VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

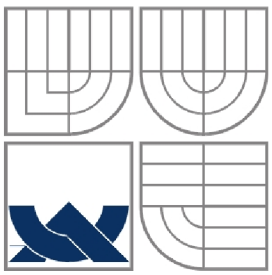
LIBOVOLNÁ BITOVÁ ŠÍŘKA DATOVÉHO TYPU INTEGER V PLATFORMĚ LLVM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

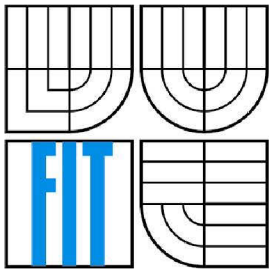
AUTOR PRÁCE
AUTHOR

MARTIN VEŠKRNA

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

LIBOVOLNÁ BITOVÁ ŠÍŘKA DATOVÉHO TYPU INTEGER V PLATFORMĚ LLVM

VARIABLE BIT WIDTH OF INTEGER IN LLVM PLATFORM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MARTIN VEŠKRNA
AUTHOR

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

BRNO 2010

Abstrakt

Tato práce se zabývá úpravou kompilační platformy LLVM/Clang tak, aby podporovala libovolnou bitovou šířku u datového typu integer.

Abstract

This Bachelor thesis describes changes on compilation platform LLVM/Clang for support variable bit width of integer data types.

Klíčová slova

bitová šířka, integer, LLVM, Clang, C

Keywords

bit width, integer, LLVM, Clang, C

Citace

Martin Veškrna: Libovolná bitová šířka datového typu integer v platformě LLVM, bakalářská práce, Brno, FIT VUT v Brně, 2010

Libovolná bitová šířka datového typu integer v platformě LLVM

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Prof. Ing. Tomáše Hrušky, CSc.
Další informace mi poskytl pan Ing. Adam Husár.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Veškrna
17. května 2010

Poděkování

Rád bych poděkoval Prof. Ing. Tomáši Hruškovi CSc. za vedení práce a Ing. Adamu Husárovi za rady, trpělivost a shovívavost.

Obsah

1 Úvod

V poslední době dochází k masivnímu rozvoji tzv. vestavěných zařízení (embedded systems). Jedná se o jednoúčelové systémy, ve kterých je řídicí počítač zcela zabudován do zařízení, které ovládá. Na rozdíl od univerzálních počítačů, jako jsou osobní počítače, zabudované počítače jsou většinou specializované, určené pro předem definované činnosti. Vzhledem k tomu, že systém je určen pro konkrétní účel, mohou jej tvůrci při návrhu optimalizovat pro konkrétní aplikaci a tak snížit cenu výrobku. Vestavěné systémy jsou často vyráběny sériově ve velkém množství, takže úspora bývá znásobena velkým počtem vyrobených kusů.

Na Fakultě informačních technologií Vysokého učení technického v Brně se zabývá výzkumná skupina Lissom vývojem systému pro podporu návrhu těchto vestavěných zařízení. Pomocí speciálního jazyka ISAC se snaží popsat architekturu mikroprocesoru, aby bylo možné pomocí automatizovaných postupů vytvořit sadu nástrojů, které se používají při vývoji aplikací pro daný mikroprocesor. Jedná se o assembler, linker, simulátor, disassembler a debugger. Jako kompilační platformu si zvolili LLVM (Low Level Virtual Machine) jejíž součástí je i překladač jazyků rodiny C Clang.

U některých typů mikroprocesorů je možné se setkat s registry, jejichž bitová šířka není kompatibilní s datovými typy používanými v jazyce C. Jedná se například o signálové procesory, které díky optimalizaci rychlosti a velikosti čipu mají 24 bitové registry. Použití neupraveného jazyka C pro vývoj programů pro tyto mikroprocesory je potom značně omezené.

Aby bylo možné použít nástroje vyvíjené skupinou Lissom i pro procesory s nestandardně širokými registry, je nutné upravit platformu LLVM/Clang tak, aby zvládala libovolně široký celočíselný datový typ integer.

Samotný mezijazyk, který tato platforma používá, již libovolně široký integer implementuje. Zbývá upravit přední část překladače – Clang. Tímto úkolem se zabývá tato bakalářská práce.

V kapitolách 2, 3, 4 a 5 popisuji jednotlivá teoretická východiska mojí práce. V kapitole 2 Jazyk C uvádím některé konstrukce jazyka C, kterých se úprava překladače týká. V kapitole 3 Fáze překladu rozebírám jednotlivé fáze, které je potřeba provést při překladu zdrojového kódu z jednoho jazyka do druhého. Samotnou kompilační platformu LLVM a Clang blíže přibližuji v kapitolách 4 Vnitřní reprezentace programu pomocí LLVM a 5 Clang. Kapitolu 6 jsem věnoval popisu problému, kterého se tato práce týká. V poslední kapitole vysvětluji jednotlivé kroky při úpravě překladače, aby podporoval libovolně široký datový typ integer.

2 Jazyk C

V této kapitole jsou uvedeny některé informace o jazyce C. Vybrány jsou ty pasáže, které se týkají této práce. V první části se nachází formální popis konstrukcí jazyka, které se týkají celočíselných datových typů. V další části jsou popsány některé datové typy a poslední část se věnuje kompatibilitě a konverzi datových typů.

Informace obsažené v této kapitole jsem čerpal z [2].

2.1 Konstrukce jazyka

V této kapitole jsou uvedeny některé konstrukce jazyka C, kterých se týká tato práce.

2.1.1 Konstanty

Jazyk C rozlišuje tyto konstanty.

constant :
integer-constant
floating-constant
character-constant
string-constant

Pro tuto práci jsou podstatné konstanty typu integer.

integer-constant :
decimal-constant integer-suffix$_{opt}$
octal-constant integer-suffix$_{opt}$
hexadecimal-constant integer-suffix$_{opt}$

integer-suffix :
long-suffix unsigned-suffix$_{opt}$
long-long-suffix unsigned-suffix$_{opt}$
unsigned-suffix$_{opt}$ *long-suffix*
unsigned-suffix$_{opt}$ *long-long-suffix*

long-suffix : one of
l L

long-long-suffix : one of
ll LL

unsigned-suffix : one of
u U

Pravidla pro určení základu číselné soustavy:

1. Pokud konstanta začíná **0x** nebo **0X,** jedná se o zápis v hexadecimální soustavě.
2. V případě, že konstanta začíná **0**, jedná se o oktalovou soustavu.

3. V ostatních případech se jedná o decimální soustavu.

Konstanty mohou nabývat několika datových typů:

- písmeno **l** nebo **L** indikuje konstantu typu **long,**
- písmena **ll** nebo **LL** indikují konstantu typu **long long** a
- písmena **u** nebo **U** označují bezznaménkový typ (**unsigned**)

Unsigned příponu (**u** nebo **U**) je možné kombinovat s příponami long (**l**) nebo long long (**ll**) v libovolném pořadí.

2.1.2 Specifikace typu, deklarátor a inicializátor

Specifikátor typu poskytuje informaci o datovém typu identifikátoru, který bude v programu použit. Deklarátor zavádí nové jméno a zároveň přidává informaci o datovém typu. Deklarace proměnné může být doprovázena inicializátorem, který zajistí, že proměnná bude nabývat určité hodnoty okamžitě po jejím vzniku.Výraz použitý při inicializaci musí mít stejný datový typ jako inicializovaná proměnná. Výchozí hodnota pro statický integer je 0.

Příklad

```
int prom = 20+50;
```

`int` je specifikátor typu, `prom` je deklarátor a výraz `20+50` inicializuje obsah nově vzniklé proměnné.

2.2 Datové typy

Datový typ je množina hodnot a množina operací nad těmito hodnotami. Datový typ proměnných je určen deklarací, datový typ výrazu se získá vyhodnocením výrazových pravidel.

<table>
<tr><th>Datový typ</th><th colspan="3">Kategorie</th></tr>
<tr><td>short, int, long, long long</td><td rowspan="4">Integrální typy</td><td rowspan="5">Aritmetické typy</td><td rowspan="6">Skalární typy</td></tr>
<tr><td>char</td></tr>
<tr><td>bool</td></tr>
<tr><td>enum {...}</td></tr>
<tr><td>float, double, long double</td><td>Typy s plovoucí řádovou čárkou</td></tr>
<tr><td>T *</td><td colspan="2">Typ ukazatel</td></tr>
<tr><td>T [...]</td><td colspan="2">Typ pole</td><td rowspan="2">Agregované typy</td></tr>
<tr><td>struct {...}</td><td colspan="2">Typ struktura</td></tr>
<tr><td>union {...}</td><td colspan="3">Typ union</td></tr>
<tr><td>T (...)</td><td colspan="3">Typ funkce</td></tr>
<tr><td>void</td><td colspan="3">Typ void</td></tr>
</table>

Tabulka 1: Datové typy jazyka C podle [2]

2.2.1 Datový typ integer

Jazyk C poskytuje u datového typu integer několik bitových šířek (**`short`**, **`int`**, **`long`**, **`long long`**). Toto umožní vytvořit soulad s hardwarem, pro který jsou programy v jazyce C psány. Konkrétní bitovou šířku ale není možné určit.

2.2.2 Operátor TYPEDEF

Pomocí operátoru typedef je možné pojmenovat libovolný jiný deklarátor a toto nové jméno později používat pro deklaraci proměnných.

2.3 Kompatibilita datových typů

Pro dva datové typy v jazyce C platí, že jsou kompatibilní, pokud jsou stejného typu nebo to jsou ukazatele, funkce nebo pole se stejnými vlastnostmi.

Dva aritmetické typy jsou kompatibilní, pokud jsou stejného typu. Pokud může být typ zapsán pomocí různých typů specifikátorů, jedná se o jeden datový typ. **short** a **short int** vyjadřují stejný datový typ. Ale **unsigned int** a **int** jsou dva různé datové typy. Typy **char**, **unsigned char** a **signed char** jsou vždy různé typy.

Použití **typedef** nevytvoří nový datový typ, ale synonymum datového typu.

