VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

PŘEKLAD C++ APLIKACÍ PRO VESTAVĚNÁ ZAŘÍZENÍ

COMPILATION OF C++ APPLICATIONS FOR EMBEDDED DEVICES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. MILAN NOSTERSKÝ**
AUTHOR

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

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Nosterský Milan, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Překlad C++ aplikací pro vestavěná zařízení**
Compilation of C++ Applications for Embedded Devices
Kategorie: Překladače
Zadání:

1. Seznamte se s nástrojem Codasip Studio, překladačem Clang/LLVM a jazykem C++11 a také s existujícími standardními knihovnami C++ a způsoby implementace výjimek a volání virtuálních metod.
2. Zvolte vhodnou standarní knihovnu C++ a navrhněte způsob jejího přizpůsobení pro použití na vestavěných zařízeních s podporou operačního systému a bez ní.
3. Navrhněte způsob implementace výjimek a volání virtuálních metod na vestavěných zařízeních.
4. Implementujte navržené řešení nad překladačem Clang/LLVM a integrujte jej do prostředí Codasip Studio.
5. Otestujte implementované řešení na sadě testovacích modelů a aplikací.
6. Zhodnoťte dosažené výsledky a navrhněte možná budoucí vylepšení.

Literatura:

- STROUSTRUP, Bjarne. The C++ programming language. Addison-Wesley, 2000, 1019 s. ISBN 978-0-201-70073-2.
- MEYERS, Scott. Effective Modern C++. 1. O'Reilly Media, 2014. ISBN 9781491903995.
- STROUSTRUP, Bjarne. Standard C++. Chichester: J. Wiley, 2003, 782 s. ISBN 0-470-84674-7.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Hruška Tomáš, prof. Ing., CSc.**
Konzultant: Šnobl Pavel, Ing., CODASIP
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 31. října 2018

Abstrakt

Tato diplomová práce se zabývá přidáním podpory překladu jazyka C++ a jeho standardu C++11 v rámci překladače pro vestavěné systémy. Překladač, založený na projektu LLVM se generuje v rámci prostředí Codasip Studia. Toto prostředí slouží pro návrh procesorů s aplikačně specifickou instrukční sadou, kdy umožňuje na základě popisu sémantiky instrukční sady generovat překladač pro libovolnou cílovou architekturu. Jazyk C++ je jazyk vycházející z jazyka C, rozšířený o objektovou orientaci a několik nových funkcionalit. Jazyk C++ umožňuje psát velmi efektivní kód na vysoké úrovni abstrakce. V rámci testovací fáze je implementace podpory jazyka C++ ověřena na modelech procesorových jader s využitím testovací sady.

Abstract

This master's thesis deals with the integrations of C++ programming language and its standard C++11 into the compiler for embedded systems. This compiler is based on LLVM project and it is generated from Codasip Studio. Codasip Studio is tool for design of the aplication specific processor cores, it is also allows generate compiler, which is based on the description of semantics section in processor's instruction set for any target processor architecture. C++ is programming language based on the C, which is extended by object oriented design and many other features. C++ language allows writing of very effective code on high level of abstraction. Funcionality of implementation is tested on testsuite in last phase of master's thesis.

Klíčová slova

C++, LLVM, Výjimky, Codasip, CodAL

Keywords

C++, LLVM, Exceptions, Codasip, CodAL

Citace

NOSTERSKÝ, Milan. *Překlad C++ aplikací pro vestavěná zařízení.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Tomáš Hruška, CSc.

Překlad C++ aplikací pro vestavěná zařízení

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana prof. Ing. Tomáše Hrušky, CSc. Další informace mi poskytl Ing. Pavel Šnobl. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal informace

. . . .
Milan Nosterský
21. května 2019

Poděkování

Chtěl bych poděkovat vedoucímu své práce panu prof. Ing. Tomáši Hruškovi, CSc., zaměstnancům firmy Codasip s.r.o a především Ing. Pavlu Šnoblovi za odborné rady a věnovaný čas.

Obsah

1 Úvod **3**

2 Jazyk C++ **5**
2.1 Představení jazyka C++ 5
2.2 Volání virtuálních metod 9
2.3 Informace o dynamických typech 11
2.4 Výjimky v jazyce C++ 11
2.4.1 Princip zpracování výjimek 12
2.4.2 Způsoby řešení výjimek 13
2.5 Standard C++11 14
2.6 Existující knihovny C++ 15

3 LLVM framework **16**
3.1 Průběh překladu 16
3.2 LLVM Intermediate Representation 20
3.3 DAG - Directed Acyclic Graph 21

4 Codasip Studio **23**
4.1 Vývojové prostředí 23
4.2 Jazyk pro popis architektury CodAL 24

5 Zvolené řešení **27**
5.1 Knihovna jazyka C++ 27
5.2 Řešení výjimek 28

6 Volání virtuálních metod **29**
6.1 LLVM - řešení volání virtuálních metod 29

7 Implementace na architektuře Codasip uRISC **35**
7.1 Seznámení s architekturou Codasip uRISC 35
7.1.1 Obecná úprava modelu 36
7.2 Implementace builtin funkcí 36
7.2.1 ___builtin_setjmp 37
7.2.2 ___builtin_longjmp 40
7.3 Překlad knihoven 41
7.4 Úprava zdrojových kódů nástrojů Codasip 43
7.4.1 Clang - přední část překladače 43
7.4.2 LLC - zadní část překladače 43

7.4.3 Generátor překladače . . . 44
7.4.4 Simulátor procesorového jádra . . . 44
7.4.5 Standardní knihovna jazyka C . . . 44
7.5 Zachytávání výjimek pomocí Set Jump a Long Jump . . . 45
7.5.1 Implementace llvm.eh.sjlj.lsda() . . . 46
7.5.2 Implementace llvm.eh.sjlj.setup.dispatch() . . . 49
7.6 Zachytávání výjimek pomocí DWARF . . . 52
7.6.1 Vytvoření rámce *eh_frame* . . . 53
7.6.2 Úprava custom souborů modelu . . . 59
7.6.3 Úprava Clang . . . 59
7.6.4 Úprava knihoven . . . 61

8 Integrace do nástrojů Codasip Studio **62**
8.1 Integrace custom souborů . . . 62
8.1.1 CodasipGenISelLowering . . . 62
8.1.2 CodasipAsmPrinter . . . 64
8.1.3 CodasipGenInstrInfo . . . 64
8.1.4 CodasipJumpLengthCheck . . . 65
8.1.5 CodasipCFIAdder . . . 65
8.2 Integrace knihoven . . . 65

9 Testování **67**
9.1 Testovací sada . . . 67
9.1.1 Testovací prostředí . . . 67
9.2 Výsledky testování . . . 69

10 Závěr **71**

Literatura **73**

A Set Jump Long Jump zachytávání výjimek **75**

B DWARF zachytávání výjimek **78**

C Obsah CD **80**

D Návod **81**

Kapitola 1

Úvod

V dnešní době, spojené s pojmem Internet věcí (IoT - Internet of Things), se nachází počítačové systémy kdekoliv kolem nás a jsou součástí našeho každodenního života. Spolu s internetem věcí se také rozrůstá pole působnosti aplikačně specifických procesorů (ASIP), které jsou přítomny téměř v každém zařízení. S růstem tohoto segmentu jsou na procesory kladeny stále větší nároky z hlediska výkonu a velikosti.

Výkon procesoru nezáleží pouze na implementaci samotného jádra procesoru, ale také na aplikaci, kterou vykonává. Kód této aplikace poté ovlivňuje jednak programátor, autor dané aplikace, jazyk jež byl použit při vývoji aplikace a v neposlední řadě překladač jazyka, který přeloží aplikaci do podoby spustitelného kódu pro dané jádro.

Vývojem ASIP se zabývá firma Codasip s.r.o., která k tomu využívá vývojového prostředí Codasip Studio. Codasip Studio umožňuje modelování ASIP, včetně plně automatizovaného generování vývojových a ladících nástrojů. Součástí těchto nástrojů je i překladač jazyka C/C++.

V rámci těchto překladačů je zaručena plná podpora jazyka C, ale překladač jazyka C++ nemá implementovanou podporu některých základních funkcionalit jazyka C++, jako jsou například výjimky nebo volání virtuálních metod, rovněž neumožňuje využívat standardní knihovnu jazyka C++. Tento fakt může být značně omezující pro cílové uživatele Codasip Studia, neboť jazyk C++ je momentálně třetím nejoblíbenějším programovacím jazykem [14].

V rámci této práce bude představen jazyk C++ v kapitole 2. Tato kapitola popisuje základní rysy a funkcionality jazyka C++ včetně volání výjimek, jejich dostupných implementací a také představuje standard C++11.

Kapitola 3 seznamuje čtenáře s kompilačním frameworkem[1] LLVM. V této kapitole jsou představeny jednotlivé fáze překladu, včetně vnitřní reprezentace kódu v LLVM.

Kapitola 4 představuje nástroj Codasip Studio, sloužící pro návrh procesorů s aplikačně specifickou instrukční sadou, jak bylo zmíněno výše. Kromě představení samotného Codasip Studia se čtenář seznámí se zápisem sémantiky instrukcí v jazyce CodAL, které slouží ke generování překladače.

V rámci kapitoly 6 dojde k představení řešení volání virtuálních metod u kompilačního frameworku LLVM.

V kapitole 7 se nachází realizace implementace na specifickém procesorovém jádře, kterým je Codasip uRISC. V rámci kapitoly se čtenář kromě implementace seznámí i s tímto jednoduchým výukovým jádrem. Implementace se zaměřuje především na zachytávání výji-

[1] Aplikační rámec - softwarová struktura, sloužící jako podpora při vývoji softwaru.

mek a na nezbytné kroky v rámci nástrojů Codasip k přeložení a otestování knihoven jazyka C++.

Následující kapitola 8 se zabývá integrací zdrojových kódů, vytvořených v předchozí kapitole do nástrojů Codasip, kromě vytvořených kódů se zde nachází popis integrace knihoven.

Kapitola 9 obsahuje popis vytvořeného testovacího prostředí pro knihovny jazyka C++, integrací testovací sady a testováním na různých procesorových jádrech, včetně prezentace výsledků testování.

Poslední kapitolou je závěr, kde jsou shrnuty dosažené výsledky, možnosti využití výsledků práce a další možná rozšíření implementace.

Kapitola 2

Jazyk C++

V následující kapitole bude představen programovací jazyk C++. Nejdříve bude představena historie jazyka, následovaná výhodami a odlišnostmi od jazyka C. Na závěr budou uvedeny specifika standardu C++11.

2.1 Představení jazyka C++

První verze jazyka C++ pod názvem C with Classes byla sepsána roku 1979. Autorem je Bjarne Stroustrup a tato verze vznikla při jeho práci v Bellových laboratořích. Následně byl změněn název jazyka na C++, to se stalo roku 1983. V roce 1985 byla vydána první komerční implementace jazyka C++ [17]. První standardizace jazyka proběhla v roce 1998 a je známá jako C++98, dalšími standardy jsou C++03, C++11, C++14 a C++17 [18]. Každý ze standardů jazyka C++ přináší opravy chyb, novou funkcionalitu nebo zlepšení výkonu současných funkcionalit. Do budoucna je již naplánován standard s označením C++20.

Jazyk C++ rozšiřuje funkcionalitu jazyka C o spoustu vlastností. Zde jsou nejvýznamnější z nich:

- třídy a objekty,
- dědičnost,
- polymorfismus, virtuální metody a informace o typu za běhu (RTTI - Runtime type information),
- přetěžování funkcí,
- referenční proměnné,
- šablony,
- zachytávání výjimek,
- jmenný prostor (*namespace*) pro prevenci konfliktů jmen u funkcí, proměnných nebo tříd.

Jak je z výčtu patrné, jazyk C++ patří mezi jazyky objektově orientované, oproti tomu jazyk C patří mezi jazyky procedurální.

Rozdíl mezi procedurálním jazykem a jazykem objektově orientovaným lze v jednoduchosti popsat následovně: procedurální jazyky se soustředí na procedury, realizující algoritmy pro práci s daty, které získají na svém vstupu. Na rozdíl od procedurálního přístupu

se objektově orientovaný přístup zaměřuje na data, snaží se problém přizpůsobit jazyku problémů, které se řeší. Třída specifikuje nějakou část reálného světa, která má svoje atributy (data) a metody operující nad těmito daty. Konkrétní instancí třídy je poté objekt. Například třída je automobil - obecný prvek reálného světa a konkrétní objekt je automobil s atributy značka - Škoda, model - Fabia apod. Příklad tvorby samotných tříd, jejich instanciace včetně dědičnosti, která bude představena dále, lze vidět na ukázce kódu 2.1.

```
#include <iostream>
class automobil { // Vytvoření třídy
public:
  int ujeteKilometry;
  void jizda(int km){ ujeteKilometry += km; }; // Metoda třídy
  automobil(){ ujeteKilometry = 10; } // Konstruktor třídy
  ~automobil(){}; // Destruktor třídy
}

class fabia: public automobil { // Třída fabia dědí od třídy automobil
public:
  void ukazKilometry(){
    std::cout << ujeteKilometry << std::endl;
  }
  fabia(){ ujeteKilometry = 0; };
  ~fabia(){};
}

int main(){
  fabia mojeAuto; // Vytvoření instance automobil
  mojeAuto.jizda(50); // Volání metody, kterou implementuje bázová třída
  mojeAuto.ukazKilometry();
}
```

Ukázka kódu 2.1: Ukázka práce s třídami.

Objekty mezi sebou komunikují s využitím svého rozhraní, což je schéma deklarující sadu metod, které třída realizuje, s pomocí zasílaní zpráv. S objektově orientovaným přístupem se také pojí pojem zapouzdření, který definuje uzavření objektu vůči objektům okolním. Třída může mít metody veřejné, které definují její rozhraní, metody privátní, k nimž mají přístup pouze objekty dané třídy a metody chráněné, ty mají stejné vlastnosti jako metody privátní a navíc k nim nemají přístup objekty z děděných tříd.

Kromě tříd a objektů C++ přináší, jak již bylo zmíněno výše, dědičnost. Dědičnost vyjadřuje vztah předek-potomek několika tříd, kdy potomek rozšiřuje vlastnosti předka s tím, že obsahuje veškeré vlastnosti svého předka. Dědičnost může být i víceúrovňová, tedy třída může být odvozena z potomka jiné třídy.

Další vlastností na předchozím seznamu je polymorfismus, který umožňuje používat jednotné rozhraní pro různé typy objektů. Volání virtuálních metod a získávání informací o typu za běhu bude představeno v následujícím textu podrobněji.

Přetěžování umožňuje přehlednější strukturu programu, kdy můžeme mít více funkcí jednoho názvu s rozdílnými parametry. Ukázka jednoduchého přetěžování metod je znázorněna v kódu 2.2, kde metoda *vytiskni* má tři různé implementace, které se liší v typu vstupních parametrů nebo jejich počtu.

```
#include <iostream>
class tisk {
public:
  void vytiskni(int a) { // Metoda s jedním parametrem typu int
    std::cout << "Parametry:" << a << ":" << b;
  };
  void vytiskni(double a) { // Metoda s~jedním parametrem typu double
    std::cout << "Parametry:" << a << ":" << b;
  };
  void vytiskni(int a, int b) { // Metoda se dvěma parametry typu int
    std::cout << "Parametry:" << a << ":" << b;
  };
}
```

Ukázka kódu 2.2: Přetěžování metod.

Kromě metod lze přetěžovat, případně i využívat operátory jako metody pro vlastní třídy. Přetěžování operátorů lze vidět v ukázce kódu 2.3, kde třída *obdelnik* používá operátor $++$ k inkrementaci obou jejich privátních proměnných.

```
#include <iostream>
class obdelnik {
  int stranaA, stranaB;
public:
  obdelnik(int a, int b) {
    stranaA = a;
    stranaB = b;
  };
  void operator ++() { // Přetížení operátoru ++
    stranaA++;
    stranaB++;
  };
  void out() {
    std::cout << stranaA << ":" << a << std::endl;
  };
};

int main(int argc, char const *argv[])
{
  obdelnik a(7,8);
  ++a;
  out(a); // Očekávaný výstup 8:9
  return 0;
}
```

Ukázka kódu 2.3: Přetěžování operátoru.

Spolu s přetěžováním operátorů se pojí i klíčové slovo jazyka C++ *friend*, jehož použití umožňuje obejít princip zapouzdření. Princip použití klíčového slova *friend* ukazuje kód

2.4, kde funkce *out* není součástí žádné třídy, ale má přístup k privátním proměnným třídy *obdelnik*.

```
#include <iostream>
class obdelnik {
  int stranaA, stranaB;
public:
  obdelnik(int a, int b) {
    stranaA = a;
    stranaB = b;
  };
  friend void out(obdelnik a);
};

  // Funkce out není součástí třídy, ale má přístup k privátním proměnným
  void out(obdelnik a) {
    std::cout << a.stranaA << ":" << a.stranaB << std::endl;
  };

int main(int argc, char const *argv[])
{
  obdelnik a(7,8);
  out(a); // Očekávaný výstup 7:8
  return 0;
}
```

Ukázka kódu 2.4: Využití klíčového slova *friend*.

Referenční proměnné se odkazují na proměnnou, nebo instanci objektu. Jedná se o alias proměnné, případně instance objektu, kdy se v operační paměti nachází proměnná, respektive instance objektu pouze jednou, ale lze k ní přistupovat pod jiným jménem. Na rozdíl od ukazatele se jedná o abstraktnější datový typ, který neobsahuje informace o uložení objektu v operační paměti.

Šablony slouží k větší variabilitě kódu, kdy mohou být využity u funkcí, tříd i struktur. Šablona umožňuje definovat více funkcí, tříd nebo struktur pouze jednotným zápisem, který po přiřazení datového typu bude realizovat danou funkcionalitu.

Zachytávání výjimek bude podrobněji rozebráno v následujícím textu. Poslední funkcionalitou z předchozího výčtu je jmenný prostor, který umožňuje definici prostoru, v jehož rámci bude každé jméno unikátní, ale nebude definováno v rámci prostoru jiného, kde může mít dané jméno definováno naprosto odlišnou funkcionalitu nebo hodnotu od prvotní definice. Takové chování lze vidět na ukázce kódu 2.5, kde se operuje se třemi jmennými prostory *a*, *b* a *std*. V rámci každého jmenného prostoru se nachází implementace funkce *abs* se vstupním parametrem typu *int*, ale každá implementace má jinou funkcionalitu. Výsledek funkce *abs* poté závisí na tom, v rámci kterého jmenného prostoru je volána. V C++ se také nachází standardní jmenný prostor pod názvem *std*. Do tohoto prostoru patří běžná funkcionalita jazyka C++, např. *cout* - standardní výstup, *cin* - standardní vstup apod.

```
#include <iostream>
namespace a {
  void abs(int a) {
    std::cout << a << std::endl;
  }
};

namespace b {
  void abs(int a) {
    std::cout << a + 20 << std::endl;
  }
};

int main(int argc, char const *argv[])
{
  int hodnota = -5;
  std::cout << std::abs(hodnota) << std::endl; // Výpis 5
  a::abs(hodnota); // Výpis -5
  b::abs(hodnota); // Výpis 15
  return 0;
}
```

Ukázka kódu 2.5: Využití jmenného prostoru.

2.2 Volání virtuálních metod

Virtuální metody umožňují polymorfismus, kdy virtuální metoda označená klíčovým slovem *virtual* definuje jednotnou metodu rozhraní bázové třídy. Samotná implementace se poté nachází jak v bázové třídě, tak i v třídě odvozené. Pokud se implementace virtuální metody nenachází v bázové třídě, jedná se o čistě virtuální metodu. S virtuálními metodami se pojí klíčová slova:

- *override*, které bylo do jazyka C++ přidáno v revizi C++11 zajišťuje, že se opravdu přepisuje virtuální metoda a nedochází pouze k přetěžování metod, například použitím jiných operandů, v takovém případě dojde k chybě překladu,
- *final* použitím se znemožní předefinování virtuální metody v dále odvozených třídách.

Zápis virtuálních metod lze vidět na ukázce kódu 2.6. Na této ukázce se nachází dvě třídy. První třída jménem *bazova* obsahuje čistě virtuální metodu *tisk*, virtuální metodu *ukaz* a vlastní interní metodu *interni*. Druhá třída jménem *odvozena* poté obsahuje vlastní implementaci virtuálních metod *tisk*, *ukaz* a vlastní interní metodu *interni*. V rámci hlavního těla programu se nachází ukazatel *uk* na třídu *bazova*, ale neukazuje na instanci třídy *bazova*, nýbrž na instanci třídy *odvozena*. Tedy při volání virtuálních metod budou volány implementace v rámci třídy *odvozena* a v rámci volání nevirtuálních metod se použijí metody třídy samotného ukazatele.

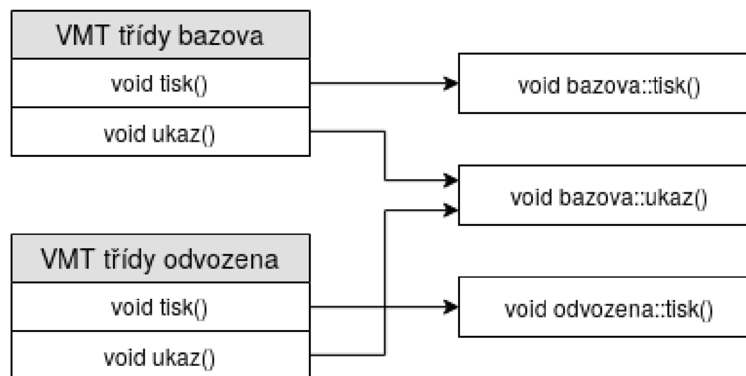
```cpp
#include <iostream>
class bazova {
public:
  virtual void tisk() = 0; // Čistě virtuální metoda
  virtual void ukaz() { // Virtuální metoda
    std::cout << "Bazova trida" << std::endl;
  };
  void interni(){
    std::cout << "Bazova trida" << std::endl;
  }
};

class odvozena: public bazova {
public:
  virtual void tisk() override{ // Implementace rozhraní v odvozené třídě
    std::cout << "Tisk odvozena trida" << std::endl;
  };
  virtual void ukaz() override{ // Implementace rozhraní v odvozené třídě
    std::cout << "Odvozena trida" << std::endl;
  };
  void interni(){
    std::cout << "Bazova trida" << std::endl;
  }
};

int main(int argc, char const *argv[])
{
  bazova *uk = new odvozena();
  uk->ukaz(); // "Odvozena trida"
  uk->tisk(); // "Odvozena trida"
  uk->interni(); // "Bazova trida"
  return 0;
}
```

Ukázka kódu 2.6: Ukázka zápisu virtuálních metod.

Volání virtuálních metod se realizuje pomocí tabulek virtuálních metod (Virtual Method Table - VMT), které obsahují odkazy na všechny virtuální metody třídy. Tuto tabulku obsahuje každá třída s virtuálními metodami. V případě odvozování třídy (dědění) převezme třída odvozená tabulku virtuálních metod své bázové třídy. Odkazy na předefinované virtuální metody se aktualizují a na konec tabulky jsou přidány odkazy na nové virtuální metody, realizované odvozenou třídou. Každý objekt dané třídy poté obsahuje odkaz na tabulku virtuálních metod. Při volání některé z virtuálních metod se vybere odkaz z tabulky virtuálních metod a provede se samotné volání [12]. Na obrázku 2.1 lze vidět ukázku tabulky virtuálních metod pro třídy z ukázky kódu 2.6.



Obrázek 2.1: Ukázka tabulky virtuálních metod

2.3 Informace o dynamických typech

Informace o dynamických typech, neboli zkráceně RTTI z anglického překladu *RunTime Type Information*, je jednou ze základních funkcionalit jazyka C++. Jazyk C++ obsahuje tři komponenty, které umožňují RTTI:

1. Klíčové slovo *dynamic_cast* říká, zda adresa objektu lze bezpečně přiřadit ukazateli, nebo referenci určitého typu. Pokud lze adresu přiřadit, pak provede přiřazení. V opačném případě, kdy tato operace není možná, se vrací buď nulový ukazatel, pokud cílovým typem je ukazatel, nebo se vyvolá výjimka v případě, že cílový typ je reference.

2. Funkce *typeid* vrací přesný typ identifikující objekt.

3. Struktura *type_info* udržující informace o daném typu.

Běžné využití RTTI se pojí na volání virtuálních metod, kdy *dynamic_cast* slouží k přiřazení ukazatele děděné třídy na ukazatel třídy rodičovské a ujištění, že je v pořádku volat virtuální metodu. Využití *typeid* vracející *type_info* spočívá například v porovnání dvou objektů, kdy *type_info* obsahuje informace o těchto objektech. Operátor *dynamic_cast*, stejně jako jeho varianty *static_cast*, *const_cast* a *reinterpret_cast* lze využít k bezpečnějšímu převodu mezi datovými typy [13].

2.4 Výjimky v jazyce C++

Výjimky slouží k oznámení neočekávané události za běhu programu. Program, při jehož vykonávání tato událost vznikne, nemůže dále pokračovat, dokud není výjimka obsloužena specifickou rutinou k tomu určenou. Zachycením výjimky může programátor definovat chování programu při této události. Příklad jednoduchého vyvolání a zachycení výjimky lze vidět v ukázce kódu 2.7. S voláním výjimek se pojí také mnohé funkce a metody objektů standardní knihovny, například při volání otevření souboru s neexistující cestou k souboru. Jazyk C++ poskytuje několik standardních typů výjimek, které jsou definovány v knihovně *exception*. Mezi tyto výjimky patří například *bad_alloc*, která může být vyvolána při volání *new* nebo *bad_exception*, sloužící k zachycení neočekávaných výjimek.

Zachytávání výjimky a obsluhu jejího kódu lze realizovat pomocí bloku *catch*, jehož parametr určuje, jaký druh výjimky zachytává, nebo v případě signálu lze využít k zachy-

cení tzv. *signalHandler*. *SignalHandler* je uživatelská funkce, zachytávající a zpracovávající signály. Identifikace druhu signálu probíhá na základě jeho identifikačního čísla.

```
#include <stdio.h>
int main() {
  try:
    throw 5; // Vyvolání výjimky
  catch(int eh) {
    ... // Blok zpracování výjimky typu int
  }
  catch(...) {
    ... // Defaultní blok pro zpracování výjimky
  }
  return 0;
}
```

Ukázka kódu 2.7: Ukázka jednoduchého zpracování výjimek.

