VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

POPIS MIKROPROCESOROVÉ ARCHITEKTURY AVR32 PRO PŘEKLADAČ LLVM

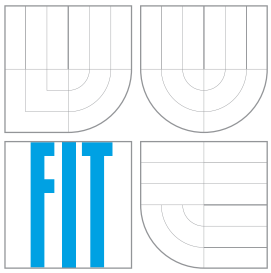
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MICHAL NAGY

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

POPIS MIKROPROCESOROVÉ ARCHITEKTURY AVR32 PRO PŘEKLADAČ LLVM

DESCRIPTION OF AVR32 MICROPROCESSOR ARCHITECTURE FOR LLVM COMPILER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MICHAL NAGY

VEDOUCÍ PRÁCE
SUPERVISOR

Prof. Ing. TOMÁŠ HRUŠKA, CSc.

BRNO 2010

Abstrakt

Tato bakalářská práce se zabývá tvorbou backendu architektury AVR32 pro překladač LLVM. Jádro práce tvoří seznamování se způsobem popisu architektur v LLVM a vlastní implementace backendu AVR32. Dále uvádím několik problémů, na které jsem během implementace narazil, a diskutuji jejich možná řešení. Výsledkem práce je funkční backend s několika omezeními zmíněnými v závěru práce.

Abstract

This bachelor's thesis deals with creation of an AVR32 backend for the LLVM compiler framework. The core of this work consists of explaining the way of architecture description in LLVM and of my own implementation of the AVR32 backend. Furthermore, several problems encountered during the implementation are discussed along with their possible solutions. As the result of this work functional backend, with a few constraints discussed in the conclusion chapter, was created.

Klíčová slova

AVR32, popis architektury, backend, překladač, GCC, LLVM, mikroprocesorová architektura

Keywords

AVR32, architecture description, backend, compiler, GCC, LLVM, microprocessor architecture

Citace

Michal Nagy: Popis mikroprocesorové architektury AVR32 pro překladač LLVM, bakalářská práce, Brno, FIT VUT v Brně, 2010

Popis mikroprocesorové architektury AVR32 pro překladač LLVM

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana profesora Tomáše Hrušky.

.....
Michal Nagy
10. května 2010

Poděkování

Děkuji panu doktorovi Miroslavu Trmači za uvedení do problematiky a za poskytnutí odborné pomoci. Dále bych chtěl poděkovat celému týmu Lissom za poskytnutí možnosti navštěvovat odborné semináře a procvičit si odborný ústní projev.

Obsah

Kapitola 1

Úvod

S vytvořením každé nové mikroprocesorové architektury je nezbytné navrhnout i překladač generující pro ní strojový kód. Přičemž v dnešním světě stále rostou nároky na rychlost výroby takového překladače a též na kvalitu jím generovaného kódu. Vytvářet pokaždé zcela nový překladač a vynalézat již vynalezené by bylo jednoduše neúnosné. Proto byly vyvinuty modulární překladače, mezi jejichž zástupce se bezesporu řadí LLVM, které poskytují nenáročnou cestu pro popis mikroprocesorových architektur. Tímto je umožněno poměrně rychlé přidání nového backendu do již existujícího překladače, který již veškeré složitosti, jako překlad z vyššího programovacího jazyka, či aplikování nejrůznějších optimalizací pro dosažení maximálně efektivního kódu, řeší a není nutné se jimi zabývat znovu. Zároveň s objevením nových a efektivnějších technik je již s minimálním úsilím dosaženo toho, aby byly promítnuty do procesu překladu pro danou architekturu.

Úkolem mé práce je popsat mikroprocesorovou architekturu AVR32 pro překladač LLVM verze 2.6, tak aby pomocí něj bylo možné provádět překlad do jazyka symbolických instrukcí této architektury.

1.1 Překladový systém LLVM

Low-Level Virtual Machine je překladový systém zaměřený na agresivní vícestupňovou optimalizaci kódu. Projekt byl započat v roce 2000 Chrisem Lattnerem z Univerzity Illinois a stále více si získává na oblíbenosti. Kód LLVM je psaný v jazyce C++ a je šířen pod licencí BSD.

LLVM se od ostatních klasických překladačů, jako je GCC, poměrně liší, protože sleduje výrazně jiný přístup. LLVM je modulární, snadno rozšiřitelná překladačová infrastruktura, což je silně reflektováno v architektuře založené na knihovnách a jasně oddělenými komponentami. Přestože je tento překladač zaměřen na efektivní optimalizace, výsledky ukazují, že je mnohonásobně výkonnější než překladač GCC.

Překlad systémem LLVM probíhá ve dvou fázích. Nejdříve je zdrojový program převeden do vnitřní jednotné reprezentace nezávislé na zdrojovém jazyce a architektuře a následně je zadní částí překladače přeložen do jazyka symbolických adres cílového procesoru. První část překladu není vlastní součástí překladače LLVM a je nutné použít např. upravenou přední část překladače GCC nebo překladač Clang. Díky tomuto principu je možné provádět překlad z libovolného zdrojového jazyka vyšší úrovně, kterým tyto frontendy rozumějí. [5]

1.1.1 Vnitřní reprezentace

Pro vnitřní reprezentaci byl vyvinut jazyk LLVM, což je jazyk podobný jazyku symbolických adres využívající virtuální instrukce nezávislé na zdrojovém jazyce a cílové architektuře. Tímto způsobem je možné dosáhnout jednotnosti velké části překladu, čímž není potřeba psát algoritmy pro každý procesor zvlášť.

Nezávislost jazyka LLVM na cílové architektuře ovšem nemůže být zaručena v situacích, kdy specifikace zdrojového jazyka dává volnost pro rozdílnou interpretaci na různých platformách. Např. datový typ `int` jazyka C může být namapován na datové typy `i32` nebo `i64` v závislosti na tom, pro kterou architekturu byl daný frontend zkompilován.

LLVM je jazyk nízké úrovně, je ovšem velice přesný a obsáhlý, tak aby nesl dostatek informací pro umožnění velkého počtu optimalizací. Může být zapsán formou textu (`.ll`), formou binárního kódu (`.bc`) nebo může být uložen v tzv. *in-memory* reprezentaci formou grafu. [5]

SSA forma

Významnou charakteristikou jazyka LLVM je *static single assignment* (SSA) forma, která obecně znamená, že do virtuálního registru je možné zapsat pouze jedenkrát. SSA forma výrazným způsobem zjednodušuje analýzu datového toku. Problém např. s řídícími proměnnými cyklů, do kterých je třeba zapisovat opakovaně v každé iteraci, je vyřešen zavedením funkce Φ. Tato funkce má na vstupu dvojice: základní blok, proměnná, a na základě toho, z jakého základního bloku bylo přijato řízení, je vrácena příslušná hodnota proměnné. [5]

Příklad 1.1.1. Prázdná smyčka čítající od 0 do `N` zapsaná v LLVM:

```
LoopHeader:
  %cond = icmp sgt i32 %N, 0
  br i1 %cond, label %Loop, label %LoopEnd
Loop:
  %index = phi i32 [ 0, %LoopHeader ], [ %nextIndex, %Loop ]
  %nextIndex = add i32 %index, 1
  %exitCond = icmp eq i32 %nextIndex, %N
  br i1 %exitCond, label %LoopEnd, label %Loop
LoopEnd:
```

1. V úvodním základním bloku `LoopHeader` je vyhodnocena podmínka `%cond`, která určuje, zda se cyklus bude alespoň jednou provádět.

2. Instrukcí podmíněného skoku `br` je na základě této podmínky proveden skok buď do těla cyklu `Loop` nebo za konec cyklu `LoopEnd`.

3. Na začátku každé iterace je do proměnné `%index` přiřazena návratová hodnota funkce `phi`. V první iteraci, kdy řízení přišlo ze základního bloku `LoopHeader`, vrací funkce nulu. V každé další iteraci, kdy už řízení přichází z toho samého základního bloku, tj. `Loop`, je vrácena hodnota proměnné `nextIndex` definovaná v minulé iteraci.

4. Je vyhodnocena podmínka ukončení cyklu `%exitCond`.

5. Je proveden podmíněný skok z závislosti na podmínce `%exitCond` buď znovu na začátek cyklu nebo za jeho konec.

1.2 Proces generování kódu

Aby bylo možné porozumět struktuře backendu, je nezbytné porozumět tomu, jak funguje generátor kódu.

Pro psaní této sekce jsem čerpal z oficiální dokumentace [8] a z práce věnované vytváření backendu TriCore [5].

1.2.1 Sestavování grafu

Proces přípravy vstupního programu pro fázi výběru instrukcí konvertováním lineárního seznamu do orientovaného stromu je nazýván *lowering*. Tento strom je označovaný zkratkou *DAG* (Directed Acyclic Graph). Jedná se o úplně první krok generování kódu. Pro každou instrukci LLVM programu je vytvořen objekt `SDNode` (SelectionDAG node), který obsahuje následující informace:

- **Opcode.** Celé číslo identifikující instrukci reprezentující daný uzel.
- **Výsledky (definice).** Zatímco většina instrukcí produkuje právě jeden výsledek, některé mohou definovat několik hodnot (např. operace s vedlejším efektem nebo kombinované instrukce dělení/modulo) nebo nemusí definovat vůbec žádnou hodnotu (např. instrukce skoku). Objekt `SDNode` obsahuje seznam datových typů pro všechny jeho výsledky.
- **Operandy (použití).** Každý `SDNode` udržuje záznam všech dalších uzlů, na kterých závisí. Jedná se buď o datovou závislost (uzel používá hodnotu definovanou jiným uzlem) nebo o řídící závislost (instrukce reprezentovaná jiným uzlem musí být provedena před touto instrukcí). Takový vztah k jinému `SDNode` objektu je reprezentovaný objektem `SDValue`, který obsahuje ukazatel na daný uzel společně s indexem příslušného výsledku. Každý `SDNode` proto vlastní seznam objektů `SDValue` a příslušný seznam datových typů. Jediný rozdíl mezi datovou a řídící závislostí je ten, že řídící závislost, tzv. *chain*, má datový typ `MVT::Other` místo standardního jako např. `MVT::i32`.

1.2.2 Legalizace

Fáze legalizace je zodpovědná za konvertování hodnot nepodporovaných datových typů na hodnoty podporovaných typů: konvertování malých typů na větší, tzv. *promoting*, a rozkládání velkých typů na menší, tzv. *expanding*. Tyto změny mohou vkládat znaménkové nebo neznaménkové rozšiřování, aby bylo zaručeno že výsledný kód má stejné chování jako vstup. Implementace daného backendu sděluje legalizátoru, které datové typy jsou legální, a kterou třídu registrů pro ně použít, voláním metody `addRegisterClass` v jeho `TargetLowering` konstruktoru.

Během legalizace musí být rovněž zaručeno, že výsledný DAG obsahuje pouze operace nativně podporované cílovou architekturou. Architektury mají většinou zvláštní omezení, jako např. nepodporování každé operace pro každý datový typ. Legalizace dosažení tohoto cíle dosahuje nahrazením dané operace sekvencí jiných operací, povýšením jednoho typu na typ větší, který daná operace podporuje, nebo ručním ošetřením operace daným backendem. Implementace daného backendu sděluje legalizátoru, které operace nejsou podporovány, a které ze zmíněných akcí provést, voláním metody `setOperationAction` v jeho `TargetLowering` konstruktoru.

1.2.3 Výběr instrukcí

Během této fáze dochází k nahrazení všech instrukcí jazyka LLVM na strojové instrukce výsledné architektury. Pro tuto transformaci se využívá techniky zvané *pattern matching*, která spočívá ve vyhledávání předem definovaných vzorů ve zdrojovém stromu a v jejich následném nahrazení za příslušný výstupní vzor. Může se jednat o prostý vztah 1:1, tj. ke každé instrukci LLVM existuje přesný ekvivalent v cílové instrukční sadě. Může se ale také jednat o celé podstromy, a to jak ve vstupním, tak i ve výstupním vzoru.

1.2.4 Plánování

Nově vytvořené grafy, jejichž uzly obsahují instrukce cílové architektury, jsou dekonstruovány a jejich instrukce jsou převedeny do lineárního seznamu. Každá funkce je reprezentována objektem třídy `MachineFunction`, který obsahuje seznam základních bloků. Základní blok, reprezentovaný objektem třídy `MachineBasicBlock`, obsahuje sekvenci strojových instrukcí (objekty třídy `MachineInstr`). Plánovač je povinen určit pořadí, ve kterém jsou všechny instrukce emitovány. Rozhodování je založeno na několika omezeních, například tzv. minimal register pressure.

Seznam je stále v SSA formě, tudíž ještě neobsahuje úplně validní assemblerový kód. S několika výjimkami (jako přesuny do registru pro návratovou hodnotu), všechny instrukce stále pracují nad množinou virtuálních registrů a všechny reference na zásobníkový rámec adresují abstraktní objekty místo konkrétních offsetů.

1.2.5 Alokace registrů

Všechny reference na virtuální registry jsou z programu eliminovány. Pro mapování virtuálních registrů na právě jeden fyzický registr jsou použity známé barvící techniky. Pokud počet virtuálních registrů překročí počet fyzických registrů, které jsou k dispozici, je vygenerován tzv. *spill code*. Eliminace virtuálních registrů je doprovázena dekonstrukcí SSA formy. Všechny dříve vložené Φ instrukce jsou jednoduše nahrazeny instrukcemi kopírování.

1.2.6 Vkládání prologu a epilogu

Po alokaci fyzických registrů a jejich možných uloženích na zásobník je možné pro každou funkci vypočítat, kolik je třeba rezervovat místa v zásobníkovém rámci. Díky tomu je nyní možné vložit prolog a epilog a nahradit abstraktní reference do zásobníkového rámce skutečnými adresami s bází ve formě ukazatele zásobníku nebo ukazatele rámce.

1.2.7 Emise kódu

V poslední fázi je provedena samotná emise výsledného kódu. Při statickém překladu se jedná o assemblerový soubor a při JIT (just in time) kompilaci dochází přímo k emisi strojových instrukcí do paměti daného procesu.

Pro funkčnost výsledného backendu stačí implementovat jen jednu z těchto variant. Například backend AVR32 implementuje pouze statický překlad.