Příklad

V následující deklaraci typy **p** a **q** jsou stejné, typy **x** a **y** jsou také stejné, ale nejsou identické s typem **u**. Typ **TS** a **struct S** jsou stejné. Typy **u**, **v**, **w** jsou také stejné.

```
char * p, q;
struct {int a, b;} x, y;
struct S {int a, b;} u;
typedef struct S TS;
TS y;
```

Příklad

V následující deklaraci typ **my_int** je stejný jako typ **int**. Typ **my_function** je stejný jako typ „**float** *()“.

```
typedef int my_int;
typedef float *my_function();
```

Příklad

Proměnné **w**, **x**, **y** a **z** jsou identického datového typu.

```
struct S { int a, b; } x;
typedef struct S t1, t2;
struct S w;
t1 y;
t2 z;
```

2.4 Uložení dat v paměti

Všechny datové objekty kromě bitových polí jsou za běhu počítače uloženy v abstraktních, tzv. *storage units*. Každá tato jednotka má předem stanovenou pevnou velikost v bitech. Jednotlivé bity nabývají dvou hodnot – 0 a 1. Každá jednotka musí být jednoznačně adresovatelná. Počet bitů této jednotky musí být větší nebo stejný jako je počet bitů potřebných pro uložení znaku ze základní znakové sady.

Podle definice je velikost datového objektu počet *storage unit* potřebných pro jeho reprezentaci. Velikost datového objektu typu **`char`** je 1.

Protože všechny objekty jednoho datového typu potřebují stejný počet *storeage unit* pro jejich reprezentaci, můžeme použít operátor **`sizeof`** pro určení velikosti datového objektu ještě před jeho deklarací a definicí. Říkáme, že datový typ je „větší“ nebo „delší“ než jiný, pokud jeho velikost je větší. Obdobně říkáme, že datový typ je „menší“ nebo „kratší“, pokud jeho velikost je menší.

2.5 Konverze datových typů

Jazyk C poskytuje pro hodnoty jednoho datového typu, které mají být konvertovány do jiného datového typu, tyto možnosti konverze:

- pro explicitní přetypování do jiného datového typu může být použit výraz,
- operand může být implicitně přetypován do jiného datového typu v případě přípravy nebo provádění nějaké aritmetické nebo logické operace,
- objekt jednoho datového typu může být v případě přiřazení do jiného objektu implicitně přetypován,
- argument předávaný funkci může být implicitně přetypován do typu uvedeného v hlavičce funkce a
- návratová hodnota funkce může být implicitně přetypována podle datového typu objektu, kam má být uložena.

2.5.1 Změny v reprezentaci hodnot při konverzi datových typů

Konverze hodnoty z jednoho datového typu do druhého může a nemusí vyvolat změnu reprezentace. Pokud datové typy mají různou velikost, změna reprezentace je nutná. Při konverzi z celočíselného datového typu do typu s plovoucí řádovou čárkou je změna reprezentace nutná i v případě stejné velikosti těchto typů. Změna reprezentace při převodu ze znaménkové do bezznaménkového typu nemusí být vždy vyvolána (například z **int** do **unsigned int**).

2.5.2 Triviální konverze

K triviální konverzi dojde, pokud datový typ, který se má konvertovat, je kompatibilní s datovým typem, do kterého se má konvertovat. K žádné změně v reprezentaci dat nedojde.

2.5.3 Konverze do celočíselných datových typů

Skalární typy (aritmetické a ukazatel) mohou být konvertovány do celočíselných datový typů (integer).

Konverze do datového typu bool

Tato konverze je trochu odlišná, než konverze do jiných celočíselných datových typů. Pokud konvertujeme aritmetickou hodnotu do typu **bool**, tak hodnota 0 je konvertována jako 0, ostatní jako 1. V případě konverze z datového typu ukazatel, tak hodnota null je konvertována jako 0, ostatní jako 1. Pokud konvertujeme datový typ bool do jiného datového typu, výsledek je 0 nebo 1.

Konverze do ostatních celočíselných datových typů (kromě bool)

Pravidla pro tuto konverzi se rozlišují podle toho, jestli matematická hodnota po konverzi je stejná, jako před konverzí. Například, pokud bezznaménkové celé číslo 15 je převedeno na znaménkové celé číslo, tak výsledek bude opět 15. Ke změně hodnoty nedošlo.

Pokud není možné vyjádřit originální hodnotu v novém datovém typu, nastávají dvě možnosti. V případě, že cílový datový typ je znaménkový, dojde k přetečení a nová hodnota není technicky definovatelná. Pokud je cílový datový typ znaménkový, výsledná hodnota je dána výrazem $a \bmod 2^n$, kde a je původní hodnota a n je počet bitů použitých k reprezentaci cílového datového typu. V případě, že jsou znaménkové datové typy reprezentovány pomocí dvojkového doplňku, není potřeba v případě konverze do bezznaménkových typů provádět změnu reprezentace dat. Toto je nutné, pokud znaménkové typy používají jinou reprezentaci.

Pokud je konvertován bezznaménkový datový typ do stejně velkého znaménkového typu, dojde k přetečení, když je hodnota příliš velká, než aby mohla být uložena ve znaménkovém typu. Toto nastává, pokud nejvýznamnější bit původní hodnoty je 1. Neošetření této situace zapříčiní vznik záporné hodnoty.

V případě konverze do většího datového typu, než je zdrojový, nastává jediná situace, kdy není možné hodnotu reprezentovat v novém typu – konverze záporné znaménkové hodnoty do bezznaménkového typu. Zde je nutné nejprve konvertovat do znaménkového typu stejně velkého jako cílový typ a až poté do cílového typu.

Pokud je cílový typ menší než zdrojový a oba, zdrojový i cílový typ, jsou bezznaménkové, stačí konverzi provést odstraněním nejvýznamnějších bitů původní hodnoty. Stejným způsobem se konvertují větší znaménkové typy do menších znaménkových. Toto je ovšem možné jen v případě, že znaménkový typ používá ke své reprezentaci dvojkový doplněk.

Konverze z *datového typu s plovoucí řádovou čárkou* se provede odříznutím části za řádovou čárkou. Výsledek této konverze není definovaný, pokud hodnota po oříznutí nemůže být uložena v cílovém datovém typu. Toto nastává, pokud je hodnota příliš velká nebo by mělo dojít k uložení záporné hodnoty bezznaménkového typu.

Při konverzi z *datového typu ukazatel,* pokud není cílovým typem bool, tak se pro účely konverze na ukazatel nahlíží jako na bezznaménkový integer o stejné délce jako ukazatel. Potom dojde ke konverzi, jak bylo popsáno předem.

3 Fáze překladu

Prakticky každý programátor dnes programuje v některém vyšším programovacím jazyce (např. PASCAL či C). Procesor počítače ovšem provádí programy ve svém strojovém jazyce. Má-li být tedy programátorův program počítačem zpracován, musí být nejprve transformován na funkčně ekvivalentní program ve strojovém jazyce, což provádí systémový program, který se nazývá kompilátor.

Tato kapitola se věnuje jednotlivým fázím, které je nutné provést při překladu z jednoho programovacího jazyka do druhého. Informace jsou převzaty z [1] a [4].

Překlad zdrojového programu zpravidla probíhá v šesti hlavních fázích:

1. lexikální analýza,
2. syntaktická analýza,
4. sémantická analýza,
5. generování vnitřní formy programu,
6. optimalizace a
7. generování cílového programu.

3.1 Lexikální analýza

Hlavním úkolem lexikální analýzy je nalézt, rozpoznat a zakódovat jednotlivé lexikální symboly zdrojového programu. Lexikální symboly jsou popsány pomocí regulárních výrazů. Aby bylo možné načítat zdrojový text programu a ten poté rozdělit podle pravidel regulárních výrazů na jednotlivé části (lexikální symboly), je nutné sestrojit lexikální převodník. Ten se od lexikálního automatu odlišuje tím, že poskytuje výstup. Pomocí lexikálního automatu je jen možné zjistit, jestli právě načtený lexikální symbol odpovídá pravidlům popsaným regulárními výrazy. Lexikální převodník navíc přidá informaci o právě načteném symbolu, kterou je možné předat dále syntaktickému analyzátoru.

3.2 Syntaktická analýza

Úkolem syntaktického analyzátoru je určit syntaktickou strukturu zdrojového programu. To tedy znamená, že se syntaktický analyzátor snaží sestrojit pro dané slovo lexikálních symbolů (reprezentující zdrojový program) derivační strom. Pokud se mu to podaří bez chyby, je program po syntaktické stránce zapsán správně.

Překladová gramatika je bezkontextová gramatika

$$G = (N, V_I V_O, Q, S),$$

ve které je množina terminálních symbolů rozdělena na dvě disjunktní podmnožiny, množinu vstupních symbolů V_I a množinu výstupních symbolů V_O.

Syntaktický analyzátor používá pro vytváření derivačního stromu zásobníkový převodník. Ten je zkonstruován na základě překladové gramatiky. Zásobníkový převodník je v podstatě zásobníkový automat, který umožňuje v každém kroku výstup řetězu.

Syntaktickou analýzu lze rozdělit podle dvou používaných postupů, zdola nahoru a shora dolů.

3.2.1 Syntaktická analýza shora dolů

Analyzátor začíná počátečním symbolem a snaží se jej převést na vstup. Schematicky řečeno začíná největšími prvky, které postupně rozbíjí na menší části, dokud se nedostane k terminálním symbolům, které může porovnat se vstupem. Příkladem syntaktické analýzy shora dolů je LL analýza.

LL syntaktický analyzátor

Jedná se o syntaktický analyzátor shora-dolů pro bezkontextové gramatiky. Analyzuje vstup zleva (**L**eft) doprava a konstruuje nejlevější derivaci (**L**eftmost) věty. Gramatiky, které jsou takto analyzovatelné, se nazývají LL gramatiky.

LL(*k*) gramatika

Tato gramatika generuje jazyk typu LL(*k*). LL gramatika se nazývá LL(*k*), jestliže pro deterministickou analýzu věty je potřeba znát maximálně *k* následujících symbolů a není nutné použít backtracking. LL gramatiky konstruují nejlevější derivaci.