2.4.1 Princip zpracování výjimek

Jak bylo zmíněno v předchozím textu, výjimky slouží k indikaci a zachycení neočekávaného stavu programu. Výjimky mohou být vyvolány programátorem pomocí klíčového slova *throw*, nebo mohou vzniknout v některé z programátorem použitých knihovních funkcí. V následující sekci bude popsáno jak výjimky fungují obecně. Jedná se o základní pochopení schématu zachytávání výjimek. Při zpracování výjimky se tedy postupuje následovně:

1. Napsané klíčové slovo *throw* překladač přeloží na volání interních funkcí standardní knihovny C++, které alokují výjimky a započnou proces uvolňování zásobníku volání voláním standardní knihovny jazyka C.

2. Pro každý blok *catch* překladač generuje speciální informace, které následují po tělu dané funkce. Jedná se o tabulku výjimek, které může tento blok zachytit a metody pro vyčištění tabulky.

3. Při procházení zásobníku volání se zavolá speciální funkce standardní knihovny jazyka C++, osobní rutina (personality routines), která pro každou funkci na zásobníku určí, která výjimka může být zachycena.

4. Pokud se pro výjimku nenajde žádný *catch* blok, který by ji mohl zpracovat, dojde k volání *std::terminate*[1].

5. V případě, že se nalezne *catch* blok zpracovávající danou výjimku, začíná prohledávání zásobníku od shora.

6. Při druhém prohledávání zásobníku volání dochází k dotazu personality routine na provedení čistící funkce.

7. Personality routine zkontroluje tabulky pro aktuální metody, pokud může zpracovat výjimku, skočí na aktuální rámec zásobníku volání a započne zpracování výjimky. To vede na volání destruktoru pro každý objekt alokovaný v aktuálním kódovém bloku.

[1] *std::terminate* - volá se pro abnormální ukončení programu, většinou vede na volání funkce *abort*. Toto chování může být předefinováno pomocí funkce *set_terminate*.

8. Pokud se při prohledávání zásobníku volání narazí na jeho rámec, který může zpracovat výjimky, dojde ke skoku na příslušný *catch* blok.

9. Po dokončení *catch* bloku se zavolá funkce pro uvolnění paměti alokované pro výjimku [6].

2.4.2 Způsoby řešení výjimek

Pro samotné řešení zpracování výjimek existují tři základní druhy přístupu, které zde budou představeny. Tyto způsoby zpracování doplňují obecný princip zpracování výjimek, jenž se nachází v předchozí části.

SEH - *Structured Exception Handling* Za první variantu zpracování výjimek lze označit Structured Exception Handling, který se využívá na operačních systémech Microsoft Windows. Tento princip zachytávání výjimek pracuje kromě softwarových výjimek, které zde byly uváděny také s výjimkami hardwarovými. Hardwarové výjimky jsou ve většině případů výjimky vyvolané procesorem, mohou být vyvolané určitou sekvencí instrukcí, například dělení nulou nebo přístupem na špatné místo v paměti. Oproti tomu softwarové výjimky jsou vyvolané aplikací, nebo operačním systémem.

SEH tedy umožňuje pracovat s oběma druhy výjimek stejně. Principielně SEH se skládájí z řetězce tvořeného z bloků pro zachycení výjimek (tzv. SEH řetězec), který je umístěn blízko dna zásobníku. Při vyvolání výjimky operační systém Windows předá hlavičku SEH řetězce. V tomto řetězci se následně hledá příslušný blok pro zachycení výjimky, který umožní správné ukončení aplikace. Pro SEH existuje také rozšíření o Vectored Exception Handling, které nepracují s rámci jako SEH, ale můžou být zavolány z libovolného místa, kde se nacházíme [4].

DWARF Exception Handling Využívá ladících informací programu ve formátu DWARF (*Debugging With Attributed Record Formats*). Jedná se o formát využívaný překladači a ladícími nástroji pro podporu ladění programů na úrovni zdrojových kódů. DWARF popisuje program jako stromovou strukturu, kdy každý uzel stromu může mít následníka nebo souseda. Každý uzel stromu poté může být proměnná, datový typ nebo funkce [16].

V těchto informacích se pro každou funkci zapisuje tabulka, která obsahuje informace o callee-saved registrech[2], návratovou adresu, frame pointer[3] a informaci o tom, kde jsou tyto hodnoty uloženy na zásobníku. Tyto informace se získávají z CFI (*Call Frame Information*) direktiv generovaných překladačem. Překladač poté dokáže efektivně pracovat s těmito informacemi pro obnovení registrů, nastavení návratových hodnot apod. Při tomto přístupu je nezbytně nutná podpora zadní části překladače, která musí poskytnout informace o dané architektuře tak, aby byly známy calle-saved registry, frame pointer apod. Dále musí knihovna jazyka C++ obsahovat interpret formátu DWARF, aby bylo možné přečíst a vyhodnotit tyto informace za běhu.

Set Jump and Long Jump Exception Handling Jedná se o zachytávání výjimek s využitím funkcí *setjmp* a *longjmp*. V tomto případě se klíčové slovo *try* v překladači nahradí za *setjmp* a *throw* za *longjmp*. Provázaný list *jmp_buf* struktur reprezentuje dynamický list

[2]Callee-saved registry - jsou registry, které si uchovávají hodnotu. Jejich obsah bude po návratu z volané rutiny shodný s obsahem před voláním této rutiny.

[3]Frame pointer - ukazatel na vrchol zásobníku před voláním funkce.

try bloků [7]. Jedná se o mechanismus ze standardu jazyka C, kde funkce *setjmp* ukládá kontext[4] vykonávané aplikace do struktury *jmp_buf*. Funkce *longjmp* poté na základě této struktury přesune tok programu na místo, kde byl původně *setjmp* volán. Přesunu předchází obnovení výchozího stavu programu, který je uložen v *jmp_buf*. Po tomto návratu se mění i hodnota, kterou funkce *setjmp* vrací. Tedy v případě prvního volání *setjmp* se nastaví struktura *jmp_buf* a vrací se hodnota 0. Následuje provedení *longjmp*, které obnoví výchozí stav a zajistí následovné vykonání *setjmp*, který již vrací hodnotu, která se liší od 0 a odpovídá hodnotě uvedené v *longjmp*.

2.5 Standard C++11

Standard C++11, oproti předchozím standardům jazyka C++, podstatně vylepšuje knihovní funkce předchozích standardů, přináší nové funkcionality, změny samotného jazyka a opravy. Mezi hlavní změny jazyka patří:

- r-hodnotové reference k odstranění nadbytečného kopírování,
- inicializační seznamy,
- jednotná inicializace kontejnerů,
- *auto* a *decltype*, kde *auto* nastaví typ proměnné automaticky podle přiřazovaného typu a *decltype* vrací typ parametru,
- cyklus *for*, který pracuje na základě rozsahu,
- lambda funkce, umožňující deklarovat i definovat dočasnou funkci,
- *nullptr* jedná se o nulový ukazatel, který lze převést na libovolný typ ukazatele a hodnotu typu *boolean*,
- *static_assert* v případě nesplnění podmínky v době překladu vede k pádu překladu aplikace a další [12].

Mezi přidané knihovny standardu, kromě knihoven přidávajících podporu výše zmíněných funkcionalit, patří například:

- *typetraits* a *typeindex*, které rozšiřují podporu RTTI,
- *chrono* pro práci s časovými údaji,
- *scoped_allocator* rozšiřující podporu dynamické správy paměti,
- *random*, *ratio*, *cfenv* numerické knihovny,
- *regex* přidávající regulární výrazy[5],
- *atomic*, který přidává do C++ atomické operace[6],
- *thread*, *mutex*, *future* a *contition_variable* pro podporu vláken,
- a další knihovny rozšiřující jazyk.

[4]Kontext aplikace - je tvořen obsahem registrů v okamžiku volání funkce *setjmp*.

[5]Regulární výraz - jedná se o sekvenci znaků, pomocí kterých zapíšeme hledaný řetězec. Obsahují speciální znaky, jako jsou například . značící libovolný znak, + označující 1 a více opakování, * pro 0 a více opakování apod.

[6]Atomické operace - zaručují, že činnost bude provedena najednou a nebude ničím přerušena.

2.6 Existující knihovny C++

V následující části budou představeny existující implementace standardní knihovny jazyka C++. Mezi nejpopulárnější řešení patří:

- knihovna vyvíjená přímo pro překladač LLVM libc++ a poté
- knihovna, která je součástí GCC - GNU Compiler Collection a to libstdc++.

Knihovna libc++ Knihovna libc++ je vyvíjena v rámci projektu LLVM. Knihovna se zaměřuje na standard C++11 a výše, kdy má kompletně implementovaný standard C++11, C++14 a aktuálně se postupně přidává podpora standardu C++17. Implementace ABI[7] knihovny libc++ udržuje i kompatibilitu s knihovnou libstdc++. Knihovna je licencována pod licencemi MIT[8] a UIUC[9], které dovolují její úpravy a případné využití v proprietárním softwaru.

Knihovna libstdc++ Knihovna libstdc++ vzniká v rámci GCC. Tato knihovna je dostupná pod licencí GPLv3[10], která neumožňuje využití v proprietárním softwaru. Přesněji knihovna může být využita, ale veškeré změny v rámci knihovny musí být zveřejněny pod stejnou licencí. Knihovna obsahuje implementace všech dosavadních standardů od C++98 do C++14 a standard C++17 se stále vyvíjí. Knihovna kromě překladače g++ spolupracuje s překladači Clang, EDG C++ překladačem nebo Intel ICC C++ překladačem.

Knihovna libstdcxx Další existující knihovnou je libstdcxx, která byla vyvíjena firmou Apache. Její poslední aktualizace proběhla v květnu 2008 a v květnu roku 2014 byl její vývoj zastaven. Knihovna byla postavena na knihovně STLport.

Knihovna STLport Knihovna STLport je stále se vyvíjející multiplatformní implementace standardní knihovny jazyka C++, která je volně dostupná a její licence dovoluje volné šíření knihovny i její dokumentace. V knihovně STLport není zcela implementovaná podpora standardu C++11, na jehož plné podpoře se pracuje.

Knihovna firmy Microsoft Za zmínku zde dále stojí implementace standardní knihovny firmou Microsoft pro systémy Windows, která je součástí nástrojů MSVC překladače. Knihovna má implementovány kompletně všechny standardy včetně standardu C++17.

---

[7] Application Binary Interface (ABI) - nízkoúrovňové rozhraní. Toto rozhraní definuje pravidla pro spolupráci knihovny s procesy, které ji využívají.

[8] Licence MIT - vznikla na Messachusettském technologickém institutu. Umožňuje použít software pod touto licencí jednak v proprietárním software, tak i pod GPL licencovaným softwarem. Podmínkou použití licence je, že text licence se musí nacházet u softwaru spolu se jménem autora.

[9] Licence UIUC - vznikla na Univerzitě Illinois, licence vychází z licence MIT a BSD. Stejně jako licence MIT je kompatibilní s většinou licencí. Podmínky obsahují požadavek na zachování licence, jmen autorů a zákaz použití jmen autorů nebo organizace k propagaci.

[10] Licence GPLv3 - původně pro projekt GNU, vyžaduje aby odvozená díla obsahovala stejnou licenci.

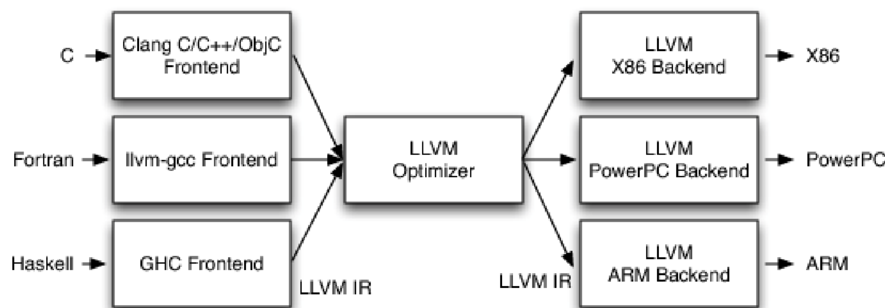
Kapitola 3

LLVM framework

LLVM byl dříve akronym pro Low Level Virtual Machine, dnes se jedná o jméno celého projektu, který toho má s virtualizací málo společného. LLVM se skládá ze sady knihoven a nástrojů, s jejiž kombinací lze realizovat překladač, optimalizační nástroje, případně jakýkoliv jiný nástroj příbuzný s překladačem. Základem jsou takzvané Core knihovny, které poskytují optimalizační nástroj, který je nezávislý jak na vstupních datech, tak i na cílové architektuře. Tyto knihovny pracují s vnitřní reprezentací - LLVM intermediate representation, neboli zkráceně LLVM IR. Tato reprezentace bude představena v následující části textu. Mezi další části LLVM patří například Clang, což je překladač C/C++/Objective C, projekt poskytující nástroje pro ladění kódu - LLDB, knihovny implementující standard C++ a další. LLVM bylo původně výzkumným projektem na University of Illinois [15]. Celý projekt je napsán v jazyce C++.

3.1 Průběh překladu

Překlad programu s pomocí LLVM má tři fáze, jejichž průběh znázorňuje obrázek 3.1. Můžeme si všimnout, že zdrojový kód projde nejdříve přední částí (frontend), následně se dostane do části prostřední (midend), kterou v případě LLVM tvoří optimalizátor a v poslední fázi kód projde zadní část(backend). Všechny části jsou v ideálním případě na sobě nezávislé. Nezávislost jednotlivých částí umožňuje na změnu programovacího jazyka překládaného projektu reagovat pouze změnou frontendu bez nutnosti měnit zbývající části. Také v případě změny cílové architektury se změní pouze backend a optimalizátor s frontendem zůstanou beze změny.



Obrázek 3.1: Průběh překladu v LLVM [8].

Frontend Přední část provádí zpracování programu napsaném ve vstupním programovacím jazyce. Provádí lexikální analýzu, která je následována analýzou syntaktickou. Po syntaktické analýze dochází k analýze sémantické a budování abstraktního syntaktického stromu (AST). AST jsou abstraktní reprezentací vstupního programu, neobsahují specifické syntaktické detaily, jako jsou například středníky. Vnitřní uzly AST představují operátory a listy jsou jednotlivé operandy daných výrazů. Poté co se vytvoří AST, dojde na jeho základě ke generování vnitřního kódu LLVM IR, který je předán optimalizátoru. Pro LLVM existuje řada implementací frontendů, kdy za zmínku stojí Clang, který podporuje jazyky C, Objective C, C++ a Objective C++. Ten porušuje nezávislost frontendu na cílové architektuře, protože v době překladu potřebuje znát *target datalayout*, který je reprezentován jako textový řetězec, obsahující základní informace o cílové architektuře. Mezi tyto informace patří endianita, velikost a zarovnání ukazatelů, velikost a zarovnání datových typů apod. Na ukázce kódu 3.1 lze vidět zdrojový kód v jazyce C a jeho odpovídající zápis v LLVM IR na 3.2 pro funkci *function_to_inline* a na ukázce 3.3 pro funkci *main*, který byl vygenerovaný frontendem.

```
#include <stdio.h>
void function_to_inline() {
  printf("Hello world\n");
}

int main(int argc, char const *argv[]) {
  function_to_inline();
  return 0;
}
```

Ukázka kódu 3.1: Ukázka zpracování jednoduchého kódu v jazyce C.

```
define void @function_to_inline() #0 {
entry:
  %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds (
    [9 x i8], [9 x i8]* @.str, i32 0, i32 0))
  ret void
}
```

Ukázka kódu 3.2: Výstup frontendu pro funkci *function_to_inline*.

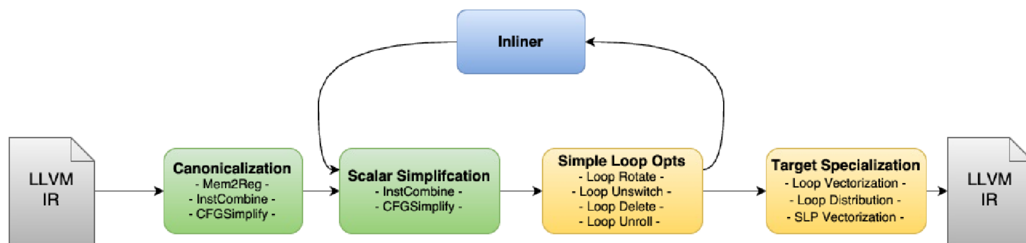
```
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 4
  store i32 0, i32* %retval, align 4
  store i32 %argc, i32* %argc.addr, align 4, !tbaa !2
  store i8** %argv, i8*** %argv.addr, align 4, !tbaa !6
  call void @function_to_inline()
  ret i32 0
}
```

Ukázka kódu 3.3: Výstup frontendu pro funkci *main*.

LLVM Optimizer Vstupem optimalizátoru je zdrojový kód ve vnitřní reprezentaci LLVM IR, nad kterou se provádí jednotlivé optimalizace. Optimalizace jsou aplikovány postupně formou jednotlivých průchodů. Samotné průchody mohou být buď optimalizace kódu, které ovlivňují výsledný tvar kódu, nebo průchody provádějící analýzu kódu, která nijak přímo neovlivňuje výsledný kód, ale slouží jako zdroj informací pro následující optimalizační průchody tak, aby bylo dosaženo rychlejšího vykonávání výsledného programu, případně co nejmenší velikosti kódu. LLVM Optimizer umožňuje s využitím příkazového řádku specifikovat, které průchody, případně skupiny průchodů budou při překladu využity.

Na obrázku 3.2 vidíme průchod vstupního kódu ve formátu LLVM IR optimalizátorem. Jednotlivé fáze optimalizátoru:

- canonicalization - se zaměřuje především na přípravu instrukcí pro optimalizace, kombinování instrukcí, zjednodušování, převod proměnných z paměti do registrů,
- scalar simplification - zjednodušování pomocí kombinování instrukcí apod.,
- simple loop opts - optimalizace cyklů,
- inliner - vkládání těl funkcí namísto jejich volání,
- target specialization - optimalizace zaměřené na cílovou architekturu, které zaručí využití výhod specifických architektur.



Obrázek 3.2: LLVM optimizer optimalizační fáze [3].

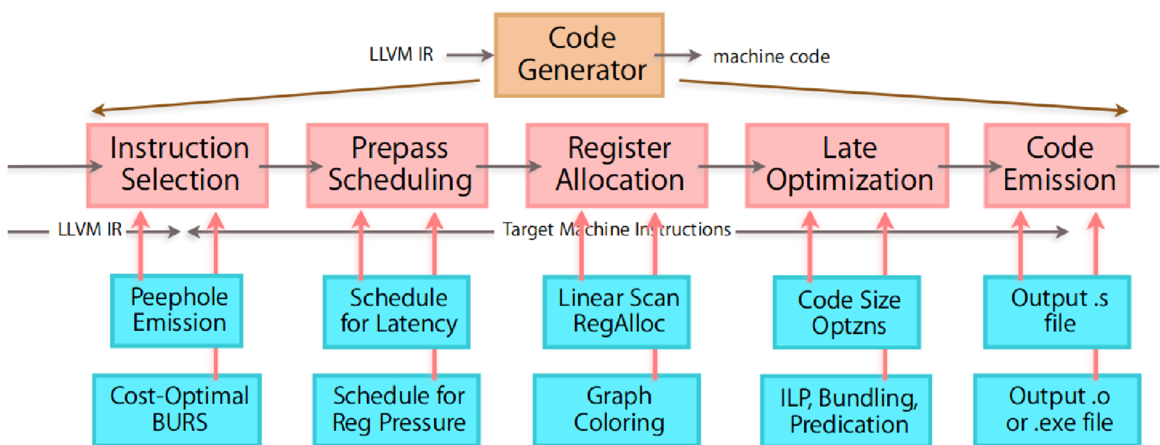
Na ukázce kódu 3.4 lze vidět kód z ukázky 3.1 po zpracování optimalizátorem pro optimalizační úroveň *O3*, která se zaměřuje na optimalizace pro dosáhnutí maximální rychlosti vykonávaného kódu. Lze vidět, že kód byl značně zjednodušen, kdy se volání funkce *function_to_inline* nahradilo za volání funkce *puts* a výsledný kód LLVM IR obsahuje pouze dvě instrukce.

```
define i32 @main(i32 %argc, i8** nocapture readnone %argv) ↩
   local_unnamed_addr #0 {
entry:
  %puts.i = tail call i32 @puts(i8* getelementptr inbounds ([8 x i8],
    [8 x i8]* @str, i32 0, i32 0)) #1
  ret i32 0
}
```

Ukázka kódu 3.4: Výstup optimalizátoru pro funkci main.

Backend Provádí převod vstupního kódu ve formátu LLVM IR na kód cílové architektury. Fáze převodu jsou znázorněny na obrázku 3.3, kde:

- instruction selection - selekce instrukcí neboli výběr instrukcí podle vzoru,
- prepass scheduling - plánování instrukcí,
- register allocation - alokace registrů, přiřazení fyzických registrů proměnným a virtuálním registrům,
- late optimization - finální optimalizace kódu cílové architektury, např. peephole optimalizace[1],
- code emission - zápis do výstupního souboru.



Obrázek 3.3: LLVM backend fáze převodu LLVM IR na kód cílové architektury [5].

Výsledný kód pro ukázku 3.1 v jazyce symbolických adres znázorňuje kód 3.5 pro cílovou architekturu Codasip uRISC, která bude představena v následujících kapitolách.

```
$main:
  r4 = movsi $str & 0xffff
  r0 = addi r0, -8
  r4 = movhi $str >> 16 & 0xffff
  st r3, [ r0 + 4 ]
  call $puts
  r3 = ld [ r0 + 4 ]
  r4 = movsi 0
  r0 = addi r0, 8
  jump r3
```

Ukázka kódu 3.5: Výstup backendu.

[1]Peephole optimalizace - spočívá v nahrazování určitých vzorů instrukcí za efektivnější kombinace.

3.2 LLVM Intermediate Representation

LLVM Intermediate Representation neboli LLVM IR označuje vnitřní reprezentaci kódu překladače LLVM. Tato vnitřní reprezentace se vyskytuje ve všech částech překladače, jak bylo zmíněno výše. LLVM IR se vytvoří v rámci frontendu. Následně se nad ní provedou optimalizace v optimalizátoru a na závěr se v rámci backendu převede na reprezentaci cílové architektury v rámci selekce instrukcí.

Samotné LLVM IR lze reprezentovat ve třech formách, které jsou navzájem ekvivalentní. Tyto formy jsou:

- forma datových struktur uložených v paměti,
- čitelná textová verze podobná jazyku symbolických adres a
- bitcode ukládáný na disku, vhodný k rychlému načtení pro *Just-In-Time* kompilaci [9].

LLVM IR počítá s neomezeným počtem registrů, na rozdíl od reálných architektur, které mají značně omezený počet fyzických registrů. LLVM IR využívá přístupu SSA (*Static Single Assigment*), který říká, že každému registru může být přiřazena hodnota pouze jednou. Tedy daný registr se může vyskytnout na levé straně výrazu pouze jedenkrát. V rámci LLVM IR to znamená, že se pro každý výsledek instrukce vytvoří zcela nový virtuální registr. Ukázka principu SSA lze vidět na kódu 3.7, který odpovídá zápisu v jazyce C znázorněného na ukázce 3.6. V této ukázce kódu se provádí dvojí zápis do proměnné a, ale v rámci LLVM IR se vytvoří dva virtuální registry a a $a2$. Nedojde tedy k přepsání hodnoty a, jako je tomu v jazyce C na ukázce. O mapování těchto vnitřních virtuálních registrů na registry fyzické se stará v rámci backendu alokátor registrů.

```
int a = someFunction();
a += 1;
```

Ukázka kódu 3.6: Zdrojový kód v jazyce C [10].

```
%a = call i32 @someFunction()
%a2 = add i32 %a, 1
```

Ukázka kódu 3.7: Zápis v LLVM IR [10].

LLVM IR využívá tříadresní kód, ten pracuje vždy se třemi operandy, kterými jsou dva operandy pro vstupní hodnoty a jeden operand pro výsledek. Syntaxe samotných instrukcí vychází ze syntaxe RISC architektury[2]. LLVM IR podporuje tyto instrukce:

- aritmetické - *add*, *mul*, *div* apod.,
- porovnání - *seteq*, *setlt* atd.,
- pro práci s pamětí - *load*, *store*,
- pro podporu řízení toku programu - například *br*, *call*,
- bitovou manipulaci, vektorové operace a další.

[2]Architektura typu RISC se vyznačuje především nižším počtem jednoduchých instrukcí, jednotnou délkou i tvarem instrukcí a snahou dosahovat zpracování instrukce/takt. V rámci architektury se složitější instrukce nahrazují posloupností jednoduchých instrukcí. Za další významný rys architektury lze označit omezenou komunikaci s pamětí, kdy do paměti mohou přistupovat pouze instrukce typu *load* a *store*.

Důležitou vlastností LLVM IR instrukcí je, že podporují polymorfismus, kdy například jednoduchá instrukce *add* může pracovat se širokou škálou typů vstupních operandů, aniž by bylo nutné definovat speciální instrukce. Samotná sémantika instrukce se odvodí ze vstupních operandů instrukce [10].

LLVM IR shlukuje sekvence jednotlivých instrukcí do tzv. *basic blocks*, neboli základních bloků. Instrukce v daném bloku se vykonávají v pořadí, v jakém jsou zapsány. Blok se provede vždy celý, začíná první instrukcí bloku a končí instrukcí poslední, která patří do skupiny terminátorů.

Terminátorem jsou většinou instrukce pro řízení toku programu, patří mezi ně:

- *ret* - instrukce pro navrácení z funkce zpět k volajícímu,
- *br* - instrukce skoku, ta obsahuje dva cílové basic bloky, jeden pro případ kladného vyhodnocení podmínky a druhý pro případ záporného vyhodnocení,
- *switch* - na základě vstupní hodnoty vybírá z více možných cílových basic bloků,
- *invoke* - tok programu se přesune do specifické funkce, ze které se buď vrátí zpět do programu, nebo dojde k obsluze výjimky, tedy skoku na landing pad[3], který určí další chování programu pro výjimky,
- *resume* - instrukce, která pokračuje propagací stávající výjimky, jejíž zpracování bylo přerušeno instrukcí landing padu,
- *unreacheble* - nemá definovanou sémantiku, říká optimalizátoru, že kód po *unreacheble* nelze dosáhnout,
- další instrukce *indirectbr*, *catchswitch*, *catchret* a *cleanupret*.