1.3 Architektura procesoru AVR32

AVR32 je 32-bitová mikroprocesorová architektura s redukovanou instrukční sadou vyvíjená firmou Atmel. Mezí hlavní podtypy této architektury patří AVR32A a AVR32B. AVR32A

je mikroprocesor zaměřený na levná vestavěná zařízení a neposkytuje dedikované registry pro stínování souboru registrů či navratových adres v kontextu přerušení. Oproti tomu AVR32B je zaměřená na zařízení, kde latence přerušení hraje významnou roli a tyto věci implementuje. Další zajímavou vlastností je podpora harwarové implementace Java Virtual Machine.

Procesor obsahuje 16 32-bitových registrů pro všeobecné použití `R0 - R15`. Do horních tří registrů jsou mapovány ukazatel zásobníku, linkový registr a ukazatel instrukcí, čímž je efektivní počet registrů pro všeobecné použití snížen na 13. Linkový registr `LR` je používaný pro uchování návratové adresy. Při volání podprogramu je do něj návratová adresa uložena a při návratu je hodnota tohoto registru zpět zkopírována do ukazatele instrukcí. Kromě těchto speciálních případů je daný registr volný pro všeobecné použití. Dále AVR32 obsahuje jeden 32-bitový stavový registr `SR`, přičemž jeho horní polovina obsahující informace o módu a stavu procesoru, je přístupná pouze v privilegovaném režimu.

Procesor umožňuje v omezené míře i práci s 64-bitovými čísly. 64-bitová hodnota je ukládána do dvou po sobě jdoucích registrů, z nichž první z nich má sudé pořadové číslo. Jako argument instrukce se zadává pouze registr s nižším pořadovým číslem, ve kterém je uložena spodní polovina číselné hodnoty. Příkladem takovéto instrukce je instrukce násobení `MULU.D`, popř. její znaménková varianta `MULS.D`, která násobí dvě 32-bitová čísla a do registrů ukládá 64-bitový výsledek.

AVR32 je architektura typu load/store, což znamená, že většina instrukcí pracuje pouze s registrovými, popř. přímými operandy, a s pamětí pracují jen instrukce pro tento účel dedikované: load a store. Pomocí těchto instrukcí je možné načítat a ukládat data o velikostech 8, 16, 32 a 64-bitů. Data jsou obvykle ukládána ve formátu *big-endian*, tj. vyšší byte je uložený na nižší adrese, nicméně pro snadnou přenositelnost je k dispozici několik instrukcí, které jsou schopné ukládat a načítat data ve formátu *little-endian*. Při načítání hodnot kratších než nativní velikost registrů, jsou tyto hodnoty v závislosti na použité instrukci znaménkově, popř. neznaménkově rozšířeny. Některé implementace podporují nezarovnaný přístup k datům prostřednictvím instrukcí load/store. Položky na zásobníku by měly být z důvodu výkonnosti vždy zarovnány na hranici slov.

Instrukční sada AVR32 obsahuje dva typy instrukcí lišící se délkou: kompaktní a rozšířené. Kompaktní instrukce mají délku 16-bitů a rozšířené 32-bitů. Oba typy instrukcí musí být zarovnány na hranici půlslov (16-bitů). Některé instrukce existují v podmíněné i nepodmíněné variantě. Podmíněné instrukce přijímají 4-bitový kód podmínky, v závislosti na kterém (společně s nastavenými příznaky ve stavovém registru) se buď provedou nebo ne.

Pro přehlednost níže uvádím nejdůležitější rysy architektury AVR32:

- 32-bitová RISC registrová mikroprocesorová architektura typu big-endian.
- 16 32-bitových registrů pro všeobecné použití (`R0 - R15`), z nichž do horních tří jsou mapovány ukazatel zásobníku, linkový registr a instrukční ukazatel.
- Velké množství instrukcí s podmíněným vyhodnocením.

Podrobnější informace o architektuře AVR32 je možné najít v oficiální dokumentaci firmy Atmel [3].

1.3.1 Syntaxe instrukcí

Instrukce AVR32 jsou v jazyce symbolických adres zapisovány podle následujícího vzoru:

```
opcode{cond} dest, op1, op2
```

,kde `opcode` je operační kód dané instrukce, `cond` představuje kód podmínky, `dest` je cílový operand a `op1`, `op2` jsou zdrojové operandy. Podmínkový kód se uvádí pouze u podmíněných instrukcí a příslušný počet operandů se může lišit u každého formátu instrukce.

Někdy se poslední zdrojový operand uvádí ve formě `op << sa`, kde `sa` představuje počet bitů, o které je operand posunut vlevo (bez zpětného ukládání) před tím, než je použit.

Adresový operand je zpravidla možné najít pouze u instrukcí load a store a zapisuje se podle následujících vzorů:

1. `Rp++`
   Provedení přesunu paměťové buňky na adrese uložené v registru `Rp` a následný posun ukazatele `Rp` o jednu položku dále. Analogie operace pop při práci se zásobníkem.

2. `-Rp`
   Posun ukazatele `Rp` o jednu položku zpět a následný přesun paměťové buňky na adrese uložené v tomto registrovém ukazateli. Analogie operace push při práci se zásobníkem.

3. `Rp[disp]`
   Adresa tvořená bázovým registrem `Rp` a přímým znaménkovým offsetem `disp` obvykle 16 bitů dlouhým.

4. `Rb[Ri << sa]`
   Adresa tvořená bázovým registrem `Rb` a znaménkovým indexem získaným posunutím hodnoty v registru `Ri` o `sa` bitů vlevo.

Upozorňuji, že výše uvedený popis formátu instrukcí je pouze orientační, a konkrétní formát každé instrukce se může lišit, přičemž i samotná instrukce může definovat více formátů. Pro přesnou definici odkazuji na referenční manuál [3].

Příklad 1.3.1. Podmíněná instrukce odečítání `sub`. Instrukce se provede pouze pokud výsledkem předchozí instrukce porovnání bylo kladné znaménkové číslo. Při provedení uloží do registru `R0` hodnotu `R1 - 6`.

```
subgt R0, R1, 6
```

1.4 Instalace systému LLVM

Překladač LLVM je dodáván ve dvou částech. První část – *llvm* – je tvořena všemi knihovnami, nástroji a hlavičkovími soubory potřebnými pro používání LLVM. Tato část obsahuje nástroje jako assembler, dissasembler, analyzátor a optimalizátor bitového kódu. Zároveň také obsahuje testovací sadu umožňující testování nástrojů LLVM a přední části GCC. Zatímco druhá část – *llvm-gcc* – je upravený GCC frontend umožňující překlad zdrojového programu v jazyce C či C++ do jazyka LLVM. Ještě existuje jedna volitelná část – *llvm-test* – sloužící k dalšímu testování funkčnosti a výkonnosti překladače. Všechny tyto části je možné získat na oficiálních stránkách projektu LLVM na adrese: `http://llvm.org/releases/download.html`. Též je možné stáhnout nejnovější verzi těchto balíčků přímo z repozitáře svn.

Informace uváděné v této sekci je možné nalézt v oficiální dokumentaci [4].

1.4.1 Instalace překladače LLVM

Po stažení a rozbalení souborů zdrojových souborů LLVM je třeba vytvořit prázdný adresář pro objektové soubory. V tomto adresáři budou uchovávány všechny meziprodukty během překladu LLVM. Oddělení zdrojových souborů od objektových souborů je doporučovaný způsob a druhý jsem ani nezkoušel. Nyní je třeba přejít do tohoto nově vytvořeného adresáře, nakonfigurovat a zkompilovat překladač. Mezi nejdůležitější konfigurační volby patří:

`-prefix`
: Adresář, do kterého se má sestavené LLVM nainstalovat.

`-with-llvmgccdir`
: Cesta k LLVM GCC frontendu. Pokud tato cesta není specifikována, bude GCC frontend hledán v adresářích uvedených v proměnné prostředí `PATH`. Pokud ani tak program nebude nalezen, dojde ke kompilaci LLVM bez něj a nebude možné jej využívat v rámci nástrojů LLVM.

`-enable-optimized`
: Povoluje kompilaci s optimalizací a odstranění ladících informací. V LLVM distibuci je tato konfigurační volba implicitně povolena a ve verzi stažené z repozitáře svn je naopak implicitně zakázána.

`-enable-targets`
: Seznam architektur, které mají být zkompilovány. Výchozí hodnota je `all` způsobující kompilaci všech dostupných architektur. Pokud je zadána hodnota `host-only`, dojde ke kompilaci pouze nativní architektury.

Příklad 1.4.1. Instalace LLVM distribuce s ladícími informacemi do adresáře `/opt`. Nainstaluje se pouze backend procesoru Sparc. `OBJ_ROOT` zastupuje objektový adresář a `SRC_ROOT` adresář se zdrojovými soubory.

```
cd OBJ_ROOT
SRC_ROOT/configure -prefix=/opt -disable-optimized \
  -enable-targets=sparc
make
make install
```

Po úspěšné instalaci a přidání cesty k výsledným spustitelným souborům do proměnné prostředí `PATH` je možné překladač začít používat. Pokud ovšem v systému ještě není přítomen LLVM GCC frontend, bude zatím možné provádět překlad pouze ze zdrojových souborů psaných v jazyce LLVM.

1.4.2 Instalace LLVM GCC frontendu

LLVM GCC frontend se konfiguruje obdobně jako originální GCC překladač. Do konfiguračních voleb se ovšem musí navíc uvést cesta k objektovému adresáři obsahujícího sestavený LLVM překladač. Bez uvedení tohoto adresáře výsledný nástroj nebude reagovat na přepínač `-emit-llvm`, čili je potřeba tento krok nezanedbat. Mezi nejdůležitější konfigurační volby patří:

`-prefix`
: Adresář, do kterého se má sestavený LLVM GCC frontend nainstalovat.

`-program-prefix`
: Prefix, který bude přidán před název výsledných spustitelných souborů. Např. při prefixu `llvm-` bude nástroj `gcc` pojmenován `llvm-gcc`.

`-enable-llvm`
: Cesta k objektovému adresáři LLVM, obsahujícího zkompilovaný překladač.

`-enable-checking`
: Zapnutí kontrol uvnitř překladače. Není doporučováno tyto kontoly vypínat.

`-enable-languages`
: Seznam jazyků, které mají být povoleny. Výchozí hodnotou jsou všechny jazyky.

Příklad 1.4.2. Instalace LLVM GCC frontendu s podporou jazyků C a C++ do adresáře `/opt`. Před název výsledných nástrojů bude předřazen prefix `llvm-`. V adresáři `/usr/src/llvm-obj` je předem nachystaný zkompilovaný překladač LLVM.

```
cd OBJ_ROOT
SRC_ROOT/configure -prefix=/opt -enable-llvm=/usr/src/llvm-obj
make
make install
```

1.5 Používání systému LLVM

Pro překlad zdrojového programu v jazyce C slouží nástroj `llvm-gcc` představující upravený GCC frontend pro LLVM. Pokud byl při konfiguraci `llvm-gcc` balíčku uveden jiný prefix než `llvm-`, název nástroje se bude patřičně lišit. Implicitně tento nástroj generuje nativní objektový kód. Pro generování programu v jazyce LLVM je mu třeba explicitně přidat přepínač `-emit-llvm` společně s přepínačem `-S` nebo `-c`. Pokud je zadán přepínač `-S`, je vygenerován soubor v textové reprezentaci jazyka LLVM. Tento soubor je možné dále zkonvertovat nástrojem `llvm-as` do binární reprezentace, tzv. *bitcode*. Pokud byl ovšem zadán přepínač `-c`, bude vygenerován přímo tento bitcode.

Překlad z binární reprezentace jazyka LLVM do assembleru cílové architektury se uskuteční nástrojem `llc`. Zde je třeba pomocí přepínače `-march` definovat cílovou architekturu. Bez uvedení architektury se za výchozí považuje architektura uvedená ve vstupním zdrojovém souboru, což většinou bývá architektura nativní. Nástroj `llc` rovněž umožňuje možnost výpisu a vizualizace průběhu překladu, což je výborný pomocník v době ladění vytvářeného backendu. Aby se ovšem systém LLVM zkompiloval s podporou grafové vizualizace, je třeba mít v době konfigurace LLVM nainstalován vizualizační software *Graphviz*, který je možný získat na adrese: http://www.graphviz.org/. Konfigurační skript systému LLVM si tento nástroj sám najde. Při spuštění nástroje `llc` s příslušným přepínačem se během překladu programu postupně zobrazí okna obsahující grafy stromů jednotlivých základních bloků u všech funkcí. Mezi tyto přepínače patří:

`-view-dag-combine1-dags` Zobrazí graf po sestavení a před prvním průchodem optimalizací.

`-view-legalize-dags` Zobrazí graf před legalizací.

`-view-dag-combine2-dags` Zobrazí graf před druhým průchodem optimalizací.

`-view-isel-dags` Zobrazí graf před výběrem instrukcí.

`-view-sched-dags` Zobrazí graf před plánováním instrukcí.

Kromě těchto přepínačů existuje ještě jeden užitečný přepínač pro účely ladění, a to `-print-machineinstrs`, který způsobí vytisknutí vygenerovaného strojového kódu mezi jednotlivými fázemi kompilace. Bližší informace je možné nalézt v [8].

Příklad 1.5.1. Zdrojový program jazyka C `main.c` je nejdříve přeložen do textové formy jazyka LLVM uložené v souboru `main.ll`. Tento soubor je dále převeden z textové reprezentace do binární reprezentace jazyka LLVM uložené v souboru `main.bc`. Na závěr dojde k překladu tohoto souboru do assembleru architektury Mips. Před fází výběru instrukcí jsou zobrazeny grafy všech základních bloků uvnitř modulu. Výsledný assembler je uložen v souboru `main.S`.

```
llvm-gcc main.c -emit-llvm -S -o main.ll
llvm-as main.ll -o main.bc
llc -march=mips -view-isel-dags -o main.S
```

Pro přeložení assemblerového souboru do objektového souboru a pro sestavení spustitelného souboru lze využít balíčku *binutils*, který byl zkompilován s podporou dané architektury.

1.6 Nástroj TableGen

Důležitou součástí systému LLVM je nástroj TableGen, který byl vytvořen týmem LLVM pro zjednodušení popisu architektury procesoru. Pomocí tohoto nástroje je možné popsat vlastnosti procesoru objektově s využitím tříd a dědičnosti. Tímto způsobem je dosaženo snížení duplicity při popisu společných vlastností, což dopomáhá ke snadnějšímu psaní a údržbě kódu a snižuje riziko vzniku chyb.

