VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ZPĚTNÝ PŘEKLAD APLIKACÍ PRO ARCHITEKTURU AARCH64 V NÁSTROJI RETDEC

DECOMPILATION OF AARCH64 BINARIES IN RETDEC DECOMPILER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MATEJ KAŠŤÁK**
AUTHOR

VEDOUCÍ PRÁCE — **DUŠAN KOLÁŘ, doc. Dr. Ing.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Kašťák Matej**
Program: Informační technologie
Název: **Zpětný překlad aplikací pro architekturu AArch64 v nástroji RetDec**
Decompilation of AArch64 Binaries in RetDec Decompiler
Kategorie: Překladače

Zadání:

1. Studujte problematiku reverzního inženýrství. Zaměřte se na zpětný překlad binárního kódu do vyšší formy reprezentace.
2. Seznamte se s architekturou procesorů AArch64 (ARM64).
3. Seznamte se se zpětným překladačem RetDec společnosti Avast a s technologiemi v něm použitými (LLVM, Capstone, Keystone atd.).
4. Analyzujte současné překážky zpětného překladu aplikací pro AArch64. Pro vylepšení podpory překladu těchto binárních souborů identifikujte komponenty, které je nutné rozšířit, nebo nově implementovat.
5. Navrhněte nutná rozšíření a nové komponenty pro dosažení cíle z bodu 4.
6. Po konzultaci s vedoucím a konzultantem implementujte návrhy z předchozího bodu.
7. Vytvořené řešení důkladně otestujte sadou testů, včetně reálných programů pro danou architekturu, nebo vzorků potenciálně škodlivých programů.
8. Zhodnoťte svou práci a diskutujte budoucí vývoj.

Literatura:

- E. Eilam: Reversing: Secrets of Reverse Engineering, Wiley 2005, ISBN 978-076457481.
- ARM Architecture Reference Manual.
- Dokumentace k projektům RetDec, LLVM, Capstone, Keystone atd.
- Dle doporučení vedoucího či konzultanta.

Pro udělení zápočtu za první semestr je požadováno:

- Prvních pět bodů zadání a rozpracování šestého bodu

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

Vedoucí práce: **Kolář Dušan, doc. Dr. Ing.**
Konzultant: Matula Peter, Ing., Avast
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 24. října 2018

Abstrakt

Cieľom tejto práce je navrhnúť a implementovať spätný prekladač pre architektúru AArch64. Práca najprv uvedie koncept reverzného inžinierstva, následne všeobecne analyzuje platformu procesorov ARM a architektúru spätného prekladača RetDec od firmy Avast. V ďalších kapitolách je popísaný návrh a implementácia modulu pre RetDec. Výstupom modulu je preklad strojového kódu do LLVM inštrukcií, ktoré sú následné spracované priechodmi LLVM. Toto vedie k výslednému prekladu do vyššieho jazyka.

Abstract

The goal of this thesis is to propose and implement a decompiler for the AArch64 architecture. The thesis firstly introduces the concept of reverse engineering, then analyzes the ARM processor platform and architecture of RetDec decompiler from Avast company. In the next chapters, we describe the design and implementation of a module for RetDec. The purpose of this module is to decompile machine code into LLVM IR instructions which are further processed by LLVM passes. This leads to decompilation to a higher level language.

Kľúčové slová

ARM64, AArch64, ARMv8, reverzné inžinierstvo, spätný preklad, LLVM, Capstone, RetDec

Keywords

ARM64, AArch64, ARMv8, reverse engineering, decompilation, LLVM, Capstone, RetDec

Citácia

KAŠŤÁK, Matej. *Zpětný překlad aplikací pro architekturu AArch64 v nástroji RetDec*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Dušan Kolář, doc. Dr. Ing.

Zpětný překlad aplikací pro architekturu AArch64 v nástroji RetDec

Prehlásenie

Prehlasujem, že túto bakalársku prácu som vypracoval samostatne pod vedením pána doc. Dr. Ing. Dušana Koláře. Ďalšie informácie mi poskytol Ing. Peter Matula. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Matej Kašťák
15. mája 2019

Poďakovanie

Ďakujem vedúcemu práce pánovi doc. Dr. Ing. Dušanovi Kolářovi za odborné vedenie a venovaný čas. Ďalej by som sa chcel poďakovať Ing. Petrovi Matulovi za ústretovosť a jeho rady.