Mezi terminátory nepatří instrukce *call*, jelikož po dokončení volané funkce se pokračuje instrukcí, jež následuje volání funkce a tedy dojde k vykonání celého bloku až po terminátor. Z jednotlivých bloků jsou poskládány funkce vstupního programu, nad těmito celky jsou prováděny jednotlivé optimalizační kroky. Funkce jsou následně shlukovány do modulů.

3.3 DAG - Directed Acyclic Graph

Kód ve formátu LLVM IR překladač konvertuje do podoby orientovaného acyklického grafu (*Directed Acyclic Graph*). Tato konverze se odehrává v zadní části překladače. Využití DAG usnadňuje výběr instrukcí, kdy jednotlivé instrukce architektury jsou reprezentovány rovněž pomocí DAG. Selekce instrukcí je poté problémem pokrytí grafu, kdy grafem se zde rozumí basic block reprezentující část funkce vstupního programu, podgrafy, reprezentujícími jednotlivé instrukce cílové architektury. Jeden graf vstupního programu tedy představuje jeden *basic block*. Nad tímto grafem se odehrávají další průchody překladu. Mezi hlavní části překladu probíhajícího nad strukturou DAG patří:

- Kombinace - slouží k nahrazování neoptimálních konstrukcí za optimální. Zde se nachází optimalizace, které nemohou být provedeny nad LLVM IR, jelikož LLVM IR pracuje na vyšší úrovni abstrakce na rozdíl od DAG.

[3] Landing pad - obsahuje specifické klauzule a informace určující, které výjimky mohou být zachyceny a zpracovány daným blokem *catch*.

- Legalizace - skládá se ze dvou částí, kdy první část zajišťuje, aby graf obsahoval pouze nativně podporované datové typy a druhá část slouží k legalizaci operací, tato část zajišťuje, že graf obsahuje pouze podporované operace.
- Selekce instrukcí - nahrazování pseudoinstrukcí za instrukce cílové architektury.
- Plánování instrukcí - pro dosažení lepších výsledků v závislosti na době zpracování instrukcí.

Kapitola 4

Codasip Studio

V následující kapitole se nachází představení vývojového prostředí Codasip Studio. V první části se čtenář seznámí s vývojovým prostředím Codasip Studio. Druhá část popisuje zápis instrukční sady v rámci vývojového prostředí, které slouží jako vstup generátoru překladače jazyka C nebo C++.

4.1 Vývojové prostředí

Codasip Studio je automatizované vývojové prostředí, sloužící k tvorbě aplikačně specifických procesorů a platforem, které mohou kromě samotných procesorů obsahovat paměti, sběrnice a další části. Codasip Studio umožňuje generování SDK (*Software Development Kit*) pro vyvíjený projekt. Tato sada obsahuje assembler, překladač, simulátor, knihovny apod. Dále Studio umožňuje generování simulačních nástrojů, generování syntetizovaného RTL (*Register Transfer Logic*) popisu architektury a funkční verifikaci.

Codasip Studio se skládá z několika částí, mezi které patří:

- uživatelské rozhraní - vývojové prostředí založené na Eclipse[1],
- Codasip commandline - jedná se o interpret jazyka Python, který se stará o správné provedení příkazů, ty mohou být vyvolány na základě akce v uživatelském rozhraní nebo přímo z příkazové řádky,
- sady nástrojů - například linker,
- generátory nástrojů - zde patří generátor překladače jazyka C/C++ [2].

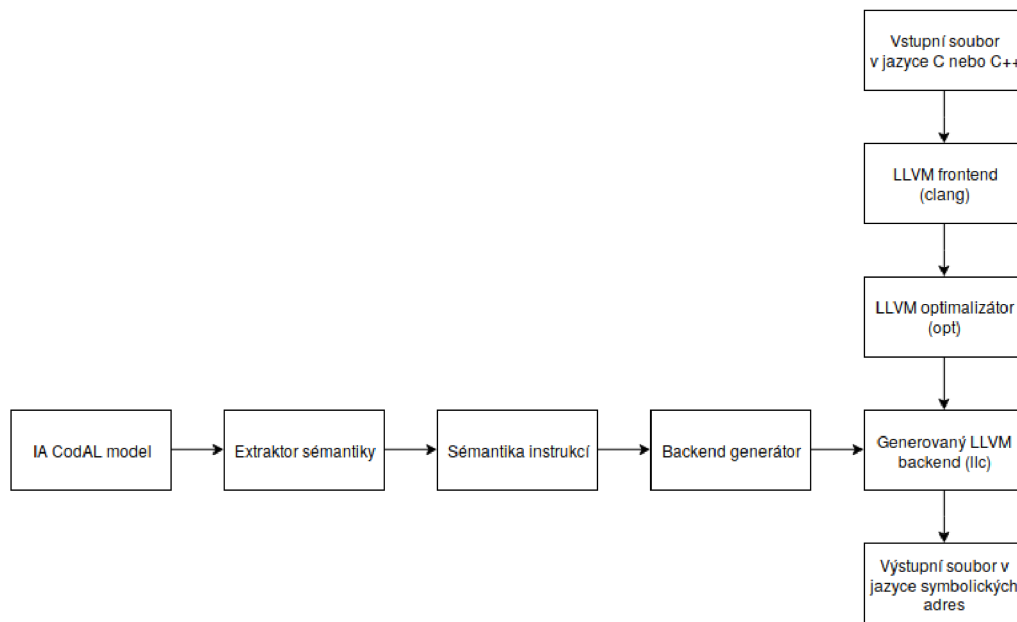
Generování překladače jazyka C/C++ znázorňuje obrázek 4.1. Na obrázku lze vidět, že proces generování překladače jazyka C/C++ začíná návrhem IA (Instruction Accurate) modelu procesoru v jazyce CodAL. Jazyk CodAL bude popsán v kapitole 4.2. IA model označuje úroveň abstrakce návrhu a označuje model na úrovni instrukční sady. Tento model se následně zpracuje extraktorem sémantiky. Na základě získané sémantiky jednotlivých instrukcí dojde s využitím generátoru k vytvoření zadní části překladače. Ta spolu s přední částí a optimalizátorem tvoří samotný překladač jazyka C/C++. Výstupem překladače jazyka C/C++ není spustitelný soubor, ale soubor v jazyce symbolických adres, který musí být zpracován pomocí assembleru, jehož výstupem je objektový soubor. Objektové soubory

[1]Eclipse - open source vývojová platforma

slouží jako vstup pro linker, jehož výstup tvoří jeden spustitelný binární soubor. Takto vygenerovaný binární soubor může být následně spuštěn na cílové architektuře, nebo simulátoru generovaným Codasip Studiem.

Codasip Studio momentálně podporuje programy zapsané v jazyce C. K podpoře využívá knihovny Newlib, která implementuje standardní knihovnu jazyka C pro vestavěné systémy a knihovny Compiler RT, která poskytuje definici funkcí, jež nejsou přímo podporovány na úrovni hardwaru.

Podpora jazyka C++ je implementována pouze částečně, kdy má uživatel možnost vygenerovat knihovnu libc++abi a převzít hlavičkové soubory z knihovny libc++. Knihovna libc++abi se překládá bez možnosti využití RTTI, kdy kód s RTTI byl z knihovny odstraněn. Kromě RTTI v knihovně také chybí podpora zpracování výjimek. Knihovna libc++ není překládána, pouze jsou využity hlavičkové soubory knihovny a knihovna libunwind se v Codasip Studiu nenachází.



Obrázek 4.1: Schéma generování překladače a překlad programu v Codasip Studiu [1].

4.2 Jazyk pro popis architektury CodAL

Jak bylo zmíněno výše, jazyk CodAL neumožňuje pouze popis procesoru na úrovni instrukcí (IA), ale i na úrovni samotné architektury (CA). Z pohledu generování překladače se zajímáme o popis na úrovni IA modelu, který obsahuje popis:

- zdrojů architektury - registry, paměti, programový čítač,
- instrukční sady - textovou podobu instrukcí a jejich binární kódování,
- sémantiky instrukcí - popis chování instrukcí.

Zdroje architektury Do této kategorie patří především registry, tedy základní registrové pole, registrové aliasy, které umožňují přistupovat do již definovaného registrového pole

pod jiným jménem, případně registry speciální jako je programový čítač. Programový čítač je registr, který má omezení na počet výskytů v rámci jedné architektury, kdy musí být deklarován pouze jednou. Dále mohou být implementovány signály, porty nebo rozhraní pro komunikaci s periferními zařízeními.

Instrukční sada Jednotlivé instrukce jsou popsány v rámci elementů, jak je znázorněno na ukázce kódu 4.1. Tyto elementy se následně seskupují do větších skupin, které jsou označeny klíčovým slovem *set*. Ty pak tvoří v rámci jednoho *set* stromovou strukturu popisující celou instrukční sadu procesoru.

```
element nazev_elementu { // jméno elementu
  use xy_part as reg; // použití existujícího prvku
  assembly { "INS" reg }; // reprezentace instrukce v jazyce assembler
  binary {0xFFFFFFF reg}; // binární kódování instrukce
  semantics { ... }; // definice sémantiky elementu
  return { val; }; // definice návratové hodnoty
  timing { ... }; // události spojené s elementem
};

set novy_set = nazev_elementu; // přidání elementu do setu
```

Ukázka kódu 4.1: Ukázka elementu jazyka CodAL [11].

Sémantika Popis sémantiky se zapisuje pomocí jazyka ANSI C, z něhož CodAL podporuje veškeré operátory, kromě ukazatelů a operátoru *sizeof*. Popis sémantiky pracuje s vnitřními zdroji architektury, jako jsou registry, paměti nebo periferní zařízení. Samotnou sekci pro popis sémantiky lze nalézt buď v rámci elementů pro popis instrukcí, nebo v rámci událostí (*event*). Mezi události patří nastavení procesoru po restartování nebo hlavní smyčka procesoru, která se stará o inkrementaci programového čítače a dekódování jednotlivých instrukcí.

Na ukázce kódu 4.2 lze vidět zápis instrukce pro sčítání hodnot dvou registrů. Odpovídající zápis po extrakci sémantiky se nachází na ukázce 4.3, kde lze vidět v sekci *semantics*, načtení dvou vstupních operátorů pomocí *regop*, provedení sčítání s využitím *add* a následným zápisem do cílového registru.

```
element i_add
{
  // Používá operační kód add jako proměnnou opc
  use OPC_ADD as opc;
  // Využití registrů z registrového pole gpr_all
  use gpr_all as gpr_dst, gpr_src1, gpr_src2;
  // Zápis v jazyce assembler
  assembly { gpr_dst "=" "add" gpr_src1 "," gpr_src2 };
  // Binární zápis instrukce
  binary { opc gpr_dst gpr_src1 gpr_src2 };
  semantics // Zápis sémantiky
  {
    uint32 src1, src2, vysledek;
    // Načtení obsahu registru
    src1 = rf_gpr[gpr_src1];
    src2 = rf_gpr[gpr_src2];
    // Uložení výpočtu výsledné hodnoty
    vysledek = src1 + src2;
    // Zápis výsledné hodnoty do cílového registru
    rf_gpr[gpr_dst] = vysledek;
  };
};
```

Ukázka kódu 4.2: Zápis instrukce add s využitím jazyka CodAL.

```
instruction i_add__gpr_all__opc_add__gpr_all__gpr_all__
{
  operands { reg_0 = regop(gpr_all),
    reg_1 = regop(gpr_all), reg_2 = regop(gpr_all) };
  assembly { reg_0 "=" "add" reg_1~"," reg_2 };
  binary { 0b00000101:bit[8] reg_0[4..0] reg_1[4..0]
    reg_2[4..0] 0b000000000:bit[9] };
  semantics
  {
    %5 = i32 regop(reg_2);
    %6 = i32 regop(reg_1);
    %4 = i32 add(i32 %5, i32 %6);
    regop(reg_0) = i32 %4;
  };
  element_tree =
    "el:i_add(el:gpr_all, el:opc_add, el:gpr_all, el:gpr_all)";
  slots = {{0}};
};
```

Ukázka kódu 4.3: Element po extrakci sémantiky.

Kapitola 5

Zvolené řešení

Následující kapitola se zabývá volbou standardní knihovny jazyka C++, která byla zvolena k integraci do nástrojů Codasip. Dále se kapitola zabývá volbou způsobu řešení výjimek, která bude dále implementována.

5.1 Knihovna jazyka C++

Po prozkoumání možností jsem zvolil knihovnu libc++. Tato knihovna se zaměřuje na standard C++11 a výše. Je stále ve vývoji, což umožňuje v budoucnu rozšířit podporu jazyka C++ o další standardy. Knihovna je součástí projektu LLVM a je tedy zaručena kompatibilita s překladačem Clang, umožňuje kompatibilitu s knihovnou libstdc++ a její licence umožňuje použití v proprietárním softwaru.

Knihovna od Apache nemohla být zvolena z důvodu ukončení vývoje. Knihovna STLport i přes naprosto volnou licenci a pravidelné aktualizace nemohla být zvolena, neboť její vývoj je velmi zdlouhavý a není zatím implementována plná funkcionalita standardu C++11. Knihovna od firmy Microsoft poté není volně dostupná. Knihovna libstdc++ se stejně jako knihovna libc++ stále vyvíjí a má implementovány všechny standardy po standard C++17, ale její licence nedovoluje využití v proprietárním softwaru.

Kromě vlastní implementace knihovny libc++ je k jejímu fungování nezbytně nutné využívat knihovny libc++abi a libunwind. Jak lze z názvu vyčíst, knihovna libc++abi implementuje C++ ABI. Obsahem knihovny jsou například operátory *new* a *delete*, nebo funkce pro práci s výjimkami apod. Mezi funkce pro práci s výjimkami patří funkce pro alokaci, vyvolání, zpracování i uvolnění výjimek. Pokud nastane výjimka k jejímu zpracování se využívají funkce knihovny libc++abi. Knihovna libc++abi pak využívá funkcí knihovny libunwind poskytující podporu pro zachytávání výjimek. Knihovna libunwind se převážně stará o práci se zásobníkem, která záleží na typu využívaných výjimek. U výjimek typu Set Jump Long Jump knihovna slouží k vytváření a zahazování jednotlivých rámců zásobníku volání. U výjimek využívajících ladících informací DWARF se knihovna libunwind stará o vytváření rámců zásobníku volání na základě ladících informací v případě vyvolané výjimky. Knihovny libc++abi a libunwind jsou obdobně jako libc++ implementovány v rámci projektu LLVM a poskytovány pod stejnou licencí tedy MIT a UIUC.

5.2 Řešení výjimek

Pro řešení výjimek byla zvolena varianta implementující výjimky s podporou Set Jump Long Jump. Tyto výjimky vyžadují implementaci builtin funkcí[1] *setjmp* a *longjmp*, které se starají o řízení toku programu v případě vzniku výjimky. Při volbě implementace výjimek byla nejdříve vyřazena implementace výjimek SEH, která vyžaduje podporu operačního systému.

Práce se dále také zabývá implementací výjimek s využitím DWARF informací pro ladící nástroje. Toto řešení výjimek vede k menšímu počtu cyklů programu v případě, že nedojde k vyvolání výjimky, neboť informace o daném rámci jsou již zakódovány v daných regionech pro výjimku. Tento přístup je rychlejší v případě, že nedojde k výjimce. Pokud k výjimce dojde, pak je rychlejším způsob v podobě Set Jump Long Jump implementace výjimek, který vytváří a odstraňuje rámce za běhu programu. Nevýhodou zpracování výjimek využívajících ladící informace DWARF je nutná podpora na všech úrovních překladu. Tedy u překladače jazyka C++, u kterého je nutné zajistit generování ladících informací DWARF. V programu assembler, kdy se ladící informace musí správně zapsat do objektového souboru a následně i v programu linker, kdy musí dojít ke správnému spojení informací o výjimkách v jednotlivých objektových souborech. Kromě podpory na straně těchto nástrojů, musí být implementována podpora v rámci knihovny libunwind, která vyžaduje implementaci specifické třídy reprezentující registrový prostor procesorového jádra. Tato třída následně poskytuje informace o zpracovaní jednotlivých rámců, například kde se nachází návratový registr, registr ukazatele vrcholu zásobníku apod.

Ve finální implementaci by měl mít uživatel překladače možnost zvolit, který typ výjimek bude využívat. Předpokládá se, že preferovanější způsob zpracování výjimek bude řešení s využitím DWARF informací, neboť dosahují rychlejšího vykonání kódu. Rychlejší vykonání kódu způsobuje absence vytváření a zahazování rámců obsahujících informace o kontextu v průběhu programu, ty se vytváří pouze při vyvolání výjimky. To způsobuje pomalejší zpracování výjimky, ale rychlejší zpracování kódu v případě, že k výjimce nedojde. Obecně je tento přístup lepší, neboť k výjimkám by nemělo docházet často.

[1] Builtin funkce - jedná se o funkce, které jsou vždy k dispozici, pro jejich použití není nutné využívat knihovny.

Kapitola 6

Volání virtuálních metod

Následující kapitola popisuje způsob volání virtuálních metod implementovaných v rámci frameworku *LLVM*. Volání virtuálních metod je řešeno v rámci frameworku bez ohledu na cílovou architekturu a nemuselo být nijak zasahováno do jeho implementace. Kapitola se tedy zabývá pouze analýzou již existující implementace.

6.1 LLVM - řešení volání virtuálních metod

Ukázka kódu 6.1 využívající jednoduchého volání virtuálních metod bude dále sloužit jako reference.

Odpovídající kód LLVM IR se nachází na ukázce 6.2. Na ukázce si lze všimnout, že v případě prvního volání *fun_1* se využívá instrukce *call* na návěští *_ZN6bazova5fun_1Ev*, které je pevně dáno a volá se tedy přímo. U volání druhé (*fun_2*) a třetí (*fun_3*) metody, ale dochází nejdříve k načtení adresy cíle z tabulky virtuálních metod *vtable*, až poté dojde k volání *call* na příslušnou funkci, kterou udává adresa extrahovaná z tabulky virtuálních metod. Tento kód LLVM IR se nachází již před prvním optimalizačním průchodem. Z toho lze vyvodit, že volání virtuálních metod se řeší již na úrovní přední části překladače.

V rámci přední části překladače probíhá tvorba virtuálních tabulek, které poskytují informace pro volání virtuálních metod nebo RTTI. Tyto tabulky se vytváří pro každou třídu, která má nějakou virtuální metodu, případně její rodič obsahuje virtuální tabulky. Tabulky virtuálních metod se v rámci implementace LLVM skládají ze sekvence offsetů, ukazatelů na data a funkce. Kde offset většinou slouží k rychlému naleznutí tabulky virtuálních metod třídy postavené nejvýše v hierarchii dědění. Tvorba tabulek virtuálních metod se odehrává ve dvou třídách. První třída *VTTBuilder* slouží k tvorbě pole adres tabulek virtuálních metod. Tyto pole jsou definovány pro každou třídu, která obsahuje tabulku virtuálních metod a slouží k volání konstruktorů a destruktorů bázových tříd ve správném pořadí. Druhá třída *VTBuilder* se stará o tvorbu samotných tabulek virtuálních metod.

Detailnější popis bude uveden pro kód z příkladu 6.1. Tomuto kódu odpovídá zápis virtuálních tabulek na ukázce 6.3. Ten lze získat s využitím přepínačů *-Xclang -fdump-vtable-layouts* při volání překladače, kde první argument specifikuje použití druhého argumentu na program clang. Zde lze vidět, že každá třída má vlastní tabulku virtuálních metod a tabulku indexů dané třídy. První tabulka obsahuje na řádku 0 zápis offsetu, který je zde roven 0. Následující řádek uvádí hierarchii dědičnosti, na které lze vidět že třída *bazova* zde má pouze sebe samu. Oproti tomu třída *odvozena2* má nejdříve sebe, poté třídu *odvozena*, od níž dědí a na konci se nachází třída *bazova*, která je předkem třídy *odvozena*.

```
class bazova
{
public:
    void fun_1() { printf("bazova-1\n"); }
    virtual void fun_2() { printf("bazova-2\n"); }
    virtual void fun_3() { printf("bazova-3\n"); }
};

class odvozena : public bazova
{
public:
    void fun_1() { printf("odvozena-1\n"); }
    void fun_2() { printf("odvozena-2\n"); }
};

class odvozena2 : public odvozena
{
};

int main()
{
    bazova *p;
    odvozena2 obj1;
    p = &obj1;

    p->fun_1();
    p->fun_2();
    p->fun_3();
}
```

Ukázka kódu 6.1: Volání virtuálních metod.

```
define dso_local i32 @main() #0 {
entry:
  %p = alloca %class.bazova*, align 4
  %obj1 = alloca %class.odvozena2, align 4
  call void @_ZN9odvozena2C2Ev(%class.odvozena2* %obj1) #4
  %0 = bitcast %class.odvozena2* %obj1 to %class.bazova*
  store %class.bazova* %0, %class.bazova** %p, align 4
  %1 = load %class.bazova*, %class.bazova** %p, align 4
  call void @_ZN6bazova5fun_1Ev(%class.bazova* %1)
  %2 = load %class.bazova*, %class.bazova** %p, align 4
  %3 = bitcast %class.bazova* %2 to void (%class.bazova*)***
  %vtable = load void (%class.bazova*)**, void (%class.bazova*)*** %3, ↩
      align 4
  %vfn = getelementptr inbounds void (%class.bazova*)*, void (%class.↩
      bazova*)** %vtable, i64 0
  %4 = load void (%class.bazova*)*, void (%class.bazova*)** %vfn, align 4
  call void %4(%class.bazova* %2)
  %5 = load %class.bazova*, %class.bazova** %p, align 4
  %6 = bitcast %class.bazova* %5 to void (%class.bazova*)***
  %vtable1 = load void (%class.bazova*)**, void (%class.bazova*)*** %6, ↩
      align 4
  %vfn2 = getelementptr inbounds void (%class.bazova*)*, void (%class.↩
      bazova*)** %vtable1, i64 1
  %7 = load void (%class.bazova*)*, void (%class.bazova*)** %vfn2, align ↩
      4
  call void %7(%class.bazova* %5)
  ret i32 0
}
```

Ukázka kódu 6.2: Volání virtuálních metod LLVM IR.

Po této hierarchii následují jednotlivé virtuální metody, které se již odkazují na jednotlivé indexy tabulek virtuálních metod. První třída *bazova* od nikoho nedědí, takže zápis odpovídá např. *bazova::fun_2*. V tabulce indexů má dvě virtuální metody, které obsahuje tj. *fun_2* a *fun_3*. Druhá třída *odvozena* má:

- Jednu metodu, kterou sama přepisuje a je tedy uvedena v její tabulce indexů (*fun_2*).
- Třída se také oproti třídě *bazova* rozšiřuje o virtuální metodu *fun_2*, ta se nachází v záznamech tabulky virtuálních metod i v záznamech indexů.
- Poslední virtuální metoda třídy je nepřepsaná metoda převzatá z třídy *bazova*, ta se nachází pouze v záznamech tabulky virtuálních metod jako *bazova::fun_3*.

Poslední třída *odvozena2* se odkazuje na implementace ve svých nadřazených třídách, tedy na *bazova::fun_3* a na *odvozena* ve zbylých případech.

Tyto informace jsou dále propagovány v překladači, kdy se ve třídě *CodeGenModule*, která slouží k převodu vstupního kódu v reprezentaci AST do vnitřní reprezentace překladače LLVM IR, udržují v proměnné *VTables*. Tato proměnná je instancí třídy *CodeGenVTables*, která obsahuje metody pro práci s tabulkami virtuálních metod v rámci LLVM.

Před generováním LLVM IR dojde k propagaci adresy tabulky virtuálních metod do konstruktoru dané třídy. Adresa virtuální tabulky je uložena v instanci třídy, kdy k jejímu uložení dojde při volání konstruktoru. O toto uložení se postará metoda *InitializeVTablePointers*, jež je součástí třídy *CodeGenFunction*. V rámci metod třídy *CodeGenFunction* se generuje konstruktor pro každou třídu vstupního programu obsahující virtuální metody, jež předchází konstruktoru třídy ze vstupního programu. Tento nově přidaný konstruktor provádí ukládání adresy tabulky virtuálních metod a obstarává volání konstruktoru původního.

Propagace tabulek do LLVM IR se provádí s využitím metody *emitVTableDefinitions* třídy *ItaniumCXXABI*. Tato třída reprezentuje v programu clang univerzální ABI pro jazyk C++. V metodě dojde k vytvoření globální konstanty, která bude reprezentovat tabulku virtuálních metod, data této konstanty jsou nastaveny s využitím metody *EmitVTableTypeMetadata*, která se nachází ve třídě *CodeGenModule*. Dále se v překladači s tabulkami pracuje jako s globální konstantou.

Stejně jako ostatní konstanty jsou tabulky uloženy do výsledného souboru v jazyce symbolických adres. Výsledné uložení tabulky virtuálních metod lze vidět na ukázce 6.4. Na ukázce znázorňující tabulku pro třídu *odvozena* lze vidět, že tabulka se nachází ve výsledném binárním kódu v sekci *rodata*(*read only data*) určené pro statická data. Tabulka obsahuje 5 záznamů, kdy na prvním místě se nachází ofset roven hodnotě 0, za ním následuje odkaz na tabulku obsahující hierarchii konstruktorů a zbývající záznamy odpovídají adresám jednotlivých virtuálních metod třídy. Mezi metodami třídy lze vidět, že pro první dvě metody se volá implementace třídy *odvozena* a pro třetí metodu se volá implementace ze třídy *bazova*.

```
Vtable for 'bazova' (4 entries).
   0 | offset_to_top (0)
   1 | bazova RTTI
       -- (bazova, 0) vtable address --
   2 | void bazova::fun_2()
   3 | void bazova::fun_3()

VTable indices for 'bazova' (2 entries).
   0 | void bazova::fun_2()
   1 | void bazova::fun_3()

Vtable for 'odvozena2' (5 entries).
   0 | offset_to_top (0)
   1 | odvozena2 RTTI
       -- (bazova, 0) vtable address --
       -- (odvozena, 0) vtable address --
       -- (odvozena2, 0) vtable address --
   2 | void odvozena::fun_2()
   3 | void bazova::fun_3()
   4 | void odvozena::fun_1()


Vtable for 'odvozena' (5 entries).
   0 | offset_to_top (0)
   1 | odvozena RTTI
       -- (bazova, 0) vtable address --
       -- (odvozena, 0) vtable address --
   2 | void odvozena::fun_2()
   3 | void bazova::fun_3()
   4 | void odvozena::fun_1()

VTable indices for 'odvozena' (2 entries).
   0 | void odvozena::fun_2()
   2 | void odvozena::fun_1()
```

Ukázka kódu 6.3: Tabulky virtuálních metod.

```
        .address_space 0
        .type $_ZTV8odvozena,@object // @_ZTV8odvozena
        .section "$.rodata._ZTV8odvozena","aG",@progbits,_ZTV8odvozena,↩
           comdat
        .weak $_ZTV8odvozena
        .p2align 2
$_ZTV8odvozena:
        .long 0
        .long ($_ZTI8odvozena)
        .long ($_ZN8odvozena5fun_2Ev)
        .long ($_ZN6bazova5fun_3Ev)
        .long ($_ZN8odvozena5fun_1Ev)
        .size $_ZTV8odvozena, 20
```

Ukázka kódu 6.4: Tabulky virtuálních metod v jazyce symbolických adres.