Práce s nástrojem TableGen probíhá následovně. Nejdříve je vytvořen zdrojový soubor pro tento nástoj (obyčejně s příponou `.td`), dále je tento soubor předložen nástroji TableGen, který z něj posléze vygeneruje hlavičkové a zdrojové soubory jazyka C++ popisující registrovou sadu, instrukční sadu, volací konvence a mnoho dalšího. Pro zpřístupnění těchto informací je třeba vygenerované soubory vložit na příslušné místo ve zdrojových souborech popisu architektury.

TableGen se skládá z 2 klíčových částí: tříd a definic, přičemž obě jsou považovány za záznamy.

- Záznamy mají jedinečné jméno, seznam hodnot a seznam nadtříd.
- Definice jsou pevná forma záznamů a většinou už mají všechny hodnoty atributů definované. Jsou to konečné instance předem definovaných tříd nesoucí vlastní potřebná data.

- Třídy jsou abstraktní záznamy použité pro vytvoření a popis dalších záznamů. Třídy je možné dědit a předávat jim parametry. Atributy třídy mohou být nechány nedefinovány a je možné definovat je až později v některé z podtříd. Pokud už byl určitý atribut nebo část atributu (určitý bitový rozsah) definován dříve, následující definice příslušnou část přepíše. Tato vlastnost definování pouze určitého bitového rozsahu atributů se využívá např. pro postupnou definici binární podoby instrukce. Nejdříve je vytvořena obecná třída zahrnující určitý typ instrukce, ve které jsou definovány již známé bity. Tato třída je následně postupně specializována a jsou definovány další známé bity instrukčního slova.
- Multitřídy jsou skupiny abstraktních záznamů vyvolané všechny naráz. Např. je možné jediným vyvoláním multitřídy definovat všechny varianty určité instrukce bez nutnosti tyto varianty definovat odděleně jednu po druhé.

Pro podrobnější informace odkazuji na oficiální dokumentaci nástroje TableGen [6].

1.7 Postup přidání nového backendu do LLVM

Pro začlenění nového backendu do infrastruktury LLVM je zapotřebí provést několik kroků:

1. Vytvořit podadresář, který bude obsahovat veškeré zdrojové soubory backendu, a upravit sestavovací skripty LLVM pro začlenění nového adresáře do procesu překladu.
2. Vytvořit podtřídu třídy `TargetMachine`, která bude popisovat vlastnosti cílové architektury.
3. Popsat registrovou sadu procesoru. Pomocí nástroje TableGen se popisují samotné registry, registrové třídy a pravidla alokace registrů. Ručně v jazyce C se následně popisují další vlastnosti jako např. seznam ukládaných registrů v těle funkce, generování prologu a epilogu a eliminace frame indexu.
4. Popsat instrukční sadu procesoru.
5. Popsat výběr instrukcí.
6. Napsat kód pro tisk assembleru.
7. Volitelně přidat podporu pro podtypy daného procesoru.
8. Volitelně přidat podporu pro tzv. *just in time* (JIT) překlad, který emituje binární kód přímo do operační paměti procesu.

Podrobnější informace jsou k nalezení v [9].

1.7.1 Začlenění backendu do procesu překladu

Nejdříve ze všeho je zapotřebí vytvořit podadresář s názvem nového backendu v adresáři `lib/Target`, kde se vyskytují všechny ostatní implementace backendů. Následně je třeba upravit vstupní soubor nástroje Autoconf s názvem `configure.ac`. V tomto souboru stačí najít všechny výskyty seznamu backendů a přidat do nich právě přidávaný backend. Po úpravě tohoto souboru se nesmí zapomenout na znovu vygenerování skiptu `configure` skriptem `AutoRegen.sh` v adresáři `autoconf`. [9]

V nově vytvořeném adresáři je bohužel nutné vytvořit poměrně mnoho souborů a tříd, aby bylo vůbec možné provést bezchybný překlad nového backendu. Asi nejjednodušším způsobem je zkopírovat sem soubory již existujícího backendu a postupně jeho funkčnost zakomentovávat a modifikovat.

Kapitola 2

Návrh a implementace backendu

Tato kapitola pojednává o samotné implementaci backendu AVR32. Ovšem předem upozorňuji, že převážná část informací, které jsou zde uvedeny, se snaží být co možná nejobecnějšími a slouží především k uvedení čtenáře do problematiky popisu mikroprocesorových architektur v překladači LLVM. Tyto informace jsou většinou převzaty z dokumentace LLVM a z kódu ostatních backendů a je tak možné je s výhodou použít pro psaní libovolného backendu. Konkrétnímu návrhu a implementaci backendu procesoru AVR32 už je věnováno znatelně méně textu. Jedná se o přirozený důsledek toho, že vlastní studium principů mi zabralo mnohonásobně více času, než samotná implementace konkrétního backendu.

Přestože se ovšem většinou jedná o informace aplikovatelné na jakýkoli backend, zaměřuji se převážně jen na problémy vlastní architektuře AVR32 a je vysoce pravděpodobné, že při popisu jiných architektur bude nutné řešit mnoho dalších problémů, jejichž řešení zde již k nalezení nebude.

Každý problém, na který jsem během implementace narazil, je zde pečlivě popsán a vždy k němu uvádím možná řešení, o kterých vím, dále cestu, kterou jsem se vydal, z jakých důvodů, a především, odkud jsem se inspiroval.

2.1 Obecné informace o backendu

Každý backend musí implementovat a zaregistrovat podtřídu třídy `TargetMachine`. Tato třída tvoří jediné rozhraní mezi samotným backendem a zbytkem LLVM. Podrobnější informace lze nalézt v [9, 5].

2.1.1 Popis rozvržení dat

Velice důležitou částí backendu představuje popis rozvržení dat, kam se řadí věci jako uspořádání slabik v rámci slova, velikost ukazatele, požadavky na zarovnání jednotlivých datových typů, směr růstu zásobníku či velikost jednotlivých zásobníkových položek. Definice těchto vlastností se zpravidla provádí v konstruktoru příslušné třídy `TargetMachine`.

Každý backend musí pro tuto třídu implementovat funkci `getTargetData`, která vrací objekt typu `TargetData`. Konstruktoru třídy `TargetData` se předává řetězec, který v backendu AVR32 vypadá následovně:

```
"E-p:32:32-i8:8:8-i16:16:16-i32:32:32"
```

Jednotlivá pole jsou oddělena pomlčkou. První pole udává, že backend používá pořadí bytů typu big endian. Další pole obsahují informace o velikosti a požadovaném zarovnání jed-

notlivých datových typů včetně typu ukazatel (`p`) [9]. V manuálu AVR32 je uvedeno, že některé implementace umožňují nezarovnaný přístup k datům. Čili jsem se raději rozhodl zvolit bezpečnější variantu - všechna data zarovnaná.

Vlastnosti zásobníku nese objekt typu `TargetFrameInfo`, který musí být vracen funkcí `getFrameInfo`.

2.1.2 Definice podtypů architektury

U mnoha architektur existuje velké množství jejich variant a vývojových verzí. Navíc některé hardwarové prvky mohou být přítomny volitelně, např. jednotka FPU.

Pro popis těchto skutečností slouží tzv. *subtarget*. Každý subtarget obsahuje jednu či více tzv. *features*. Uvnitř backendu je následně možné provádět rozhodnutí na základě toho, zda je daná feature přítomna či nikoliv (viz. [9, 5]).

V AVR32 tohoto konceptu nevyužívám, nicméně jsem byl donucen implementovat alespoň jednu prázdnou feature s názvem `Dummy`, abych mohl vůbec provést úspěšný překlad.

2.2 Popis registrové sady

Převážná část popisu registrové sady je obsažena v příslušném souboru `RegisterInfo.td`. Z tohoto souboru jsou během překladu vygenerovány nástrojem TableGen hlavičkové a zdrojové soubory jazyka C++. Zde je potřeba popsat vlastní registry programového modelu a třídy registrů, do kterých jsou tyto registry posléze přiřazeny. Myšlenka tříd registrů je taková, že pokud určitá instrukce vyžaduje jako operand registr z nějaké třídy, je za tento operand možné dosadit kterýkoli z nich. Jinými slovy, registry stejné třídy jsou funkčně ekvivalentní. Každá třída registrů obsahuje seznam registrů patřících do ní, jejich datový typ a pořadí jejich alokace. Některé registry je možné z pořadí alokace vynechat, např. speciální registry typu ukazatel zásobníku či ukazatel instrukcí, kterými je sice možné zaměnit operand ve většině instrukcí, ovšem jejich alokace kvůli jejich speciálnímu významu není žádoucí. Dále je nutné u každého registru skládajícího se z podregistrů tuto skutečnost popsat. [9]

Nejvhodnější asi bude vysvětlit popis registrů na příkladu.

Příklad 2.2.1. V tomto příkladu je uvedena velmi zjednodušená definice dvou registrových tříd.

```
def R0  : Register<"R0">;
def R1  : Register<"R1">;
def R0D  : RegisterWithSubRegs<"R0",  [R0,  R1]>;

def : SubRegSet<1, [R0D], [R0]>;
def : SubRegSet<2, [R0D], [R1]>;

def RegsW : RegisterClass<"AVR32", [i32], 32, [R0, R1]> {}

def RegsD : RegisterClass<"AVR32", [i64], 64, [R0D]> {
  let SubRegClassList = [RegsW, RegsW];
}
```

1. Pomocí třídy `Register` je definováno několik registrů. Jediným povinným parametrem této třídy je název registru, který je využíván ve fázi tisku assembleru.

2. Pomocí třídy `RegisterWithSubRegs` je možné definovat registry složené ze subregistrů. V tomto případě je definován registr `R0D` skládající se z podregistrů `R0` a `R1`.

3. Pro pozdější možnost zpřístupnění subregistrů určitého registru je nutné definovat pomocí třídy `SubRegSet`, pod kterým indexem bude zpřístupněn který podregistr. Zde například adresováním podregistru registru `R0D` s indexem 2 bude vrácen registr `R1`. Dané seznamy mohou obsahovat i více prvků, přičemž vždy dochází ke svazování dvojic na stejných idexech v obou seznamech.

4. Na závěr jsou definovány samotné třídy registrů pomocí třídy `RegisterClass`. První třída obsahuje dva 32-bitové registry `R0` a `R1` a druhá třída obsahuje jeden 64-bitový registr `R0D` obsahující dva subregistry. Proměnná `SubRegClassList` obsahuje seznam tříd registrů, do kterých patří jednotlivé podregistry tohoto registru. Pravidla alokace registrů nejsou uvedena ani u jedné třídy, což znamená, že registry budou alokovány v pořadí jejich definice uvnitř třídy.

Jak už jsem uvedl dříve, po překladu tohoto souboru nástrojem TableGen jsou vygenerovány zdrojové soubory jazyka C++, které je potřeba vložit do příslušného `RegisterInfo.cpp` modulu popisujícího registrovou sadu. Uvnitř tohoto souboru je dále nutné v popisu registrové sady pokračovat. Zde se popisují věci jako generování prologu a epilogu, eliminace frame indexu, alokace a uvolňování volacího rámce či definice ukládaných a rezervovaných registrů v těle funkce. Pro podrobnější informace odkazuji na sekci 2.8.

2.3 Popis instrukční sady

Instrukční sada se popisuje v příslušném `InstrInfo.td` souboru, přičemž pouze výběr některých složitých instrukcí či adresovacích režimů je nutné popsat ručně v příslušném zdrojovém souboru C++. Mezi tyto instrukce patří zejména instrukce definující více výsledků (např. násobení nebo dělení), či instrukce řídící.

U každé instrukce je třeba definovat zejména seznam operandů, text pro výpis do assembleru a vzor pro výběr instrukce. Pokud k výběru určité instrukce nedochází nebo je výběr implementován ručně, nechává se vzor prázdný.

Pro definici instrukce pomocí nástroje TableGen se používá třída `Instruction`. Jednotlivé vlastnosti instrukce jsou v této třídě reprezentovány jako proměnné, jejichž hodnotu je třeba definovat.

2.3.1 Definice operandů

Operandy instrukce se dělí celkem do čtyř základních typů v závislosti na tom, zda jsou vstupní nebo výstupní a explicitní nebo implicitní.

Vstupní operand je takový, jehož hodnotu daná instrukce používá a závisí na ní. Může se jednat o registr určité registrové třídy nebo o přímou hodnotu. Oproti tomu výstupní operand je takový, jehož hodnotu daná instrukce definuje, a zde se může jednat pouze o registr.

Explicitním operandem je myšlena skutečnost, že je daný operand přímo uváděný v zápisu instrukce, a v případě registru je uvedený prostřednictvím jeho registrové třídy. Během alokace registrů je následně za tento operand dosazen konkrétní fyzický registr dané registrové třídy.

Pomocí implicitních operandů je možné popisovat vedlejší efekty dané instrukce. Tento operand ve výpisu instrukce uváděný nebývá, instrukce ho ale přesto používá nebo definuje (např. stavový registr). Tento operand je uváděný ve formě konkrétního fyzického registru, čili během alokace registrů už se za něj nic dalšího nedosazuje.

```
Defs = [SR];
Uses = [SR];
OutOperandList = (outs RegsW:$dst);
InOperandList = (ins RegsW:$src1, RegsW:$src2);
```

Výše je uveden příklad definice všech čtyř typů operandů. Proměnné, do kterých je přiřazováno, jsou členské proměnné třídy `Instruction`, přičemž první dvě jsou typu `list` (seznam) a druhé dvě typu `dag` (graf). Daná instrukce implicitně definuje i používá stavový registr `SR`, explicitně definuje operand `$dst` z třídy registrů `RegsW` a explicitně používá operandy `$src1` a `$src2` ze stejné třídy registrů. Identifikátory daných explicitních operandů jsou uváděny proto, aby bylo možné se na ně později odkazovat. Tato instrukce může reprezentovat např. instrukci sčítání s přenosem. Upozorňuji ovšem, že se jedná pouze o ilustrativní příklad, a pro úplnou definici instrukce následující příkazy nestačí.

2.3.2 Definice výpisu instrukce

Způsob zápisu instrukce se uvádí pomocí textového řetězce, v němž je možné odkazovat se na operandy dané instrukce prostřednictvím identifikátorů, které jim byly přiřazeny při jejich definici. Identifikátor operandu je též možné doplnit o modifikátor, jehož přítomnost je následně možné zjistit v modulu tisku assembleru, a upravit tak způsob zápisu daného operandu v závislosti na použitém modifikátoru.

```
AsmString   = "adc $dst, $src1, $src2";
```

Takto by mohl být definován textový zápis instrukce sčítání s přenosem, jejíž operandy byly definovány o několik odstavců výše.