Často používanou LL(*k*) gramatikou je LL(1) gramatika, protože i přes jistá omezení této gramatiky stačí k deterministické analýze znát maximálně jeden následující symbol, což významně zjednodušuje konstrukci analyzátoru. Naopak LL(0) gramatiky jsou nevhodné, protože mohou generovat jen jazyk s konečným počtem slov a není zde možná rekurze. Existují nedeterministické postupy, jak transformovat gramatiky LL(*k*) na gramatiky LL(1).

3.2.2 Syntaktická analýza zdola nahoru

Analyzátor začíná vstupním textem a snaží se jej převést na počáteční symbol. Prakticky tedy hledá nejprve pravidla, která obsahují dané terminální symboly, pak pravidla, která mohou takovým pravidlům předcházet, atd. Příkladem syntaktické analýzy zdola nahoru je LR analýza. Jiný termín pro tento druh syntaktické analýzy je shift-reduce parsing (doslova „posuň-zmenši“).

LR(*k*) gramatika

Je taková gramatika, pomocí níž je možné provést syntaktickou analýzu zdola nahoru postupným načítáním terminálů (stejně jako u LL - zleva doprava). V rozkladové tabulce se nepoužívá rozklad neterminálů, ale redukce terminálů na neterminály a ty se dále redukují na jim předcházející. *k* označuje počet terminálů, které je nutné dopředu načíst ze vstupního souboru, aby nebylo nutné použít backtracking.

3.3 Sémantická analýza

Během sémantické analýzy se postupně prochází symboly či skupiny symbolů získané ze syntaktické analýzy a přiřazuje se jim význam.

Pokud například skupina symbolů představuje použití konkrétní proměnné, pak analyzátor zjišťuje, zda je proměnná už deklarována (pokud je to požadováno, nevyžaduje například programovací jazyk Perl či PHP) a zda je správně použita vzhledem k jejímu datovému typu. Dále například u operací kontroluje, zda jsou operandy správného typu, případně provede potřebnou konverzi datového typu.

Pro všechny typy operátorů jsou stanoveny předpisy pro jejich operandy. Například pro některé jazyky platí, že pokud sčítáme řetězec a číslo, pak dojde k sémantické chybě. Pokud je to možné, pak je sémantický analyzátor schopný implicitního přetypování v případě operandů různých datových typů. Implicitní přetypování se provádí na základě stanovených priorit jednotlivých typů. Pokud je

tedy jeden operand typu s větší prioritou, pak se automaticky druhý přetypuje na typ prvního operandu.

U modernějších programovacích jazyků, kde je umožněné přetěžování funkcí, musí překladač správně určit, která funkce je volána a zda je volána správně. Tedy zda předáváme správný počet a typ parametrů.

3.4 Generování vnitřní formy programu (mezikódu)

Generování mezikódu je další fází v procesu překladu zdrojového kódu programu. Nedochází k ní až po dokončení syntaktické a sémantické analýzy, ale instrukce jsou generovány průběžně. Toto přináší výhodu v podobě menších paměťových nároků (není nutné si někde pamatovat celý derivační strom).

Důvodů, proč generovat vnitřní formu programu, je několik.

1. Vnitřní forma programu je méně závislá na cílové architektuře než samotný strojový kód. Závislost vnitřní formy programu na cílové architektuře zůstává zachována u velikosti ukazatelů, šířce celočíselných datových typů a případně dalších vlastnostech. Při použití vnitřní formy programu je tedy mnohem jednodušší upravit překladač pro jednotlivé operační systémy nebo cílové architektury. K největším změnám dojde v poslední části – generátoru instrukcí. Ve zbylých částech překladače se provádějí jen malé změny.
2. Ještě před samotným vygenerováním cílového programu je možné na úrovni mezikódu provádět optimalizace.

Dvě nejpoužívanější vnitřní formy programu jsou polský zápis a trojce.

Polský zápis (polská notace) se dále dělí na prefixový a postfixový. Zápis aritmetických výrazů běžně používaný v matematice je infixový.

Dalším způsobem reprezentace vnitřní formy programu jsou tzv. *trojice*, které mají tvar

$$(i)\ b,\ a_1,\ a_2,$$

kde (i) je pořadové číslo trojice, b je operátor (binární) a a_1 a a_2 jsou operandy. Operandy instrukcí jsou buď proměnné nebo konstanty. Pokud je operandem výsledek výpočtu některé jiné trojice, pak je tento operand zapsán jako ukazatel na tuto trojici. Tyto ukazatele jsou zapisovány (podobně jako pořadová čísla trojic) v závorkách, tj. (i) označuje ukazatel na i-tou trojici (a nikoliv konstantu i).

Aritmetický výraz

$$1 + b * c$$

je pomocí trojic reprezentován takto:

$$(1)\ *,\ b,\ c$$
$$(2)\ +,\ 1,\ (1)\ .$$

Zápis vnitřní formy programu v podobě trojic je výhodný pro následnou optimalizaci, protože mimo jiné nepoužívá pomocných proměnných, což šetří paměť.

Dalšími formami vnitřní reprezentace programu jsou nepřímé trojice, stromy a čtveřice.

3.5 Optimalizace

Optimalizace je proces, ve kterém (přeuspořádáním, eliminací, změnou operací) získáme nový efektivní program, který je ale funkčně ekvivalentní s původním.

Předvýpočet je optimalizace, která odstraní ty operace, jejichž výsledek je znám ještě před spuštěním programu (například sečtení dvou konstant). *Eliminace nadbytečných operací* se snaží vyloučit ty operace, které na výsledek programu nemají žádný vliv.

3.6 Generování strojového kódu (výsledného programu)

Tato etapa je poslední v procesu překladu programu. Zoptimalizovaný program ve vnitřní formě je zde překládán na cílový program, což je zpravidla posloupnost strojových instrukcí nebo posloupnost příkazů jazyka symbolických adres. Je tedy jasné, že výstup generátoru cílového programu je závislý na cílové architektuře.

4 Vnitřní reprezentace programu pomocí LLVM

LLVM (Low Level Virtual Machine) je projekt, který implementuje nízkoúrovňový virtuální stroj. Tento framework lze použít pro mnoho věcí, např. implementaci Just-In-Time optimalizátoru pro konstrukci statického nativního překladače, například jazyka C. LLVM v podstatě poskytuje celou zadní část překladače. Obecné informace jsou převzaty z [9] a popis jednotlivých konstrukcí jazyka z [10].

Strategie kompilace je sestavená tak, aby umožnila efektivní optimalizaci programu napříč celým jeho životním cyklem. LLVM podporuje efektivní optimalizaci v době kompilace, linkování, běhu programu a i poté, co je program nainstalován. Tento proces je transparentní pro vývojáře a udržuje kompatibilitu s existujícími skripty používanými pro překlad.

Virtuální instrukční sada LLVM je nízkoúrovňová objektová reprezentace kódu, která využívá instrukce podobné instrukcím u RISC procesorů, ale poskytuje bohaté, jazykově nezávislé, typované informace o operandech.

LLVM využívá tzv. SSA formu zápisu. Jedná se o formu zápisu instrukcí, kde je do každé proměnné zapsáno pouze jednou. Tato podmínka se může na první pohled jevit jako omezující, ale přidává do kódu informaci o směru toku dat, které může využít optimalizátor.

Infrastruktura překladače LLVM je také kolekce zdrojových kódů, které implementují jazykovou a kompilační strategii. Hlavní komponentou LLVM infrastruktury jsou C a C++ přední část překladače (označený Clang, kompatibilní s GCC), link-time optimalizační systém (optimalizace prováděné v domě linkování objektových souborů), statická zadní část překladače pro řadu architektur, zadní část překladače emitující přenositelný C kód a Just-In-Time kompilátor (překlad probíhá až při spuštění aplikace) pro několik architektur. LLVM nabízí podporu pro tyto architektury: ARM, Alpha, Blackfin, Cell, MSP430, Mips, PIC16, PowerPC, Sparc, SystemZ a X86.

LLVM neobsahuje součásti, které jsou běžné u „high-level“ virtuálních strojů typu Java Virtual Machine nebo Common Language Runtime (.NET Framework). LLVM neobsahuje garbage collection nebo generování kódu za běhu (jedná se o statický překladač). LLVM komponenty mohou být použity pro vytvoření „high-level“ virtuálního stroje nebo jiných systémů.

LLVM je napsán v podmnožině C++ se silným využitím šablon a bez použití výjimek. Je psán jako všeobecně znovupoužitelné knihovny.

Reprezentace kódu v LLVM je navržená tak, aby mohla být použita ve třech rozdílných formách. Tyto tři formy jsou plně ekvivalentní:

- reprezentace uložená v paměti (využívá překladač v době kompilace),
- reprezentace kódu uložená na disku (například pro rychlé načítání při Just-In-Time překladu) a
- pro člověka čitelná reprezentace v podobě jazyka symbolických adres.

4.1 Identifikátory

LLVM identifikátory se dělí do dvou skupin: globální a lokální. Globální identifikátory (funkce, globální proměnné) začínají znakem ‚@‘. Lokální identifikátory (jména registrů, typy) začínají znakem ‚%‘. LLVM rozlišuje tři formáty identifikátorů:

1. pojmenované hodnoty jsou reprezentovány jako řetěz znaků s jejich prefixem (například %foo, @DivisionByZero, %velmi.dlouhy.identifikator),
2. nepojmenované hodnoty jsou reprezentovány pomocí bezznaménkové celočíselné hodnoty s prefixem (například %12, @4) a
3. konstanty.

Klíčová slova jsou v LLVM velmi podobná jako v ostatních jazycích. Jsou zde klíčová slova pro instrukce (‚add', ‚bitcast', ‚ret', apod.), pro primitivní typy (‚void', ‚i32', apod.) a další. Klíčová slova nemohou být v konfliktu s identifikátory, protože ty začínají ‚@' nebo ‚%'.

Příklad

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

Příklad ukazuje několik důležitých lexikálních vlastností LLVM:

1. komentáře jsou uvozeny ‚**;**' a pokračují až na konec řádku,
2. nepojmenované dočasné proměnné jsou vytvořeny, jakmile výsledek operace není možné uložit do pojmenované proměnné a
3. nepojmenované proměnné jsou postupně číslovány.

4.2 Typový systém

Zavedené typování umožňuje provádět řadu optimalizací přímo bez nutnosti předchozí analýzy. Striktní typový systém umožňuje snazší čitelnost kódu a umožňuje zavést další optimalizace, které na běžném tříadresném kódu nejsou možné.