Při překladu z jazyka symbolických adres do objektového souboru programem assembler, jsou všechny záznamy, představující odkazy na symboly v objektovém souboru, tabulky uloženy v podobě relokace[1]. Konkrétní adresy symbolů výsledného binárního souboru jsou doplněny až ve fázi linkováni nástrojem linker.

[1]Relokace - symboly jsou uloženy do relokační tabulky, kdy po přiřazení konečné adresy symbolu ve fázi linkování dochází k aktualizaci všech odkazů na tento symbol. Aktualizace se provádí na základě dříve vytvořené relokační tabulky.

Kapitola 7

Implementace na architektuře Codasip uRISC

Následující kapitola se zabývá implementací podpory jazyka C++11 v rámci uživatelských souborů překladače na architektuře Codasip uRISC. V kapitole bude popsána implementace builtin funkcí *setjmp* a *longjmp*, které jsou nutné pro překlad knihovny *libunwind*. Dále se kapitola zabývá samotným překladem a konfigurací jednotlivých knihoven, následovaných implementací výjimek a jednoduchým otestováním implementace.

7.1 Seznámení s architekturou Codasip uRISC

Architektura Codasip uRISC slouží v rámci nástrojů Codasip především pro demonstrační a výukové účely. V základním provedení se jedná, jak již název napovídá o architekturu typu RISC s délkou slova 32-bitů. Instrukční sada architektury se skládá z 38 instrukcí, které lze rozdělit do těchto skupin:

- aritmetické operace,
- přesun konstanty do registru,
- nepodmíněný přesun mezi registry,
- podmíněný přesun mezi registry,
- instrukce typu *load* a *store*,
- instrukce pro řízení toku programu pracující s konstantou,
- instrukce pro řízení toku programu pracující s registry,
- podmíněné instrukce pro řízení toku programu,
- systémové volání,
- instrukce *nop* a *halt*.

7.1.1 Obecná úprava modelu

Zde bude uvedena obecná úprava modelu nutná pro oba implementované typy volání výjimek. Úpravy týkající se specifických volání výjimek se nachází v rámci samostatných sekcí, obsahujících popis implementace daných výjimek.

Úprava modelu spočívá v úpravě souboru *crt1.S*, jedná se o soubor obsahující tzv. startup code. Instrukce tohoto souboru jsou vykonány před samotným voláním funkce *main* vykonávaného programu. Dochází zde například k inicializaci ukazatele zásobníku nebo ukončení simulace po vykonání programu.

V jazyce C++ se provádí několik rutin, které předchází volání hlavního těla programu *main*. První z těchto rutin, která se již nachází v souboru *crt1.S*, je volání konstruktorů globálních instancí. To probíhá načtením adresy *___CTOR_LIST___*, obsahující adresy všech konstruktorů pro globální instance. Nad tímto listem adres se poté iteruje, dokud se nedorazí na konec, který je označen symbolem *___CTOR_END___*. V každé iteraci se provede načtení adresy funkce konstruktoru, volání této funkce a po jejím vykonání inkrementace ukazatele do listu adres.

Druhou rutinou je registrování globálních destruktorů s využitím funkce *atexit*. Průběh je obdobný jako v případě volání konstruktorů globálních instancí, kdy začátek listu označuje symbol *___DTOR_LIST___*, konec poté značí symbol *___DTOR_END___*. Rozdíl v iteraci nad listem konstruktorů a destruktorů spočívá v registru, do kterého se jednotlivé adresy načítají. V případě konstruktoru se jedná o registr *r6* a provede se volání *call r6*, tedy tok vykonávaného programu se přesune na adresu načtenou z listu konstruktorů. U destruktorů dochází pouze k jejich registraci tak, aby v případě normálního ukončení programu mohly být volány. Nejdříve dojde k načtení adresy do registru *r4*, jenž slouží k předávání parametrů funkce. Po načtení hodnoty následuje instrukce *call $atexit*. Výsledkem je tedy volání funkce *atexit* s adresou destruktoru předanou registrem *r4*.

Adresy začátku a konce globálních konstruktorů a destruktorů jsou do výsledného binárního souboru umístěny programem linker. Tento program rozpozná globální konstruktory a destruktory na základě jejich jména a umístí je do příslušného listu.

7.2 Implementace builtin funkcí

Pro překlad knihovny libunwind s podporou výjimek řešených pomocí *setjmp* a *longjmp* je nezbytně nutné nejdříve implementovat příslušné funkce, kterými jsou *___builtin_setjmp* a *___builtin_longjmp*. Podpora těchto funkcí bude implementována v rámci následujících zdrojových souborů:

- CodasipGenIselLowering - obsahuje třídu CodasipGenTargetLowering, která se stará o mapování operací LLVM IR na DAG,
- CodasipCustomPats - obsahuje uživatelem vytvořené vzory pro převod vstupního DAG, nezávislého na cílové architektuře, na DAG výstupní pro konkrétní architekturu.

Tyto soubory po prvotním vygenerování překladače jsou přesunuty z adresáře *work* mezi zdrojové soubory modelu tak, aby při následujícím překladu modelu došlo k nahrazení generovaných zdrojových kódů za zdrojové kódy upravené v rámci modelu.

```
define dso_local i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 4
  %ex_buf__ = alloca [32 x i32], align 4
  store i32 0, i32* %retval, align 4
  store i32 %argc, i32* %argc.addr, align 4
  store i8** %argv, i8*** %argv.addr, align 4
  %0 = bitcast [32 x i32]* %ex_buf__ to i8**
  %1 = call i8* @llvm.frameaddress(i32 0)
  store i8* %1, i8** %0, align 4
  %2 = call i8* @llvm.stacksave()
  %3 = getelementptr inbounds i8*, i8** %0, i32 2
  store i8* %2, i8** %3, align 4
  %4 = bitcast i8** %0 to i8*
  %5 = call i32 @llvm.eh.sjlj.setjmp(i8* %4)
  ret i32 0
}
```

Ukázka kódu 7.2: LLVM IR pro main funkci obsahující *__builtin_setjmp*.

7.2.1 __builtin_setjmp

Jak bylo popsáno v rámci 2.4.2 *__builtin_setjmp* slouží k uložení kontextu. Uložení se provede na adresu, která je builtin funkci předána pomocí vstupního parametru funkce. Pokud se funkce volá poprvé, její návratová hodnota po provedení uložení kontextu odpovídá hodnotě 0. V případě, že se tok programu na volání funkce dostane po volání *__builtin_longjmp*, její návratová hodnota odpovídá hodnotě, kterou předává builtin funkce *__builtin_longjmp*, kdy hodnota předaná funkcí *__builtin_longjmp*, musí být různá od hodnoty 0.

Prvním krokem v rámci implementace je vygenerování LLVM IR pro jednoduchý program, který se nachází na ukázce kódu 7.1. Na ukázce se nachází jednoduché použití funkce *__builtin_setjmp*, která při vykonání uloží kontext vykonávaného programu do struktury *jmp_buf*.

```
#include <setjmp.h>

int main(int argc, char** argv)
{
  jmp_buf ex_buf__;
  _builtin_setjmp((void**)&ex_buf_);
  return 0;
}
```

Ukázka kódu 7.1: Jednoduchá ukázka s *__builtin_setjmp*.

Tomuto zdrojovému kódu poté odpovídá zápis v LLVM IR, který lze vidět na ukázce 7.2.

Z LLVM IR lze vyčíst, že nás zajímá řádek obsahující volání *llvm.eh.sjlj.setjmp*. Implementací této funkce se dále zabývá následující sekce.

Prvním krokem je v rámci konstruktoru třídy *CodasipGenTargetLowering* zajistit, aby se operace reprezentující instrukci zpracovávala s využitím implementace specifické pro dané procesorové jádro, tedy byla nastavena na hodnotu *Custom*. Instrukce s tímto nastavením zpracování se dostanou do metody *LowerOperation*. Zde lze nastavit, aby se pro zpracovávanou instrukci *llvm.eh.sjlj.setjmp* vložil do DAG nový uzel s nově vytvořenou operací *CodasipISD::EH_SJLJ_SETJMP*. Tato operace se vytvoří ve zdrojovém souboru *CodasipInstrInfo.td*. Do stejného souboru je přidána i pseudoinstrukce *Codasip::EH_SJLJ_SETJMP* se vzorem pro zpracování tohoto uzlu. Tento vzor uvádí, že pokud se v rámci DAG nachází uzel s operací *CodasipISD::EH_SJLJ_SETJMP* dojde k jejímu nahrazení pseudoinstrukcí *Codasip::EH_SJLJ_SETJMP*, která má nastaveno *custom* zpracování. V případě *custom* zpracování instrukce, dojde k jejímu zpracování metodou *EmitInstrWithCustomInserter* třídy *CodasipGenTargetLowering*. Tato metoda se stará o instrukce, které mají nastaveno uživatelské zpracování. Třída *CodasipGenTargetLowering* slouží ke zpracování LLVM instrukcí pro danou cílovou architekturu.

Pokud se zpracovávaná pseudoinstrukce *Codasip::EH_SJLJ _SETJMP* dostane do metody *EmitInstrWithCustomInserter*, dojde k volání implementované metody *emitEHSjLjSetJmp*. Metoda *EmitInstrWithCustomInserter* se stará o nahrazení jednoduché instrukce ve vnitřní reprezentaci LLVM IR, nebo uživatelských pseudoinstrukcí posloupností instrukcí cílové architektury.

Metoda *emitEHSjLjSetJmp* slouží ke generování kódu pro uložení kontextu vykonávaného programu. Metoda má dva vstupní parametry, první parametr tvoří machine instrukce, druhý parametr udává machine basic block, ve kterém se tato instrukce nachází. Zde se vytvoří nový machine basic block, který bude obsahovat kód k uložení kontextu (*main*) a machine basic block s kódem pro ukončení dané pseudoinstrukce (*sink*). Závislost a výsledné uspořádání machine basic block se nachází na obrázku 7.1, kde na levé straně lze vidět původní stav a na pravé straně stav po nahrazení pseudoinstrukce.

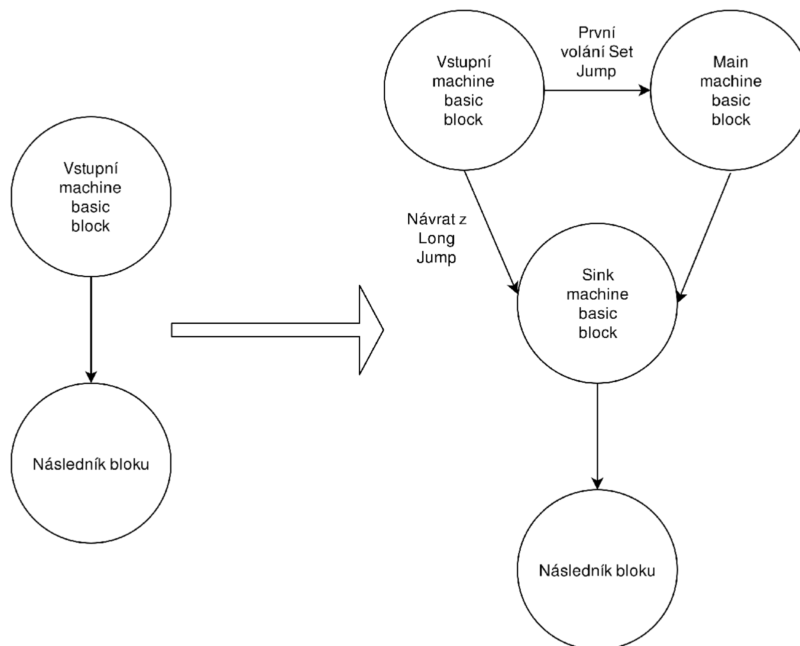
Dále se provede nastavení parametrů pro každý machine basic block. První basic block *main* se označí jako landing pad, neboť se jedná o landing pad pro zachytávání výjimek a také se jedná o cíl nepřímého skoku, což udává parametr *AddressTaken*. *Sink* machine basic block přebírá všechny následníky, které obsahuje machine basic block na vstupu, tedy machine basic block, v němž se nachází instrukce *Codasip::EH_SJLJ_SETJMP*. Dále se do *Sink* machine basic block přesunou všechny instrukce, které ve vstupním bloku následují po instrukci *Codasip::EH_SJLJ_SETJMP*. Machine basic block, obsahující instrukci *Codasip::EH_SJLJ_SETJMP*, dostane následníky, kterými jsou *main* a *sink* machine basic block. *Main* machine basic block má následníka, kterým je *sink* machine basic block.

Po nastavení parametrů může dojít k nahrazování instrukce. Tedy za instrukci *Codasip::EH_SJLJ_SETJMP* se nejprve vloží instrukce *call*[1]. Instrukce reprezentuje instrukci volání. Tato instrukce má jeden operand, kterým je absolutní hodnota adresy zapsána jako 24-bitová hodnota, tento operand je nastaven na *main* machine basic block. Instrukce také ukládá návratovou adresu do registru *r3*. Za touto instrukcí následuje instrukce pro načtení konstanty *movsi*[2], zde do cílového registru dojde k načtení konstanty hodnoty 1. Machine basic block ukončuje instrukce absolutního skoku *jump*[3], jejichž jediným operandem, stejně

[1]Přesněji se jedná o element *Codasip::i_jump_call_abs___opc_call_abs___abs_addr24___*, pro větší přehlednost textu budou elementy instrukcí uvedeny v rámci poznámky.

[2]Element *i_movsi_movhi___gpr_all___opc_movsi___simm19___*.

[3]Element *i_jump_call_abs___opc_jump_abs___abs_addr24___*.



Obrázek 7.1: Ukázka tvorby nového uspořádání machine basic block po nahrazení pseudoinstrukce.

jako u předchozího skoku, je adresa zapsána na 24-bitech, která je nastavena na *sink* machine basic block.

Dále následuje *main* machine basic block začínající instrukcí *store*[4]. Instrukce uloží obsah druhého vstupního registru na adresu tvořenou součtem obsahu prvního vstupního registru a 14-bitovou znaménkovou hodnotou, která reprezentuje druhý vstupní operand. První parametr je dán operandem vstupní instrukce *Codasip::EH_SJLJ_SETJMP*, hodnota konstanty odpovídá velikosti ukládané hodnoty. Ukládaná hodnota je registr *r3*, jenž byl nastaven voláním v předchozím machine basic blocku. Poslední instrukce provádí načtení konstanty o hodnotě 0 pomocí instrukce *movsi*.

Poslední machine basic block *sink* obsahuje instrukci *PHI*, která byla představena v sekci 3.2. Tato instrukce nastavuje návratovou hodnotu na základě toho, z jakého machine basic blocku se do bloku *sink* program dostane. Tedy vybere hodnotu 1 v případě, že předchůdce byl původní basic block a hodnotu 0, pokud byl předchůdce *main* basic block. V posledním kroku se odstraní vstupní instrukce *Codasip::EH_SJLJ_SETJMP*.

Výsledný kód poté odpovídá následující ukázce 7.3, která pro přehlednost zobrazuje pouze instrukce, které nahradily v LLVM IR instrukci *llvm.eh.sjlj.setjmp*.

Při vykonávání kódu se postupuje následovně. Při volání *___builtin_setjmp* se provede *call* instrukce, která přesune tok programu na *main* (*bb.1.entry*) machine basic block. V rámci tohoto bloku se provede uložení registru *Ra* (*rf_gpr_3*), který ukazuje na instrukci následující po instrukci *call*, na adresu danou vstupním parametrem builtin funkce. Dále se pokračuje na *sink* (*bb.2.entry*) machine basic block, který vybere a vrátí registr s hodnotou 0.

Při provedení funkce *___builtin_longjmp* se provede skok na instrukci pro načtení konstanty o hodnotě 1. Po načtení konstanty následuje skok na *sink* (*bb.2.entry*) machine basic

[4]Element *i_store___opc_st___gpr_all___gpr_all___simm14___*.

```
  i_jump_call_abs__opc_call_abs__abs_addr24__ %bb.1, <regmask>, implicit-↩
     def $rf_gpr_3
  %11:gpr_all = i_movsi_movhi__gpr_all__opc_movsi__simm19__ 1
  i_jump_call_abs__opc_jump_abs__abs_addr24__ %bb.2

bb.1.entry (address-taken, landing-pad):
; predecessors: %bb.0
  successors: %bb.2(0x80000000); %bb.2(200.00%)
  liveins: $rf_gpr_3
  i_store__opc_st__gpr_all__gpr_all__simm14__ $rf_gpr_3, %7:gpr_all, 4
  %12:gpr_all = i_movsi_movhi__gpr_all__opc_movsi__simm19__ 0

bb.2.entry:
; predecessors: %bb.0, %bb.1
  %8:gpr_all = PHI %12:gpr_all, %bb.1, %11:gpr_all, %bb.0
```

Ukázka kódu 7.3: Machine kód po nahrazení LLVM IR instrukce.

```c
#include <setjmp.h>

int main(int argc, char** argv)
{
  jmp_buf ex_buf__;
  __builtin_longjmp((void**)&ex_buf__, 1);
  return 0;
}
```

Ukázka kódu 7.4: Jednoduchá ukázka s *__builtin_longjmp*.

block a navrácení hodnoty z registru, který byl nastaven v předchozím machine basic blocku a má hodnotu 1. Hodnota je tedy různá od 0 a program ví, že se jedná o návrat z funkce *__builtin_longjmp*.

7.2.2 __builtin_longjmp

Builtin funkce *__builtin_longjmp* provádí obnovení kontextu z paměti, adresu kontextu určuje vstupní parametr. Ukázka kódu 7.4 znázorňuje jednoduchý program v jazyce C využívající funkci *__builtin_longjmp*, tomuto kódu poté odpovídá zápis v LLVM IR na ukázce 7.5. Na ukázce si lze všimnout, že druhý parametr funkce tvoří konstanta 1, která se dále nevyužívá. To je způsobeno přímo překladačem a jeho kontrolou v metodě *SemaBuiltinLongJmp* ve třídě *Sema*. Metoda provádí kontrolu, zda zápis využívá konstantu 1, která je jediná přípustná a dále provádí kontrolu podpory builtin funkce pro danou cílovou architekturu. Třída *Sema* slouží k sémantické analýze vstupního programu.

Obdobně jako v případě *__builtin_setjmp* je nutné nastavit *Custom* zpracování v rámci *CodasipGenTargetLowering*. Dále v *LowerOperation* bylo implementováno nahrazení funkce *llvm.eh.sjlj.longjmp* za operaci *CodasipISD::EH_SJLJ_LONGJMP*. Tato operace je přidána do souboru *CodasipIntstrInfo.td* spolu s pseudoinstrukcí, která bude sloužit k jejímu nahrazení *Codasip::EH_SJLJ_LONGJMP*. Zde se, obdobně jako v předchozím pří-

```
define dso_local i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 4
  %ex_buf__ = alloca [32 x i32], align 4
  store i32 0, i32* %retval, align 4
  store i32 %argc, i32* %argc.addr, align 4
  store i8** %argv, i8*** %argv.addr, align 4
  %0 = bitcast [32 x i32]* %ex_buf__ to i8**
  %1 = bitcast i8** %0 to i8*
  call void @llvm.eh.sjlj.longjmp(i8* %1)
  unreachable

return: ; No predecessors!
  %2 = load i32, i32* %retval, align 4
  ret i32 %2
}
```

Ukázka kódu 7.5: LLVM IR pro kód s *__builtin_longjmp*.

padě, využívá přidaná pseudoinstrukce *Codasip::EH_SJLJ_LONGJMP* k zachycení operace v rámci DAG a zpracování metodou *EmitInstrWithCustomInserter*.

Metoda *EmitInstrWithCustomInserter* pro danou instrukci zavolá metodu *emitEHSjLjLongJmp*, která je již značně jednodušší než metoda pro *__builtin_setjmp*.

V rámci implementace *__builtin_longjmp* je nutné pouze přečíst kontext. Toho se dosáhne s využitím instrukce *load*[5]. Instrukce má tři operandy: cílový registr, do kterého bude nahrána hodnota z paměti; zdrojový registr, který udává cílovou adresu a konstantu pro posuv v rámci adresy. Cílový registr se nastaví na nově vytvořený virtuální registr, zdrojový registr odpovídá operandu vstupní instrukce a konstanta odpovídá velikosti ukládané hodnoty. Po načtení adresy z paměti do virtuálního registru se zavolá instrukce *jump*, která reprezentuje skok na adresu danou vstupním registrem, ten je nastaven na registr vytvořený v předchozím kroku.

Obě představené instrukce se vkládají za instrukci *Codasip::EH_SJLJ_LONGJMP*, po přidání těchto instrukcí se instrukce *Codasip::EH_SJLJ_LONGJMP* může odstranit. Výsledný kód odpovídá kódu na ukázce 7.6.

7.3 Překlad knihoven

Pro překlad knihoven byl nejdříve implementován jednoduchý skript ve skriptovacím jazyce BASH. Skript má dvě konfigurace, první z nich je pro cíl Codasip uRISC, případně jiný Codasip procesor a druhá konfigurace se stará o překlad knihoven pro architekturu *x86_64*, která slouží jako referenční řešení. V rámci skriptu se definuje identifikátor *CODASIP_MACRO*. Tento identifikátor se nachází v každé z překládaných knihoven. Na základě tohoto identifikátoru se v rámci jednotlivých knihoven provede nastavení potřebných parametrů uvnitř konfiguračního souboru *CMakeList.txt*. Tyto parametry jsou následující:

[5]Element *i_load__gpr_all__opc_ld__gpr_all__simm14__*.

```
bb.0.entry:
 liveins: $rf_gpr_4, $rf_gpr_5
 %1:gpr_all = COPY $rf_gpr_5
 %0:gpr_all = COPY $rf_gpr_4
 %3:gpr_all = COPY %1:gpr_all
 %2:gpr_all = COPY %0:gpr_all
 %4:gpr_all = i_movsi_movhi__gpr_all__opc_movsi__simm19___immspec_1_
 i_store__opc_st__gpr_all__gpr_all__simm14__ killed %4:gpr_all, %stack↩
    .0.retval, 0 :: (store 4 into %ir.retval)
 i_store__opc_st__gpr_all__gpr_all__simm14__ %0:gpr_all, %stack.1.argc.↩
    addr, 0 :: (store 4 into %ir.argc.addr)
 i_store__opc_st__gpr_all__gpr_all__simm14__ %1:gpr_all, %stack.2.argv.↩
    addr, 0 :: (store 4 into %ir.argv.addr)
 %5:gpr_all = i_addi__gpr_all__opc_addi__gpr_all__simm14__ %stack.3.↩
    ex_buf__, 0
 %6:gpr_all = i_load__gpr_all__opc_ld__gpr_all__simm14__ %5:gpr_all, 4
 i_jump_call_gpr__opc_jump_gpr__gpr_all__ %6:gpr_al
```
Ukázka kódu 7.6: Výsledný kód po nahrazení instrukce.

- definuje se, že výsledný binární soubor je typu ELF,
- knihovny se překládají s nastavením *BAREMETAL*, označujícím absenci podpory operačního systému,
- knihovny nebudou využívat vlákna, jelikož procesorová jádra Codasip běží pouze na 1 vláknu a nepodporují víceprocesorová jádra,
- všechny knihovny budou mít statickou podobu, z důvodu absence podpory dynamických knihoven v rámci Codasip nástrojů,
- nastavení typu zachytávání výjimek - v rámci prvního kroku se jedná o Set Jump a Long Jump zachytávání výjimek,
- pro knihovnu libc++abi je nutné definovat, že nemá k dispozici zarovnanou alokaci paměti,
- pro knihovnu libc++ se nastaví, že se jedná o GNU zdrojový kód a informace o nedostupnosti monotonních hodin[6].

Skript pracuje s absolutními cestami k nástrojům: překladač jazyka C, překladač jazyka C++ a archivační nástroj. Tyto cesty slouží k nastavení proměnných pro program *CMake*. Pro každou knihovnu a cílovou architekturu se v rámci skriptu vytvoří adresář *build*, který obsahuje všechny soubory vytvořené během překladu.

Po spuštění samotného skriptu dojde k vygenerování statické knihovny libunwind.a, libc++abi.a a libc++.a.

[6]Monotonní hodiny - slouží k měření času od určité události. Nejsou ovlivněny systémovými hodinami.

7.4 Úprava zdrojových kódů nástrojů Codasip

Přestože se tato kapitola zabývá především implementací v rámci uživatelských souborů překladače, musí se i zde provést úpravy samotného překladače. Tyto úpravy se týkají metod a tříd, které jsou nezávislé na cílové architektuře. Kromě úpravy souborů překladače se zde nachází i popis úprav v rámci knihoven jazyka C++11 a jazyka C. V této sekci se nachází veškeré úpravy, které bylo nutné udělat pro plnou funkcionalitu knihovny na základě testování, které bude popsáno dále a chyb při překladu knihoven. Jednotlivé úpravy jsou rozděleny do podkapitol podle jednotlivých částí překladače a nástrojů Codasip.

7.4.1 Clang - přední část překladače

První úprava v rámci nástroje Clang se nachází ve třídě *CodasipTargetInfo*, která udržuje základní informace o procesorovém jádře, pro které probíhá překlad kódu. Mezi tyto informace patří například bitová šířka datových typů, informace o registrovém poli apod. Do této třídy se přidá přepis implementace metody *hasSjLjLowering*, který bude vracet hodnotu *true*, což označuje podporu Set Jump Long Jump builtin funkcí. Poslední úpravou v rámci přední části překladače je odstranění kódu pro již nepodporované rozšíření překladače v podobě libovolně dlouhého datového typu *int*, které není kompatibilní s knihovnami jazyka C++. Tato úprava v rámci metody *toCharUnitsFromBits* způsobovala špatné zarovnání paměti při ukládání hodnoty, kdy při jejím opětovném čtení došlo k přečtení nesprávné hodnoty a ukončení testované aplikace s chybovým hlášením.