Případný modifikátor, např. s názvem `arith` nad operandem `$dst`, by bylo možné uvést tímto způsobem: `${dst:arith}`.

2.3.3 Definice výběrového vzoru

Aby mohla být určitá instrukce vůbec vybrána, musí překladač znát vzor, místo něhož je možné danou instrukci vložit.

```
Pattern = (set RegsW:$dst, (adde RegsW:$src1, RegsW:$src2));
```

Takto by mohl vypadat výběrový vzor dříve zmíněné instrukce sčítání s přenosem. Uzel `adde` reprezentuje instrukci LLVM provádějící součet operandů `$src1` a `$src2` s přenosem. Uzel `set` ukládá druhý operand do operandu prvního. Nyní již LLVM zná, že při výběru instrukcí může nahradit instrukci `adde` právě definovanou instrukcí, pokud jsou všechny její operandy 32-bitové virtuální registry (za předpokladu, že třída registrů `RegsW` je definována jako 32-bitová).

Uzly LLVM, které je možné ve vzoru použít, se dají najít v hlavičkovém souboru LLVM `Target/TargetSelectionDAG.td`. Jejich sémantika už tam bohužel příliš vysvětlována není. Názvy jsou ovšem docela intuitivní a v případě pochybností je možné nahlédnout na příklad jejich použití v jiných backendech.

Přímé operandy

Pokud je třeba ve vzoru uvést, že daný vstupní operand představuje např. 32-bitovou přímou hodnotu, uvádí se místo názvu registrové třídy typ `i32imm` (analogicky tak pro 8, 16 či 64 bitovou hodnotu). Pokud je na tuto hodnotu potřeba klást určité omezení (např. že může nabývat pouze neznaménkové 5-bitové hodnoty), dá se toho docílit pomocí třídy `PatLeaf`, jejíž druhý parametr představuje predikát ve formě C++ kódu.

```
def uimm5  : PatLeaf<(i32 imm), [{
  return (((unsigned)N->getZExtValue() & 31) ==
    (unsigned)N->getZExtValue());
}]>;
```

Výše je definováno omezení na 5-bitovou neznaménkovou hodnotu. Pokud tedy bude ve vzoru uvedený typ daného operandu `uimm5`, bude tomuto omezení podléhat, čímž nebude možné provést výběr dané instrukce, pokud predikát nebude platit. Fyzicky se ovšem stále jedná o operand typu `i32`, čili v seznamu operandů je třeba ho takto definovat. Pokud by totiž mezi typem definovaným v seznamu operandů a typem definovaným ve vzoru vznikla nekonzistence, nástroj TableGen by při překladu vygeneroval chybu.

2.3.4 Definice vlastností instrukce

U některých speciálních instrukcí je třeba též uvést doplňující vlastnosti, jako např. že se jedná o instrukci skoku (`isBranch`), návratu z podprogramu (`isReturn`), že daná instrukce může číst paměť (`mayLoad`), popř. že řízení programu nikdy nepokračuje instrukcí následující (`isBarrier`). Tyto vlastnosti jsou reprezentovány členskými proměnnými třídy `Instruction`, které mohou nabývat hodnot 0 nebo 1. Bližší informace je možné nalézt v hlavičkovém souboru `Target/Target.td`.

Dvojadresné instrukce Velice užitečnou vlastností je vlastnost `isTwoAdress` udávající, že daná instrukce je dvojadresná, což má za následek, že první a druhý operand jsou svázány. Díky této vlastnosti je možné modelovat instrukce typu `a = a + b`, u kterých je první zdrojový operand totožný s operandem cílovým.

2.3.5 Nestatické informace o instrukcích

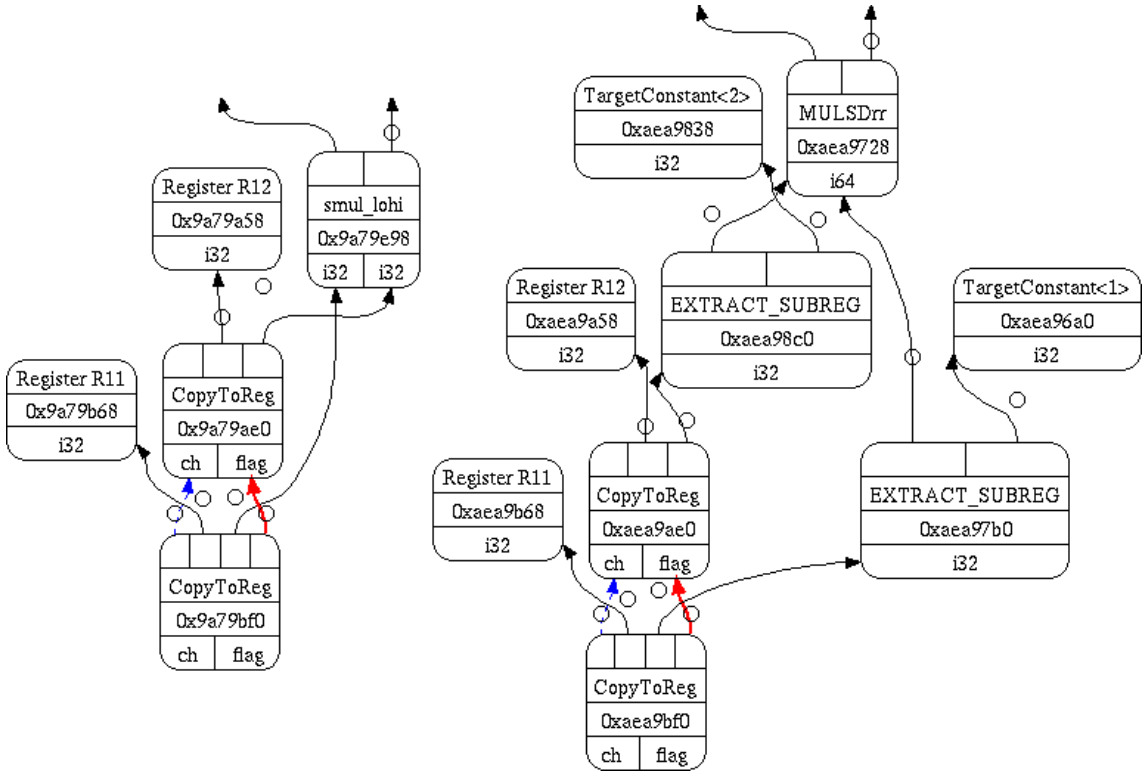
Kromě statických informací uvedených v příslušném `.td` souboru je třeba implementovat několik funkcí sloužících pro analýzu, tranformaci a vkládání určitých strojových instrukcí. Mezi tyto funkce patří např. funkce pro ukládání registru na zásobník, kopírování registru do jiného registru, či vkládání instrukcí nepodmíněného skoku. Jedná se o členské funkce příslušné třídy `InstrInfo`.

2.3.6 Instrukce násobení a dělení v AVR32

Mezi instrukce, které je většinou potřeba ošetřit ručně, patří instrukce násobení a celočíselného dělení, přičemž AVR32 zde není žádnou výjimkou. Problém spočívá v tom, že ne vždy existuje v LLVM přesný ekvivalent k dané instrukci z instrukční sady daného procesoru. Pro zvýšení přehlednosti budu uvádět instrukce LLVM malými písmeny a instrukce AVR32 písmeny velkými.

Pokud dochází k násobení dvou 32-bitových čísel, přičemž z výsledku je potřeba pouze spodních 32-bitů, LLVM vygeneruje uzel `mul`, který přijímá dva 32-bitové operandy a generuje 32-bitový výsledek. V AVR32 naštěstí existuje ekvivalent, a to instrukce `MUL`. Jediné, co tedy stačí udělat, je v popisu dané instrukce napsat šablonu, která nahradí uzel `mul` za instrukci `MUL`.

Pokud jsou ovšem z násobení potřeba obě dvě poloviny výsledku, LLVM vygeneruje uzel `umul_lohi`, popř. při znaménkovém násobení `smul_lohi`. Tento uzel přijímá dva 32-bitové operandy a produkuje dva 32-bitové výsledky, přičemž v jednom je uložena spodní polovina a v druhém horní polovina výsledku. AVR32 ovšem pro tento účel nabízí instrukci s mírně odlišným významem. Instrukce `MULU.D`, popř. její znaménková varianta `MULS.D`, přijímá dva 32-bitové operandy a produkuje 64-bitový výsledek, který ukládá do dvou po sobě jdoucích registrů, přičemž první z nich musí být sudý. Problém jsem vyřešil přidáním 64-bitové registrové třídy skládající se z registrů, jejichž subregistry jsou 32-bitové registry. O této nové registrové třídě jsem ovšem neinformoval legalizační část překladače, protože jinak by začal považovat 64-bitové operandy za legální a přestal by je eliminovat. Algoritmus pro výběr instrukce jsem popsal ručně a to následovně: Uzel `umul_lohi` jsem nahradil za instrukci `MULU.D` s tím, že z výsledku `MULU.D` jsem vyseparoval oba subregistry a ty jsem namapoval na výsledky instrukce `umul_lohi`. Znaménkovou variantu jsem řešil analogicky. Na obrázku 2.1 je zobrazena transformace instrukce násobení LLVM `smul_lohi` (levý strom) na instrukci násobení `MULS.D` architektury AVR32 (pravý strom).



Obrázek 2.1: Transformace instrukce násobení

Pokud dochází k celočíselnému dělení dvou 32-bitových čísel, LLVM k získání výsledku standardně použije instrukci `udiv`, popř. `sdiv`, a k získání zbytku použije instrukci `urem`, popř. `srem`. Pokud se ovšem legalizační části překladače sdělí, aby tyto instrukce expando-

val, LLVM tyto instrukce nahradí za instrukci `udivrem`, popř. `sdivrem`, která produkuje oba dva výsledky. Pokud některý z nich není potřeba, tak prostě není na nic navázán. V AVR32 existuje instrukce `DIVU`, popř. `DIVS`, která provádí celočíselné dělení dvou 32-bitových čísel, výsledek ukládá do sudého 32-bitového registru a zbytek do registru přímo následujícího. K problému jsem postupoval analogicky k násobení. Využil jsem již přidané 64-bitové registrové třídy a vytvořil jsem instrukci dělení, která produkuje 64-bitový výsledek. Legalizační části překladače jsem přikázal, aby expandovala instrukce `udiv` a `urem` na instrukci `udivrem`. Při výběru instrukcí jsem tuto instrukci nahradil za instrukci `DIVU`, přičemž z výsledku `DIVU` jsem vyseparoval oba subregistry a namapoval je na výsledek a zbytek instrukce `udivrem`. Znaménkovou variantu jsem řešil analogicky.

2.4 Práce s přímými operandy v AVR32

Architektura AVR32 je registrová architektura a jako taková obsahuje převážně instrukce pracující pouze nad registry. Pro operandy jiných typů existuje jen několik speciálních instrukcí, které načítají, ukládají popř. přesouvají přímé operandy či paměťové buňky z/do registrů. U těchto instrukcí ovšem existuje jedna nepříjemnost. Instrukce procesoru AVR32 mají maximální možnou délku 32 bitů, což při uvážení místa pro kód operace, adresy cílového registru a dalších příznaků, nemůže stačit pro uložení 32-bitové přímé hodnoty. Výsledkem je to, že když už nejaká instrukce přímý operand podporuje, je omezený délkou např. na 21 bitů, a větší přímé hodnoty je třeba načítat ve více krocích.

2.4.1 Načítání konstanty

Instrukční sada procesoru AVR32 obsahuje jen několik málo instrukcí, které jsou schopny pracovat s přímými operandy. Mezi tyto instrukce patří např. přesouvání do registru, odečítání nebo násobení. Maximální délka znaménkového přímého operandu, kterou je možné u těchto instrukcí najít, činí 21 bitů [3]. Delší hodnoty je proto nutné nejdříve ve více krocích uložit do registru a následně pracovat s tímto registrem.

Načítání konstanty přesahující délku 21 bitů řeším následujícím postupem:

1. Konstanta je rozdělena na dvě poloviny, které jsou posléze individuálně načteny do registrů pomocí instrukce `mov`.
2. Obsah registru obsahujícího horní polovinu konstanty je posunut o 16 bitů vlevo.
3. Nad oběma registry je provedena operace logického součtu. Výsledkem je registr obsahující danou konstantu.

Pro modifikaci číselné hodnoty je možné využít třídy `SDNodeXForm` nástroje TableGen. Tato třída umožňuje manipulaci s číselnými hodnotami v okamžiku, kdy již prošly fází výběru instrukcí. Jako parametr třídy je nutné zadat funkci jazyka C++ provádějící danou konverzi.

```
def HI16 : SDNodeXForm<imm, {
  return CurDAG->getTargetConstant(
    (unsigned)N->getZExtValue() >> 16, MVT::i32);
}]>;
```

Výše uvedený příklad definuje modifikátor `HI16`, který posouvá 32-bitovou neznaménkovou hodnotu o 16 bitů vpravo, neboli extrahuje její horní polovinu. Analogicky by bylo možné popsat např. modifikátor, který extrahuje spodní polovinu hodnoty nulováním bitů 16 až 31.

Pomocí třídy `Pat` je možné definovat vzory pro výběr instrukcí, tj. nahrazení určité části stromu instrukcí částí jinou. Tato třída přijímá dva parametry: vstupní a výstupní vzor.

```
def : Pat<(i32 imm:$val), (ORrrsl (MOVri21 (LO16 imm:$val)),
  (MOVri21 (HI16 imm:$val)), (i32 16))>;
```

Výše uvedený příklad definuje konverzi přímé hodnoty typu `i32` na instrukci `ORrrsl` a dvě instrukce `MOVri21`. První instrukce `MOVri21` načítá do registru spodní polovinu přímé hodnoty `$val`, což následně tvoří první operand logického součtu. Druhá instrukce `MOVri21` načítá do dalšího registru horní polovinu hodnoty `$val`, což následně tvoří druhý operand logického součtu. Třetím operandem instrukce `ORrrsl` je přímá hodnota udávající počet bitů, o které má být druhý operand posunut vlevo, před tím než je použit (viz. [3]).