Typy v LLVM lze rozdělit podle následující klasifikace:

Klasifikace	Typy
integer	`i1, i2, i3, ... i8, ... i16, ... i32, ... i64, ...`
floating point	`float, double, x86_fp80, fp128, ppc_fp128`
first class	integer, floating point, pointer, vector, structure, union, array, label, metadata.
primitive	label, void, floating point, metadata.
derived	array, function, pointer, structure, packed structure, union, vector, opaque.

Tabulka 2: Datové typy v LLVM

Typy klasifikované jako *first class* jsou jediné, které mohou být produkovány na výstupu instrukcí.

4.2.1 Primitivní typy

Primitivní typy jsou základní stavební bloky LLVM systému. LLVM rozlišuje tyto primitivní typy:

- integer,

- floating point,
- void,
- label a
- metadata.

Datový typ integer

Integer je typ, který umožňuje reprezentovat celočíselnou hodnotu o bitové šířce od 1 do 2^{23}-1.

Syntaxe

```
iN,
```

kde N je počet bitů.

Příklad

```
i1          ; jednobitový integer
i32         ; 32 bitů široký integer
i1942625    ; velmi široký integer
```

4.2.2 Odvozené datové typy

LLVM implementuje tyto odvozené datové typy:

- agregate,
- pole,
- funkce,
- struktura,
- packed structure,
- union,
- ukazatel,
- vektor a
- opaque.

Odvozený datový typ se skládá z primitivních datových typů nebo odvozených. Například je možné definovat vícerozměrné pole. Dále jsou popsány jen ty datové typy, které se týkají práce.

Datový typ pole

Pole je jednoduchý odvozený typ, který uspořádává jednotlivé elementy sekvenčně v paměti.

Syntaxe

```
[<# elements> x <elementtype>]
```

Příklad

```
[40 x i32]        ; 40 prvkové pole 32 bitových celočíselných hodnot
[3 x [4 x i32]]   ; 3x4 pole 32 bitových celočíselných hodnot
```

Datový typ ukazatel

Typ používaný pro určení umístěni v paměti. Výchozí hodnota je nula.

Syntaxe

```
<type> *
```

Datový typ vektor

Tento typ reprezentuje vektor elementů. Typ vektor je používán, pokud jsou mnohačetná primitivní data paralelně zpracovávána použitím jednoduchých instrukcí (SIMD).

Syntaxe

```
< <# elements> x <elementtype> >
```

Příklad

```
<8 x float>       ; vektor osmi float hodnot
```

4.2.3 Konstanty

LLVM rozlišuje několik druhů konstant. Jednoduché konstanty se používají pro booleovské hodnoty, celočíselné, s plovoucí řádovou čárkou a nulový ukazatel (null). Komplexní konstanty se týkají odvozených datových typů. Jsou potenciálně rekurzivní. Dále používá LLVM konstanty pro určení nedefinované hodnoty.

4.3 Instrukční sada

LLVM instrukční sada se dělí na tyto druhy instrukcí:

- ukončující instrukce,
- instrukce se dvěma operandy,
- instrukce binárních operací,
- instrukce pro práci s pamětí a
- ostatní instrukce.

4.3.1 Ukončující instrukce

Každý základní blok v programu končí některou z těchto instrukcí, která říká, který další blok bude vykonáván po dokončení aktuálního. Tyto instrukce obvykle nevrací hodnotu, jen řídí tok (kudy se bude dále ubírat provádění aplikace).

Příkladem může být instrukce ‚**ret**‘, která slouží k návratu řízení zpět z funkce, odkud byla volána.

4.3.2 Instrukce se dvěma operandy

Tyto instrukce se používají v částech programu, kde probíhá nějaký výpočet. Očekávají dva operandy stejného datového typu, vykonají nějakou operaci a vrací jednu hodnotu. Na vstupu mohou být i mnohonásobná data v podobě vektoru. Výsledek má stejný datový typ jako jsou operandy.

Příkladem může být instrukce ‚**add**‘ pro sčítání dvou celočíselných hodnot nebo dvou vektorů celočíselných hodnot.

Příklad

Ukázka použití instrukce **add**. Zde se sčítají dvě celočíselné hodnoty (konstanta **4** a globální proměnná **%var**).

```
<result> = add i32 4, %var
```

4.3.3 Instrukce binárních operací

Tyto instrukce provádějí různé druhy bitových posunů. Očekávají na vstupu dva operandy stejného datového typu a vracejí jednu hodnotu stejného typu jako je vstup.

4.3.4 Instrukce pro práci s pamětí

Například ‚**alloca**', ‚**load**', ‚**store**', apod.

4.4 Informace o cílové architektuře

Různé architektury používají různé způsoby reprezentace dat. Liší se v použitém způsobu ukládání dat v paměti (little-endian, big-endian), velikostí datových typů apod.

Tyto údaje je možné specifikovat v hlavičce každého souboru s LLVM kódem. Tímto se zadní části překladače předá informace, jak generovat cílový kód. Další možností je určit cílovou architekturu přímo v kódu, který překládá přední část překladače. Ta zajistí vygenerování patřičných hlaviček v souborech s LLVM kódem.

LLVM používá dvě deklarace, které popisují cílovou architekturu:

- data layout a
- target triple.

Tyto údaje sice znemožní přenositelnost kódu na jiné platformy, ale jednoznačně řeknou zadní části překladače, jaké knihovny použít pro generování cílového programu.

4.4.1 Data layout

Syntaxe použitá u *data layout* je následující:

```
target datalayout = "layout specification"
```

layout specification obsahuje jednotlivé specifikace oddělené pomlčkou. Každá specifikace začíná písmenem, které ji určuje. Potom následují parametry.

Příklad některých specifikací:

- **E** - Říká, že se používá big-endian.
- **e** - Data jsou ukládána za použití little-endian.
- **p:size:abi:pref** – Popisuje datový typ ukazatel.
- **isize:abi:pref** – Datový typ integer
- **fsize:abi:pref** – Datový typ s plovoucí řádovou čárkou. Bitová šířka musí být 32 nebo 64.

Použité parametry:

- **size** – bitová šířka,

- **abi** (Application Binary Interfeace) – počet bitů, který se použije při předávání proměnné jako parametr funkce nebo její návratová hodnota,
- **pref** – preferované zarovnání (v bitech).

4.4.2 Target triple

Dalším prostředkem, jak blíže specifikovat cílovou architekturu, jsou tzv. *target triples*. Jedná se o trojici údajů oddělených pomlčkou:

```
target triple = "procesor-výrobce-operačníSystém"
```

4.5 Ukázka LLVM kódu

Kód v jazyce C

```
int global = 13;

int main()
{
  float f = 10.32;
  return (int)f;
}
```

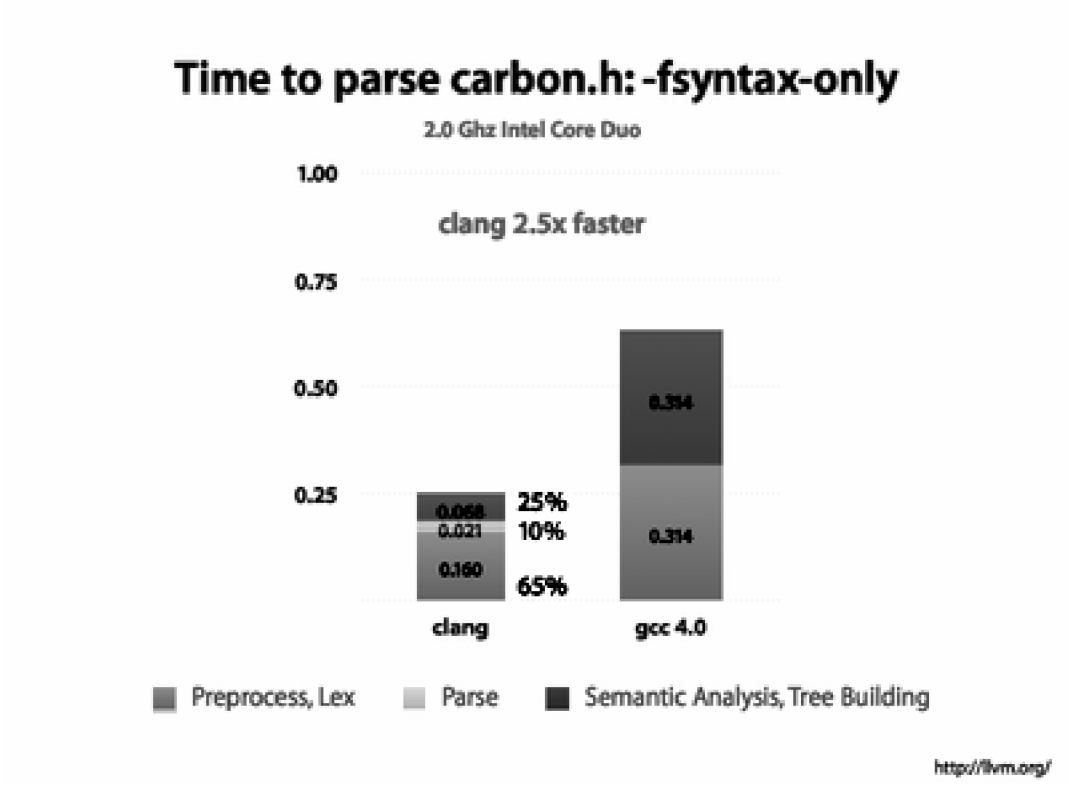
lze v LLVM zapsat takto (bez optimalizace):

```
; ModuleID = 'ukazka.c'
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
i64:64:64-f32:32:32-f64:64:64-f80:128:128-v64:64:64-v128:128:128-
a0:0:64-f80:32:32-n8:16:32"
target triple = "i686-pc-win32"

@global = global i32 13, align 4                ; <i32*> [#uses=0]

define i32 @main() nounwind {
entry:
  %retval = alloca i32, align 4                 ; <i32*> [#uses=3]
  %f = alloca float, align 4                    ; <float*> [#uses=2]
  store i32 0, i32* %retval
  store float 0x4024A3D700000000, float* %f
  %tmp = load float* %f                         ; <float> [#uses=1]
  %conv = fptosi float %tmp to i32              ; <i32> [#uses=1]
  store i32 %conv, i32* %retval
  %0 = load i32* %retval                        ; <i32> [#uses=1]
  ret i32 %0
}
```

5 Clang

Clang je přední část překladače pro jazyky C, C++, Object C a Object C++. V současné době (květen 2010) je stále ještě ve vývoji. Podměty pro vznik tohoto nového překladače byly získat kompilátor s lepší diagnostikou, lepší integrací s prostředím (IDE), licencí kompatibilní s komerčními produkty a rychlý překladač, který je jednoduchý na vývoj a údržbu. Podle tvůrců Clangu nejsou tyto vlastnosti dostupné u překladače GCC používaného v současné době. Obecné informace jsou převzaty z [5] a popis vnitřní struktury překladače z [6].