7.4.2 LLC - zadní část překladače

V rámci nástroje llc první úprava spočívá v přidání podpory jump table index[7] do metody *getName* ve třídě *BaseJumpInfo*. Metoda vrací jméno cíle skoku, který nepodporuje cíle zadané pomocí jumping table index. Pro přidání podpory postačí úprava podmínky ve funkci *assert*, která bude symboly typy jump table index brát jako podporované. Tyto symboly jsou důležité pro zachytávání výjimek typu Set Jump Long Jump, jejichž implementaci se bude věnovat následující podkapitola.

Druhá úprava se nachází v metodě *analyzeBranch* ve třídě *CodasipBaseInstrInfo*. Zde je nutné přidat podporu nepodmíněného skoku, jehož cílem není machine basic block, ale v implementovaném případě symbol *abort*. Tento skok se využívá v implementaci metody *emitEHSjLjDispatchBlock* u výjimek typu Set Jump Long Jump, kdy se v případě dosáhnutí trap machine basic block volá právě funkce *abort*. Pro takový skok metoda po úpravě vrací hodnotu *BT_None*. Tato hodnota označuje, že nelze analyzovat návěští a tedy návěští musí být analyzováno, respektive zpracováno dále v kódu LLVM.

Následná úprava se týká metody *tryEraseEmptyBB*, která je součástí SuperBlock Scheduleru. Tato metoda se snaží smazat prázdné nebo nevyužité machine basic block. Zde se nachází funkce *assert*, která opět nepočítá s nepodmíněným skokem, jehož parametrem není machine basic block. Tuto kontrolu nahrazuje podmínka, která v případě, že se o machine basic block nejedná, vrací hodnotu false. Tato hodnota udává, že daný machine basic block nemůže být smazán.

Poslední úprava v nástroji llc se týká zpracování tabulek výjimek v metodě *emitExceptionTable* třídy *EHStreamer*. Třída se stará o tisk informací spojených se zachytáváním výjimek do souboru v jazyce symbolických adres, metoda *emitExceptionTable* obstarává

[7]Jump Table Index - slouží jako ukazatel na konkrétní záznam v rámci tabulky skoků.

tisk tabulek pro zachytávání výjimek. Zde se nachází kód, který pracuje s nepodporovanými direktivami jazyka symbolických adres. Tyto direktivy jsou ULEB128[8], s jejichž pomocí dochází k zápisu velikosti vzdálenosti mezi dvěma symboly. Prvotní úprava spočívá v nahrazení nepodporovaného zakódování ULEB128 za zapsání nekódované hodnoty na 4-bytech. Úprava se také musela přenést do knihovny libunwind, kde se dekodér hodnoty ULEB128 nahrazuje za obyčejné načtení hodnoty po bytech. Toto řešení bylo zvoleno, jako dočasné před implementací plné podpory direktivy ULEB128. V rámci implementace se jednalo o dočasné řešení, které bylo po přípravě metod nahrazeno podporou direktiv typu LEB128. Přidání podpory spočívalo v implementaci kódování LEB v přichystané metodě *TryEval* třídy *ExprConst*. Metoda se snaží zpracovat známé funkce, pro zpracování kódování typu LEB128 se zde nachází funkce *CODASIP_SLEB128* a *CODASIP_ULEB128*. Implementace spočívala v přidání kódování typu LEB pro tyto funkce. Zakódovaná hodnota poté odpovídá hodnotě návratové.

7.4.3 Generátor překladače

První úprava spočívá v úpravě generovaní implicitních argumentů pro překladač. Generování implicitních argumentů pro překladač se odehrává ve skriptu *backendgen.py*. Tento skript, napsaný v programovacím jazyce Python, se stará o generování samotného překladače pro daný model. Pro dosažení požadovaného výsledku je nutné odstranit z příkazu pro generování výchozích parametrů překladače jazyka C++ v rámci metody *generate_llc_related_scripts* tyto parametry:

- *-fno-exceptions*, který zakazuje zachytávání výjimek a
- *-fno-rtti*, který znemožňuje použití RTTI.

Dále odstranit chybu v rámci metody *init* třídy *Compiler*. Tato třída se stará o volání překladače v rámci Codasip frameworku, například při spuštění testů. Chybný kód vedl na volání překladače jazyka C místo překladače jazyka C++.

7.4.4 Simulátor procesorového jádra

Úprava zdrojových kódů pro generování simulátoru daného procesorového jádra spočívá v upravě příznaků otevřeného souboru v metodě *TranslateFileOpenFlags* třídy *Syscalls*. Tato třída realizuje systémová volání simulátoru procesorových jader. Úprava přidává podporu příznaku *O_EXCL*, který slouží ke kontrole existence souboru, aby nedošlo k jeho přepsání.

7.4.5 Standardní knihovna jazyka C

Další úpravy se týkají knihovny Newlib, která realizuje implementaci standardní knihovny jazyka C. V následujících krocích se provádí úpravy knihovny tak, aby odpovídala očekávání knihoven jazyka C++, které využívají funkce standardní knihovny jazyka C. Za první úpravu knihovny lze označit její samotnou konfiguraci, která se rozšíří o podporu formátu C99. Toto rozšíření přidává do knihovny například podporu tisku čísel s plovoucí desetinnou čárkou v hexadecimálním formátu ("%a"). Další úprava v rámci konfigurace knihovny spočívá v *CMakeList.txt*, který se rozšíří o následující zdrojové kódy:

[8]ULEB128 - jedná se o direktivu, která ukládá hodnoty typu integer na menším počtu bitů. Zjednodušeně se jedná o kódování konstanty na menší počet bitů, kdy formát je little endian se základem 128b.

- *logbl* - funkce vrací exponent vstupního parametru ve formátu *long dounle*,
- *nexttoward* - vrací první reprezentovatelnou hodnotu následující vstupní parametr,
- *nexttowardf* - vstupní parametr je typu *float* a
- *nexttowardl* - vstupní parametr je typu *long double*.

Poslední úprava v rámci konfigurace standardní knihovny jazyka C spočívá v generování makra *_LDBL_EQ_DOUBLE*. Toto makro se generuje v případě, že cílová architektura má stejnou bitovou délku datových typů *long double* a *double*. Na základě tohoto makra dochází ke generování některých podmíněných funkcí uvnitř standardní knihovny jazyka C Newlib. Makro lze přidat do konfiguračního souboru knihovny *newlib.h*, který se následně využívá při volání překladače.

První z úprav, zasahujících do zdrojových kódů knihovny, je oprava implementace funkce *_user_strerror*, která v základní implementaci vrací návratovou hodnotu rovnou konstantě 0. Po úpravě se vrací hodnota *"Unknown Error"* datového typu *char **. Tato hodnota se očekává při špatně zvolené chybové hlášce generické kategorie error.

Druhá úprava zdrojových kódů spočívá v opravě funkce *__sigtramp_r*, nacházející se ve zdrojovém kódu souboru *signal.c*. Zde implementace obsahuje navíc nastavení ukazatele funkce na hodnotu *SIG_DFL* v případě, že vstupní ukazatel ukazuje na neznámou hodnotu.

Poslední úprava spočívá ve vytvoření nové tabulky *_CTYPE_DATA* pro jazyk C++. Tato úprava poté vede i na opravu knihovny libc++, jejíž implementace *ctype* nerozlišuje jednotlivé typy u znaků. Přesněji všechny znaky jsou přepsány hodnotou *PRINT*, která nastavuje všechny bity jednotlivých znaků. Na straně knihovny *newlib* spočívá úprava ve vytvoření nového pole pro jazyk C++, které bude obsahovat definice jednotlivých ASCII znaků tak, jak jsou očekávány na straně libc++. Pro každý ASCII znak se nastaví požadované vlastnosti. Vlastnosti jednotlivých znaků jsou přebrány z testu na tuto tabulku, tento test se nachází v rámci testovací sady knihovny libc++. Mezi výše zmíněné vlastnosti patří: malé písmeno, velké písmeno, číslice, číslice v hexadecimálním tvaru, mezera, kontrolní znak, tisknutelný znak, prázdný symbol a interpunkční znaménko. Na straně knihovny libc++ se pouze rozšíří zdrojový soubor *__locale* o podmínku, kdy v případě definice symbolu *__codasip__*, tedy pokud překladač byl generovaný nástroji Codasip, dojde k využití rozšiřující implementace z knihovny *newlib*.

Dále se zde nachází několik minoritních úprav, spočívajících v nastavení proměnné *errno* pro některé funkce, například při chybném vstupním souboru apod. Některé z testů knihoven kontrolují kromě návratových kódů i proměnnou *errno*, kdy knihovna Newlib má od knihovny libc menší četnost využívání této proměnné. Chybějící implementace se týká především chybně zadaných nebo chybějících vstupů. Implementace byla doplněna na základě knihovny libc tak, aby odpovídala požadavkům knihovny libc++.

7.5 Zachytávání výjimek pomocí Set Jump a Long Jump

Po dokončení předchozích kroků lze přeložit knihovny libunwind, libc++abi a libc++. Tyto knihovny umožňují využívat funkcionality jazyka C++, nicméně pro plnohodnotné využití jazyka zbývá implementace zachytávání výjimek. Na ukázce kódu 7.7 lze vidět příklad vyvolání jednoduché výjimky typu *int*, tato výjimka bude zachycena v rámci bloku *catch*. Návratová hodnota funkce na ukázce bude při správném provedení rovna 0, v opačném případě bude hodnota odpovídat hodnotě -1.

```
int eh()
{
  try
  {
    throw 0;
  }
  catch(int e)
  {
    return e;
  }
  return -1;
}
```

Ukázka kódu 7.7: Vyvolání výjimky.

Pro tento jednoduchý program jsou implementačně důležité instrukce, které jsou vygenerovány v rámci LLVM průchodu s názvem *Exception handling preparation*. Tento průchod, na základě nastaveného typu výjimek, generuje pseudoinstrukce k dalšímu zpracování v rámci překladače. Pro přehlednost se v následující ukázce 7.8 nachází pouze část LLVM IR obsahující vyvolání výjimky. Po výjimce následuje část, obsahující zpracování výjimky, která bude uvedena dále na ukázce 7.9.

Z hlediska implementace jsou nejdůležitější v LLVM IR funkce, které nejsou prozatím podporovány cílovou architekturou. Mezi tyto funkce patří:

- *llvm.eh.sjlj.lsda()* - vrací adresu LSDA[9] pro danou funkci,
- *llvm.eh.sjlj.setup.dispatch()* - tato funkce se nahrazuje za posloupnost instrukcí, které se postarají o přípravu k vyvolání výjimky.

Další funkce jsou již implementovány v samotném překladači, případně v rámci knihoven jako například funkce *___cxa_allocate_exception*, která se nachází v rámci knihovny libc++abi.

V kódu 7.9 lze vidět zpracovávání zachycené výjimky. V rámci landing padu (*lpad*) dojde k uložení výjimky do struktury *ehselector*, následně se v rámci *catch.dispatch* porovná s existujícími bloky typu *catch*. Pokud existuje příslušný blok *catch*, dojde ke skoku na něj, v opačném případě dojde ke skoku na blok *eh.resume*, v němž dojde k ukončení programu s nezpracovanou výjimkou. Blok *catch* obsahuje funkce knihovny libc++abi *___cxa_begin_catch* a *___cxa_end_catch*, jež slouží ke zpracovaní zachycené výjimky.

7.5.1 Implementace llvm.eh.sjlj.lsda()

Jak bylo zmíněno v předchozí části, instrukce vrací adresu příslušného landing padu. Instrukce se zachytí nejdříve v metodě *LowerIntrinsic*, která slouží k zachycení LLVM funkcí, jež nebyly zachyceny v rámci metody *LowerOperation*. Obě metody se nacházejí ve třídě *CodasipTargetLowering*, která slouží ke zpracování LLVM IR instrukcí pro cílovou architekturu, jak bylo zmíněno výše. Pokud se v rámci zpracování nalezne instrukce, jejíž operační

[9]Language Specific Data Area (LSDA) - obsahuje informace pro personální rutinu. Mezi tyto informace patří například, která přerušení mohou být zachycena danou funkcí.

```
%fn_context = alloca { i8*, i32, [4 x i32], i8*, i8*, [5 x i8*] }, ↩
    align 4
%retval = alloca i32, align 4
%exn.slot = alloca i8*
%ehselector.slot = alloca i32
%e = alloca i32, align 4
%exception = call i8* @__cxa_allocate_exception(i32 4) #2
%0 = bitcast i8* %exception to i32*
store i32 0, i32* %0, align 16
%pers_fn_gep = getelementptr { i8*, i32, [4 x i32], i8*, i8*, [5 x i8*]↩
     }, { i8*, i32, [4 x i32], i8*, i8*, [5 x i8*] }* %fn_context, i32 ↩
    0, i32 3
store volatile i8* bitcast (i32 (...)* @__gxx_personality_sj0 to i8*), ↩
    i8** %pers_fn_gep
%lsda_addr = call i8* @llvm.eh.sjlj.lsda()
%lsda_gep = getelementptr { i8*, i32, [4 x i32], i8*, i8*, [5 x i8*] },↩
     { i8*, i32, [4 x i32], i8*, i8*, [5 x i8*] }* %fn_context, i32 0, ↩
    i32 4
store volatile i8* %lsda_addr, i8** %lsda_gep
%jbuf_gep = getelementptr { i8*, i32, [4 x i32], i8*, i8*, [5 x i8*] },↩
     { i8*, i32, [4 x i32], i8*, i8*, [5 x i8*] }* %fn_context, i32 0, ↩
    i32 5
%jbuf_fp_gep = getelementptr [5 x i8*], [5 x i8*]* %jbuf_gep, i32 0, ↩
    i32 0
%fp = call i8* @llvm.frameaddress(i32 0)
store volatile i8* %fp, i8** %jbuf_fp_gep
%jbuf_sp_gep = getelementptr [5 x i8*], [5 x i8*]* %jbuf_gep, i32 0, ↩
    i32 2
%sp = call i8* @llvm.stacksave()
store volatile i8* %sp, i8** %jbuf_sp_gep
call void @llvm.eh.sjlj.setup.dispatch()
%1 = bitcast { i8*, i32, [4 x i32], i8*, i8*, [5 x i8*] }* %fn_context ↩
    to i8*
call void @llvm.eh.sjlj.functioncontext(i8* %1)
%call_site = getelementptr { i8*, i32, [4 x i32], i8*, i8*, [5 x i8*] ↩
    }, { i8*, i32, [4 x i32], i8*, i8*, [5 x i8*] }* %fn_context, i32 ↩
    0, i32 1
store volatile i32 1, i32* %call_site
call void @llvm.eh.sjlj.callsite(i32 1)
call void @_Unwind_SjLj_Register({ i8*, i32, [4 x i32], i8*, i8*, [5 x ↩
    i8*] }* %fn_context) #2
invoke void @__cxa_throw(i8* %exception, i8* bitcast (i8** @_ZTIi to i8↩
    *), i8* null) #3
        to label %unreachable unwind label %lpad
```

Ukázka kódu 7.8: Zjednodušený LLVM IR pro vyvolání výjimky.

```
lpad: ; preds = %entry
  %2 = landingpad { i8*, i32 }
          catch i8* bitcast (i8** @_ZTIi to i8*)
  %__data = getelementptr { i8*, i32, [4 x i32], i8*, i8*, [5 x i8*] }, {↩
       i8*, i32, [4 x i32], i8*, i8*, [5 x i8*] }* %fn_context, i32 0, ↩
      i32 2
  %exception_gep = getelementptr [4 x i32], [4 x i32]* %__data, i32 0, ↩
      i32 0
  %exn_val = load volatile i32, i32* %exception_gep
  %3 = inttoptr i32 %exn_val to i8*
  %exn_selector_gep = getelementptr [4 x i32], [4 x i32]* %__data, i32 0,↩
       i32 1
  %exn_selector_val = load volatile i32, i32* %exn_selector_gep
  store i8* %3, i8** %exn.slot, align 4
  store i32 %exn_selector_val, i32* %ehselector.slot, align 4
  br label %catch.dispatch

catch.dispatch: ; preds = %lpad
  %sel = load i32, i32* %ehselector.slot, align 4
  %4 = call i32 @llvm.eh.typeid.for(i8* bitcast (i8** @_ZTIi to i8*)) #2
  %matches = icmp eq i32 %sel, %4
  br i1 %matches, label %catch, label %eh.resume

catch: ; preds = %catch.dispatch
  %exn = load i8*, i8** %exn.slot, align 4
  %5 = call i8* @__cxa_begin_catch(i8* %exn) #2
  %6 = bitcast i8* %5 to i32*
  %7 = load i32, i32* %6, align 4
  store i32 %7, i32* %e, align 4
  %8 = load i32, i32* %e, align 4
  store i32 %8, i32* %retval, align 4
  call void @__cxa_end_catch() #2
  br label %return

return: ; preds = %catch
  %9 = load i32, i32* %retval, align 4
  call void @_Unwind_SjLj_Unregister({ i8*, i32, [4 x i32], i8*, i8*, [5 ↩
      x i8*] }* %fn_context)
  ret i32 %9
```

Ukázka kódu 7.9: Zachycení výjimky.

kód odpovídá *INTRINSIC_WO_CHAIN*[10], pak lze zkontrolovat, zda se nejedná o interní funkci překladače *eh_sjlj_lsda*. Informace o přesném typu instrukce se nachází v jejím prvním operandu. V případě kladného porovnání dojde v DAG k nahrazení uzlu s funkcí *eh_sjlj_lsda* za uzel obsahující instrukci *CodasipISD::GAWrap*, parametr instrukce tvoří symbol *GCC_except_table*, ke kterému se připojí sufix reprezentující číslo funkce, kterou DAG reprezentuje. Číslo funkce představuje unikátní identifikátor funkce, který má přiřazena každá machine funkce. Symbol označuje návěští pro příslušný landing pad dané funkce. Využití *GAWrap* slouží k zachycení návěští instrukcí, která jej má jako vstupní parametr.

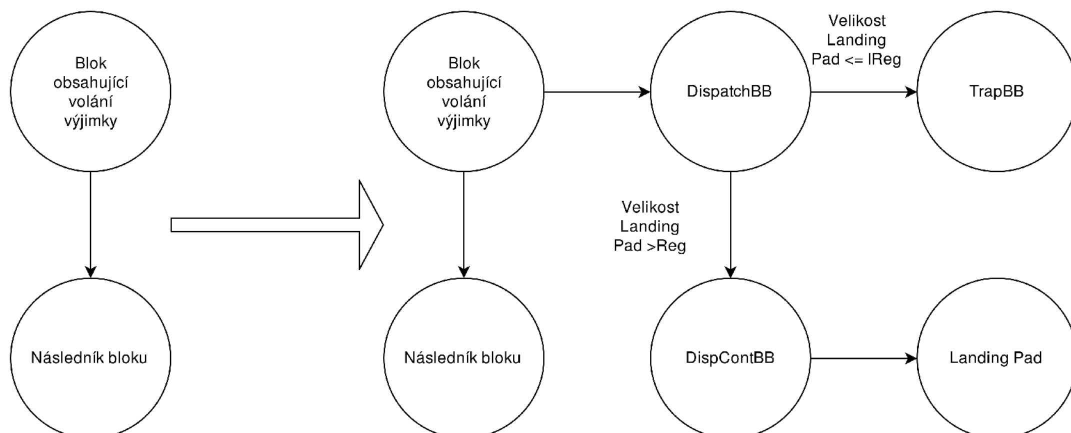
7.5.2 Implementace llvm.eh.sjlj.setup.dispatch()

Obdobně jako u předchozích implementací *__builtin_setjmp* a *__builtin_longjmp*, prvním krokem bude zachycení instrukce v rámci metody *EmitIntWithCustomInserter* a volání příslušné implementace instrukce, která se nachází v metodě *emitEHSjLjDispatchBlock*.

Na počátku metody *emitEHSjLjDispatchBlock* dojde k vytvoření tabulky skoků. Tabulka se vytvoří na základě vektoru z indexů symbolů skoku, které se nachází jako první parametr pseudoinstrukce *EH_LABEL* a machine basic block, ve kterém se tato pseudoinstrukce nachází. Po vytvoření tabulky dojde k vytvoření následujících bloků v daném pořadí:

- dispatch - blok, realizující landing pad pro zachytávání výjimek,
- dispCont - blok, který provádí skok na příslušnou tabulku pro zpracování výjimek,
- trap - blok, k jehož vykonání by program neměl dospět, pokud se program dostane do tohoto bodu, dojde k volání funkce *abort* a ukončení programu.

Přidání bloků lze vidět na obrázku 7.2. Na obrázku lze vidět nové bloky, kdy landing pad není nově vytvořený blok, ale ukazatel na již existující landing pads, které se nacházejí v rámci programu. Tyto landing pads po zpracování *llvm.eh.sjlj.setup.dispatch()* mají pouze jednoho předchůdce, kterým je dispCont machine basic block.



Obrázek 7.2: Ukázka tvorby nového uspořádání machine basic block po nahrazení pseudoinstrukce.

[10] *INTRINSIC_WO_CHAIN* - tento operační kód říká, že se jedná o LLVM instrukci bez vedlejšího efektu, která vrací pouze hodnotu.

Nejprve se provede uložení adresy dispatch basic block v rámci machine basic block, obsahujícího vstupní instrukci *Codasip::EH_SJLJ_SETUP_DISPATCH*. K tomu se využije instrukce *e_movi32___*[11], která načte danou adresu. Po načtení adresy dojde k jejímu uložení do struktury *SjLj_Function_Context* s pomocí instrukce *store*.

V rámci dispatch basic block se nejprve provede načtení registru, ten dále slouží jako index do paměti. Registr se nachází ve struktuře *SjLj_Function_Context*, ze které se hodnota načte s pomocí instrukce *load*. Následně se provede načtení velikosti landing pad listu. Na velikosti landing pad listu závisí další kroky, kdy se nejprve využije instrukce *cmp*[12], která porovná velikost získanou v předchozím kroku s hodnotou rovnou indexu. Pokud je hodnota velikosti landing pad listu menší nebo rovna, nastaví se příznak do cílového registru. Na základě příznaku se provede skok pomocí instrukce podmíněného skoku *jumpnz*[13], který provede skok v případě, že vstupní registr není roven hodnotě 0 na adresu zadanou 14-bitovou konstantou. V tomto případě vstupním registrem je registr s uloženým příznakem z předchozí instrukce a cílová adresa odpovídá adrese trap machine basic block. Tok programu se přesune, v případě že nedojde ke skoku na trap basic block, do dispCont basic block.

V prvním kroku dojde k načtení adresy jumping table s využitím instrukce *e_movi32___*. Následuje výpočet hodnoty offsetu pro výpočet adresy v rámci jumping table. Hodnota offsetu odpovídá dvojnásobku (u 32-bitové architektury), nebo čtyřnásobku (u 64-bitové architektury) indexu do paměti. K násobení se využije instrukce *mul*[14], instrukce má tři registry, kdy dva jsou vstupní operandy instrukce a třetí označuje registr cílový. Výsledek násobení se s pomocí instrukce *add* přičte k adrese jumping table a získáme adresu, ze které je načten, s využitím instrukce *load*, cíl skoku.

Před provedením skoku na návěští musí být přečten typ jumping table, kdy se rozlišují dva typy tabulek. První typ uchovává přímé adresy, v takovém případě dojde pouze ke skoku na danou adresu s pomocí *jump* instrukce. Toho se využívá u kódu, který není pozičně nezávislý. Druhý typ tabulek obsahuje relativní adresy, ke kterým musí být přičtena samotná adresa jumping table s využitím instrukce *add*, skok je proveden až na samotný výsledek sčítaní. Na závěr se instrukce *Codasip::EH_SJLJ_SETUP_DISPATCH* odstraní z machine basic block.

Výsledný kód lze vidět na machine kódu v ukázce 7.10, v té se pro přehlednost nachází pouze nahrazení funkce *llvm.eh.sjlj.setup.dispatch()*. Na ukázce lze vidět, že první blok odpovídá popisu bloku dispatch (*bb.7.entry*), který slouží jako landing pad. Označení landing pad se nachází vedle názvu machine basic block. První instrukce machine basic block provádí načítání dvou hodnot (indexu do paměti a velikosti landing padu), jejich porovnání a podmíněný skok. Podmíněný skok vede na trap (*bb.9*) basic block, který obsahuje pouze *call* instrukci na funkci *abort*. V případě, že nedojde ke skoku se tok programu přesune do druhého basic block, který odpovídá bloku dispCont (*bb.8.entry*). Zde lze vidět načtení adresy jumping table (*jump-table.0*). Za načtením adresy následuje výpočet hodnoty offset v podobě násobení. V dalším kroku dojde k přičtení výsledku násobení k hodnotě adresy jumping table, jedná se tedy o druhý druh tabulky skoku, kdy jsou adresy zadány relativně. Na závěr se provede načtení cíle skoku a skok samotný.

Výsledný kód v jazyce symbolických adres pro výjimky typu Set Jump Long Jump lze vidět na ukázce kódu, nacházející se v příloze práce, A.1.

---

[11] Ve skutečnosti se jedná o dvě instrukce, které načtou 32-bitovou konstantu.

[12] Element *i_cmp___gpr_all___opc_sle___gpr_all___gpr_all___*.

[13] Element *i_jump_cond___opc_jumpnz___gpr_all___rel_addr14___*.