Instrukce `movhi` V instrukční sadě AVR32 jsem našel ještě jednu vhodnou instrukci pro tento účel - `movhi`, která posouvá 16-bitovou přímou hodnotu o 16-bitů vlevo a takto vzniklou 32-bitovou hodnotu ukládá do cílového registru. Tento proces by ovšem opět vyšel celkem na tři strojové instrukce. Nicméně s instrukcí `movhi` jsem narazil na problém v nástroji IAR, ve kterém jsem prováděl simulaci, proto jsem se rozhodl pro první uvedenou variantu.

Pseudoinstrukce `lda.w` Kromě vícekrokového ukládání přímé hodnoty existuje ještě mnohem pohodlnější možnost, a to využití pseudoinstrukce `lda.w` podporované překladačem GCC (viz. 2.4.2). Tuto variantu jsem ovšem zamítl, protože použití pseudoinstrukce podporované jedním nástrojem není příliš přenosné řešení [1].

Tabulka konstant V neposlední řadě se musím zmínit o technice, kterou je též možné použít. Pokud si překladač připraví potřebnou konstantu na určité místo v paměti, může tuto hodnotu načíst do registru v jediném kroku. Tím ovšem vzniká další problém, a to jakým způsobem zadat adresu této paměťové oblasti, která sama o sobě může být velká hodnota. Pro tento účel se často daná konstanta ukládá do tabulky a následně je adresována relativně vůči začátku této tabulky. LLVM pro tento účel poskytuje tzv. *constant pool*, způsob jeho používání jsem ovšem zatím nestihl nastudovat.

2.4.2 Načítání globálního symbolu

Globalní symbol slouží pro adresování paměti. V assembleru je reprezentovaný návěštím, které je při generování strojového kódu nahrazeno konkrétní adresou. Jedná se tedy též o konstantu, ovšem narozdíl od číselných konstant, její hodnota v době překladu není známa. Proto je třeba vždy předpokládat ten nejhorší případ, že zabírá všech 32 bitů a je třeba jí načítat jako velké číslo.

Problémem ovšem je, že s globálním symbolem není možné číselně manipulovat. Aby tedy bylo možné rozdělit jeho načítání do registru do více kroků, příslušný assembler musí

[1] Instrukci `lda.w` nicméně stejně používám pro načítání globálních symbolů, čili přenositelnost je snížena v každém případě.

poskytovat makra, která extrahují jednotlivé části symbolu. Pokud už se ale takový assembler najde, je pravděpodobné, že tyto makra bude poskytovat odlišným způsobem od ostatních assemblerů. Možnou alternativou je využití constant poolu, který je zmiňován v sekci 2.4.1, nebo využití techniky *linker relaxing* zmiňované níže.

Linker relaxing

V dokumentu [1] věnovanému krátkému popisu zadní části překladače GCC pro architekturu AVR32 jsem se dočetl o použité technice nazývané *linker relaxing*. Při použití této techniky překladač generuje pseudoinstrukce přijímající 32-bitový přímý operand, popř. globální symbol reprezentující libovolnou absolutní adresu, a o eliminaci těchto pseudoinstrukcí se stará až samotný spojovací program (linker). Mezi tyto pseudoinstrukce patří pseudoinstrukce načítající 32-bitovou přímou hodnotu do cílového registru `lda.w` a pseudoinstrukce `call` volající podprogram na absolutní adrese definované globálním symbolem. Důvod k použití této techniky je takový, že v okamžiku překladu překladač nemá informace o konkrétním umístění symbolů, a vždy musí uvažovat nejhorší případ při načítání jejich adresy jako přímé hodnoty. Pokud je ovšem tato technika uplatněna, o použití konkrétních instrukcí rozhodne až samotný linker, který již konkrétní hodnoty symbolů zná, čímž je umožněn další stupeň optimalizace kódu.

Technika *linker relaxing* je v překladači GCC implicitně zapnuta, rozhodl jsem se ji tedy též využít. Dopad tohoto rozhodnutí je v tom, že generovaný assemblerový soubor překladačem LLVM musí být přeložen assemblerem a slinkován linkerem, které tuto techniku podporují. Pokud by bylo třeba provést překlad assemblerem bez podpory této techniky, bylo by nutné buď využít maker pro extrakci částí globálního symbolu, popř. přidat podporu pro constant pool.

2.5 Adresace paměti

V AVR32 je možné adresovat paměť pomocí bázového registru doplněného přímým 16-bitovým offsetem, popř. indexovým registrem. Způsob implementace adresování v LLVM se liší v závislosti na typu adresy. Může se jednat např. o globální symbol či místo na zásobníkovém rámci.

Pro podporu instrukcí pracujících s pamětí je zapotřebí udělat následující kroky:

2.5.1 Legalizace adresy

Pokud program potřebuje adresovat globální symbol, LLVM vygeneruje uzel `GlobalAddress` obsahující název daného symbolu. Tento uzel je nejdříve potřeba během legalizační fáze nahradit sekvencí uzlů, která daný symbol načte do určitého registru, odkud bude možné tento symbol použít. V mém případě se jedná o nahrazení tohoto uzlu instrukcí `lda.w`. Pro ruční ošetření uzlu `GlobalAdress` je nutné přidat do konstruktoru třídy `AVR32TargetLowering` následující příkaz:

```
setOperationAction(ISD::GlobalAddress, MVT::i32, Custom);
```

Následně je potřeba ve funkci `LowerOperation` daný uzel ručně zpracovat.

Pokud program adresuje objekt na zásobníku, LLVM generuje uzel `FrameIndexSDNode` obsahující index jednoznačně identifikující tento objekt. Na rozdíl od globálního symbolu,

k eliminaci frame indexu dojde až v pozdní části překladu těsně před generováním assembleru.

Po legalizaci stromu instrukcí je adresa uložena buď v některém registru nebo ve formě frame indexu.

2.5.2 Popis adresovacího schématu

Aby bylo možné popsat instrukce pracující s pamětí, je třeba nejdříve vytvořit alespoň jedno adresovací schéma a ručně popsat výběr adresy. Výběr adresy je funkce, která konvertuje adresu na operand, který určitá instrukce přijímá . Tento operand může být tvořen více složkami, např. báze a index. V mé implementaci je adresový operand tvořen právě dvěma složkami, z nichž první reprezentuje registr uchovávající bázi adresy a druhá reprezentuje přímý offset. Zapsáno v příslušném souboru `InstrInfo.td` to vypadá následovně:

```
def ADDRri : ComplexPattern<i32, 2, "SelectADDRri", [], []>;

def MEMri : Operand<i32>
  let PrintMethod = "printMemOperand";
  let MIOperandInfo = (ops RegsW, i32imm);
```

Pomocí třídy `ComplexPattern` je definováno adresovací schéma s názvem `ADDRri`, pro jehož výběr je dále nutné definovat funkci C++ s názvem `SelectADDRri`. Výsledkem výběru adresy jsou dvě hodnoty typu `i32`. Pomocí třídy `Operand` je následně definován nový operand tvořený dvěma složkami, z nichž první je registr třídy `RegsW` a druhá je 32-bitová přímá hodnota. Pro tisk daného operandu je použita funkce `printMemOperand`, kterou je nutné definovat v modulu tisku assembleru.

Při definici instrukce se následně v seznamu vstupních či výstupních operandů daný operand uvede typu `MEMri` a uvnitř výběrové šablony se pro označení typu daného operandu používá adresové schéma `ADDRri`.

2.5.3 Výběr adresy

Funkci pro výběr adresy je nutné implementovat v příslušné třídě `SelectionDAGISel`. Název této funkce se musí shodovat s názvem uvedeným v definici příslušného adresovacího schématu. Implementace této funkce v backendu AVR32 dělá následující:

- Pokud je adresován globální symbol, po legalizaci je uložen ve formě registru, a takto příjde na vstup výběru adresy. Při výběru adresy stačí tento registr vrátit jako bázi a offset nastavit na nulový. Během tisku assembleru bude operand vytisknut následovně: `báze[offset]`.
- Pokud je adresován objekt na zásobníkovém rámci, vstupem funkce výběru adresy je frame index, který zatím není možné eliminovat. Jako báze je tedy vrácen tento frame index a offset je opět nastaven na nulový. Později před samotným tiskem instrukce dojde k nahrazení tohoto indexu a offsetu za skutečný bázový registr a skutečný offset od něj.

Sčítání frame indexu s konstantou Během testování práce se zásobníkem jsem ovšem narazil na jednu vlastnost LLVM, kterou jsem od něj původně nepředpokládal. Objekty v zásobníkovém rámci nejsou vždy adresovány jen pomocí frame indexu. Může dojít i k tomu, že LLVM vygeneruje instrukci sčítání sčítající frame index s konstantou a teprve výsledek tohoto sčítání se použije pro vstup funkce výběru adresy. K tomuto může dojít např. při ukládání 64-bitové hodnoty na zásobník. Horní polovina hodnoty je uložena na určitý frame index a spodní polovina hodnoty je uložena na tento frame index povýšený o hodnotu 4. Implementace procesoru Sparc tuto vlastnost LLVM řeší ošetřením instrukce sčítání uvnitř výběru instrukcí tak, že extrahuje operandy sčítání, levý operand uloží jako bázi a pravý operand jako offset. Ve fázi eliminace frame indexu je potom k výslednému offsetu, který by příslušel danému indexu, přičten ještě tento offset uložený během výběru adresy. Touto implementací jsem se tedy inspiroval.

Pro lepší pochopení principu adresování uvedu příklad adresace frame indexu pomocí sčítání s konstantou. Upozorňuji, že syntaxe výrazů není skutečná a je upravená pro snadnou přehlednost.

Příklad 2.5.1. Úkolem je uložit spodní polovinu 64-bitového parametru funkce uloženého na frame indexu -1 do registru `R12` pomocí instrukce načítaní z paměti `ld.w`. Adresa spodní poloviny tohoto parametru bude nabídnuta ve formě `add(index -1, constant 4)` (adresa horní poloviny parametru + 4). Funkce alokuje zásobníkový rámec o konstantní velikosti 20 bytů.

1. Ve fázi legalizace se nic zajímavého neděje, protože frame index musí zůstat tak, jak je.

2. Na vstupu funkce výběru adresy je instrukce `add(index -1, constant 4)`, ze které jsou extrahovány operandy a je vrácena dvojice `[index -1, constant 4]`, která se posléze stává adresovým operandem dané instrukce.

3. Potřebná instrukce načítání nyní vypadá takto:
   `ld.w(R12, addr([index -1, constant 4]))`.

4. Během fáze eliminace frame indexu je adresový operand změněn na dvojici `[SP, constant 24]`. Hodnota 24 byla získána jako součet velikosti zásobníkového rámce (20 B), offsetu parametru s daným indexem od začátku rámce a konstanty 4 z předchozího kroku. Případ s využitím ukazatele rámce by vypadal odlišně. Pro lepší pochopení tohoto posledního kroku odkazuji na sekci věnovanou práci se zásobníkem 2.8.

2.6 Implementace podmínek

V LLVM existují dva hlavní typy podmínek. Prvním typem jsou nevětvící podmínky, které neupoužívají podmíněných skoků, a druhým typem jsou podmínky, které skoky používají. Pro přidání podpory podmínek v backendu určitého procesoru je potřeba upravit legalizaci instrukcí tak, aby byly generovány uzly, které bude možné vybrat při výběru instrukcí. Někdy je též nevyhnutelné ručně napsat vkládání kódu pro nevětvící podmínky. Nyní nastíním problematiku a budu se věnovat popisu vlastní implementace podmíněných příkazů v backendu pro procesor AVR32.

Při implementaci jsem vycházel z backendů procesorů Sparc a ARM.

2.6.1 Podmínky v jazyce LLVM

Jazyk LLVM používá pro implementaci podmínek instrukce: `icmp` a `select`, popř. `br`. Instrukce `icmp` vyhodnotí danou podmínku nad dvěma celými čísly a instrukce `select` na základě výsledku této podmínky vrací buď první nebo druhý operand bez větvení programu. Vyhodnocení podmínek nad čísly s plovoucí řádovou čárkou je prováděno instrukcí `fcmp`. Pokud je větvení programu nevyhnutelné, je místo instrukce `select` použita instrukce skoku `br`, která v závislosti na výsledku podmínky skáče buď na první nebo druhé uvedené návěští. Podrobnou dokumentaci těchto funkcí je možné nalézt v manuálu jazyka LLVM [7]. Níže uvádím jejich zjednodušený popis:

Instrukce `icmp`

Syntaxe: `<result> = icmp <cond> <ty> <op1>, <op2>`

Tato instrukce přijímá celkem tři operandy. Prvním operandem je podmínkový kód `<cond>`, který může nabývat následujících hodnot:

`eq` : rovno

`ne` : nerovno

`ugt` : neznaménkově větší než

`uge` : neznaménkově větší nebo rovno

`ult` : neznaménkově menší než

`ule` : neznaménkově menší nebo rovno

`sgt` : znaménkově větší než

`sge` : znaménkově větší nebo rovno

`slt` : znaménkově menší než

`sle` : znaménkově menší nebo rovno

Zbývající dva operandy `<op1>` a `<op2>` jsou celočíselné hodnoty, nad nimiž bude daná podmínka vyhodnocena, přičemž obě hodnoty musí být stejného typu `<ty>`. Vrácená hodnota `<result>` je jednobitové číslo obsahující výsledek vyhodnocení podmínky. Instrukce je schopná pracovat i nad vektory, tento případ tu ovšem vysvětlovat nebudu, protože pro vysvětlení podmínek není relevantní.

Instrukce `select`

Syntaxe: `<result> = select selty <cond>, <ty> <val1>, <ty> <val2>`.

Tato instrukce přijímá celkem tři operandy. Prvním operandem je jednobitová hodnota nesoucí výsledek podmínky. Druhým operandem `<val1>` je hodnota typu `<ty>`, která je vrácena v případě pravdivosti podmínky, a třetím operandem `<val2>` je hodnota rovněž typu `<ty>`, která je vrácena v případě nepravdivosti podmínky.

Instrukce `br`

Syntaxe: `br i1 <cond>, label <iftrue>, label <iffalse>`

Tato instrukce tvoří analogii s instrukcí `select` s rozdílem, že místo vracení příslušné hodnoty, skáče na příslušné návěští `<iftrue>`, popř. `<iffalse>`. Kromě této syntaxe existuje ještě nepodmíněná varianta instrukce `br`.