Výkonnostní srovnání Clangu a GCC (Obrázek 5-1) ukazuje zrychlení v přední části překladače.



Obrázek 5-1 Porovnání doby překladu (jen přední část překladače)

Diagnostika překladače byla vylepšena tak, aby samotné hlášení předalo uživateli co největší množství informací o chybě nebo varování.

Příklad

Ukázka výpisu chybového hlášení z překladačů Clang a GCC.

```
$ gcc-4.2 -fsyntax-only t.c
t.c:7: error: invalid operands to binary + (have 'int' and 'struct A')
$ clang -fsyntax-only t.c
t.c:7:39: error: invalid operands to binary expression ('int' and 'struct A')
  return y + func(y ? ((SomeA.X + 40) + SomeA) / 42 + SomeA.X : SomeA.X);
                      ~~~~~~~~~~~~~~~ ^ ~~~~~
```

Aby bylo možné pomocí Clangu přeložit programy, které byly vyvíjeny pomocí překladače GCC, obsahuje Clang rozšíření, která obsahuje GCC. Tato snaha vede k nahrazení GCC Clangem. V současné době (květen 2010) FreeBSD experimentuje s Clangem jako systémovým překladačem

[7]. Zatím je tato snaha omezena na architektury i386 a amd64. Aktuální stav je takový, že Clang dokáže zkompilovat veškerý C kód ve FreeBSD, všechen Object C kód (kterého je velmi málo) a 2 z 5ti C++ aplikací (devd a gperf).

5.1 Vnitřní struktura překladače

Překladač Clang je rozdělen do několika knihoven, které mohou být použity v rámci jiného projektu. Další výhodou rozdělení je to, že začínajícím vývojářům ulehčuje seznamování s projektem (stačí se detailně zorientovat jen v té části, na které budou pracovat). Některé knihovny jsou popsány níže.

5.1.1 Základní knihovna

Tato knihovna obsahuje několik nízkoúrovňových utilit pro manipulaci s kusy kódu (třídy pro určení, z jaké části zdrojových souborů daný kus kódu pochází), diagnostiku, popis cílové architektury apod.

Diagnostický systém je důležitou součástí překladače při komunikaci s uživatelem. Produkuje varovná nebo chybová hlášení, pokud je kód špatně zapsaný nebo podezřelý. V Clangu každé chybové hlášení obsahuje unikátní ID, instanci třídy **`SourceLocation`** (odkud pochází kód, kterého se hlášení týká), anglický text popisující hlášení a informaci, jestli se jedná o chybu nebo varování.

5.1.2 Knihovna preprocesoru a lexikální analýzy

Tato knihovna obsahuje několik úzce spojených tříd, které jsou využívány při předběžném zpracování kódu (preprocessing) a lexikální analýze. Na rozdíl od jiných překladačů jsou v Clangu preprocesor a lexikální analyzátor sloučeny do jedné části. Hlavní rozhraní této knihovny pro ostatní knihovny poskytuje třída **`Preprocessor`**. Jádrem rozhraní je metoda **`Preprocessor::Lex`**, která vrací následující symbol (token).

Třída **`Token`** reprezentuje jeden načtený symbol ze vstupního souboru po provedení preprocessingu. Tato třída je určena pro použití v preprocesoru, lexikální analýze a během syntaktické analýzy. Nepoužívá se později v knihovně **`AST`**. Symboly (instance třídy **`Token`**) se vyskytují ve dvou formách. Normální symbol, který je generovaný během lexikální analýzy a symbol s poznámkou (Annotation Token), který vzniká během syntaktické analýzy nahrazením normálního symbolu. Normální token obsahuje tyto informace:

- instance třídy **`SourceLocation`** – informace o místě, odkud symbol pochází,
- **délku**,
- **informace o identifikátoru** – pokud symbol pochází z identifikátoru, je přidán odkaz do tabulky identifikátorů,
- **druh symbolu** – informace, kterou přidává lexikální analyzátor (např. identifikátor, operátor přiřazení, klíčové slovo atd.),
- **příznaky** – symbol pochází ze začátku řádku, symbolu předchází bílé znaky a další.

Třída **`Lexer`** prochází vstupní soubor a získává z něj jednotlivé symboly (implementuje syntaktický analyzátor). Tento analyzátor má několik vylepšení, zde je výčet některých z nich:

- Může pracovat v *raw* módu. Tento mód mimo jiné pracuje rychleji. Používá se například pro analýzu **`#if 0`** bloku.

- Může vrátit komentář jako symbol.
- Při analýze direktivy preprocesoru se analyzátor přepne do *ParsingPreprocessorDirective* módu a místo konce řádku vrací **`EOM`** symbol.

5.1.3 Knihovna syntaktického analyzátoru

Syntaktický analyzátor Clangu je ručně vytvořený (není generovaný automaticky z gramatiky za použití některého z nástrojů pro generování syntaktického analyzátoru). Využívá rekurzivní sestup. Podle autorů je tento způsob výhodný, protože usnadňuje začínajícím vývojářům pochopení překladače, umožňuje dodatečně přidat další pravidla a „speciality“ vyžadované ze strany C/C++ a vytvoří základ pro implementaci dobrého diagnostického systému.

Rozhraní syntaktického analyzátoru implementuje třída **`Parser`**. Jednotlivé metody používané při analýze kódu jsou ve tvaru **`Parse*()`** (například **`ParseExpression()`**, **`ParseTypeofSpecifier()`** apod.). Tato struktura metod (jejich pojmenování a postup jejich volání) je odvozeno od gramatiky jazyka C [8].

Metoda **`ConsumeToken()`** slouží pro volání lexikálního analyzátoru a načtení dalšího tokenu. Některé tokeny se musí načítat k tomu určenými metodami (**`ConsumeStringToken()`**, **`ConsumeBracket()`** a další). Aktuálně načtený token je uložen ve vlastnosti třídy **`Parser`** **`tok`**.

Překladač Clang poskytuje rozhraní, jak mohou jiné programy využít již existující syntaktickou analýzu. Během provádění analýzy jsou volány metody **`ActOn*()`**, které je možné přepsat a tak získat výstup syntaktického analyzátoru. Jedná se například o metody **`ActOnCallExpr()`**, **`ActOnParenOrParenListExpr()`** a další.

5.1.4 Knihovna sémantického analyzátoru

Třída **`Type`** reprezentuje typ během syntaktické a sémantické analýzy. Pro každý typ ve zdrojovém programu je vytvořena jedna instance této třídy. Nejsou v ní uloženy informace o atributech typu (**`const`**, **`volatile`**, ...), informace o definici nového typu (**`typedef`**) se ukládá. Typ není možné změnit.

Třída **`QualType`** je určena pro ukládání atributů typu. Obsahuje odkaz na typ a bitové pole, kde jsou uloženy jednotlivé atributy. Tento způsob reprezentace zrychluje přístup k atributům typu a není potřeba ukládat více instancí třídy **`Type`** pro jeden typ s rozdílnými atributy.

Jednotlivé deklarace jsou odvozeny od třídy **`Decl`**. Třída **`NamedDecl`** dědí od **`Decl`** a popisuje deklarace, které jsou nějakým způsobem pojmenované (**`TypeDecl`**, **`ValueDecl`**, **`DeclaratorDecl`**, **`FunctionDecl`** a další.).

Samotná syntaktická analýza je implementována ve třídě **`Sema`**. Jednotlivé sémantické akce jsou volány přímo při provádění syntaktické analýzy.

6 Důvody pro rozšíření překladače

V současné době dochází k velkému rozvoji tzv. vestavěných zařízení. Snaha snižovat jejich cenu vede k vývoji lepších postupů, jak vestavěná zařízení navrhovat. Touto problematikou se zabývá výzkumná skupina Lissom na Fakultě informačních technologií Vysokého učení technického v Brně.

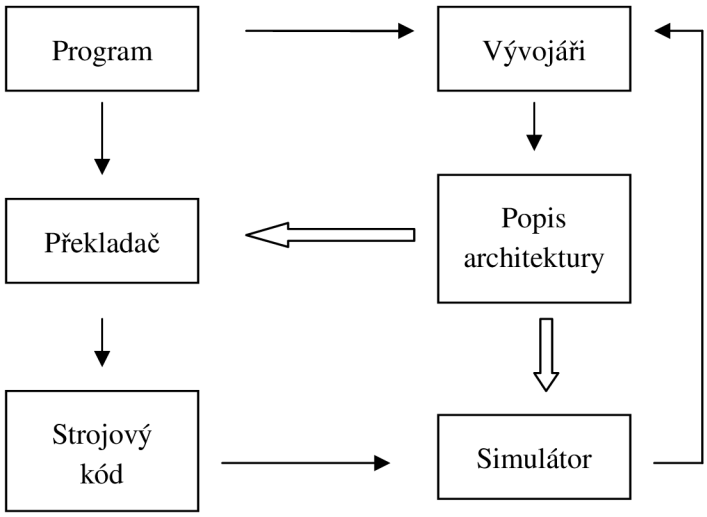
6.1 Současný návrh hardwaru a softwaru

Současný trend vývoje vestavěných zařízení se snaží co nejvíce snižovat jejich cenu, příkon, velikost apod. Tohoto cíle je možné dosáhnout pomocí vhodného rozdělení funkcionality do hardwaru a softwaru. Jedním z problémů, který vyvstává, je, jak rozhodnout, co se bude zpracovávat přímo v hardwaru a které části budou implementovány pomocí softwarových prostředků.