[14] Element *i_alu___gpr_all___opc_mul___gpr_all___gpr_all___*.

```
bb.7.entry (landing-pad):
; predecessors: %bb.0
  successors: %bb.9(0x40000000), %bb.8(0x40000000); %bb.9(200.00%), %bb↩
      .8(200.00%)

  %34:gpr_all = i_load__gpr_all__opc_ld__gpr_all__simm14__ %stack.0.↩
      fn_context, 4
  %44:gpr_all = ↩
      i_movsi_movhi__gpr_all__opc_movsi__simm19__SYNTAX_CLONE_e_movi32__3_↩
       1
  %36:gpr_all = ↩
      i_movsi_movhi__gpr_all__opc_movhi__simm19__SYNTAX_CLONE_e_movi32__4_↩
       %44:gpr_all, 1
  %37:gpr_all = i_cmp__gpr_all__opc_sle__gpr_all__gpr_all__ %36:gpr_all, ↩
      %34:gpr_all
  i_jump_cond__opc_jumpnz__gpr_all__rel_addr14__ %37:gpr_all, %bb.9

bb.8.entry:
; predecessors: %bb.7
  successors: %bb.1(0x80000000); %bb.1(200.00%)

  %45:gpr_all = ↩
      i_movsi_movhi__gpr_all__opc_movsi__simm19__SYNTAX_CLONE_e_movi32__3_↩
       %jump-table.0
  %38:gpr_all = ↩
      i_movsi_movhi__gpr_all__opc_movhi__simm19__SYNTAX_CLONE_e_movi32__4_↩
       %45:gpr_all, %jump-table.0
  %46:gpr_all = ↩
      i_movsi_movhi__gpr_all__opc_movsi__simm19__SYNTAX_CLONE_e_movi32__3_↩
       2
  %42:gpr_all = ↩
      i_movsi_movhi__gpr_all__opc_movhi__simm19__SYNTAX_CLONE_e_movi32__4_↩
       %46:gpr_all, 2
  %41:gpr_all = i_alu__gpr_all__opc_sll__gpr_all__gpr_all__ %34:gpr_all, ↩
      %42:gpr_all
  %39:gpr_all = i_alu__gpr_all__opc_add__gpr_all__gpr_all__ %38:gpr_all, ↩
      %41:gpr_all
  %40:gpr_all = i_load__gpr_all__opc_ld__gpr_all__simm14__ %39:gpr_all, 0
  i_jump_call_gpr__opc_jump_gpr__gpr_all__ %40:gpr_all

bb.9.entry:
; predecessors: %bb.7

  i_jump_call_abs__opc_jump_abs__abs_addr24__ &abort
```
Ukázka kódu 7.10: Set Jump Long Jump zachytávání výjimek, machine code pro Setup Dispatch.

7.6 Zachytávání výjimek pomocí DWARF

Řešení zachytávání výjimek pomocí ladících informací ve formátu DWARF vyžaduje především úpravu v nástrojích Codasip. Přesněji nástroji assembler, který slouží k překladu souborů v jazyce symbolických adres do objektových souborů ve formátu ELF[15]. V rámci překladu ze zdrojového jazyka C++ do jazyka symbolických adres jsou generovány ladící informace obsahující direktivy CFI (*Call frame information*). Tyto direktivy následně slouží k vytvoření rámců pro debugger[16] nebo rámců pro zpracování zachytávání výjimek. CFI direktivy udávají informace o zásobníku, jako jsou posuv ukazatele na zásobník a uložení důležitých registrů v rámci zásobníku.

Důležitými registry jsou v rámci implementace myšleny především registr ukazatele zásobníku, registr s návratovou hodnotou (Ra) a callee-saved registry. Pokud uvažujeme implementaci v rámci procesorového jádra Codasip uRISC, pak se ukazatel na zásobník udržuje v registru *R0*, návratová hodnota se ukládá do registru *R3*, calle-saved registr má architektura pouze jeden a to registr *R1*.

Ukázka kódu 7.11 v jazyce symbolických adres znázorňuje prolog funkce *eh* z ukázky 7.7. V rámci této ukázky si lze všimnout využití CFI direktiv, kdy jsou v ukázce využity direktivy:

- *.cfi_startproc* - označení začátku funkce, která by měla mít záznam v příslušném rámci (buď pro zachytávání výjimek nebo pro ladící program), funkce poté končí odpovídajícím *.cfi_endproc*, jež označuje její konec,
- *.cfi_personality* - určuje osobní rutinu pro danou funkci a její kódování (první parametr),
- *.cfi_lsda* - označuje landing pad a jeho kódování,
- *.cfi_return_column* - říká, ve kterém registru je uložena návratová hodnota,
- *.cfi_def_cfa* - říká, že hodnota CFA[17] se vypočítá jako součet registru R0 (první parametr) a offsetu, jenž je dán druhým parametrem,
- *.cfi_adjust_cfa_offset* - označuje posun zásobníku,
- *.cfi_offset* - udává lokaci uložení registru R3.

Celý kód v jazyce symbolických adres lze nalézt v rámci přílohy práce na ukázce B.1. Při porovnání této ukázky kódu s kódem A.1 pro výjimky typu Set Jump Long Jump si lze všimnout, že výjimky využívající ladící informace mají mnohem menší velikost kódu než výjimky typu Set Jump Long Jump. To je způsobeno tím, že výjimky využívající DWARF ladících informací mají menší režii pokud nedojde k výjimce, neboť v takovém případě nevytváří rámce volání. Ty jsou vytvářeny až v případě vyvolání výjimky v rámci knihovny libunwind na základě informací obsažených v sekci *.eh_frame*, což bude představeno dále.

CFI direktivy jsou zpracovány programem assembler, výsledkem tohoto zpracování je vytvoření rámce pro obsluhu zpracovávání výjimek *.eh_frame*. Tento rámec se skládá z posloupnosti CFI záznamů. Každý CFI záznam obsahuje CIE (Common Information Entry)

[15] ELF - formát pro uložení spustitelných souborů, linkovatelných objektů, dynamických knihoven a ladících výpisů.

[16] Debugger - nástroj pro ladění chyb v programu, umožňující například procházení programu po instrukcích apod.

[17] Canonical Frame Address (CFA) - udává hodnotu ukazatele zásobníku v předchozím rámci.

```
$_Z2ehv:
$func_begin0:
  .cfi_startproc
  .cfi_personality 0, $__gxx_personality_v0
  .cfi_lsda 0, $exception0
  .cfi_return_column 3
  .cfi_def_cfa 0, 0
  r0 = addi r0, -32
  .cfi_adjust_cfa_offset 32
  st r3, [ r0 + 28 ]
  .cfi_offset 3, -4
```

Ukázka kódu 7.11: Výstup backendu.

záznam a k němu jeden a více příslušných FDE (Frame Description Entry) záznamů. Každý z těchto záznamů má pevně daný formát, který je nutné dodržet, aby došlo ke správnému zpracování rámců jednak v rámci nástroje linker, a poté i v rámci samotné knihovny libunwind, která rámce zpracovává při vyvolání výjimek. Prvním krokem před vytvořením rámců je zpracování nepodporovaných direktiv, které generuje překladač. Mezi tyto direktivy patří *.cfi_personality* a *.cfi_lsda*. Implementace podpory direktiv se provádí na několika místech. Nejprve dojde k úpravě parseru[18], kde se přidá assembleru token s direktivou. Dále je nutné vytvořit metodu pro zpracování tohoto tokenu. Pokud uvažujeme direktivu *.cfi_personality* vytvoří se nová třída *CFIPersonalityCommand*. Tato třída obsahuje metody pro tisk, porovnání a především předání parametrů třídě *DwarfFrameInfo* a její nově vytvořené metodě *Personality*. Metoda *Personality* ze vstupních dat vytvoří záznam do vektoru pro konfigurační řetězec. Do tohoto záznamu se uloží typ instrukce, její kódování (pokud je přítomno) a symbol který k instrukci patří, v případě metody *Personality* se jedná o adresu osobní rutiny. Pro direktivu *.cfi_lsda* se postupuje obdobně, jen symbol obsahuje adresu příslušného lsda.

Vektor pro konfigurační řetězec byl přidán do třídy *DwarfFrameInfo*, jako vektor struktur reprezentujících jednotlivé položky konfiguračního řetězce. Tato vytvořená struktura obsahuje informace o typu záznamu (lsda, osobní rutina nebo velikost adresy), kódování a volitelném parametru, kterým je symbol pro záznamy typu lsda nebo adresa osobní rutiny pro záznamy typu osobní rutina.

7.6.1 Vytvoření rámce *eh_frame*

Po zpracování direktiv dochází ke generování rámců. Nejdříve se z direktiv vytvoří příslušný *.debug_frame*, ukládající informace pro ladící programy, po tomto rámci je nutné vytvořit rámec *.eh_frame*, který bude sloužit knihovně libunwind ke zpracování výjimek.

Jak již bylo zmíněno výše, rámec *.eh_frame* se skládá z CFI záznamů, které jsou tvořeny dvěma druhy záznamů CIE a FDE. Přesněji *.eh_frame* obsahuje jeden a více CFI záznamů. CFI záznam se poté skládá z jednoho záznamu CIE, po kterém následuje jeden nebo více FDE záznamů. Pro vytvoření jednotlivých záznamů jsou využity shodná data i některé metody, které slouží ke generování *.debug_frame*. Prvním krokem je vytvoření celé sekce *.eh_frame*, která obsahuje jednotlivé záznamy. K vytvoření sekce se využije již implemen-

[18]Parser - část překladače, která se stará o syntaktickou analýzu vstupního zdrojového kódu.

tované metody *GenerateFrameInfoSectionContents*, která se rozšíří o několik informací pro generování *.eh_frame*. Metoda v původní implementaci slouží pouze ke generování rámce *.debug_frame*. V prvním kroku se tedy metoda rozšíří o argument, kterým je jméno generované sekce. Dále se přidá volání metody *GenerateFrameInfoSectionContents* s parametrem *.eh_frame* za volání stejné metody s parametrem *.debug_frame* v metodě *TransformTablesToObjFile* třídy *ObjFileTransfUnit*. Třída se stará o zápis do objektových souborů a metoda *TransformTablesToObjFile* zajišťuje transformaci tabulek z jazyka symbolických adres do objektového souboru.

Úprava implementace *GenerateFrameInfoSectionContents* spočívá ve volání nově vytvořené metody *CreateEHSection* pro sekci s názvem *.eh_frame* místo původní *CreateDebugSection* pro *.debug_frame*. Nově vytvořená metoda *CreateEHSection* vytvoří novou sekci *.eh_frame* do objektového souboru. Při vytváření sekce se nastaví její typ na *Debug*, dále jsou sekci nastaveny následující parametry:

- CONTENTS - sekce obsahuje parametry definované programem,
- ALLOC - sekce zabírá místo během vykonávání programu,
- LOAD - obsah sekce je přítomen ve zdrojovém kódu a může být čten do operační paměti již při spuštění programu,
- READONLY - parametr říká, že sekce slouží pouze ke čtení a není do ní povolen zápis,
- DATA - označuje, že sekce obsahuje pouze data, ne instrukce.

Pokud jsou parametry správně nastaveny, získáme z programu objdump[19] s parametrem *-x*, který umožňuje zobrazení hlaviček objektových souborů, výstup který odpovídá ukázce 7.12.

Po vytvoření dané sekce lze začít s plněním sekce jednotlivými CFI. Při zapisování jednotlivých záznamů jsou využity již existující metody třídy *DwarfUtils*. Přesněji metody *GenerateCIE* a *GenerateFDE*, obě metody byly rozšířeny o vstupní parametr typu *boolean*, který udává zda se jedná o *.eh_frame*. Metody umožňují přímý zápis do objektového souboru s využitím metod třídy *DwarfUtils*, kdy se metody třídy liší podle zapisovaného typu, například:

- pro zápis hodnoty typu *string* slouží metoda *AppendString*,
- pro zápis hodnoty s kódováním LEB128 slouží metoda *AppendLEB128*,
- pro zápis libovolné číselné hodnoty slouží metoda *AppendNumber*, vstupním parametrem této metody lze určit na jaké bitové délce bude číslice zapsána.

Formát záznamu CIE odpovídá tabulce 7.1.

Při generování záznamu se postupuje přesně v pořadí, v jakém jsou parametry zapsány v tabulce 7.1. První dojde k vygenerování prázdného místa o velikosti 4B, které bude sloužit k zapsání velikosti záznamu, jakmile bude tato hodnota známá. Velikost záznamu se udává na celý záznam bez pole délky. Za délku rámce se ukládá CIE ID, tato hodnota je uložena opět na 4B a vždy se pro *.eh_frame* ukládá hodnota 0 na rozdíl od *.debu_frame*, který

[19] Objdump - program zobrazující informace o objektovém souboru. Tento program lze vygenerovat jako součást SDK v rámci Codasip Studio.

```
Sections:
Idx Name Size VMA LMA File off Algn
  0 .text 000000b8 0000000000000000 0000000000000000 000007ee 2**2
                CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .gcc_except_table 00000028 0000000000000000 0000000000000000 000008a6↩
     2**2
                CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
  2 .debug_frame 000000ac 0000000000000000 0000000000000000 000008ce 2**0
                CONTENTS, RELOC, READONLY, DEBUGGING
  3 .eh_frame 000000a4 0000000000000000 0000000000000000 0000097a 2**0
                CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
  4 .codasip_as 00000011 0000000000000000 0000000000000000 00000a1e 2**0
                CONTENTS, READONLY
  5 .codasip_relexprtab 000000fd 0000000000000000 0000000000000000 00000↩
    a2f 2**0
                CONTENTS, READONLY
```

Ukázka kódu 7.12: Výstup programu objdump.

ukládá hodnotu $0xFF$. Tato informace slouží k rozlišení záznamů CIE od záznamu typu FDE.

Verze záznamu je reprezentována hodnotou uloženou na 1B. V rámci sekce *.debug_frame* se využívá záznamů, které jsou zapsány pomocí formátu DWARF ve verzi 4. Tato verze formátu DWARF není podporována v rámci knihovny libunwind, která podporuje pouze formáty verze 1, 2 a 3. Z důvodu co nejmenšího zasahování do knihoven jsem zvolil generování rámce ve formátu DWARF 3, u kterého se CIE liší pouze absencí samostatných polí pro velikost adresy a velikost segmentu. Došlo tedy k úpravě metody *GenerateCIE*, kdy se pro rámec *.eh_frame* generuje rámec DWARF verze 3. Další rozšíření metody spočívá v podpoře konfiguračního řetězce, kterou původní implementace neobsahuje. Nejprve dojde ke kontrole, zda konfigurační řetězec neobsahuje informaci o velikosti adresy. Pokud tyto informace neobsahuje a není prázdný, specifikuje se velikost adresy na velikost slova architektury. Pokud řetězec není zadán, musí velikost adres odpovídat 64b nezávisle na architektuře. Zde došlo k opravě chyby v původní implementaci, která adresu specifikovala vždy na velikost slova architektury. Tato skutečnost je způsobena tím, že výsledný objektový soubor nástrojů Codasip je ve formátu elf64, tedy linker při zpracování jednotlivých záznamů předpokládá, že velikost slova architektury odpovídá 64b. Zde dochází k problému, kdy linker rozhoduje o velikosti slova architektury na základě informace, kterou získá z ELF souboru, ale knihovna libunwind předpokládá velikost slova na základě definice *___SIZEOF_POINTER___*. Zde jsem zvolil úpravu knihovny libunwind, kdy v případě že kódování adresy není zadáno a odpovídá tedy velikosti ukazatele, dojde u architektur s velikostí slova o délce 32-bit ke čtení slova o velikosti 64b místo 32b. Stejná úprava musí být provedena i v rámci ladícího programu *LLDB* (Low Level Debugger), kdy při zpracování záznamů CIE a FDE se předá informace, že se jedná o 64b architekturu. Tato úprava v rámci ladícího nástroje ovlivňuje pouze zpracování adres, které mají kódování podle velikosti ukazatele. Úpravy v rámci ladícího nástroje jsou nezbytné, neboť se informace o zásobníku nejprve extrahují z *.eh_frame*, tedy rámce pro zpracování výjimek. Až při případném neúspěchu se začnou informace získávat z rámce *.debug_frame*. Pokud tedy výsledný binární

Délka záznamu	Povinný parametr
Rozšířená délka záznamu	Volitelný parametr
CIE ID	Povinný parametr
Verze	Povinný parametr
Konfigurační řetězec	Povinný parametr
EH Data	Volitelný parametr
Code Alignment Factor	Povinný parametr
Data Alignment Factor	Povinný parametr
Registr obsahující návratovou hodnotu	Povinný parametr
Délka konfiguračních dat	Volitelný parametr
Konfigurační data	Volitelný parametr
Inicializační instrukce	Povinný parametr
Zarovnání	

Tabulka 7.1: Uspořádání záznamu CIE.

kód byl přeložen bez ladících informací pouze s využitím zachytávání výjimek, lze nahradit tyto chybějící informace a využít rámce *.eh_frame* i pro činnost debuggeru.

Konfigurační řetězec se generuje na základě CFI direktiv s pomocí nově vytvořené metody *GenerateAugStr*. Řetězec začíná vždy písmenem *z*, za kterým mohou následovat tyto možnosti:

- *L* - odpovídá direktivě *.cfi_lsda*, označuje přítomnost jednoho bytu v konfiguračních datech, který udává kódování LSDA jež se nachází v FDE,
- *P* - generován na základě direktivy *.cfi_personality*, v konfiguračních datech se nachází dva argumenty, kdy první je kódování druhého argumentu na jednom bytu a druhý argument udává adresu osobní rutiny, velikost této adresy odpovídá použitému kódování,
- *R* - udává kódování adres použitých v FDE,
- *eh* - pokud jsou přítomna EH data v rámci CIE.

Code alignment factor a Code data alignment factor jsou shodné hodnoty o velikosti 1. První faktor je zapsán jako neznaménková hodnota LEB128 a druhý faktor jako znaménková hodnota LEB128. Za těmito faktory následuje uložená hodnota odpovídající návratovému registru. Tato hodnota se generuje na základě direktivy *.cfi_return_column* a ukládá se jako neznaménkový LEB128. Jelikož hodnota zapsaná v rámci LEB128 může mít různou délku, musí zápis délky dat konfiguračního řetězce předcházet jejich samotnému zápisu. Tedy nelze si alokovat místo, které bude později doplněno jako v případě délky rámce CIE, která má pevný formát. Po zápisu dojde k zápisu samotných dat konfiguračního řetězce. V případě zápisu adresy osobní funkce (personality routine) se nejdříve musí zjistit, zda může být adresa zapsána přímo, nebo musí být vytvořena relokace. K tomu se využije implementované metody *ResolveSymbol*, která nad každým vstupním symbolem zavolá jeho metodu *TryEval*. Metoda *TryEval* vrací hodnotu symbolu a jeho typ. Na základě typu symbolu se buď uloží symbol samotný, pokud se jedná o typ *ABSOLUTE* označující konstantní hodnotu, nebo dojde k vytvoření relokace. V případě vytvoření relokace lze odhalit její správné generování na základě kontroly ve výstupu programu objdump s parametrem *-x*, kdy zápis relokace lze vidět na ukázce 7.13. Z ukázky je patrné, že byla vytvořena relokace

```
RELOCATION RECORDS FOR [.eh_frame]:
OFFSET TYPE VALUE
0000000000000013 R_CODASIP_EXPR (bc: 4, msb: 31, lsb: 0) i256 $0 ↩
    __gxx_personality_v0
```

Ukázka kódu 7.13: Výstup programu objdump pro relokaci.

pro symbol *___gxx_personality_v0* v sekci *.eh_frame* s určitým offsetem, výsledná hodnota bude zapsána na 4B s nejvýznamnějším bitem na pozici 31.

Po zápisu konfiguračních dat se provede zápis inicializačních instrukcí. V implementovaném případě se jedná o instrukce prologu dané funkce, případně instrukce NOP (no operation) v případě, že prolog není přítomen. O zápis instrukcí se stará nově vytvořená metoda *AppendCFInstructions*, která obsahuje část již existující implementace z metody *GenerateFDE* sloužící k zápisu CFI instrukcí. Prolog funkce spočívá v uložení calee-save registrů, spolu s ukazatelem na vrchol zásobníku a návratovou adresou. Pro přesnou detekci konce prologu se využívá direktiva *.loc* končící volitelným argumentem v podobě řetězce *prologue_end*. Zpracování této direktivy se provádí ve dvou krocích. První slouží k detekci dané direktivy. Tato direktiva se detekuje v rámci metody *Ecexute* třídy *LocCommand*. Třída se stará o zpracování *.loc* direktiv při generování objektového souboru ze souboru v jazyce symbolických adres. Metoda se poté stará o generovaní informací na základě direktiv. V metodě se nachází kód pro zachycení direktivy *.loc* s identifikátorem *prologue_end*. Zde dojde k rozšíření kódu pro tuto direktivu o volání metody *CodasipPrologueEnd* třídy *DwarfFrameInfo*. Metoda *CodasipPrologueEnd* byla do třídy *DwarfFrameInfo* přidána v rámci implementace. Pokud assembler narazí na direktivu *.loc* s *prologue_end* dojde k volání metody *CodasipPrologueEnd*, která provede přesun všech předchozích instrukcí z vektoru pro generování FDE do vektoru pro generování inicializačních instrukcí. Vektor pro instrukce prologu musel být také přidán do třídy *DwarfFrameInfo*, aby bylo možné jednoznačně rozlišit instrukce prologu od ostatních instrukcí.

S využitím existující direktivy se pojí i další úpravy, nutné pro správnou detekci konce prologu. Direktiva *.loc* s *prologue_end* se vkládá za poslední instrukci, která nemá nastavený příznak typu *SetupFrame*, který říká, že daná instrukce představuje prolog funkce. Vlivem různých optimalizací v rámci překladače může dojít k přeuspořádání instrukcí, kdy se do popředí dostanou instrukce, které nemají příslušný příznak. V takovém případě dojde k chybnému generování direktivy *.loc* s *prologue_end* a následně i chybnému zápisu inicializačních instrukcí. Řešením je úprava metody *runOnMachineFunction* ve třídě *CfiAdder*. Tato třída tvoří poslední průchod před generováním výstupního souboru v jazyce symbolických adres a úpravy tedy neovlivní jiné optimalizační kroky. Metoda samotná slouží ke generování CFI instrukcí pro výstupní soubor na základě jednotlivých instrukcí vstupní machine funkce. Úprava metody spočívá v nalezení poslední CFI direktivy s nastaveným příznakem *FrameSetup* a všem instrukcím předcházejícím se nastaví příznak *FrameSetup*, díky tomu se dostanou všechny instrukce prologu mezi inicializační instrukce CIE záznamu.

Druhým problémem, který lze také řešit na úrovni metody *runOnMachineFunction*, je závislost direktivy *.loc* na přítomnosti ladících informací. K řešení této závislosti se využije příznak *CodasipFlag*. Ten se nastaví v případě, že nejsou přítomny ladící informace na instrukci následující poslední CFI direktivu prologu společně s příznakem *FrameSetup*, aby nedošlo k chybnému generování pro instrukce využívající tento příznak. Tento příznak se následně detekuje při tisku instrukcí v rámci metody *EmitInstruction* třídy *Co-*

dasipAsmPrinter, kde se v případě, že nejsou přítomny ladící informace, ale je přítomen příznak *CodasipFlag* spolu s příznakem *FrameSetup* vytiskne před danou instrukci *.loc 0 0 end_prologue*.

Inicializační instrukce jsou následovány zarovnáním tak, aby byl záznam zarovnán na velikost slova architektury. Na závěr se doplní velikost záznamu na místo, které bylo vyhrazeno pro tento účel při vytváření záznamu.

V dalším kroku dojde k vytvoření záznamu FDE, který se váže k záznamu CIE vytvořeném v předchozím kroku. V tabulce 7.2 lze vidět uspořádání jednotlivých parametrů v rámci záznamu.

Délka záznamu	Povinný parametr
Rozšířená délka záznamu	Volitelný parametr
Ukazatel na záznam CIE	Povinný parametr
Začátek PC	Povinný parametr
Rozsah PC	Povinný parametr
Délka konfiguračního řetězce	Volitelný parametr
Data konfiguračního řetězce	Volitelný parametr
CFI	Povinný parametr
Zarovnání	

Tabulka 7.2: Uspořádání záznamu FDE.

Pro vytvoření záznamu FDE byla využita již existující metoda *GenerateFDE* třídy *DwarfFrameInfo*. Využití této metody jsem zvolil z důvodu, že záznam FDE by se neměl lišit při využití pro ladící nástroj nebo pro zachytávání výjimek. Metoda navíc již obsahovala zpracovaní některých CFI direktiv, kdy zpracování bylo nahrazeno voláním metody *AppendCFInstructions*, která obsahuje původní kód pro zpracování CFI direktiv rozšířený o některé nové direktivy. V této metodě bylo uděláno několik úprav tak, aby záznam odpovídal standardu se zajištěním plné spolupráce s nástrojem linker a knihovnami jazyka C++, přesněji knihovnou libunwind, která se stará o práci s rámcem *.eh_frame* spolu s jeho záznamy.

Při vytvoření záznamu se postupuje obdobně jako u záznamu CIE. Nejprve se vytvoří 4B velké pole, které bude sloužit k zápisu velikosti. Toto pole následuje ukazatel na příslušný záznam CIE. Zde byla provedena první úprava, kdy ukazatel odpovídá relativní adrese na záznam CIE, místo vytvoření relokace, která se děje v případě, že záznam slouží nástroji debugger (rámce *.debug_frame*). Ukazatel poté odpovídá hodnotě: $pocatecniAdresaFDE - pocatecniAdresaCIE + 4$, kde hodnota 4 udává posun o velikost pole délky CIE záznamu, počáteční adresy jednotlivých záznamů ukazují na pole jejich délky.

Dále dojde k zápisu rozsahu programového čítače, kterého se daný záznam týká. Nejprve dojde k vytvoření symbolu, který slouží jako počáteční označení rozsahu, symbol se vloží do tabulky relokací. Následně se provede zápis rozsahu programového čítače. Rozsah samotný odpovídá rozdílu adres direktiv *.cfi_stratproc* a *.cfi_endproc*. Při zápisu symbolu i jeho rozsahu došlo k úpravě velikosti zapisované hodnoty z důvodů, které byly uvedeny výše, tedy výsledným binárním souborem ve formátu ELF64 i pro 32-bitové architektury. Zapisovaná hodnota má velikost buď 64b, nebo odpovídá velikosti udávané parametrem *R* v rámci konfiguračního řetězce příslušného CIE záznamu. O určení správné velikosti se stará přidaná metoda *GetAddrSize*, která vrací správnou velikost adresy.

Po zapsání rozsahů se přejde na zápis konfiguračních dat. Zde byla přidána podpora pro adresu landing pad, pokud byla zadána v rámci parametru *L* konfiguračního řetězce přísluš-

ného záznamu CIE. Data jsou zapsána na délku odpovídající kódování, které bylo zadáno v direktivě *.cfi_lsda*, na danou adresu se ukládá buď symbol, pokud se jedná o konstantní symbol, nebo relokace symbolu.

V předposledním kroku se vkládají instrukce, kdy bylo nutné provést úpravu ve výpočtu offsetu instrukcí obsahujících adjust offset. Zde se výsledný offset správně vypočítá následovně: $OffsetInstrukce/ZarovnaniDat$. Na závěr dojde k zarovnání záznamu na délku slova.