2.6.2 Implementace nevětvících podmínek

Po sestavení grafu ze zdrojového souboru jazyka LLVM je instrukce `icmp` reprezentována uzlem `setcc` a instrukce `select` je reprezentována stejnojmenným uzlem `select`. Tyto uzly mají stejné vstupní a výstupní operandy jako jejich ekvivalenty v jazyce LLVM. Jejich stručný popis lze nalézt v hlavičkovém souboru LLVM `SelectionDAGNodes.h`.

Pro další manipulaci s grafem se ovšem velice hodí nechat si oba uzly sloučit do jediného. Sloučení je zajištěno přidáním následujících dvou příkazů do konstruktoru podtřídy třídy `TargetLowering` pro příslušný procesor:

```
setOperationAction(ISD::SELECT, MVT::i32, Expand);
setOperationAction(ISD::SETCC, MVT::i32, Expand);
```

Od nyní bude zajištěno, že budou uzly `setcc` a `select` sloučeny v jediný uzel `select_cc`. Tento uzel přijímá dvě celočíselné hodnoty, nad kterými má být vyhodnocena podmínka, hodnotu která má být vrácena v případě pravdivosti podmínky, hodnotu, která má být vrácena v případě nepravdivosti podmínky a podmínkový kód. Vrací příslušnou hodnotu v závislosti na výsledku podmínky. Jeho popis je možné nalézt rovněž ve výše uvedeném hlavičkovém souboru LLVM.

Uzel `select_cc` je třeba následně během legalizace instrukcí ručně nahradit takovou sekvencí uzlů, aby bylo následně možné uspěšně provést výběr instrukcí. Většina mikroprocesorových architektur využívá pro implementaci podmíněných instrukcí stavový registr. Nejdříve je třeba provést porovnávací instrukci, která provede nad vstupními operandy rozdíl a na základě výsledku nastaví příznaky stavového registru. Následná podmíněná instrukce se v závislosti na nastavených příznacích a zadaného podmínkového kódu buď provede nebo ne. Touto podmíněnou instrukcí může být např. instrukce skoku či přesunu dat. Uzel `select_cc` je tedy zpravidla během legalizace instrukcí opět rozdělen na dva uzly, tentokrát jsou ovšem tyto uzly svázány implicitně tzv. *flagem* a nikoli explicitně zadaným jednobitovým operandem, jako tomu bylo ve zdrojovém LLVM souboru. Flag jímž jsou obě instrukce svázány má za následek to, že druhá instrukce musí být vykonána po ukončení instrukce první, přičemž mezi ně lze vložit jen takové instrukce, které celkový výsledek neovlivní, např. instrukci přesunu dat. Pro ruční rozdělení uzlu `select_cc` je třeba nejdříve přidat následující příkaz do konstruktoru podtřídy třídy `TargetLowering` pro příslušný procesor:

```
setOperationAction(ISD::SELECT_CC, MVT::i32, Custom);
```

Tímto je dosaženo toho, že je legalizace daného uzlu přenechána funkci `LowerOperation`. Uvnitř této funkce je následně potřeba tento uzel ručně ošetřit. Užitečné je poznamenat, že je potřeba přeložit podmínkový kód LLVM na podmínkový kód cílové architektury, před tím než je vložen jako operand podmíněné instrukce, protože se tyto podmínkové kódy mohou samozřejmě číselně lišit.

Pro první uzel nastavující příznaky většinou existuje přímý ekvivalent v instrukční sadě cílového procesoru. Např. tato instrukce se u procesoru AVR32 nazývá `cp.w`. Tato instrukce provede rozdíl vstupních operandů a nastaví příznaky ve stavovém registru `SR`. Je tedy dostatečné napsat jednoduchou šablonu, pomocí které dojde k výběru dané instrukce.

Převést druhý uzel na cílovou instrukci už ovšem tak jednoduché být nemusí, zvlášť pokud daná architektura postrádá instrukci podmíněného přesunu dat. Architektura AVR32 naštěstí tuto instrukci obsahuje. Nejedná se ale o přesný ekvivalent tohoto uzlu. Zatímco vygenerovaný uzel vrací první operand v případě pravdivosti podmínky a druhý operand v případě její nepravdivosti, instrukce podmíněného přesunu `mov` procesoru AVR32 provádí přesun, pokud podmínka platí, a neprovádí nic, pokud podmínka neplatí [3]. Řešení tohoto problému jsem našel v backendu procesoru ARM, který rovněž obsahuje instrukci podmíněného přesunu dat. V definici této instrukce je třeba uvést omezení svazující registr výsledku s registrem, ve kterém je uložena hodnota, která má být vrácena v případě nepravdy. Tímto dojde k tomu, že pokud podmínka neplatí, prováděl by se přesun z jednoho registru do toho samého registru. LLVM je daným omezením donuceno k tomu, že nejdříve do cílového registru nepodmíněně uloží hodnotu pro nepravdivou podmínku a poté do něj podmíněně přesune hodnotu pro pravdivou podmínku, což je přesně to, čeho bylo třeba dosáhnout. Můj popis této instrukce pro procesor AVR32 vypadá následovně:

```
let Uses = [SR] in
def MOVCrr : InstAVR32<
  (outs RegsW:$dst), (ins RegsW:$true, RegsW:$false, CCOp:$cc),
  "mov$cc $dst, $true",
  [(set RegsW:$dst, (AVR32selectcc RegsW:$true, RegsW:$false,
    imm:$cc))]>,
  RegConstraint<"$false = $dst">;
```

Tato instrukce přijímá v jednom registru z třídy registrů `RegsW` hodnotu pro pravdivou podmínku `true`, v dalším registru ze stejné třídy registrů hodnotu pro nepravdivou podmínku `false` a jako poslední operand přímou hodnotu nesoucí kód podmínky `cc`. Implicitně ještě používá stavový registr `SR`. Omezení `$false = $dst` způsobuje svázání registru `false` s registrem `dst`. V šabloně je uvedeno, že uzel `AVR32selectcc` je možné nahradit touto instrukcí. Tento uzel je generován při ručním rozkladu uzlu `select_cc` jako uzel svázaný flagem s instrukcí porovnání a provádějící podmíněný přesun dat.

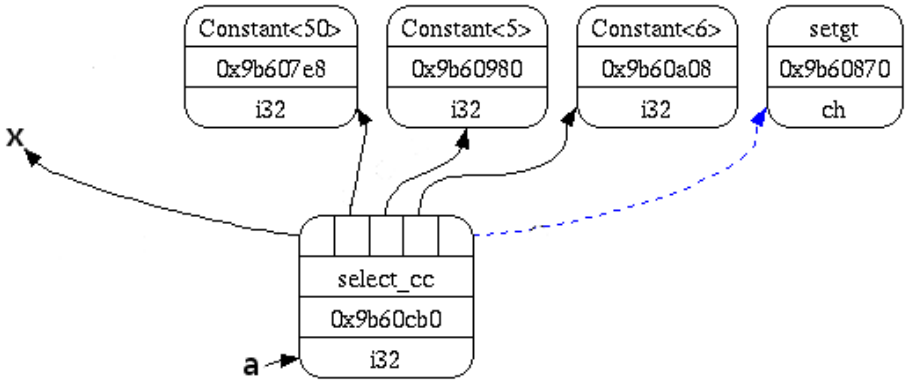
Příklad 2.6.1. Průběh překladu následující podmínky backendem AVR32:

```
if(x > 50) a = 5;
else a = 6;
```

Přeložená podmínka bude v jazyce LLVM vypadat zhruba takto:

```
%0 = icmp sgt i32 %x, 50
%a = select i1 %0, i32 5, i32 6
```

Po sestavení grafu bude podmínka vypadat takto:



Poté bude uzel `select_cc` rozložen na dva uzly, které budou při výběru instrukcí nahrazeny instrukcemi `cp.w` a `mov`. Následně bude v důsledku omezení vygenerována ještě jedna tentokrát nepodmíněná instrukce `mov`. Výsledný assembler bude vypadat následovně:

```
mov r0, 6
mov r1, 5
mov r2, 50
cp.w r12, r2  ; x - 50
mov r12, r0   ; r12 <- 6
movgt r12, r1 ; r12 <- 5, pokud x > 50
```

Vstupem je proměnná `x` uložená v registru `r12` a výstupem proměnná `a` uložená opět v registru `r12`.

Implementace bez využití instrukce podmíněného přesunu

Pokud je ovšem třeba implementovat nevětvící podmínku pro architekturu, která instrukci podmíněného přesunu dat neobsahuje, nezbyde jiná cesta než vkládání kódu této instrukce napsat ručně. Zpravidla jde o to, rozvinout instrukci podmíněného přesunu na tzv. *diamond control-flow pattern*:

```
StartBB:
  %TrueValue = ...
  brCC SinkBB
FalseBB:
  %FalseValue = ...
SinkBB:
  %Result = phi [%FalseValue, FalseBB], [%TrueValue, StartBB]
```

V základním bloku `StartBB` je nastavena hodnota pro pravdivou podmínku, následně v případě pravdivosti podmínky je proveden skok do základního bloku `SinkBB`. Pokud je podmínka nepravdivá, skok není proveden a řízení propadne do základního bloku `FalseBB`, kde je nastavena hodnota pro nepravdivou podmínku. Následně řízení propadá též do `SinkBB`. V zaklákladním bloku `SinkBB` funkce `phi` ukládá do proměnné `%Result` příslušnou hodnotu, v závislosti na tom, ze kterého bloku přišlo řízení. (Funkce `phi` je probírána v sekci věnované SSA formě: 1.1.1.). Konkrétní příklad implementace této techniky může být nalezen např. u backendu procesoru Sparc ve funkci `EmitInstrWithCustomInserter`.

2.7 Volání funkcí

Tato sekce pojednává o mechanismu volání funkcí, což zahrnuje témata jako popis volacích konvencí či implementace volání a návratu z funkcí. Při popisu jejich funkčnosti jsem čerpal výhradně ze zdrojových kódů již hotových backendů. Jmenovitě backendy procesoru Sparc a MSP430.

2.7.1 Volací konvence

Volací konvence definuje pravidla pro volání funkcí a předávání parametrů a návratových hodnot, která umožňují vzájemnou komunikaci volající funkce s funkcí volanou. Mezi tyto pravidla patří:

- Pořadí a způsob umístění jednotlivých argumentů (do registrů nebo na zásobník).
- Seznam registrů, jejichž obsah je volaná funkce povinna zachovat.
- Způsob uložení návratové adresy.
- Která z funkcí má na starost odstranění argumentů ze zásobníku.

Volací konvence použitá překladači GCC a IAR pro AVR32

Překladač IAR používá níže uvedená pravidla pro volání funkcí [2], přičemž experimentováním jsem zjistil, že danou volací konvenci dodržuje i překladač GCC.

- 32-bitové parametry jsou předávány nejdříve v registrech `R12-R8` v tomto pořadí. Následně jsou ukládány na zásobník v opačném pořadí.
- 8-bitové a 16-bitové parametry jsou rozšířeny na 32-bitů a jsou předávány podle pravidel pro 32-bitové hodnoty.
- 64-bitové parametry jsou ukládány nejdříve ve volných dvojicích registrů `R11:R10`, `R9:R8`. Následně jsou ukládány na zásobník v opačném pořadí.
- Návratová adresa je předávána v registru `LR` [2].
- Návratová hodnota je rošířena na 32-bitů a je předávána v registru `R12`. 64-bitová hodnota je předávána ve dvojici registrů `R11:R10`.
- Obsah registrů `R0-R7` volaná funkce zachovává.
- Uložené argumenty na zásobníku odstraňuje volající funkce.

[2] Toto pravidlo je vynuceno instrukcemi volání podprogramu, které před samotným skokem automaticky ukládají návratovou adresu do registru `LR`, a instrukcí pro návrat z podprogramu, která provádí skok na adresu uloženou v tomto registru.

2.7.2 Popis volacích konvencí v LLVM

Jednoduchou volací konvenci je možné popsat s využitím třídy `CallingConv` nástroje TableGen. Pro složitější pravidla volacích konvencí ovšem tento popis někdy nestačí a je třeba chování ručně modifikovat, popř. napsat celé ručně. Při popisu volací konvence pomocí nástroje TableGen je třeba popsat zvlášť volání funkcí a zvlášť vracení hodnot. Pravidla, která lze ovšem použít lze aplikovat bez rozdílu na oba dva případy.

Popis volací konvence spočívá v uvedení sekvence podmínek a pravidel, které jsou následně aplikovány v pořadí jejich uvedení. Podmínky je možné vnořovat. Níže uvádím popis nejdůležitějších pravidel:

- `CCIfType<typeList, rule>` - Aplikace pravidla `rule`, pokud je parametr typu uvedeného v seznamu typů `typeList`.
- `CCIfNotVarArg<rule>` - Aplikace pravidla `rule`, pokud se jedná o funkci s proměnným počtem argumentů.
- `CCPromoteToType<type>` - Povýšení typu argumentu na typ `type`.
- `CCAssignToReg<reglist>` - Uložení parametru do prvního dosud neobsazeného registru uvedeného v seznamu registrů `reglist`.
- `CCAssignToStack<size, alignment>` - Uložení parametru na místo na zásobníku, které je velké `size` bytů a je zarovnáno na hranici `alignment` bytů.

Popis mnou navržené volací konvence AVR32 vypadá následovně:

```
def CC_AVR32 : CallingConv<[
  CCIfNotVarArg<CCAssignToReg<[R12, R11, R10, R9, R8]>,
  CCAssignToStack<4, 4>
]>;

def RetCC_AVR32 : CallingConv<[
  CCAssignToReg<[R12, R11]>
]>;
```

Definice `CC_AVR32` uvádí pravidla pro předávání parametrů a definice `RetCC_AVR32` definuje pravidla pro vracení hodnot. Parametry jsou nejdříve předávány v registrech `R12-R8` v tomto pořadí. Následně jsou ukládány na zásobník (v opačném pořadí). U funkcí s proměnným počtem parametrů jsou všechny parametry (včetně fixních) ukládány rovnou na zásobník. Návratové hodnoty jsou vraceny v registrech `R12` a `R11`.