Procesory, které jsou navrženy, aby vykonávaly určité operace specifické jen pro daný úkol, se nazývají ASIP (Application Specific Instruction set Processor). Tyto procesory obsahují speciální instrukce, které akcelerují část výpočtu. Například procesory používané při zpracování obrazu budou obsahovat instrukce pro různé transformace obrazu apod.

Při návrhu aplikačně specifických procesorů existují dvě metodologie – tradiční a metodologie využívající jazyky pro popis architektury (ADL).

Nevýhodou *tradiční metodologie* je poměrně dlouhý čas, který potřebuje na nalezení optimálního řešení.



Obrázek 6-1 Postup návrhu architektury vestavěného zařízení

Při použití klasické metodologie je návrh programu předložen vývojářům, kteří pro něj vytvoří popis architektury, na které bude daný program provozován. Po jejím dokončení se ručně pro tuto architekturu vytvoří překladač a simulátor. Pokud vytváření popisu architektury zabralo půl roku, výstavba překladače a simulátoru bude trvat přibližně stejně dlouho.

Výstupem simulátoru je získána zpětná vazba, pomocí níž se upraví popis architektury a celý proces se opakuje znovu.

Pokud se pro popis architektury použije některého z jazyků určených pro popis architektury, proces vytváření překladače a simulátoru odpadá. Tyto dva nástroje jsou vytvořeny automaticky (tlusté šipky na obrázku Obrázek 6-1). Tímto se výrazně zrychlí návrh vestavěného zařízení.

6.2 Procesory s aplikačně specifickou instrukční sadou

ASIP (Application Specific Instruction set Processor) jsou procesory, které se využívají všude tam, kde je potřeba nějakým způsobem optimalizovat výkon. Tyto procesory obsahují instrukce, které se v klasických procesorech nenacházejí a jsou navrženy tak, aby urychlily tu část výpočtu, která se v dané úloze opakuje nejčastěji, nebo by bylo nemožné ji realizovat pomocí programu.

Tyto procesory je možné dělit na dvě skupiny. U první není možné změnit instrukční sadu (statické procesory), u druhé to možné je (konfigurovatelné).

Příkladem ASIP procesorů mohou být procesory s architekturou VLIW (velmi dlouhé instrukční slovo – very long instruction word). Tato architektura umožňuje paralelní zpracování na úrovni jedné instrukce. Procesory podporující VLIW obsahují násobné funkční jednotky, které zachycují z instrukční cache instrukci ve formátu VLIW, tvořenou několika primitivními instrukcemi, které se vykonají paralelně.

Dalším typem procesorů ze skupiny procesorů s aplikačně specifickou instrukční sadou jsou DSP (digitální signálové procesory), což jsou procesory, jejichž architektura je optimalizována pro algoritmy používané při zpracování digitálně reprezentovaných signálů. Hlavním nárokem na systém bývá průběžné zpracování velkého množství dat „protékajících“ procesorem. Analogový signál je nejprve převeden A/D převodníkem na digitální a v této podobě je průběžně zpracováván digitálním signálovým procesorem. Zpracovaný digitální signál je D/A převodníkem zpět převeden na analogový. V mnoha zařízeních prochází signál tímto řetězcem v reálném čase.

Ve všech případech procesorů s aplikačně specifickou sadou se mohou vyskytnout instrukce a registry, které nemají standardní bitovou šířku (mocnina dvou), jakou používá jazyk C. Příkladem mohou být 24 bitové registry u signálových procesorů. Pomocí standardního překladače jazyka C potom není možné psát programy pro tyto procesory bez jeho předchozích úprav.

6.3 Výzkumná skupina Lissom

Současným návrhem hardwaru a softwaru se na Vysokém učení technickém v Brně Fakultě informačních technologií zabývá výzkumná skupina Lissom. Jejím cílem je vytvořit automatizovaný nástroj pro návrh procesorů s aplikačně specifickou instrukční sadou.

Pro popis architektury je použit jazyk ISAC (popsán v kapitole 6.4). Jako kompilační platforma byla zvolena kombinace LLVM/Clang.

6.4 Jazyk ISAC

Jazyk ISAC (Instruction Set Architecture C) spadá do skupiny jazyků určených pro popis architektury. Umožňuje popis jednotlivých částí procesoru – registrů, paměti a instrukcí. ISAC vychází z jazyka LISA. Převzato z [3].

Příklad

Ukázka popisu paměti a registrů v jazyce ISAC.

```
RESOURCES {
  REGISTER bit[32] regs;
  REGISTER bit[1] carry;
  RAM bit[32] memory {
```

```
    SIZE (0x10000); FLAGS (R, W);
  };
}
```

Pro popis chování instrukcí se využívá podmnožiny jazyka C.

Příklad

```
OPERATION instr {
  INSTANCE reg ALIAS {rd, rs, rt};
  INSTANCE opc;

  ASM { opc rd "," rs "," rt };
  COD { 0b00 rs rt rd opc };

  //instr. behavior described using C
  BEHAVIOR {
    switch (opc) {
      case 0x2: regs[rd] =
        regs[rs] + regs[rt]; break;
      case 0x3: regs[rd] =
        regs[rs] - regs[rt]; break;
    }
  };
}
```

Příklad ukazuje popis instrukce. **opc** je operační kód instrukce, **rd**, **rs** a **rt** jsou registry. V části **BEHAVIOR** se nachází popis chování instrukce v jazyce C. Pro překlad tohoto popisu se používá překladač Clang.

6.5 Libovolná bitová šířka datového typu integer na platformě LLVM

Překladač jazyka C Clang se v projektu Lissom používá nejenom pro překlad popisu chování instrukcí, ale i pro následnou simulaci procesoru (kód se píše v jazyce C).

Aby bylo možné nástroje vytvořené v rámci projektu Lissom použít i pro návrh procesorů s nestandardní šířkou registrů (jiná než mocnina dvou), je nutné upravit překladač Clang tak, aby podporoval deklaraci a definici proměnných s libovolnou bitovou šířkou.

Tato práce se zabývá úpravou překladače Clang tak, aby zvládal libovolnou bitovou šířku u datového typu integer. Samotné LLVM již libovolnou bitovou šířku datového typu integer podporuje.

7 Implementace

7.1 Získání informace o bitové šířce

Podle normy jazyk C rozlišuje několik celočíselných datových typů – char, short, int, long a long long. Všechny tyto datové typy mají předem stanovenou bitovou šířku (podle cílové architektury) a není možné ji změnit.

První možností, jak upravit překladač, aby umožňoval získat informaci o bitové šířce, je upravit syntaktický analyzátor, aby podporoval konstrukci podobnou této:

```
int 24 a;
```

kde 24 by byla bitová šířka nové proměnné a. Tento zásah by znamenal úpravu syntaktického analyzátoru ve třídě **`Parser`** v metodě **`ParseDeclarationSpecifiers`** tak, aby po přijetí symbolu **`tok::kw_int`** (symbol reprezentující klíčové slovo int) byl přijat symbol **`tok::numeric_constant`**.

Tento způsob úpravy má ovšem řadu nevýhod. Protože dochází k zásahu do lexikálního analyzátoru, mění se i gramatika jazyka. Programy napsané pro takto upravený překladač nebudou přenositelné – budou muset být překládány jen na tomto překladači. Další nevýhodou je, že tato úprava není samovysvětlující. Pokud se ke kódu dostane někdo, kdo o této úpravě není dopředu informován, ztratí přehled.

Mnohem lepší variantou je využít podporu atributů, které už jsou v Clangu zapracovány. Zápis potom může vypadat takto:

```
int prom __attribute__(( bit_width(24) ));
```

Z tohoto zápisu je jasné, o jakou změnu se jedná. Pokud se program přenese do jiného překladače, dojde k vypsání varování, že atribut **`bit_width`** neexistuje, ale program půjde přeložit. Tímto směrem jsem se ubíral a takto překladač upravil.

Po úspěšném dokončení překladu vypadá výsledný LLVM kód takto:

```
@prom = common prom i24 0, align 3
```

7.1.1 Zpracování atributu a třída AttributeList

Prvním krokem při přidávání dalšího atributu je definovat jej ve třídě **`AttributeList`** (soubor AttributeList.h) ve výčtu **`Kind`**. Zde jsem přidal další položku **`AT_bit_width`**.

Třída **`AttributeList`** dále obsahuje tyto informace o atributu:

- instanci třídy **`IdentifierInfo`** obsahující samotné jméno atributu,
- instanci třídy **`SourceLocation`** s informací o místě, odkud atribut pochází (této informace je mimo jiné využito v případě chybových hlášení),
- dvojici instancí tříd **`IdentifierInfo`** a **`SourceLocation`** s informacemi o parametru, který je v atributu předáván,
- instance třídy **`ActionBase::ExprTy`**, která obsahuje ve formě výrazu informaci předávanou v rámci atributu (zde bitová šířka) a

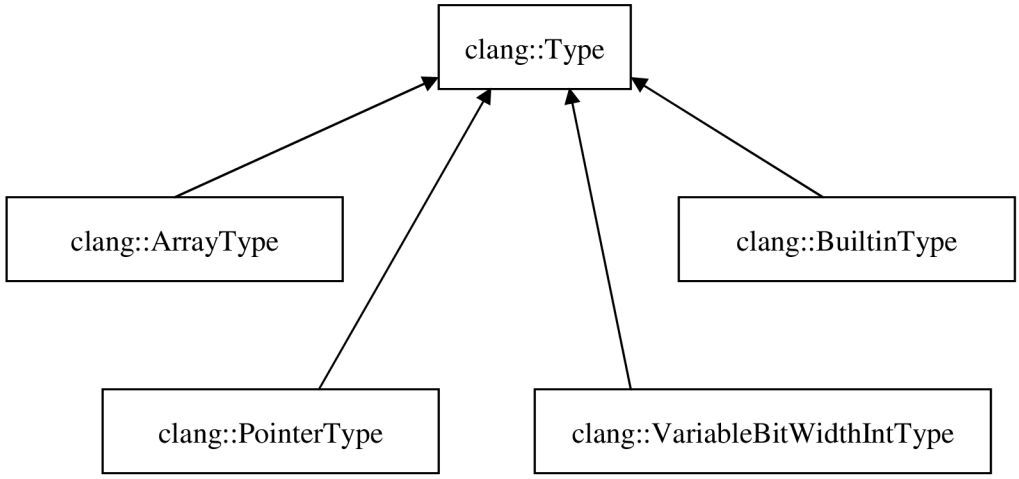
- počet argumentů obsažených v atributu (zde jeden).

