a podobně. Kód v příkladu akcemi `Expand` říká překladači, že architektura nepodporuje žádnou z těchto operací. Jedinou technikou, kterou Motorola pro tyto případy má, jsou podmíněné skoky. K těm se nejvíce hodí operace `BR_CC`. K úspěšnému překladu bohužel nestačí pouze tato jediná operace. Všechny tyto případy jdou však rozložit na kombinaci operací `BR_CC` a `SELECT_CC`. Jak tyto instrukce převést na kombinaci podporovaných instrukcí je pak dopsáno v metodách `LowerBR_CC` a `LowerSELECT_CC` s využitím generování pseudoinstrukce (viz 5.5.2 a příklad 5.5).

Kapitola 6

Závěr

Tato práce měla za úkol vytvořit back-end pro platformu LLVM, která umožní překládat programy do jazyka symbolických instrukcí pro mikroprocesory typu Motorola 68000. Předpokládaná podpora byla kompletní instrukční sada bez instrukcí pracujících v plovoucí řádové čárce.

Odevzdaný program sice nepodporuje celou instrukční sadu, ale implementuje většinu nejdůležitějších instrukcí, potřebných pro kompilaci základních programových konstrukcí. V této fázi je back-end omezen na aritmetické instrukce, logické instrukce a některé instrukce posuvů a rotací, všechny pro 32-bitové operandy. Výjimkou aritmetických instrukcí, které ještě nejsou zvládnuty, jsou operace násobení a dělení. Je to způsobeno komplikovanou adresací částí registrů v LLVM bez použití subregistrů. Zároveň je program schopen generovat instrukce jednoduchých přesunů 32-bitových operandů. Z adresovacích módů je podporována adresace přímá, nepřímá, nepřímá s posunutím a absolutní – dlouhá i krátká. Nejsou podporovány indexované adresovací režimy a nepřímá adresace s *updatem* (tedy s predekrementací a postinkrementací).

Z řídících instrukcí je implementována podpora především podmíněných a nepodmíněných skoků a volání subrutin a návratu z nich a s nimi spojené instrukce testování a porovnávání operandů. Díky tomu je možné přeložit základní konstrukce jako *if-else*, programové smyčky, nebo volání funkcí s libovolným množstvím 32-bitových parametrů a 32-bitovou návratovou hodnotou.

Literatura

[1] Kolektiv autorů: MOTOROLA M68000 FAMILY Programmer's Reference Manual [online]. `http://www.freescale.com/files/archives/doc/ref_manual/M68000PRM.pdf`, 1992 [cit. 2010-04-29].

[2] Kolektiv autorů: Oficiální stránky projektu LLVM [online]. `http://www.llvm.org`, 2010-04-27 [cit. 2010-04-29].

[3] Lattner, C.; Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, 2004.

[4] Meduna, A.; Lukáš, R.: Formální jazyky a překladače – Studijní opora. 2006.

[5] The MacHax Group: Instrukce Motorola 68000 [online]. `http://68k.hax.com/`, [cit. 2010-05-16].

Příloha A

Obsah CD

Přiložený disk obsahuje celou platformu LLVM ve verzi 2.6 upravenou pro podporu instrukční sady Motorola 68000 s již upraveným autokonfiguračním souborem. V dalším adresáři jsou zdrojové soubory textové části a zkompilovaná textová část ve formátu pdf.

Příloha B

Ukázka funkčnosti

Na CD je LLVM již předpřipraveno pro zkompilování s podporou nové instrukční sady. Pro přeložení stačí trojice příkazů:

```
./autoconf/AutoRegen.sh
./configure --enable-targets=m68k
make
```

Ve složce `llvm/test/CodeGen/M68k` je nachystáno několik testovacích souborů ve formátu LLVM mezikódu s ukázkami funkčních instrukcí a programových konstrukcí.