Poslední úpravou v rámci nástrojů je rozšíření podpory CFI direktiv o direktivu *.cfi_restore*. Direktiva nahradí chybně využitou direktivu *.cfi_same_value* v epilogu volaných funkcí, kdy epilog je část funkce, ve které dochází k obnovení předchozího kontextu, například obnovení callee save registrů, stack pointer apod. Nahrazení bylo provedeno v metodě *emitEpilogue*, starající se o přidání epilogu na konec funkce, třídy *CodasipGenFrameLowering*, která obsahuje informace o zásobníku a jeho rámcích. V rámci rozšíření podpory bylo nutné jednak přidat do parseru assembleru token s direktivou a dále zpracování tohoto tokenu, o který se postará nově vytvořená třída *CFIRestoreCommand*. Tato třída obsahuje metody pro tisk, porovnání a především předání parametrů třídě *DwarfFrameInfo* a její nově vytvořené metodě *Restore*. Metoda *Restore* převezme parametry a vytvoří z nich FDE instrukci, která se zpracuje v rámci vytváření FDE záznamu, jež je popsáno v odstavci výše.

7.6.2 Úprava custom souborů modelu

Po provedení úprav v rámci nástrojů Codasip následují úpravy na úrovni custom souborů modelu, obdobně jako v případě zachytávání výjimek pomocí Set Jump a Long Jump. V rámci custom souborů dochází k jednoduché úpravě třídy *CodasipGenTargetLowering*, kde došlo k přetížení metod *getExceptionPointerRegister* a metody *getExceptionSelectorRegister*, které vrací registry *R4* a *R5*. Tyto registry byly zvoleny, jelikož představují první dva registry pro předání parametrů volání funkcí. Další úprava v rámci custom souborů spočívá v úpravě prologu a epilogu funkcí v rámci třídy *CodasipGenFrameLowering*. U metody prologu *emitPrologue* dochází ke generování CFI direktiv o stavu callee-save registrů. U metody epilogu *emitEpilogue* se nahradí použití CFI direktivy *.cfi_same_value*, která udává, že hodnota registru je shodná s hodnotou registru v předchozím rámci za *.cfi_restore*.

Kromě custom souborů modelu dojde k úpravě i linker scriptu. Tento skript upřesňuje jak mají být zapsány jednotlivé sekce do výstupního souboru. Do linker skriptu musí být přidána sekce *.eh_frame* a *.eh_frame_hdr* spolu se symboly označujícími začátek a konec sekce *.eh_frame*. Sekce *.eh_frame_hdr* slouží k jednoduššímu a efektivnějšímu přístupu k záznamům v sekci *.eh_frame*. Jejím obsahem jsou ukazatele na jednotlivé záznamy a vyhledávací tabulka nad těmito ukazateli. Sekce *.eh_frame_hdr* není v rámci implementace využívána, neboť nedojde k jejímu vygenerování programem *ld*.

7.6.3 Úprava Clang

V rámci přední části překladače Clang byla implementována metoda *getEHDataRegisterNumber*, která vrací registry pro DWARF zachytávání výjimek. Metoda se využívá při volání builtin funkce *___builtin_eh_return_data_regno*, kterou využívá knihovna libc++abi pro získání informací u osobní rutiny. Tato metoda se nachází ve třídě *CodasipTargetInfo*, která obsahuje informace o cílové architektuře. Návratové registry odpovídají registrům *R4*, pokud je vstupní parametr roven 0 a *R5*, pokud se vstupní parametr rovná hodnotě 1. Registry jsou shodné s registry, jež vrací implementace metod *getExceptionPointerRegister* a *getExceptionSelectorRegister*. Rozdíl v návratové hodnotě registrů tvoří použití v návratu

jejich DWARF číslování. Tyto informace je nutné získat z generátoru zadní části překladače a předat je do částí přední, která nemá k těmto informacím přístup. Toho lze dosáhnout přidáním výchozích argumentů pro překladač. Ty se nachází uvnitř souborů s názvem překladače a sufixem *_args.txt*. Tyto soubory se nachází ve shodném adresáři s překladačem. Ke generování souborů dochází uvnitř metody *generate_llc_related_scripts* třídy *BackendgenBuilder*, která se nachází ve skriptu *backendgen.py* a slouží jako část Codasip commandline. Třída se stará o generování překladačů pro jazyky C a C++, metoda *generate_llc_related_scripts* slouží ke generování souborů s výchozími argumenty pro tyto překladače.

Do metody *generate_llc_related_scripts* byla přidána proměnná *ehdataregs*, jejíž hodnota odpovídá návratové hodnotě příkazu *codasip.check_output([llc, '-get-ehdataregs'])*. Tento zápis zavolá program llc s parametrem *-get-ehdataregs* a vrátí jeho výstup. Tato získaná hodnota se následně zapíše do proměnné *cmd* spolu s přepínačem *-ccc-ehdata-regs*. Proměnná *cmd* se na závěr metody zapíše do příslušných souborů pro překladače. Přepínač llc *-get-ehdataregs* a clang *-ccc-ehdata-regs* jsou implementovány v rámci následujících kroků.

Do zdrojového kódu *llc.cpp* se přidá externí proměnná typu *char** jménem *___codasip_ehdata_regs* a přepínač *PrintEHDataRegs*, jehož parametrem je *-get-ehdataregs*. Pokud je přepínač zadán, detekuje se ve funkci *main*, kde dojde k tisku obsahu proměnné *___codasip_ehdata_regs*. Data této proměnné se získají ve zdrojovém souboru *CodasipTargetMachine.cpp.tp*, v této šabloně se generují metody obsahující informace o cílové architektuře. Zde se s pomocí nově přidané metody generátoru backendu[20] *GetEHDataRegs()* inicializuje proměnná *___codasip_ehdata_regs*. Implementace metody *GetEHDataRegs()* se nachází ve třídě reprezentující generátor backendu *BGWrappper*. Tato nově vzniklá metoda získá z informací o ABI architektury registry pro předávání parametrů a vrátí jejich DWARF číslování v podobě řetězce, kdy jednotlivé registry jsou odděleny znakem ":".

V případě, že lze z llc získat požadované informace, které jsou následně uloženy mezi výchozí parametry překladače, lze implementovat zpracování těchto parametrů v přední části překladače. Prvním krokem je rozšíření programu clang o argumenty ve zdrojovém souboru *CodasipClangOptions.td*, kde jsou přidány možnosti *-ccc-ehdata-regs* spolu s define *ccc_ehdata_regs*, který se získá na vstupu a *-ehdata-regs* s příslušným define *ehdata_regs* pro nastavení atributů překladače. Při zadání parametru *-ccc-ehdata-regs* dojde k jeho zachycení v rámci metody *setAttrs* třídy *CodasipToolChain*, kde metoda slouží k nastavení argumentů pro nástroje, ke kterým třída poskytuje přístup. Zde dojde k naplnění přidané proměnné třídy *EHDataRegs* hodnotou zadanou parametrem *-ccc-ehdata-regs*. Tyto data se využijí při konfiguraci nástroje clang před spuštěním v rámci metody *ConstructJob* třídy clang, kde je přidán parametr pro překladač *-eh-data-regs* s hodnotou *EHDataRegs* do vektoru argumentů. Při zpracování argumentů ve třídě *ParseTargetArgs* se argument přesune do proměnné *EHDataRegs* ve třídě *TargetOption*. Tato třída udržuje informace o všech použitých argumentech.

Na základě třídy *TargetOption* poté dojde k získání argumentů v metodě *init* třídy *CodasipTargetInfo*, kde se v případě, že instance třídy *TargetOption* obsahuje informaci o registrech pro zachytávání výjimek tyto registry uloží do přidaného vektoru *EHDataRegs*. Při volání přidané metody *getEHDataRegisterNumber* se poté zkontroluje jestli vstupní parametr není mimo rozsah vektoru a dojde k navrácení příslušného registru.

[20]Backend generátor - Nástroj sloužící k tisku informací závislých na cílové architektuře do předpřipravených šablon. Informace o cílové architektuře získává z implementovaného modelu procesorového jádra.

7.6.4 Úprava knihoven

Posledním krokem pro zajištění zachytávání výjimek typu DWARF je vytvoření specifické třídy *Registers_codasip_urisc* v rámci knihovny libunwind. Tato třída popisuje registrové pole daného procesoru. V tomto případě se jedná o procesorové jádro typu Codasip uRISC. Třída poskytuje metody pro přístup k ukazateli na zásobník, ukazateli na prováděnou instrukci a přístup k ostatním registrům. V rámci Codasip uRISC se jedná o pole s 32 registry šířky 32b.

Další krok implementace spočívá ve vytvoření dvou funkcí v jazyce symbolických adres, které jsou používány knihovnou libunwind. První z těchto dvou funkcí provádí uložení kontextu programu, nazývá se *unw_getcontext* a provádí uložení všech registrů na adresu danou registrem *R4*. Její implementace se nachází ve zdrojovém souboru *UnwindRegisterSave.S* knihovny libunwind, kde se podmíní definicí makra *___codasip_urisc_ia___*. Druhá funkce *jumpto* slouží k obnovení kontextu z adresy předané registrem *R4*, provede postupné obnovení všech registrů a skok na registr *R3*, který udržuje návratovou hodnotu. Nachází se v knihovně libunwind ve zdrojovém kódu s názvem *UnwindRegisterRestore.S* a je podmíněn shodným makrem jako předchozí funkce.

Tyto úpravy pro knihovnu musí být provedeny pro každé nově přidané procesorové jádro, případně i každou možnou kombinaci procesorového jádra. Podpora nově vzniklé třídy musí být přidána i k vytvoření nového kurzoru pro procházení zásobníku ve funkci *unw_init_local*, kde musí pro každou nově vzniklou třídu být přidána podmíněná definice pro *REGISTER_KIND*. V implementovaném případě to znamená, že pokud je definováno makro *___codasip_urisc_ia___*, pak *REGISTER_KIND* odpovídá třídě *Registers_codasip_urisc*.

Pro jednodušší práci s knihovnou v ní byla provedena úprava pro zpracování FDE instrukcí, kdy hodnota nezadaného CFA registru, nastavená při inicializaci CFA registru uvnitř struktury obsahující informace o zpracování CFI instrukcí, je změněna na nově definovanou hodnotu -3. Rozšiřuje tak již existující hodnoty -1 a -2, odpovídající ukazateli na instrukci a ukazateli na zásobník. Hodnota se přepíše, pokud se mezi CFI instrukcemi vyskytne některá z instrukcí definující CFA registr. Tato úprava umožňuje mít DWARF číslování registrů začínající od hodnoty rovné 0, tak jako má například architektura Codasip uRISC.

Kapitola 8

Integrace do nástrojů Codasip Studio

Kapitola zabývající se integrací knihoven a implementace zachytávání výjimek do nástrojů Codasip tak, aby bylo možné využívat jazyka C++ na libovolném procesorovém jádře s minimálním zatížením programátora starajícího se o návrh procesorového jádra. Kapitola se skládá ze dvou částí, kdy první část se zabývá integrací custom souborů do nástrojů Codasip a druhá část integrací knihoven a přidáním podpory pro jejich překlad v rámci Codasip Studio. V kapitole není záměrně zmíněna integrace testovacích nástrojů, neboť tím se zabývá kapitola 9 o testování implementace.

8.1 Integrace custom souborů

Integrace custom souborů spočívá v přenesení změn, které byly v rámci těchto souborů provedeny do šablon, které slouží ke generování zdrojových kódů překladače. Veškerý kód přenesený do šablon musí být nezávislý na cílové architektuře. Tedy nelze využít fyzické názvy registrů nebo konkrétní názvy instrukcí ze sémantiky jako tomu je v předchozí kapitole 7.

8.1.1 CodasipGenISelLowering

V prvním kroku lze nahradit některé jednodušší operace procesorového jádra za základní instrukce LLVM. Tyto základní instrukce budou nahrazovány za konkrétní instrukce daného procesorového jádra v rámci selekce instrukcí na základě použitých operandů. Mezi tyto instrukce patří:

- LOAD,
- STORE a
- ADD.

Po nahrazení těchto jednoduchých instrukcí zbývá nahradit instrukce složitější. První z nich je instrukce nepřímého skoku. Pro tuto instrukci neexistuje základní instrukce LLVM, kterou by bylo možné využít v této fázi překladu. Řešením této situace je využití existující metody *getCodasipInstr* třídy *CodasipInstrInfo*. Tato metoda naplní vstupní vektor instrukcemi, které lze využít k provedení požadované instrukce. V rámci implementace se

nad vektorem provede iterace, která se pokusí vybrat nejvhodnější instrukci k využití na základě parametrů instrukce. Pokud se tedy jedná o známý tvar instrukce, dojde k jejímu využití, pokud žádná instrukce nelze využít, dojde k vyvolání chyby překladu. Mezi známé tvary instrukcí patří instrukce s jedním nebo dvěma operandy, kdy první operand je cílový registr a v případě přítomnosti druhého operandu se jedná o konstantu, jež bude vždy nastavena na hodnotu 0.

Pokud metoda *getCodasipInstr* neobsahuje některou instrukci, lze ji upravit v custom souborech modelu procesorového jádra. Toho bylo využito v implementaci, kdy tento custom soubor obsahuje rozšíření metody *getCodasipInstr* o operaci *BRIND*, která vrací nepřímý skok. Nepřímý skok se využívá v implementaci na několika místech a nebylo možné jej nahradit bez využití custom souboru.

Obdobným způsobem jako instrukce nepřímého skoku lze nahradit i instrukci pro podmíněný skok s porovnáním dvou registrů, kdy se využije existující metoda *getCondBranchOpc* třídy *CodasipInstrInfo*, která opět naplní vektor instrukcemi pro požadovanou operaci. V tomto případě operací podmíněného skoku. Zde musí být vykonány všechny instrukce vektoru. Implementované zpracování vektoru nejprve provede kontrolu počtu instrukcí, tedy velikosti vektoru. Pokud vektor obsahuje jednu instrukci, lze této instrukci předat všechny parametry. Parametry instrukce jsou cíl skoku a dva registry na porovnání. V případě, že vektor obsahuje dvě instrukce, první instrukce provede porovnání a jeho výsledek vloží do cílového registru, na základě obsahu cílového registru se provede skok v instrukci následující. Metoda *getCondBranchOpc* je metodou, která se nachází v rámci custom souborů a obsah vektoru může tedy ovlivnit i návrhář procesorového jádra. V rámci architektury Codasip uRISC se pro použitou operaci využívá instrukce *sle* (*set if less or equal*), která nastaví cílový registr na hodnotu rovnou 1, pokud je hodnota prvního registru menší, nebo rovna obsahu druhého vstupního registru. Zbývající instrukce skoku představuje instrukce *jumpnz* (*jump if not zero*), která provede skok na návěští, pokud vstupní registr není roven hodnotě 0.

Obdobným způsobem jako jsou realizovány předchozí instrukce skoků, lze generovat i instrukci pro násobení. První krok pro získání instrukce násobení spočívá v nahrazení samotné operace násobení, kterou lze nahradit v implementovaném případě za operaci posuvu, neboť se u 32-bitové i 64-bitové architektury jedná o násobky číslice dvě. Instrukce pro logický posuv se poté získá s využitím metody *getCodasipInstr* se vstupní operací *ISD::SHL*. Dále se postupuje stejně jako v předchozím případě, kdy se na základě počtu operandů a jejich typů instrukce vykoná.

Poslední instrukcí využitou v rámci custom souborů, bránících přenesení do šablon zdrojových kódů překladače je instrukce načtení konstanty. K nahrazení této instrukce se využívá tzv. backend generátor. Metody tohoto generátoru lze v šablonách využít pro generování instrukcí cílové architektury, jak bylo zmíněno v předchozím textu. Příklad lze vidět na ukázce kódu 8.1, kde se nachází tvorba instrukce v rámci bloku *DispatchBB* se dvěma vstupními parametry *MJTIAddr*, značící virtuální registr pro uchování konstanty a *MJTI* označující index tabulky skoků. Výsledný formát tohoto řádku ve vygenerovaném zdrojovém kódu se nachází v ukázce kódu 8.2, kde lze již vidět konkrétní instrukci překladače, přesněji instrukci *e_movi32___* jež představuje emulaci[1] načtení 32b konstanty.

Kromě instrukcí bylo nutné upravit generování registrů pro metody *getExceptionPointerRegister* a *getExceptionSelectorRegister*. Tyto metody vrací první a druhý registr pro

[1]Emulace instrukcí - jedná se o instrukce, které pro vykonání dané sémantiky potřebují 2 a více jednoduchých instrukcí. Například instrukce *e_movi32___* jejíž sémantika obsahuje pouhé přiřazení konstanty do cílového registru se skládá ze dvou dílčích instrukcí: načtení horních 16b a načtení spodních 16b.

```
BuildMI(DispContBB, dl, $$( m_LegalRecog.PrintMachLoadImm2ARDesc(OUT); $$),
    MJTIAddr).addJumpTableIndex(MJTI);
```

Ukázka kódu 8.1: Generování instrukcí, zápis v rámci šablony, metody backend generátoru se nacházejí mezi dvojicemi symbolů $$.

```
BuildMI(DispContBB, dl, TII.get(Codasip::e_movi32__),
    MJTIAddr).addJumpTableIndex(MJTI);
```

Ukázka kódu 8.2: Generování instrukcí, zápis ve vygenerovaném zdrojovém souboru.

předávání parametrů funkcím. Tyto informace lze získat z informací o ABI procesoru v generátoru backendu. Vyžije se k tomu metoda *getParamReg* vracející vektor registrů pro předávání vstupních parametrů. Při výběru konkrétního registru lze získat jeho jméno pomocí metody *GetLlvmName*. Pro získání kompletního označení registru např. *Codasip::rf_gpr_4* značícího registr *r4* bude vypadat kód jako na ukázce 8.3.

Po těchto úpravách lze bezpečně přenést všechny přidané metody a úpravy existujících metod do šablony, kde budou sloužit ke generování zdrojových kódů překladače.

8.1.2 CodasipAsmPrinter

Zde se nachází podstatně méně úprav. Jediná úprava spočívá v přidání podpory symbolu *MO_MCSymbol* do metody *LowerSymbolOperand*. Tento symbol se využívá při vytváření uzlu s operací *GAWrap*, jejímž operandem je symbol *GCC_except_table*, jak bylo popsáno v 7.5.1. Metoda se stará o zpracování jednotlivých symbolů, mezi které patří například adresy bloků, index do tabulky skoků apod. V rámci zpracování symbolu *MO_MCSymbol* se pouze předá adresa jeho proměnné a hodnota jeho offsetu. Podpora tohoto symbolu musí být také přidána do funkce *WriteInstrAux*, která se stará o tisk symbolů. Doplněn musí být také v rámci *GetOpName*, aby nedošlo k vyvolání výjimky nezpracováním neznámého symbolu.

8.1.3 CodasipGenInstrInfo

Zde se před přenesením změn do šablon musí nahradit výskyty registrové třídy. K nahrazení se využije opět generátoru backendu a jeho metody *PrintBaseRegClass*, jejíž implementace byla doplněna.

Implementace metody *PrintBaseRegClass* třídy *BGWrappper* spočívá v iteraci nad registrovými poli daného procesorového jádra a v případě, že narazí na obecný registrový prostor vytiskne jeho jméno na dané místo.

```
Codasip::$$[m_Abi.GetReturnRegs().at(
        m_Cpu->GetPointerType())[0]->GetLlvmName() $$];
```

Ukázka kódu 8.3: Generování názvů registrů.

8.1.4 CodasipJumpLengthCheck

Zde se doplní metoda *getLongJumpOpcodeStatic*, která vrací operační kód největšího možného skoku v rámci architektury. Pro získání takového skoku se využije generátor backendu a jeho metoda *GetJump*, která vrací požadovanou instrukci. Této metody se využívá v implementaci metod pro zachycení výjimek typu Set Jump Long Jump, v metodě *emitEHSjLjDispatchBlock*, kde se využívá ke skoku na záznamy z tabulky skoků, externí symbol *abort* nebo skoku na machine basic block. Také se využívá pro implementaci builtin funkce *__builtin_longjmp* v metodě *emitEHSjLjLongJmp*, kde se využívá k získání skokové instrukce na cíl uložený v registru.

8.1.5 CodasipCFIAdder

První provedená změna byla popsána v kapitole 7.6.1, kdy byla upravena metoda *runOnMachineFunction* z důvodu generování označení konce prologu funkce. Druhá změna spočívá v omezení generování CFI direktiv, přesněji zakázání generování CFI direktiv v epilogu funkce. Stav bez úpravy generoval například CFI direktivu o posuvu zásobníku zpět na hodnotu předcházející volání funkce. Tato direktiva poté vedla na špatné sledování průběhu zásobníku i obnovování kontextu v rámci zpracování výjimek využívajících DWARF informace. Tyto informace se generují v metodě *runOnMachineFunction*. Tato metoda rozlišuje, zda se jedná o prolog nebo epilog funkce na základě parametru *FrameSetup*, respektive *FrameDestroy* a pro tyto části generuje jednotlivé CFI. Zde došlo k přidání podmínky, která povoluje přidávat CFI jen v případě, kdy se nejedná o epilog.

8.2 Integrace knihoven

Při integraci knihoven se v prvním kroku provede nakopírování adresářů knihoven do repositáře *tools* k ostatním knihovnám podporovaných v rámci nástrojů Codasip. Po tomto kroku dojde k úpravě *CMakeList* tak, aby nové knihovny byly nainstalovány spolu s ostatními knihovnami do cílového adresáře instalace nástroje Codasip Studio. V rámci úpravy *CMakeList* lze určit, které adresáře budou ignorovány a nebudou zbytečně instalovány, zde se odstraní adresáře obsahující dokumentaci a testovací sady. Testovací sada byla vyřazena, neboť se nachází v rámci repositáře *zero-tolerance-check* spolu s testovacími skripty. Popis integrace testovací sady se nachází v kapitole 9.

V dalším kroku se přidá rozšíření konfiguračního souboru modelů procesorových jader. Po rozšíření lze knihovny přeložit přímo z Codasip Studio IDE nebo příkazové řádky, kdy překlad knihoven jazyka C++ probíhá v rámci kompilace knihoven spolu s ostatními knihovnami jako je knihovna Newlib nebo knihovna CompilerRT. Rozšíření spočívá v přidání příslušných možností do souboru *dtd.xml* o nové elementy reprezentující knihovny jazyka C++. Poslední krok integrace knihoven spočívá v přidání tříd pro překlad knihoven jazyka C++ do souboru *libraries.py*. Tento soubor se stará o překlad knihoven podporovaných v rámci nástrojů Codasip Studio. Zde je vytvořena třída *LibraryCxxSupport*, která rozšiřuje vlastnosti třídy *LibraryBaseBuilder*. Tato nová třída obsahuje pouze metodu *build_impl*. Metoda se stará o překlad jednotlivých knihoven jazyka C++ na základě vstupních definic nástrojem *CMake*. Po překladu knihoven dojde ke kopírování přeložené statické knihovny a jejich příslušných hlavičkových souborů do cílového adresáře SDK daného procesorového jádra.

Třídu *LibraryCxxSupport* dále rozšiřují třídy pro jednotlivé knihovny *LibraryUnwind*, *LibraryCxxabi*, *LibraryCxx*. Každá z nových tříd obsahuje pouze metodu *run_impl*, která

tvoří rozhraní tříd pro překlad knihoven. Tato metoda v rámci každé knihovny nastaví příslušné proměnné, definice a zavolá metodu *build_impl* třídy mateřské. Dále je nutné přidat nově vzniklé knihovní třídy do proměnné *AVAIBLE_LIBRARIES* třídy *LibrariesBuilder*, starající se o překlad jednotlivých knihoven.

Důležitou součástí kompilace knihoven je zajištění jejich nezávislosti na pořadí kompilace. To se zajistí předáním cest k hlavičkovým souborům jednotlivých knihoven, kdy jednotlivé cesty využívají hlavičkové soubory přítomné v instalační cestě nástrojů Codasip, místo využívání hlavičkových souborů přítomných v rámci SDK příslušné architektury.

Jedním z cílů implementace bylo zajištění plné podpory obou verzí volání výjimek bez nutnosti opětovného překladu knihoven a samotného překladače. Toho lze dosáhnout kompilací dvou verzí knihovny. První verze, která slouží jako výchozí verze, podporuje výjimky využívající ladící informace ve formátu DWARF. Tuto knihovnu překladač využívá v případě, že nejsou specifikovány žádné speciální požadavky, nebo je zadán parametr *-fdwarf-exceptions* říkající, že překladač využívá výjimky typu DWARF. Výjimky typu DWARF jsem zvolil za výchozí řešení, neboť by měly při zpracování dosahovat lepších výsledků než výjimky typu Set Jump Long Jump, z důvodů které byly uvedeny v kapitole 5.2.

O podporu dvou verzí knihoven se musí také rozšířit samotný překladač. Úprava na straně překladače spočívá jednak v přidání zpracování argumentů předaných překladači, přesněji přední části překladače v podobě programu *clang*, v rámci metody *ConstructJob* třídy *ClangLlc*, která se stará o předání správných parametrů programu *llc*, představujícího zadní část překladače. Zde se zpracovávají jednotlivé typy výjimek a předají se v podobě argumentu programu *llc* (Low Level Compiler). Argumenty se v rámci programu *llc* propagují do třídy *MCAsmInfo*, která udržuje informace o cílové architektuře. Druhé rozšíření spočívá v úpravě metody *ConstructJob* třídy *Link*. Tato metoda nastavuje výchozí knihovny pro nástroj linker tak, aby nemusely být zadávány uživatelem při každém překladu. Zde jsou přidány podmínky, které se využijí pouze v případě překladače jazyka C++. Tyto podmínky jsou:

- v přítomnosti argumentu *-fsjlj-exceptions* se využijí knihovny jazyka C++ se sufixem *_sjlj*,
- v přítomnosti parametru *-nodefaultlibs* se nevyužije žádná knihovna,
- pokud není přítomen žádný z předchozích argumentů, použijí se výchozí knihovny jazyka C++, tedy knihovny využívající pro zachytávání výjimek ladících informací DWARF.

Kromě výchozích knihoven jsou zadány i výchozí cesty k hlavičkovým souborům knihovny jazyka C++. Děje se tak v metodě *AddClangSystemIncludeArgs* třídy *CodasipToolChain*, kde se na základě jména knihovny (*libc++*) přidá výchozí cesta k hlavičkovým souborům.