Dalším krokem při zpracování atributu je samotné rozpoznání, že se jedná o daný identifikátor. V metodě **`getKind`** třídy **`AttributeList`** jsem přidal jeden řádek do příkazu **`switch`**, který zajistí, že pokud je nalezený atribut „bit_width“, vrátí se **`AttributeList::AT_bit_width.`**

Samotné zpracování atributu je implementováno ve třídě **`Sema`** v metodě **`ProcessDeclAttribute`**. Pro každou deklaraci je postupně zjišťováno, jaké atributy u ní byly použity a volá se jejich zpracování. Proto je nutné přidat jeden řádek do příkazu **`switch`**, že pokud existuje atribut **`AttributeList::AT_bit_width`**, tak se má zavolat jeho zpracování. Atribut zpracovávám v metodě **`HandleBitWidthAttr.`** Toto vlastní zpracování je popsáno v kapitole 7.3, kde popisuji kontrolu správného použití atributu a nastavení nového datového typu.

7.2 Nový datový typ VariableBitWidthIntType

Všechny datové typy v překladači Clang jsou odvozeny od třídy clang::Type.



Obrázek 7-1: Některé třídy reprezentující datové typy

Nový datový typ, který jsem do překladače přidal, jsem pojmenoval **`VariableBitWidthIntType`**.

Prvním krokem pro přidání nového datového typu do překladače je vytvořit pro něj třídu v souboru Type.h. Tato třída obsahuje dva atributy:

- bitovou šířku a
- informaci, zda se jedná o znaménkový nebo bezznaménkový typ.

Tato třída implementuje ještě několik metod, pomocí kterých je možné zjistit dodatečné informace o datovém typu.

- **`isSugared()`** – metoda, která vrátí, jestli datový typ obsahuje i „syntaktický cukr“. Jedná se například o atributy typu (**`const`**, **`volatile`**, apod.). Tento typ neuchovává informace o žádném „syntaktickém cukru“, proto tato metoda vrací vždy **`false`**.

- **desuger()** – vrátí instanci třídy **QualType()** zbavenou syntaktického cukru. Protože žádný „syntaktický cukr“ není uchováván, stačí konstruktoru třídy **QualType()** předat odkaz přímo na sebe.
- **getAsStringInternal()** – vrátí popis typu, který se používá pro výpis chybových a varovných hlášení. Metoda v proměnné typu **string** vrátí údaje popisující tento datový typ (bitovou šířku, zda se jedná o znaménkový nebo bezznaménkový datový typ a samotné označení datového typu)
- dvakrát přetížená statická metoda **classof**, která vrací, jestli instance nějaké třídy předaná jako parametr je typu **Type** a v druhém případě typu **VariableBitWidthIntType**.

Dalším krokem po vytvoření třídy nového datového typu je její zaregistrování – přidání do databáze datových typů. Každý datový typ je v databázi reprezentován svým jménem (**Builtin**, **Enum**, atd.) a třídou od které je odvozen (**Type**, **TypeTag** apod.). V závislosti na této databázi se datový typ může objevit v abstraktním syntaktickém stromě. Pro přidávání datových typů do databáze existují tato makra.

- **TYPE(Class, Base)** – typ, který se může vyskytnout kdekoliv v abstraktním syntaktickém stromě. Všichni klienti by měli tento typ chápat.
- **ABSTRACT_TYPE(Class, Base)** – tento typ se může vyskytnout v hierarchii typů, ale nelze od něj vytvořit instanci.
- **NON_CANONICAL_TYPE(Class, Base)** – může se vyskytnout kdekoliv v abstraktním syntaktickém stromě, ale nesmí být součástí kanonického typu. Konstrukce **typedef** a podobné konstrukce pro přejmenovávání typů mohou tuto skupinu datových typů ignorovat.
- **DEPENDENT_TYPE(Class, Base)** – typ, který se může vyskytnout jen uvnitř C++ šablony.

Poslední makro je nezávislé na předchozích a nemusí být využito všemi datovými typy.

- **LEAF_TYPE(Class)** – označuje typ, který neobsahuje žádné další vnitřní typy. Například **int**, **enum** apod.

Datový typ **VariableBitWidthIntType** jsem označil pomocí maker **TYPE** a **LEAF_TYPE**, protože se může vyskytovat kdekoliv v kódu a je dále nedělitelný – neobsahuje žádné další složky.

Ve třídě **clang::Type** (soubor Type.cpp) jsem upravil metody, které rozdělují datové typy do různých kategorií. Všechny tyto metody vracejí **true**, pokud dostanou na vstup datový typ **VariableBitWidthIntType**.

- **isIntegerType()** – mezi celočíselné datové typy se řadí vestavěné datové typy jako **bool**, **short**, **int** apod., datový typ vektor, pokud jeho prvky jsou celočíselné a datový typ výčet.
- **isIntegralType()** – do skupiny integrálních datových typů se řadí stejné typy jako do skupiny celočíselných datových typů s výjimkou datového typu vektor.

- **`isSignedIntegerType()`** – skupina znaménkových celočíselných typů.
- **`isUnsignedIntegerType()`** – skupina bezznaménkových celočíselných datových typů.
- **`isRealType()`** – do skupiny reálných datových typů patří všechny ze skupiny celočíselných a navíc vestavěné datové typy s plovoucí řádovou čárkou (**`float`**, **`double`**, apod.) a datový typ vektor, pokud jeho složky patří do této skupiny.
- **`isArithmeticType()`** – sem patří všechny vestavěné datové typy (celočíselné i s plovoucí řádovou čárkou), datový typ vektor a komplexní datový typ.
- **`isScalarType()`** – tato skupina rozšiřuje aritmetické typy o datový typ ukazatel a výčet.

Další krok v přidávání datového typu je úprava třídy **`ASTContext`** v souboru ASTContext.cpp.

Metoda **`getCorrespondingUnsignedType(QualType T)`** vrátí stejný datový typ jako na vstupu jen s tou úpravou, že všechny znaménkové typy nahradí bezznaménkovými. Tuto metodu jsem upravil tak, aby umožnila vrátit bezznaménkovou variantu i u datového typu **`VariableBitWidthIntType`**.

Metoda **`getTypeInfo(const Type *T)`** vrací informace o jednotlivých datových typech. A to bitovou šířku v bitech a zarovnání v bytech. Zarovnání je nejbližší vyšší nebo stejně velká mocnina dvou než je bitová šířka.

Metoda **`mergeTypes(QualType LHS, QualType RHS)`** rozhoduje, jestli je možné přiřadit proměnnou typu **`RSH`** do proměnné typu **`LSH`**. Postupuje podle tohoto algoritmu.

1. Pokud jsou oba datové typy identické, přiřazení je možné.
2. Kontrola atributů typu (const, volatile, apod.). Pokud se liší, tak přiřazení není možné, až na některé výjimky.
3. Kontrola, jestli se přiřazuje do datového typu výčet některý celočíselný datový typ. Potom je přiřazení možné.
4. V posledním kroku je pro každý **`LHS`** typ zjišťováno, jestli do něj může být **`RSH`** přiřazen.

Ve čtvrtém kroku je v případě datového typu VariableBitWidthIntType řečeno, že přiřazení nikdy není možné. Potom by ale nebylo možné do proměnné tohoto typu přiřadit proměnnou vestavěného datového typu například **`int`** nebo **`char`**. Možné to je, protože se provádí ještě jedna kontrola v jiné metodě využívají informace z metod **`isIntegerType()`**, **`isIntegralType()`** apod.

Poslední metodou, kterou jsem implementoval ve třídě **`ASTContext`**, je **`getVariableBitWidthIntType(unsigned Width, bool Signed)`**. Tato metoda vrátí instanci datového typu **`VariableBitWidthIntType`**. Protože by bylo zbytečné plýtvat pamětí a pro každý výskyt tohoto datového typu vytvářet novou instanci, jsou jednotlivé instance tohoto typu ukládány. K tomuto účelu slouží třída **`DenseMap`**, která je popsána v kapitole 7.2.1. Konkrétní instance datového typu je vrácena jako **`QualType`** bez nastavených atributů typu.

7.2.1 Třída DenseMap - uchovávání použitých datových typů

Ve zdrojovém souboru se obvykle vyskytuje více proměnných stejného datového typu. Bylo by paměťově neekonomické pro každou proměnnou vytvářet novou instanci třídy **`Type`**. Navíc by všechny tyto instance byly identické. Instance třídy **`Type`** je vytvořena při prvním výskytu

konkrétního datového typu v překládaném programu a poté je uložena do kolekce **`DenseMap`**. Při dalším výskytu stejného datového typu se použije již existující instance třídy **`Type`**.

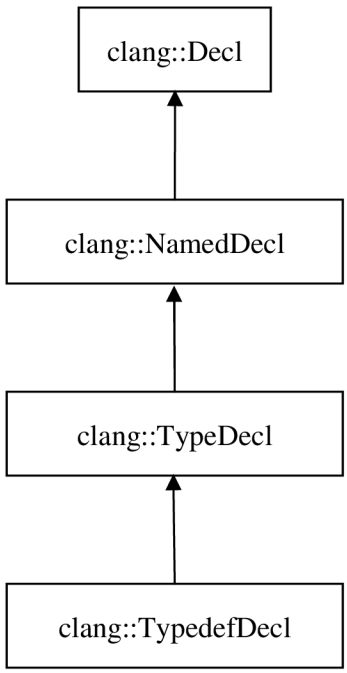
Přidal jsem dvě kolekce **`DenseMap`**. První pro znaménkový (**`SignedFixedWidthIntTypes`**) a druhou pro bezznaménkový (**`UnsignedFixedWidthIntTypes`**) **`VariableBitWidthIntType`**. Tyto kolekce jsou definovány ve třídě **`ASTContext`** v souboru ASTContext.cpp.