Jak je vidět, nikde nemám uvedeno ošetřování různých datových typů. To proto, protože jsem v třídě `TargetLowering` legalizoval pouze třídu 32-bitových registrů, tudíž dojde k předávání a vracení pouze 32-bitových hodnot. Kratší hodnoty jsou již v registrech uloženy jako 32-bitové číslo a s 64-bitovými hodnotami překladač pracuje prostřednictvím jejich polovin. Pokud má např. funkce vracet 64-bitovou hodnotu, fyzicky dojde k vrácení dvou 32-bitových hodnot.

Popis volací konvence ovšem sám o sobě ještě není dostačující. Ještě je nutné implementovat funkce `LowerFormalArguments`, `LowerCall` a `LowerReturn`, které provádějí vlastní činnost s využitím tohoto popisu.

Výše uvedený popis volací konvence není shodný s volací konvencí překladačů IAR a GCC. Konkrétně se liší ve způsobu předávání 64-bitových hodnot a ve způsobu předávání parametrů u funkcí s proměnným počtem parametrů. Jak již bylo zmíněno dříve, 64-bitové hodnoty jsou fyzicky předávány jako dvě 32-bitové hodnoty, a tudíž je není možné rozlišit od pravých 32-bitových hodnot a definovat pro ně odlišné chování. U funkcí s proměnným počtem parametrů existuje zase problém, že není možné odlišit fixní parametr od proměnného parametru. Třída `CCIfNotVarArg` rozděluje celé funkce na funkce s pevným počtem a na funkce s proměnným počtem parametrů, nikoli jednotlivé parametry. Pro rozlišení fixních parametrů od proměnných by bylo možné použít metody `IsFixed` objektu třídy `OutputArg` nesoucího argument, který je třeba volané funkci předat. Tento zásah by bylo nutné provést ve funkci `LowerCall`. Jelikož jsem ale nenašel možnost podmínit tuto vlastnost přímo v TableGen popisu volací konvence, vznikla by mezi tímto popisem a vlastní implementací poměrně velká nekonzistence a asi jednodušší by bylo TableGen popisu vůbec nevyužívat. Protože ale jednotnost volací konvence mezi mým backendem a backendy jiných překladačů nebyla nikdy vyžadována, rozhodl jsem se příliš si nekomplikovat situaci, a jednotnost volací konvence příliš neřešit.

2.7.3 Volání funkce

Proces volání funkce se implementuje ve funkci `LowerCall` třídy `AVR32TargetLowering`. Tato funkce přijímá adresu volané funkce `Callee` a vektory `Ins` a `Outs`. Vektor `Ins` obsahuje hodnoty, které je potřeba volané funkci předat a vektor `Outs` obsahuje datové typy, které volaná funkce vrací. Funkce `LowerCall` má za úkol naplnit prázdný vektor `InVals` vrácenými hodnotami.

Má implementace funkce `LowerCall` provádí zhruba následující:

1. Na základě vektoru argumentů `Ins` a popisu dané volací konvence je zjištěn způsob a místo uložení každého argumentu. Tato analýza se provádí za pomoci objektu třídy `CCState`.

2. Je vygenerován uzel `callseq_start`, který může být později, v případě potřeby alokace místa na zásobníku pro uložení argumentů, rozvinut na instrukci snižující ukazatel zásobníku.

3. Argumenty předávané na zásobníku jsou uloženy na konrétní místo v paměti. K adresaci je využit ukazatel zásobníku `SP` a offset získaný analýzou vstupních argumentů.

4. Argumenty předávané v registru jsou zkopírovány do příslušných fyzických registrů.

5. Globální symbol nesoucí adresu volané funkce je nahrazen uzlem `TargetGlobalSymbol`, čímž je zabráněno jeho pozdější nežádoucí modifikaci, např. načtením do registru a použitím registru.

6. Je vygenerována instrukce volání funkce s operandem cílové adresy.

7. Je vygenerován uzel `callseq_end`, který může být později, v případě potřeby odstranění uložených argumentů na zásobníku, rozvinut na instrukci zvyšující ukazatel zásobníku.

8. Vrácené hodnoty jsou zkopírovány z příslušných fyzických registrů do virtuálních registrů, jimiž je naplněn vektor `InVals`.

2.7.4 Načítání vstupních parametrů

Proces načítání parametrů se implementuje ve funkci `LowerFormalArguments`. Tato funkce přijímá vektor `Ins` obsahující datové typy parametrů. Úkolem je zkopírovat pevné parametry z fyzických registrů či zásobníku do virtuálních registrů, jimiž je třeba naplnit prázdný vektor `InVals`. Proměnné parametry samozřejmě kopírovány nejsou, protože jejich počet není předem znám.

Implementace této funkce je velmi podobná funkci `LowerCall`. Rovněž jsou nejdříve analyzovány vstupní parametry, které jsou následně zkopírovány do virtuálních registrů. U funkcí s proměnným počtem parametrů je ovšem nutné zapamatovat si offset prvního proměnného parametru. Toho má implementace dosahuje vytvořením fixního objektu na zásobníku s daným offsetem a nasledným uložením jeho frame indexu do členské proměnné `VarArgsFrameIndex`.

2.7.5 Ukládání návratových hodnot

Proces ukládání hodnot před návratem funkce se implementuje ve funkci `LowerReturn`. Tato funkce přijímá vektor `Outs` obsahující seznam hodnot, které je třeba uložit do fyzických registrů. Funkce opět nejdříve provede analýzu pro zjištění místa pro uložení, následně provede samotné uložení hodnot a na závěr generuje instrukci návratu funkce.

2.7.6 Práce s proměnnými parametry

Práce s proměnnými parametry probíhá prostřednictvím ukazatele. Tento ukazatel musí být před započetím práce inicializován, aby ukazoval na první proměnný parametr. Následně jsou jednotlivé parametry jeden po druhém načítány do lokálních proměnných, přičemž po načtení každého parametru je tento ukazatel inkrementován. Aby tento princip mohl fungovat, musí být při volání funkce argumenty uloženy na zásobník v opačném pořadí jejich deklarace, tak aby na vrcholu zásobníku byl umístěný první parametr. Počet proměnných parametrů ani jejich datový typ volané funkci nejsou známy, proto musí tuto informaci volající předat v některém z pevných parametrů. Načítání proměnných parametrů se odehrává zcela v režii tvůrce programu. Například v jazyce C pro tento účel slouží makra `va_start`, `va_arg` a `va_end`.

Pro inicializaci ukazatele proměnných parametrů v LLVM slouží uzel `vastart`. Tento uzel přijímá frame index objektu, do kterého je povinen uložit adresu prvního proměnného parametru. Jeho funkčnost je ovšem nutné popsat ručně.

Pro načtení parametru, na který aktuálně ukazuje ukazatel v LLVM slouží uzel `vaarg`. Tento uzel načítá proměnný parametr do virtuálního registru a následně inkrementuje ukazatel. Pokud je tvůrce backendu s tímto chováním spokojen, je možné nechat legalizaci tohoto uzlu v režii LLVM.

2.8 Práce se zásobníkem

Zásobník představuje v kontextu mechanismu funkcí datovou oblast proměnné velikosti, která je vnitřně rozdělena na tzv. *rámce*. Každý rámec přísluší právě jedné funkci, v níž se právě nachází vykonávání programu. Zásobníkový rámec slouží pro ukládání lokálních proměnných a pro předávání argumentů, případně i návratové adresy, při volání další funkce.

2.8.1 Ukazatel rámce

Objekty uvnitř rámce mohou být adresovány buď pomocí ukazatele rámce (angl. *frame pointer*) nebo pomocí ukazatele vrcholu zásobníku. Tyto ukazatele se od sebe liší tím, že první zmíněný ukazuje na začátek rámce a druhý na konec rámce (vrchol zásobníku). Pro ukazatel zásobníku je na většině architektur rezervován speciální registr, ovšem naproti tomu ukazatel rámce bývá rezervován až v prologu každé funkce, která ho používá, a představuje další rezervovaný registr, který nemůže sloužit pro všeobecné použití. Proto je vhodné vyhnout se použití ukazatele rámce vždy, pokud je to možné. Existuje pouze jedna situace, ve které tento ukazatel eliminovat nelze, a to pokud rámec obsahuje objekty proměnné velikosti, v důsledku čehož v době překladu není předem známa jeho velikost. Ve všech ostatních případech stačí přičíst k offsetu daného objektu na zásobníku velikost rámce, čímž se offset začne vztahovat k ukazateli zásobníku (v případě, že zásobník roste směrem dolů).

Existují tedy dva způsoby adresace objektů v zásobníkovém rámci, v závislosti na tom, zda je použita eliminace ukazatele rámce, či nikoliv. V třídě `RegisterInfo` příslušné architektury je nutné implementovat funkci `hasFP`, která vrací `true`, v případě, že daná funkce používá ukazatel rámce, a `false`, pokud ho nepoužívá. Pro učinění správného rozhodnutí stačí vrátit výsledek funkce `FrameInfo::hasVarSizedObjects`, která podává informaci o tom, zda příslušná funkce používá objekty proměnné velikosti na zásobníku. Ať už funkce `hasFP` vrací kteroukoli hodnotu, je třeba tuto informaci reflektovat v implementaci funkcí, které na tomto rozhodnotí závisejí, jako např. vkládání prologu, epilogu, či eliminace frame indexu.

2.8.2 Volací rámec

Volací rámec (angl. *call frame*) je místo na vrcholu zásobníku alokované těsně před zavoláním funkce, kam jsou následně ukládány argumenty, popř. návratová adresa, předávané na zásobníku. Použitá volací konvence určuje, zda za opětovné odstranění tohoto volacího rámce zodpovídá volaný nebo volající.

Kromě výše zmíněné varianty existuje ještě jedna, kdy je volací rámec již součástí zásobníkového rámce volající funkce, tudíž k jeho alokaci dojde staticky v těle prologu a uvnitř samotné funkce se již velikost rámce nijak nemění ani kolem volání dalších funkcí. Aby ovšem toto mohlo fungovat, musí být velikost rámce volajícího předem známa. Jinými slovy, daný rámec smí obsahovat pouze objekty pevné velikosti. Pokud je ale tato druhá varianta použita a volací konvence říká, že volací rámec odstraňuje volaný, je třeba po návratu ze zavolané funkce ukazatel zásobníku opět navrátit do původní hodnoty, protože LLVM v tomto případě počítá s tím, že velikost rámce je po celou dobu běhu funkce neměnná.

Ve třídě `RegisterInfo` příslušné architektury je potřeba implementovat funkci jménem `hasReservedCallFrame`, která vrací `true`, v případě že má daná funkce rezervovaný volací rámec, a `false` v případě, že je tento rámec nutné alokovat a odstraňovat vždy před a po zavolání funkce. Pro učinění správného rozhodnutí opět stačí vrátit návratovou hodnotu funkce `FrameInfo::hasVarSizedObjects`, tentokrát ovšem v negované podobě.

Pro alokaci a destrukci volacího rámce slouží uzly `callseq_start` a `callseq_end`, které by správně měly být vkládány před započetím a po ukončení volání funkce (viz. 2.7.3). Význam operandů těchto uzlů není nijak pevně daný, ovšem zpravidla prvním operandem bývá hodnota udávající počet bytů k rezervování a u uzlu `callseq_end` bývá ještě druhá hodnota udávající velikost, kterou již volaný uvolnil. U AVR32 postupuji též podle této

zvyklosti, ovšem druhý operand u uzlu `callseq_end` nevyužívám, protože volanou konvenci mám navrženou tak, že volací rámec uvolňuje pouze volající.

Uzly `callseq_start` a `callseq_end` jsou při výběru instrukcí nahrazeny pseudoinstrukcemi definovanými v příslušném `InstrInfo.td` souboru. U architektury AVR32 se jedná o pseudoinstrukce `ADJCALLSTACKUP` a `ADJCALLSTACKDOWN`. V konstruktoru příslušné třídy `RegisterInfo` je třeba příslušné třídě `GenRegisterInfo`, vygenerované nástrojem TableGen z popisu instrukční sady, předat tyto pseudoinstrukce. Tímto způsobem LLVM zná, že na zpracování těchto pseudoinstrukcí má zavolat funkci `eliminateCallFramePseudoInstr`. Jedná se o funkci, jejíž chování lze definovat v příslušné třídě `RegisterInfo`, a jejíž úkolem je odstranit danou pseudoinstrukci a nahradit ji v případě potřeby strojovou instrukcí modifikující registr ukazatele zásobníku. Při implementaci této funkce je potřeba reflektovat vlastnost právě zpracovávané funkce udávající, zda je volací rámec rezervovaný, či nikoliv, a pokud ano, kolik bytů již uvolnila volaná funkce. Např. pokud je volací rámec rezervovaný a volaná funkce nic neuvolňuje, obě instrukce se jednoduše eliminují.

2.8.3 Objekty proměnné velikosti

Jak jsem se již zmínil, objekty proměnné velikosti jsou jediná překážka pro eliminaci ukazatele rámce. Takovým objektem může být např. staticky alokované pole, jehož velikost je dána parametrem funkce. Překladač velikost takovéhoto objektu za doby překladu nezná a rezervace místa na zásobníku musí být řešena dynamicky. V LLVM pro dynamickou rezervaci místa na zásobníku slouží uzel `dynamic_stackalloc`, který je nutné při legalizaci ručně ošetřit. Jedním z jeho operandů je hodnota udávající velikost potřebnou pro rezervaci. Legalizace tohoto uzlu spočívá v získání ukazatele zásobníku, odečtení od něj dané hodnoty a v jeho zpětném uložení. K uvolnění rezervovaného místa dojde až v epilogu dané funkce.