Kapitola 9

Testování

V následující kapitole bude popsáno testování implementace. V první části kapitoly bude představena testovací sada a její integrace do nástrojů Codasip Studio. Ve druhé části jsou uvedeny výsledky testování nad jednotlivými procesorovými jádry společnosti Codasip.

9.1 Testovací sada

Testovací sadu tvoří testy, které jsou dodávány společně s jednotlivými knihovnami. Především jsem se zaměřil na testovací sadu přítomnou v rámci knihovny libc++. Tato testovací sada obsahuje 5909 testů. Z těchto testů se v první fázi spouštěly testy pouze v adresáři libcxx. Tento adresář obsahuje pouze 336 testů, které v prvotní fázi dostačovaly pro odhalení zásadních chyb v implementaci. Při odladění implementace na této jednoduché testovací sadě byly provedeny i ostatní testy. Nutno dodat, že testovací sada se značně rozšiřuje při využití vytvořeného testovacího prostředí pro nástroje Codasip, jelikož se každý test vykonává nad všemi dostupnými optimalizacemi. Dostupnými optimalizacemi jsou:

- optimalizace zaměřující se na rychlost překladu (*O0*),
- optimalizace zaměřující se na rychlost vykonání programu (*O1*, *O2*, *O3*) a
- optimalizace zaměřené na velikost kódu (*Os*, *Oz*).

Při vykonání celé testovací sady se poté dostaneme na 35454 testů. Tato sada je dále rozšířena o 40 testů, které byly vytvořeny na základě chyb objevených v průběhu implementace, ty mají jednoduše ověřit klíčové vlastnosti implementace. Testy se především zaměřují na volání globálních konstruktorů, využití implementovaných builtin funkcí, volání výjimek a volání virtuálních metod. Zaměřují se tedy především na implementovanou funkcionalitu a funkcionalitu, která se mohla změnit v případě využití jiného typu výjimek. Důvodem vytvoření této menší testovací sady je zjednodušení průběhu testování během implementace, kdy průběh celé testovací sady trvá na procesorovém jádře Codasip uRISC cca 18 hodin. Druhým důvodem je lepší kontrola průběhu výjimek sledováním různých výpisů, které jsou porovnány s referenčním souborem.

9.1.1 Testovací prostředí

Jak bylo zmíněno výše, v rámci testovací sady bylo vytvořeno prostředí, které se stará o její průběh. Toto prostředí využívá framework pytest[1].

[1] Pytest - jedná se o framework, který umožňuje psát kódy pro testování v programovacím jazyce python.

```
@pytest.mark.files(pattern='\.cpp$')
@pytest.mark.compiler(cxx=True, cxxabi=True, newlib=True, compiler_rt=True)
@pytest.mark.zero_cxx
def test_libcxx(compiler, simulator, files, work_dir)
```

Ukázka kódu 9.1: Prototyp testovací funkce

V první části tvorby prostředí dojde k vytvoření těla funkce, která se bude starat o spouštění správných testů. Funkce je označena tzv. marks[2], jak lze vidět na ukázce 9.1. Význam jednotlivých marks:

- *files* - označuje typ souborů, které se získají na vstupu, tedy soubory s příponou ".cpp",
- *compiler* - zavádí omezující podmínky na překladač, překladač musí mít knihovny libc++, libc++abi, libunwind, newlib a
- *zero_cxx* - slouží k selekci testovacích funkcí.

V těle dané funkce dojde nejdříve k filtraci testů, které se nebudou testovat a budou tedy přeskočeny. Tato filtrace probíhá na dvou úrovních. První úrovní jsou požadavky, které má test uvedené v rámci své hlavičky. Zde patří například verze knihovny, podpora vláken, případně některých knihoven jako je například libatomic. Knihovna libatomic a libc++experimental[3] byly z testování vyřazeny. Vyřazení knihovny libatomic je z důvodu, že žádné procesorové jádro Codasip nepodporuje atomické instrukce ani více vláken a testy by tedy zbytečně zabíraly výpočetní zdroje. Knihovna libc++experimental byla po dohodě s vedoucím vyřazena s možností pozdějšího otestování. Stejně tak byly vyřazeny testy vyžadující podporu, kterou plně nepodporují nástroje Codasip nebo knihovna Newlib. Do této skupiny patří testy vyžadující *locale* libovolného typu, sloužící k nastavení lokalizace prostředí, Dále nejsou podporovány testy vyžadující podporu *fenv*, sloužící k nastavení floating point prostředí. Oba druhy byly z testování vyřazeny, neboť aktuální používaná verze standardní knihovny jazyka C neposkytuje jejich podporu.

Poslední nepodporovanou skupinou testů jsou testy spoléhající se na vyvolání signálu *SIGABRT* při volání funkce *abort*. Zde by mohlo dojít k úpravě souboru *crt1.S* příslušného modelu procesoru. Tato úprava by spočívala v načtení konstanty odpovídající signálu *abort* (6) do registru prvního parametru volání. Následně by po načtení došlo k volání funkce *raise*, který vyvolá daný signál. Tento přístup nebyl zvolen, jelikož v případě, že signál *SIGABRT* nemá registrovaný handler k jeho zachycení, tak dojde k volání výchozího handleru. Výchozí handler poté odpovídá volání funkce *kill_r*. Tato funkce využívá systémové volání *kill*, které není v rámci simulátoru Codasip podporováno.

Druhá filtrace se zaměřuje na konkrétní testy, které byly vyřazeny po jejich prozkoumání, kdy například vyžadují podporu operačního systému, případně vyžadují funkcionalitu, kterou jim generované procesorové jádro nemůže poskytnout z důvodu chybějící podpory v rámci nástrojů Codasip, například systémové volání typu *unlink*.

V rámci procházení hlaviček souborů se také sleduje, zda se zde nenachází klíčové slovo *compile*. Tímto slovem jsou označeny parametry, které musí překladač mít pro správné

[2] Mark - v prostředí pytest označuje metody a zavádí některé podmínky, které musí být splněny, aby došlo k vykonání metody.

[3] libc++experimental - jedná se o knihovnu s experimentálním kódem filesystem, která nemá zaručenou stabilitu.

přeložení daného testu. Tyto parametry se načtou a v pozdější fázi skriptu se předají spolu s výchozími parametry překladači. Mezi výchozími parametry jsou knihovny, které se mají v rámci překladu využít a některé definice.

Pokud test prošel fázi filtrace dojde k jeho přeložení. Při překladu se kontroluje koncovka testovaného souboru. V rámci testovací sady se nacházejí testy, které mají koncovku ".pass.cpp" a testy s koncovkou ".fail.cpp". U druhých jmenovaných testů se očekává, že způsobí pád překladače, v opačném případě se jedná o chybu. Pád překladače vyvolá v rámci testovacího skriptu výjimku, která je zachycena a test se označí jako správně provedený, tedy *PASSED*. Pokud fail test nevyvolá výjimku, nedojde ke spuštění, ale k přímému označení testu jako *FAILED* s chybovou hláškou.

Pokud překlad proběhl v pořádku dojde ke spuštění testu. Některé testy mohou vyžadovat vstupní data, která jsou obsažena v rámci souboru s názvem testu a koncovkou ".dat". Tyto soubory se před spuštěním testu kopírují do adresáře, kde se nachází výsledný binární soubor. Testy nemají vždy návratovou hodnotu rovnu 0. Tyto testy poté musí být spuštěny s parametrem simulátoru, který říká že se ignoruje návratový kód testu, jež se kontroluje až po vykonání programu. Po vykonání testu se v posledním kroku kontroluje výstup s referenčním souborem, pokud takový soubor existuje.

V případě příliš velkých testů, jsou takové testy rozděleny na dva běhy, které jsou rozlišeny makrem *SECOND_RUN*, jež testu říká, která část se má přeložit.

Celé testovací prostředí se integruje do nástrojů Codasip studio pouhým nakopírováním k ostatním testům v rámci repositáře *zero-tolerance-check*.

9.2 Výsledky testování

Testování bylo provedeno na jádře Codasip uRISC s velikostí slova 32b a Codix Berkelium. Architektura Codix Berkelium odpovídá standardu RISC-V, v rámci této architektury je možnost využití různých konfigurací. Základní konfigurace obsahuje pouze *I*, které udává, že se jedná o základní procesorové jádro bez dalších rozšíření. Mezi rozšíření patří například *M* pro násobení a dělení, *F* pro práci s čísly s plovoucí desetinnou čárkou, *C* pro komprimované instrukce apod. Procesorové jádro Codix Berkelium plně podporuje jak 32b šířku slova, tak 64b šířku.

Z časových důvodů byla zvolena pro testování základní architektura Codix Berkelium s konfigurací RV32I. Pro tuto cílovou architekturu musel být rozšířen soubor *Registers.hpp* v knihovně libunwind, kde byla přidána třída reprezentující registrové pole pro Codix Berkelium. Stejně tak musela být knihovna libunwind rozšířena, po vzoru architektury Codasip uRISC, o funkce *JumpTo* a *unw_getcontext* pro uložení a obnovení kontextu pro výjimky typu DWARF.

SjLj exception handling

Následující tabulka 9.1 zobrazuje výsledky testování při SjLj zachytávání výjimek.

Architektura	Šířka slova	Prošlo	Neprošlo	Přeskočeno	Očekávané pády
Codasip uRISC	32	26850	0	8580	24
Codix Berkelium	32	26766	84	8580	24

Tabulka 9.1: Výsledky testování pro SjLj zachytávání výjimek.

Z tabulky je patrné, že testy na architektuře Codasip uRISC prošly bez neočekávaných pádů. Mezi 24 očekávaných pádů patří testy využívající signál *SIGABRT*, který není nastaven, jak bylo popsáno výše.

Architektura Codix Berkelium dosahuje o něco horších výsledků, kdy 84 testů neprošlo testováním. Po bližším prozkoumání jsem zjistil, že se jedná o testy, které v rámci překladu neprošly fází linkování. Nástroj linker u těchto testů končí s chybou, značící problém s relokacemi. Tento problém může být způsoben velikostí testů, kdy se symbol pro relokaci nedaří zapsat do některé z instrukcí modelu, například z důvodu malé bitové šířky pro načtení konstanty u instrukce.

DWARF exception handling

Zde tabulka 9.2 zobrazuje výsledky testování při DWARF zachytávání výjimek.

Architektura	Šířka slova	Prošlo	Neprošlo	Přeskočeno	Doba vykonání
Codasip uRISC	32	26844	6	8580	24
Codix Berkelium	32	26792	58	8580	24

Tabulka 9.2: Výsledky testování pro Dwarf zachytávání výjimek.

U výjimek typu DWARF došlo k neočekávanému pádu aplikace u Codasip uRISC architektury. Tento pád je způsoben vnitřní implementací knihovny nebo překladače, neboť se tato chyba projevuje i na referenční architektuře x86_64.

U architektury Codix Berkelium jsou stejně jako v předchozím případě testy, které neprošly překladem z důvodu problému s relokacemi. Kromě těchto testů prošly všechny testované aplikace.

Kapitola 10

Závěr

V rámci diplomové práce jsem se seznámil s jazykem C++, jeho standardem C++11 a především s řešením výjimek a RTTI v rámci tohoto jazyka. Dále jsem se podrobněji seznámil s projektem LLVM, jeho vnitřní reprezentací v podobě LLVM IR a vývojovým prostředím Codasip Studio. Tato práce rozšiřuje vývojové prostředí Codasip Studio o podporu překladu aplikací v jazyce C++, bez návaznosti na některý z předchozích projektů řešených v rámci diplomových nebo bakalářských prací.

Tato práce se zabývala přidáním podpory překladu jazyka C++ do nástrojů Codasip tak, aby bylo možné generovat překladač jazyka C++ s jeho knihovnami bez většího zásahu uživatele využívajícím tyto nástroje. Toho se podařilo dosáhnout integrací implementace do nástrojů Codasip. Integrace probíhala v několika krocích, kdy prvním krokem byla implementace zaměřená na procesorové jádro Codasip uRISC. V rámci této implementace byly implementovány builtin funkce *__builtin_setjmp* a *__builtin_longjmp*, které jsou dále využívány při překladu knihoven a volání výjimek. Také byla implementace rozšířena o podporu výjimek typu Set Jump Long Jump a výjimek využívajících ladící informace. U prvního jmenovaného typu výjimek proběhla implementace především na úrovni zadní části překladače, kdežto u druhého typu výjimek se implementace odehrávala především v programu assembler, který je rovněž jako překladač součástí nástrojů Codasip.

Druhým krokem byl překlad samotné standardní knihovny jazyka C++ libc++ a jejich podpůrných knihoven v podobě knihovny libc++abi a libunwind. V tomto kroku kromě samotného překladu knihoven bylo nutné udělat i několik nezbytných změn v rámci standardní knihovny jazyka C Newlib. Tyto změny se týkaly především konfigurace knihovny, rozšířením některých funkcí o nastavení signálů *errno* a úpravě typu *ctype*, jak na straně knihovny jazyka C, tak na straně standardní knihovny jazyka C++.

Posledním krokem byla integrace samotná, kdy došlo k přenesení implementace v rámci uživatelských souborů do šablon pro generování překladače jazyka C/C++. To vyžadovalo nahrazení kódu závislého na architektuře za kód, který bude možné využít na libovolné architektuře. Při nahrazování kódu bylo využito generátoru backendu, který byl rovněž rozšířen o metody pro získání potřebných informací z cílové architektury tak, aby mohlo dojít k bezproblémovému vygenerování zdrojových kódu překladače. Dále bylo nutné přidat podporu do nástrojů Codasip pro překlad daných knihoven z Codasip comandline. Na závěr byla integrována testovací sada knihovny libc++ spolu s vytvořenými testy, kdy v rámci integrace bylo vytvořeno testovací prostředí pro jednoduchou obsluhu.

Po integraci knihoven a její testovací sady bylo možné knihovnu otestovat na kompletní testovací sadě. Pro testování byl zvolen model Codasip uRISC a Codix Berkelium, kdy na obou architekturách byly testovány oba druhy výjimek. U prvního jmenovaného mo-

delu proběhlo testování téměř bez problémů, jediným problémem je test u výjimek typu DWARF, ten ale vykazuje stejné chování jako referenční architektura x86_64. U druhého jmenovaného modelu se podařilo dosáhnout téměř stejného, kdy jedinou výjimku tvoří testy, které neprošly fází linkování z důvodu příliš velkých adres pro instrukční sadu modelu.

Na základě dosažených výsledků je možné knihovny využívat v rámci Codasip nástrojů na libovolném procesorovém jádře.

Možným rozšířením implementace je podpora sekce *.eh_frame_hdr*, která se nevytvoří v rámci programu linker a přidáním podpory pro knihovnu libc++experimental. Dalším možným rozšířením může být přidání podpory pro *locale* libovolného typu a *fenv*, které jsou ovlivněny aktuální verzí knihovny Newlib.

Literatura

[1] *Codasip Compiler Generation Tutorial*. [Online; Verze 7.2.2].

[2] *Codasip Studio User Guide*. [Online; Verze 7.2.2].

[3] *Polly 6.0-devel documentation*. [Online; navštíveno 3.1.2019].
URL https://polly.llvm.org/docs/Architecture.html

[4] *Structured Exception Handling*. [Online; navštíveno 26.11.2018].
URL https://docs.microsoft.com/en-us/windows/desktop/debug/structured-exception-handling

[5] *LLVM Language Reference Manual*. Leden 2018, [Online; navštíveno 2.1.2019].
URL https://llvm.org/docs/LangRef.html

[6] Brailovsky, N.: *C++ exception handling internals*. [Online; navštíveno 25.11.2018].
URL https://monoinfinito.wordpress.com/series/exception-handling-in-c/

[7] de Dinechin, C.: *C++ Exception Handling for IA-64* . [Online; navštíveno 25.11.2018].
URL http://static.usenix.org/events/osdi2000/wiess2000/full_papers/dinechin/dinechin_html/

[8] Lattner, C.: *LLVM* . [Online; navštíveno 25.11.2018].
URL http://www.aosabook.org/en/llvm.html

[9] Lattner, C.: *Introduction to the LLVM Compiler Infrastructure*. 2006, [Online; navštíveno 3.1.2019].
URL https://llvm.org/pubs/2006-04-25-GelatoLLVMIntro.pdf

[10] Lattner, C.; Adve, V.: *The LLVM Instruction Set and Compilation Strategy*. Srpen 2002, [Online; navštíveno 27.12.2018].
URL https://llvm.org/pubs/2002-08-09-LLVMCompilationStrategy.pdf

[11] Nosterský, M.: *Vytvoření modelu procesoru RISC-V* . FIT VUT v Brně, 2016, [bakalářská práce].

[12] Peringer, P.: *Seminář C++* . [Online; navštíveno 25.11.2018].
URL http://www.fit.vutbr.cz/study/courses/ICP/public/Prednasky/ICP.pdf

[13] Prata, S.: *C++ Primer Plus*. Developer's library, 2011, ISBN ISBN 0-321-77640-2.

[14] Putano, B.: *Most Popular and Influential Programming Languages of 2018*. 2017, [Online; navštíveno 26.12.2018].
URL https://stackify.com/popular-programming-languages-2018/

[15] Sarda, S.; Pandey, M.: *LLVM Essentials*. Packt Publishing, 2015, ISBN 1785280805, 9781785280801.

[16] Srinivasaraghavan, R.: *Exploring the DWARF debug format information* . [Online; navštíveno 25.11.2018].
URL
https://www.ibm.com/developerworks/aix/library/au-dwarf-debug-format/

[17] Stroustrup, B.: *A History of C++:1979-1991*. [Online; navštíveno 27.12.2018].
URL http://www.stroustrup.com/hopl2.pdf

[18] Stroustrup, B.: *Evolving a language in and for the real world: C++ 1991-2006*. [Online; navštíveno 27.12.2018].
URL http://www.stroustrup.com/hopl-almost-final.pdf

Příloha A

Set Jump Long Jump zachytávání výjimek

```
$_Z2ehv: // @_Z2ehv
$func_begin0:
  .cfi_return_column 3
// %bb.0: // %entry
       r0 = addi r0, -104
       st r3, [ r0 + 100 ]
       st r1, [ r0 + 96 ]
       r1 = addi r0, 96
       r2 = movsi 4
       r4 = addi r2, 0
       call $__cxa_allocate_exception
       r2 = movsi 0
       st r2, [ r4 + 0 ]
       r3 = movsi $__gxx_personality_sj0 & 0xffff
       r3 = movhi $__gxx_personality_sj0 >> 16 & 0xffff
       r5 = addi r1, -56
       st r3, [ r5 + 24 ]
       r3 = movsi GCC_except_table0 & 0xffff
       r3 = movhi GCC_except_table0 >> 16 & 0xffff
       st r3, [ r5 + 28 ]
       r3 = addi r1, 0
       st r3, [ r5 + 32 ]
       r3 = addi r0, 0
       st r3, [ r1 + -16 ]
       r3 = movsi LBB0_7 & 0xffff
       r3 = movhi LBB0_7 >> 16 & 0xffff
       st r3, [ r1 + -20 ]
       r3 = movsi 1
       st r3, [ r5 + 4 ]
       st r4, [ r1 + -76 ] // 4-byte Folded Spill
       r4 = addi r5, 0
       st r2, [ r1 + -80 ] // 4-byte Folded Spill
```

```
        call $_Unwind_SjLj_Register
$tmp0:
        r2 = movsi $_ZTIi & 0xffff
        r2 = movhi $_ZTIi >> 16 & 0xffff
        r4 = ld [ r1 + -76 ] // 4-byte Folded Reload
        r5 = addi r2, 0
        r6 = ld [ r1 + -80 ] // 4-byte Folded Reload
        nop
        call $__cxa_throw
$tmp1:
        jump LBB0_6
LBB0_1: // %lpad
$tmp2:
        r2 = addi r1, -56
        r2 = ld [ r2 + 8 ]
        r3 = ld [ r1 + -44 ]
        st r2, [ r1 + -64 ]
        st r3, [ r1 + -68 ]
        st r5, [ r1 + -84 ] // 4-byte Folded Spill
        st r4, [ r1 + -88 ] // 4-byte Folded Spill
        jump LBB0_2
LBB0_2: // %catch.dispatch
        r2 = ld [ r1 + -68 ]
        r3 = movsi 1
        r2 = xor r3, r2
        jumpnz r2, LBB0_5
        jump LBB0_3
LBB0_3: // %catch
        r4 = ld [ r1 + -64 ]
        nop
        call $__cxa_begin_catch
        r2 = ld [ r4 + 0 ]
        nop
        st r2, [ r1 + -72 ]
        r2 = ld [ r1 + -72 ]
        nop
        st r2, [ r1 + -60 ]
        call $__cxa_end_catch
        jump LBB0_4
LBB0_4: // %return
        r4 = ld [ r1 + -60 ]
        r2 = addi r1, -56
        st r4, [ r1 + -92 ] // 4-byte Folded Spill
        r4 = addi r2, 0
        call $_Unwind_SjLj_Unregister
        r4 = ld [ r1 + -92 ] // 4-byte Folded Reload
        r0 = addi r1, 0
        r3 = ld [ r0 + 4 ]
```

```
        r1 = ld [ r0 + 0 ]
        r0 = addi r0, 8
        jump r3
        //
LBB0_5: // %eh.resume
        r2 = movsi -1
        st r2, [ r1 + -52 ]
LBB0_6: // %unreachable
LBB0_7: // %entry
        nop
        r2 = ld [ r1 + -52 ]
        r3 = movsi 1 & 0xffff
        r3 = movhi 1 >> 16 & 0xffff
        r3 = sle r3, r2
        st r2, [ r1 + -96 ] // 4-byte Folded Spill
        jumpnz r3, LBB0_9
LBB0_8: // %entry
        r2 = movsi JTI0_0 & 0xffff
        r2 = movhi JTI0_0 >> 16 & 0xffff
        r3 = movsi 2 & 0xffff
        r3 = movhi 2 >> 16 & 0xffff
        r4 = ld [ r1 + -96 ] // 4-byte Folded Reload
        nop
        r3 = sll r4, r3
        r2 = add r3, r2
        r2 = ld [ r2 + 0 ]
        nop
        jump r2
LBB0_9: // %entry
        jump $abort
$func_end0:
```

Ukázka kódu A.1: Zachycení výjimky v jazyce symbolických adres.

Příloha B

DWARF zachytávání výjimek

```
$_Z2ehv: // @_Z2ehv
$func_begin0:
	.cfi_startproc
	.cfi_personality 0, $__gxx_personality_v0
	.cfi_lsda 0, $exception0
  .cfi_return_column 3
// %bb.0: // %entry
	.cfi_def_cfa 0, 0
	r0 = addi r0, -32
	.cfi_adjust_cfa_offset 32
	st r3, [ r0 + 28 ]
	.cfi_offset 3, -4
  .loc 0 0 0 prologue_end
	r2 = movsi 4
	r4 = addi r2, 0
	call $__cxa_allocate_exception
	r2 = movsi 0
	st r2, [ r4 + 0 ]
$tmp0:
	r3 = movsi $_ZTIi & 0xffff
	r3 = movhi $_ZTIi >> 16 & 0xffff
	r5 = addi r3, 0
	r6 = addi r2, 0
	call $__cxa_throw
$tmp1:
	jump LBB0_5
LBB0_1: // %lpad
$tmp2:
	st r4, [ r0 + 16 ]
	st r5, [ r0 + 12 ]
	jump LBB0_2
LBB0_2: // %catch.dispatch
	jump LBB0_3
LBB0_3: // %catch
```

```
        r4 = ld [ r0 + 16 ]
        nop
        call $__cxa_begin_catch
        r2 = ld [ r4 + 0 ]
        nop
        st r2, [ r0 + 8 ]
        r2 = ld [ r0 + 8 ]
        nop
        st r2, [ r0 + 20 ]
        call $__cxa_end_catch
        jump LBB0_4
LBB0_4: // %return
        r4 = ld [ r0 + 20 ]
        r3 = ld [ r0 + 28 ]
        .cfi_restore 3
        r0 = addi r0, 32
        .cfi_def_cfa_register 0
        jump r3
```

Ukázka kódu B.1: Zachycení výjimky v jazyce symbolických adres.

Příloha C

Obsah CD

Na přiloženém CD se nacházejí tyto adresáře a soubory:

- /src/patches - soubory, sloužící k aktualizaci zdrojových kódů Codasip Studio,
- /src/lib - jednotlivé upravené knihovny jazyka C++,
- /tests - testovací sada knihovny jazyka C++,
- /doc - technická zpráva ve formátu pdf, včetně návodu k použití,
- /tex - zdrojové kódy technické zprávy.

Příloha D

Návod

V návodu se předpokládá, že má uživatel aktuální nástroje Codasip s platnou licencí. Návod se vztahuje na architektury zmíněné v diplomové práci, které mají podporu v rámci knihovny libunwind, ostatní architektury mohou využívat pouze výjimky typu Set Jump Long Jump.

Překlad modelu procesorového jádra:

- Úprava konfiguračního souboru modelu:
  - Povolení knihoven jazyka C++ nastavením *unwind*, *cxxabi* a *cxx* na hodnotu *true* v souboru *codal.conf*
  - V případě, že se jedná o model s podporou v rámci libunwind, nastavení *dwarfeh* na hodnotu *true* ve stejném souboru zajistí podporu výjimek využívajících ladící informace DWARF
- Vygenerování *SDK*.

Pro překlad libovolné aplikace v jazyce C++ se využívá překladač clang++. Parametry pro překlad:

- *-fsjlj-exceptions* - výjimky typu Set Jump Long Jump,
- *-fdwarf-exceptions* - výjimky využívající ladící informace DWARF,
- *-fno-exceptions* - zákaz používání výjimek a
- bez uvedení parametru se použijí výjimky typu DWARF.

Ke spuštění testovací sady se využije příkaz:

```
InstalaceCodasipNastroju/tools/bin/cmdline -m pytest --tb=short \
--self-contained-html -v --model-id=ia --work-dir=./ \
--html=./report.html --sdk=CestaKGenerovanemuSDK \
-p codasip.testsuite.pytest_addoptions -p codasip.testsuite.plugin_base \
-p codasip.testsuite.sdk -p codasip.testsuite.pytest_fixtures \
-p codasip.testsuite.plugin_cache_redirector cestaKCD/tests
```

V případě testování pouze jednoho typu výjimek, lze k testovací sadě dodat parametr *-k 'test_sjlj'* pro výjimky typu Set Jump Long Jump, nebo *-k 'test_dwarf'* pro výjimky využívající ladící informace DWARF.