Tyto dvě kolekce využívá metoda **`ASTContext::getVariableBitWidthIntType`**, která vrací instanci datového typu VariableBitWidthIntType. Tato metoda určí, jestli již byla vytvořena instance typu nebo ne a podle toho vytvoří novou a uloží ji do kolekce nebo použije již vytvořenou instanci, která se v kolekci nachází.

7.3 Kontrola správného použití atributu a nastavení datového typu

7.3.1 Deklarace - třída Decl

Informace o jednotlivých deklaracích je uchovávána ve třídě **`clang::Decl`**. V případě deklarace typu se jedná o potomka clang::TypeDecl.



Obrázek 7-2: Třídy používané pro popis deklarace

`clang::TypedefDecl` se použije v případě pojmenování datového typu použitím konstrukce **`typedef`**.

Třída **`Decl`** nebo její potomci obsahují mimo jiné tyto údaje o deklaraci:

- datový typ,

- popis místa, kde se deklarace nachází ve zdrojovém souboru,
- atributy, které byly k deklaraci přidány (pro pozdější zpracování),
- informace, zda je deklarace implicitní (deklarace je implicitní, pokud byla vytvořena překladačem - tedy nebyla obsažena ve zdrojovém souboru),
- informace o syntaktické správnosti deklarace a
- zda je deklarace použita (jestli je zapotřebí).

7.3.2 Zpracování atributu

Vlastní zpracování atributu provádí metoda **`HandleBitWidthAttr`** (soubor SemaDeclAttr.cpp). Na vstup jsou předány tyto parametry:

- **`Decl *D`** – deklarace, které se atribut týká,
- **`AttributeList &Attr`** – informace o atributu a
- **`Sema &S`** – odkaz na instanci třídy, která provádí syntaktickou analýzu a konstruuje abstraktní syntaktický strom.

Prvním krokem při zpracování atributu je jeho samotná kontrola. To znamená, jestli obsahuje právě jeden parametr (bitovou šířku) a jestli tento parametr je správně zapsaný. Musí to být celé číslo z rozsahu od 1 do 2^{23}-1. Toto jsou bitové šířky, které podporuje LLVM platforma u datového typu integer.

Parametr atributu je možné v programu zadat jako výraz. V parametru **`&Attr`** není tento atribut spočítaný, ale předaný ve formě výrazu (používá se vnitřní struktura překladače pro výrazy). Před jeho výpočtem se zkontroluje, jestli je datového typu integer. Metody pro kontrolu datového typu výrazu a jeho vlastní výpočet jsou již implementovány.

V dalším kroku se zkontroluje, jestli je atribut použit u správné deklarace. Jeho použití má smysl při deklaraci nové proměnné datového typu **`int`** nebo pojmenování nového datového typu pomocí **`typedef`** (i v tomto případě musí být typ, od kterého se odvozuje, **`int`**).

V posledním kroku se nastaví deklaraci nový datový typ. Původní typ **`BuiltIn`** je nahrazen datovým type **`VariableBitWidthIntType`** s údaji o bitové šířce a zda se jedná o znaménkový nebo bezznaménkový typ.

7.4 Generování LLVM kódu

Během generování kódu mezijazyka je nutné předat generátoru instrukcí informaci o bitové šířce a o tom, jestli se jedná o znaménkový nebo bezznaménkový typ.

Metoda **`ConvertNewType`** třídy **`CodeGenTypes`** (soubor CodeGenTypes.cpp) převádí interní reprezentaci typů používaných překladačem na typy mezijazyka LLVM. Pro typ **`VariableBitWidthIntType`** je vrácen **`llvm::IntegerType`** se stejnou bitovou šířkou.

Datové typy v LLVM nepoužívají rozlišení na znaménkové a bezznaménkové. Tato vlastnost je zohledněna použitím různých instrukcí. Informace o tom, jestli je typ znaménkový nebo bezznaménkový, předávají generátoru instrukcí metody **`isSignedIntegerType()`** a **`isUnsignedIntegerType()`** třídy **`Type`**.

7.5 Rozsah celočíselných konstant

Tímto zásahem do překladače jsem umožnil deklarovat celočíselné proměnné téměř libovolného rozsahu. Nedošlo ale ke změně v případě celočíselných konstant. Ty mohou být zadávány jen v datových typech, které určuje standard jazyka C. Nelze tedy vložit do zdrojového souboru číslo větší než dovoluje rozsah datového typu **`long long`**.

7.6 Testování

Po dokončení implementace jsem přešel k jejímu testování. Správnou funkci překladače jsem testoval dvěma způsoby.

Prvním bylo vytvoření několika zdrojových souborů, které obsahovaly konstrukce jazyka, které překladač musel zvládnout přeložit bez chyby a při kterých mělo dojít k vypsání chybového hlášení.

Příkladem povolených konstrukcí může být:

```
typedef int __attribute__(( bit_width(1300000) )) myInt1;
typedef int __attribute__(( bit_width(11) )) myInt2;
int foo(myInt2 p) __attribute__(( bit_width(14) ));
```

Naopak konstrukce, u kterých by měl překladač zahlásit chybu:

```
float __attribute__(( bit_width(13) )) myFloat1;
int __attribute__(( bit_width(13) )) myArray[4];
```

V dalším kroku testování jsem použil již vytvořené skripty v Jazyce Python, abych si ověřil, že jsem do překladače nezanesl nějakou chybu.

7.7 Možnost rozšíření o další datové typy

Po přidání podpory libovolně širokého datového typu integer se nabízí otázka, jestli by bylo možné rozšířit překladač i o jiné datové typy.

Platforma LLVM podporuje datové typy s plovoucí řádovou čárkou jen s pevně nastavenou bitovou šířkou. Nelze použít libovolnou jako v případě celočíselných datových typů. Proto by tato úprava znamenala mnohem větší zásah, protože by si vynutila změny i v samotné platformě LLVM.

U datových typů s pevnou řádovou čárkou je situace ještě horší. Ty nejsou ze strany LLVM podporovány vůbec.

8 Závěr

Cílem této práce bylo upravit kompilační platformu LLVM/Clang tak, aby podporovala libovolně široký datový typ integer. Samotný mezijazyk a zadní část překladače (LLVM) již libovolně široký integer podporovaly. Zaměřil jsem se tedy na úpravu přední části překladače (Clang).

Tento úkol se mi podařilo splnit rozšířením jazyka o atribut, pomocí kterého je možné bitovou šířku určit. Dále jsem definoval nový datový typ a implementoval jeho vnitřní reprezentaci v překladači. Po přidání nového datového typu jsem se zaměřil na sémantické kontroly, aby bylo možné konvertovat hodnoty mezi stávajícími a nově přidaným datovým typem. V posledním implementačním kroku jsem upravil generátor instrukcí mezijazyka LLVM tak, aby podporoval nově přidaný datový typ. Nakonec jsem se pomocí série testů ujistil, že implementovaná změna funguje tak, jak byla navržena, a že nevnáší do překladače nějakou novou chybu.

Význam práce

Tvorba projektu pro mne měla značný význam, protože jsem mohl v praxi využít vědomosti nabývané v oblastech formálních jazyků a překladačů. Jako největší přínos považuji svůj posun ve znalostech v oblasti konstrukce překladačů.

Dalším přínosem práce je odstranění další z překážek projektu Lissom. Projekt Lissom se na Fakultě informačních technologií Vysokého učení technického v Brně zabývá vývojem nástrojů pro návrh procesorů s aplikačně specifickou instrukční sadou. Jedná se o automaticky generovaný assembler, linker, simulátor, disassembler a debuger z popisu architektury procesoru. Jako kompilační platformu používají právě LLVM/Clang. Nedostupná podpora libovolně širokého integeru bránila vytvářet tyto nástroje pro procesory, které používají nestandardně široké registry (například signálové procesory).

Budoucnost

Dalším pokračováním práce může být přidání podpory libovolně širokých datových typům s plovoucí řádovou čárkou a tzv. fixpoint proměnných (s pevnou řádovou čárkou). Poslední zmiňované se ve značné míře používají u signálových procesorů z důvodu optimalizace rychlosti, ceny a velikosti čipu.

9 Literatura

[1] AHO, A., et al. *Compilers: Principles, Pechniques, & Tools / 2nd ed.* Boston : Pearson ; Addison Wesley, 2007. xxiv, 1009 s. ISBN 0-321-49169-6

[2] Harbison, Samuel P. *C reference manual / 5th ed.* New Jersey : Prentice-Hall, 2002. xviii, 533 s. ISBN 0-13-089592-X

[3] Husár, A. Extraction of Processor Architecture Model for Compiler Generation from ISAC Language. Technická zpráva, VUT FIT, 2010.

[4] Meduna, A. *Formální jazyky a překladače*. Brno, 2006. 152 s. Studijní opora. VUT FIT

[5] Clang: a C language family frontend for LLVM [online]. 2010 [cit. 2010-05-12]. Clang: a C language family frontend for LLVM. Dostupné z WWW: <http://clang.llvm.org/>

[6] Clang: a C language family frontend for LLVM [online]. 2010 [cit. 2010-05-12]. "Clang" CFE Internals Manual. Dostupné z WWW: <http://clang.llvm.org/docs/InternalsManual.html>

[7] FreeBSD Wiki [online]. 2010-04-24 [cit. 2010-05-12]. Building FreeBSD with clang/llvm. Dostupné z WWW: <http://wiki.freebsd.org/BuildingFreeBSDWithClang>

[8] Lysator [online]. 1995 [cit. 2010-05-12]. ANSI C Yacc grammar. Dostupné z WWW: <http://www.lysator.liu.se/c/ANSI-C-grammar-y.html>

[9] The LLVM Compiler Infrastructure Project [online]. 2010 [cit. 2010-05-12]. The LLVM Compiler Infrastructure. Dostupné z WWW: <http://www.llvm.org/>

[10] The LLVM Compiler Infrastructure Project [online]. 2010 [cit. 2010-05-12]. LLVM Language Reference Manual . Dostupné z WWW: <http://www.llvm.org/docs/LangRef.html>

Seznam příloh

Příloha 1. DVD se zdrojovými kódy, návody, ukázkami