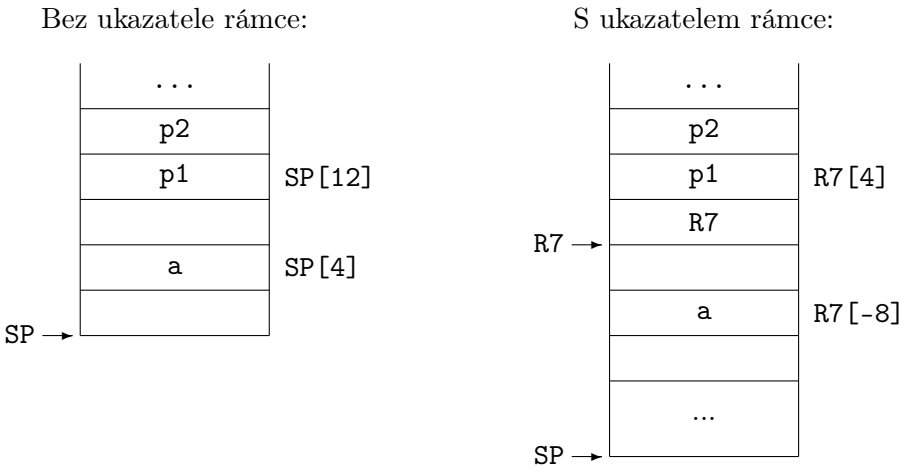
2.8.4 Vkládání prologu a epilogu

Fáze vkládání prologu a epilogu se uskutečňuje jako jedna z posledních těsně před samotnou emisí assembleru. Během této fáze je program již ve formě lineárního seznamu, všechny virtuální registry jsou již alokované a všechny Φ funkce SSA formy jsou eliminovány. Protože již všechny registry byly alokovány, je známo, kolik místa na zásobníku bude třeba pro uložení proměnných, které se již do fyzických registrů nevešly. Ukládané registry, jejichž obsah je danou funkcí modifikován, jsou v tomto okamžiku též již ošetřeny, a mají rezervované místo pro jejich uložení na zásobníku. Do rezervovaného místa se nezapočítávají objekty proměnné velikosti, k jejichž alokaci dojde až uvnitř funkce. Jediné co zbývá, je přidat kód z obou stran funkce, který toto potřebné místo alokuje a na závěr odstraní, a nahradit reference na abstraktní frame indexy skutečnými adresami. V závislosti na návratové hodnotě funkce `hasFP` (viz. 2.8.1) je nyní třeba implementovat danou funkčnost.

Pro implementaci vkládání prologu slouží funkce `emitPrologue` a pro implementaci vkládání epilogu slouží funkce `emitEpilogue`. Obě funkce je možné najít v příslušné třídě `RegisterInfo`.

2.8.5 Implementace prologu/epilogu v AVR32

Při implementaci prologu jsem musel řešit dva zcela rozdílné případy, v závislosti na tom, zda daná funkce používá ukazatel rámce či nikoliv. Oba případy jsou přehledně zobrazeny na obrázku 2.2.



Obrázek 2.2: Adresování lokální proměnné `a` a prvního parametru `p1` předávaného na zásobníku v backendu AVR32

V případě, že ukazatel rámce není použit, je situace jednoduchá. Stačí vygenerovat instrukci snižující ukazatel zásobníku o hodnotu vrácenou metodou `getStackSize` objektu třídy `FrameInfo` příslušejícímu dané funkci.

Prolog s využitím ukazatele rámce

Pokud ovšem ukazatel rámce použit je, musím pro něj alokovat některý z fyzických registrů. Pro tento účel jsem zvolil registr `R7`, ovšem vyhovoval by i kterýkoli jiný, který nemá speciální účel nebo není použit pro předávání parametrů. Protože tento registr bude mít uvnitř funkce speciální význam, je nutné označit jej jako rezervovaný pomocí funkce `getReservedRegs`, aby byl vyřazený z množiny registrů volných pro alokaci. Podle volací konvence, která je použita v backendu AVR32, je registr `R7` ukládaný. Jinými slovy, funkce je povinna zachovat jeho hodnotu. V běžném případě by stačilo tento registr označit jako ukládaný funkcí `getCalleeSavedRegs` a LLVM by ho už automaticky v případě potřeby uložilo. V tomto případě bude ovšem práce s registrem `R7` probíhat zcela manuálně a jeho ukládání je třeba implementovat též manuálně. Při vybírání vhodného registru jsem úmyslně vybíral ukládaný registr, abych nemusel mít obavy, že některá zavolaná funkce jeho obsah zničí.

Po vybrání vhodného registru a po jeho označení jako rezervovaného provádím následující kroky:

1. Uložení (push) obsahu registru `R7` na zásobník.
2. Nastavení registru `R7` na aktuální hodnotu ukazatele zásobníku `SP`.
3. Snížení ukazatele zásobníku `SP` o velikost funkčního rámce

Výsledkem je funkce s ukazatelem rámce `R7` a s potřebným místem na zásobníku pro ukládání lokálních proměnných. V důsledku vložení místa pro uložení registru `R7` mezi parametry funkce a lokální proměnné je ovšem potřeba při eliminaci frame indexu patřičně upravit výsledný offset (viz. 2.8.6).

Epilog

Při vkládání epilogu je třeba odstranit celý rámec dané funkce a obnovit původní hodnotu registru `R7`, pokud byl použit pro účel ukazatele rámce. Kroky jsou tedy prováděny v přesně opačném pořadí, než tomu bylo u prologu. Pokud je ovšem použit ukazatel rámce (a funkce tedy používá objekty proměnné velikosti), je třeba zvýšit ukazatel zásobníku nejen o hodnotu alokovanou během prologu, ale i o velikost všech dynamických alokací uskutečněných uvnitř funkce. Místo přičítání konkrétní hodnoty k registru ukazatele zásobníku `SP` do něj tedy kopíruji obsah ukazatele rámce `R7`.

Kromě toho je třeba zajistit, aby linkový registr `LR` obsahoval před návratem z funkce původní návratovou adresu. Toho je dosaženo jednoduše tak, že registr `LR` je označen funkcí `getCalleeSavedRegs` jako ukládaný. LLVM se tedy o jeho případné ukládání postará automaticky.

2.8.6 Eliminace frame indexu

V sekci věnované adresování paměti jsem se zmínil o tom, že objekty na zásobníku jsou adresovány pomocí tzv. *frame indexů* případně doplněných offsetem, a že k jejich eliminaci dochází až při velmi pozdní fázi generování kódu. Nyní nastal vhodný okamžik věnovat se právě této části.

Eliminace frame indexu se implementuje ve funkci `eliminateFrameIndex` patřící do příslušné třídy `RegisterInfo`. Tato funkce je zavolána na každou instrukci, která obsahuje operand ve formě frame indexu. Pro eliminaci frame indexu v backendu AVR32 provádím následující kroky:

1. Projít všechny operandy dané instrukce a najít příslušný frame index. Z fáze výběru adresy je navíc bezprostředně za tento frame index přidán další operand ve formě přímé hodnoty udávající offset.

2. Zvolit bázový registr na základě návratové hodnoty funkce `hasFP`.

3. Funkcí `FrameInfo::getObjectOffset` získat offset daného frame indexu vzhledem k bázi funkčního rámce, posléze k němu přičíst přímou hodnotu z následujícího operandu, a v případě báze ve formě ukazatele zásobníku `SP` přičíst ještě velikost funkčního rámce, aby se offset vztahoval na `SP`.

4. Při adresaci pomocí ukazatele rámce přičíst k aktuálnímu offsetu hodnotu 4, pokud je nezáporný. Jinými slovy, přeskočit, uloženou během vykonávání prologu, původní hodnotu registru `R7`, pokud jsou adresovány funkční parametry (viz. 2.2).

5. Nahradit frame index zvoleným bázovým registrem a nahradit starý offset nově vypočteným.

Omezená délka offsetu Podstatným omezením je délka offsetu omezená architekturou AVR32 na 16 bitů umožňující tak hodnoty pouze v rozsahu 32 kB v obou směrech. Pro velikost funkčního rámce společně s velikostí všech předaných parametrů se mi daná hodnota zdá dostačující, proto jsem se dané omezení nesnažil nějak obcházet. Pouze jsem na závěr eliminace indexu přidal kontrolu, zda offset leží v povoleném rozsahu hodnot, aby překlad skončil alespoň chybou. Tvůrce programu by byl následně nucen k předávání velkých parametrů odkazem a k dynamické alokaci paměti. Pokud by ale i přesto bylo nutné dané

omezení vyřešit, v LLVM existuje nástroj jménem `Scavenger`, který vrací právě nevyužívaný volný registr, popř. některý v případě potřeby bezpečně uvolní. Do tohoto registru je následně možné uložit úplnou adresu daného objektu na zásobníku a adresovat ho pomocí něj.

2.9 Tisk assembleru

Aby mohl být výsledný kód vytisknut do assemblerového souboru, je třeba implementovat příslušnou třídu `AsmPrinter`, jejíž zdrojový kód bývá většinou v samostatném podadresáři. Při tisku assembleru bude následně pro každou strojovou funkci zavolána funkce `runOnMachineFunction`. Tato funkce by měla vytisknout direktivy v záhlaví udávající název sekce, požadavky na zarovnání a definici globálního symbolu. Dále by měly být vytisknuty těla jednotlivých základních bloků společně s jejich návěštími.

Pro tisk jednotlivých instrukcí slouží funkce `printInstruction`, která je automaticky generována nástrojem TableGen. Tato funkce následně volá funkce pro tisk jednotlivých operandů v závislosti na jejich typu. Standardně bývá volána funkce `printOperand`, ovšem např. operandy, u kterých byla uvedena jiná funkce pro jejich tisk, budou vytisknuty touto příslušnou funkcí.

Všechny potřebné funkce pro tisk operandů je nutné implementovat ručně, ovšem s minimálními úpravami je možné využít kód z ostatních backendů.

Při implemetaci výše zmíněných funkcí jsem se snažil dodržet formát direktiv, který používá překladač GCC. Dále upozorňuji, že funkce pro tisk globálních symbolů a operandů instrukcí inline assembleru jsem z časových důvodů již příliš nestudoval, čili nemůžu zaručit jejich správnost v backendu AVR32.

Kapitola 3

Závěr

Výsledkem mého dlouhého snažení je vytvořený funkční backend architektury AVR32 pro překladač LLVM verze 2.6. Jelikož tento překladač danou architekturu ještě oficiálně nepodporuje, mohla by v něm má práce najít určité uplatnění.

Backend byl úspěšně otestován ve zkušební verzi komerčního simulátoru IAR Embedded Systems na algoritmus výpočtu kontrolního součtu CRC32, na rekurzivní výpočet faktoriálu a na použití funkce využívající deklarovaného pole proměnné velikosti na zásobníku. V důsledku odlišných direktiv a formátu návěští se ovšem simulace neobešla bez předešlé modifikace výstupních assemblerových souborů ve formátu GNU.

Backend AVR32 má ovšem několik následujících nedostatků:

1. Pro adresaci globálních symbolů využívá techniky linker relaxing podporovanou překladačem GCC, což výrazně omezuje jeho použití s assemblery a linkery ostatních překladačů.

2. Použitá volací konvence se neshoduje s volací konvencí překladačů IAR a GCC u volání funkcí s proměnným počtem parametrů a při předávání 64-bitových hodnot, čímž je omezeno kombinování modulů přeložených mým backendem s moduly přeloženými ostatními překladači.

3. Offset použitý při adresování objektů na zásobníku je omezen na délku 16 bitů, což znemožňuje překlad funkcí, které potřebují větší prostor na zásobníku.

4. Pro snižování či zvyšování ukazatele zásobníku unitř prologu, epilogu a při alokaci či uvolnění volacího rámce, je použita instrukce `sub`, jejíž přímý operand je délkově omezen na 21 bitů. Tímto je znemožněno upravovat ukazatel zásobníku o větší hodnotu. Ovšem v důsledku předchozího omezení je toto omezení irelevantní.

Ovšem za předpokladu, že bude můj backend použit výhradně s assemblerem a linkerem překladače GCC, nebudou míchany moduly zkompilované různými překladači a tvůrci programů nebudou vytvářet funkce s velkými nároky na velikost zásobníkového rámce, jsou tato omezení přípustná.

Během studia struktury překladače LLVM jsem často narážel na nedostatečnou míru dokumentace a komentářů, což mi výrazným způsobem zpomalovalo a ztěžovalo práci. Mnohdy mi nezbyla jiná možnost, než studovat sémantiku určitých funkcí a výrazů přímo z kódu již hotových backendů, což se většinou neobešlo bez předchozího nastudování daných architektur. I přestože jsou názvy funkcí a jejich argumentů poměrně intuituvní a jejich

význam se dá odvodit, chybí zde určitá formálnost a člověk si nemůže být jistý, zda význam pochopil opravdu správně.

Jako možné rozšíření určitě vidím odstranění některých těchto omezení, což by backendu poměrně zvýšilo míru možného uplatnění v praxi.

Většina mé práce spočívala ve studování způsobu psaní backendů architektur v překladači LLVM. Jakmile jsem určitým částem porozuměl, vlastní implementace již šla velmi rychle. Přičemž většina kódu se dala s poměrně malým zásahem znovu použít z již hotových backendů.

Významným přínosem, který mi tato práce dala, jsou cenné znalosti nabyté v oblasti překladačů a v architektuře stále více oblíbeného překladače LLVM, které do budoucna určitě nebudou k zahození.

Literatura

[1] *Getting started with GCC for AVR32*. `http://www.atmel.com/dyn/resources/prod_documents/doc32074.pdf`.

[2] *AVR32 IAR C/C++ Compiler Reference Guide*. 2006, dokument dodávaný společně s překladačem IAR.

[3] *AVR32 Architecture Document* [online]. `http://www.atmel.com/dyn/resources/prod_documents/doc32000.pdf`, Rev. 3200B-11/07 [cit. 2009-04-05].

[4] Criswell, J.; Lattner, C.; et al: *Getting Started with the LLVM System* [online]. `http://llvm.org/releases/2.6/docs/GettingStarted.html`, Last modified: 2009-10-24 [cit. 2010-04-05].

[5] Erhardt, C.: *Design and Implementation of a TriCore Backend for the LLVM Compiler Framework* [online]. `http://www.opus.ub.uni-erlangen.de/opus/volltexte/2010/1659/pdf/tricore_llvm.pdf`, Last modified: 2009-09-01 [cit. 2010-04-05].

[6] Lattner, C.: *TableGen Fundamentals* [online]. `http://llvm.org/releases/2.6/docs/TableGenFundamentals.html`, Last modified: 2009-10-24 [cit. 2010-04-05].

[7] Lattner, C.; Adve, V.: *LLVM Language Reference Manual* [online]. `http://llvm.org/releases/2.6/docs/LangRef.html`, Last modified: 2009-10-24 [cit. 2010-04-05].

[8] Lattner, C.; Wendling, B.; et al: *The LLVM Target-Independent Code Generator* [online]. `http://llvm.org/releases/2.6/docs/CodeGenerator.html`, Last modified: 2009-10-24 [cit. 2010-04-05].

[9] Woo, M.; Brukman, M.: *Writing an LLVM Compiler Backend* [online]. `http://llvm.org/releases/2.6/docs/WritingAnLLVMBackend.html`, Last modified: 2009-10-24 [cit. 2010-04-05].