Pro spuštění překladu některého ze souborů je třeba použít dvojici nástrojů LLVM – `llvm-as` pro přeložení mezikódu do binárního kódu a `llc` pro překlad do assembleru cílové architektury. Je-li tedy záměr přeložit testovací soubor `if.ll` ve výše zmíněném adresáři, je možné to zařídit příkazy:

```
./Release/bin/llvm-as ./test/CodeGen/M68k/if.ll -o if.bc
./Release/bin/llc -march m68k if.bc
```

V hlavním adresáři se tímto vytvoří soubor `if.s`, který obsahuje přeložený zdrojový kód do assembleru Motoroly 68000. V následujících sekcích jsou popsány některé testovací soubory.

B.0.1 if.ll

Zde je předveden překlad klasické konstrukce `if-else`:

```
int f(int a, int b) {
if (a~> 0 )
  return a;
else
  return b;
}
```

Tabulka B.1: Program `if` v jazyce `C++`

Jelikož jsou parametry `a` a `b` předány funkci dle zvolené konvence (viz 5.6) v registrech `D0` a `D1` a návratová hodnota je vrácena v registru `D0`, vypadá přeložený kód následovně:

```
.file "./foo.bc"

.text
.global f
.func f
f
tst.l d0
bgt .BB1_2

.BB1_1                                                    # %entry
move.l d1, d0

.BB1_2                                                    # %entry
rts

.endfunc
```

Tabulka B.2: Program `if` v assembleru M68k

Instrukce `tst.l` porovná první parametr funkce s nulou, následuje posmíněný skok, který buďto zavolá návrat z funkce, protože v registru pro návratovou hodnotu je již správná proměnná, anebo vloží do registru `D0` druhý argument a funkcí ukončí.

B.0.2 loop.ll

Soubor `loop.ll` je vygenerován z kódu nekonečné smyčky a ukazuje funkčnost nepodmíněného skoku:

```
int f(int a) {
  for(;;)
    ;
  return a;
}
```

Tabulka B.3: Program `loop` v jazyce `C++`

Tato funkce je LLVM přeložena do jediné instrukce a jednoho návěští. Absence instrukce `rts` je způsobena optimalizacemi:

```
.file "./foo.bc"

.text
.global f
.func f
f

.BB1_1                                                    # %bb

bra .BB1_1

.endfunc
```

Tabulka B.4: Program `loop` v assembleru M68k

B.0.3 passing_args.ll

Zde je předvedeno předávání většího množství argumentů, než se dokáže vejít do datových registrů. Podle konvence (sekce 5.6) jsou tedy ostatní parametry předány na zásobníku:

```
int f(int a, int b, int c, int d, int e, int f, int g) {
  return a+b+c+d+e+f+g;
}
```

Tabulka B.5: Program `passing_args` v jazyce `C++`

V přeloženém kódu lze spatřit ukládání *callee-saved* registrů na zásobník a jejich následné obnovení. Instrukce `adda` je zde používána namísto klasické `add`, protože nemění příznaky podle výsledku operace.

```
.file "./foo.bc"
.text
.global f
.func f
f
suba.l #16, a7
move.l d2, (12,a7)
move.l d3, (8,a7)
move.l d4, (4,a7)
move.l d5, (0,a7)
movea.l d1, a0
adda.l d0, a0
adda.l d2, a0
adda.l d3, a0
adda.l d4, a0
adda.l d5, a0
adda.l (16,a7), a0
move.l a0, d0
move.l (0,a7), d5
move.l (4,a7), d4
move.l (8,a7), d3
move.l (12,a7), d2
adda.l #16, a7
rts

.endfunc
```

Tabulka B.6: Program `passing_args` v assembleru M68k

B.0.4 fact2.ll

Tento příklad nepředvádí překlad výpočtu faktoriálu, jak název napovídá, z důvodu absence implementované funkční verze instrukce pro násobení. Operace násobení jsou zde tedy nahrazeny sčítáním. Výsledek překladu zde není uveden, protože kód je relativně rozsáhlý. Příklad demonstruje i volání subrutiny s předáváním parametrů.