Obsah

1 Úvod **3**

2 Reverzné inžinierstvo **5**
2.1 Definícia . . . 5
2.2 Analýza programov . . . 5
2.3 Nástroje . . . 6
2.3.1 Monitorovanie systému . . . 6
2.3.2 Disassembler . . . 6
2.3.3 Debugger . . . 6
2.3.4 Spätný prekladač . . . 7
2.3.5 Súčasné nástroje . . . 7

3 Architektúra ARM64 **8**
3.1 Stav vykonávania AArch64 . . . 8
3.2 Registre . . . 9
3.3 Register príznakov . . . 10
3.4 Sada inštrukcií . . . 11
3.4.1 Podmienený kód . . . 11
3.5 Adresovacie módy . . . 11
3.6 Význam registrov pri volaní funkcií . . . 12
3.7 Systémové volania . . . 13

4 Spätný prekladač RetDec **15**
4.1 LLVM . . . 15
4.2 Jazyk LLVM assembly language . . . 15
4.2.1 Identifikátory . . . 15
4.2.2 Globálne premenné . . . 16
4.2.3 Funkcie . . . 16
4.2.4 Dátová schéma . . . 16
4.2.5 Dátové typy . . . 16
4.2.6 Konštanty . . . 17
4.2.7 Metadáta . . . 17
4.2.8 Inštrukcie . . . 17
4.3 Capstone – disassembler . . . 18
4.4 Architektúra programu RetDec . . . 19
4.4.1 Predspracovanie . . . 20
4.4.2 Jadro . . . 20
4.4.3 Zadná časť . . . 21

4.5 Keystone – assembler . . . 22

5 Návrh implementácie **23**
5.1 Identifikovanie nutných rozšírení . . . 23
5.2 Modelovanie registrov . . . 24
5.3 Modelovanie zmien toku vykonávania . . . 25
5.4 Modelovanie inštrukcií . . . 25
5.5 Spracovanie operandov . . . 26
5.5.1 Operandy s bitovým posunom . . . 26
5.5.2 Rozšírenie operandov . . . 27
5.6 Modelovanie systémových volaní . . . 27
5.7 Príklady prekladu inštrukcií . . . 27
5.8 Insrunner . . . 28
5.9 ELF importované funkcie . . . 29

6 Implementácia **31**
6.1 Knižnica Capstone2LlvmIr . . . 31
6.1.1 Modul prekladača architektúry ARM64 . . . 32
6.2 Systémové volania . . . 33
6.3 Insrunner . . . 33
6.4 Importované funkcie vo formáte ELF . . . 34

7 Testovanie **35**
7.1 Jednotkové testy . . . 35
7.2 Generovanie programov pre ARM64 . . . 36
7.3 Regresné testy . . . 36
7.4 Nočné testy . . . 37
7.5 QEMU virtuálne stroje . . . 38
7.6 Počítače Raspberry PI . . . 38
7.7 Výsledky testov . . . 38

8 Výsledky implementácie **39**

9 Záver **41**

Literatúra **42**

A Podmienené vykonávanie na AArch64 **43**

B Význam inštrukcií používaných v texte **44**

C Metriky projektu **45**

D Obsah pamäťového média **46**

Kapitola 1

Úvod

Stáva sa bežným, že všetky zariadenia okolo nás obsahujú procesor. Na tieto zariadenia sú kladené veľké nároky z oblasti spotreby, veľkosti a rýchlosti. S jedným z možných riešení prichádzajú procesory s architektúrou ARM. Sú to procesory s obmedzenou inštrukčnou sadou, čo v praxi znamená, že môžu byť implementované na oveľa menšej ploche, pri vyťažení neprodukujú veľa tepla a majú mnohé iné výhody.

Tieto vlastnosti z nich robia vhodných kandidátov pre ľahké zariadenia napájané batériou. V súčasnosti ide hlavne o mobilné telefóny, tablety a vstavané systémy. Na trhu sa rozširuje zastúpenie architektúry ARM aj v niektorých radoch laptopov. Do roku 2017 bolo vyrobených 100 miliárd čipov tejto architektúry pričom len za rok 2016 ich bolo vyrobených skoro 18 miliárd[1].

Hrozba kybernetických útokov sa neustále zvyšuje. S narastajúcou komplexnosťou programov je veľmi časté, že sa v počítačových systémoch nachádzajú chyby. Útočníci tieto chyby hľadajú a snažia sa ich využiť na ovládnutie zariadení. Píšu preto programy, ktoré nazývame *malvér*. Tieto programy často vyzerajú na prvý pohľad neškodne, no obsahujú kód, ktorý dokáže prevziať kontrolu nad zariadením. Architektúra ARM svojim zastúpením dominuje trh s mobilnou výpočetnou technikou, preto sa stáva častým cieľom útočníkov. To podnecuje vznik nástrojov, ktoré pomáhajú vyhnúť sa alebo v budúcnosti odvrátiť určité typy útokov.

Keďže v minulosti neexistovali kvalitné spätné prekladače, bolo nutné analyzovať škodlivý kód kontrolou strojového kódu. Táto činnosť bola ale veľmi zdĺhavá a ukazuje sa, že pre ľudí je oveľa jednoduchšie analyzovať kód vo vyššom programovacom jazyku. Preto začali vznikať rôzne spätné prekladače, ktorých úlohou je transformovať strojový kód do formy lepšie zrozumiteľnej pre človeka.

Spätný prekladač je počítačový program, ktorý prijíma na vstupe preložené binárne súbory obsahujúce inštrukcie v strojovom kóde. Tieto súbory sú spracované a výsledkom je typicky kód vo vyššom programovacom jazyku (väčšinou C, C++, atď.).

V súčasnosti existuje viacero spätných prekladačov. Najznámejším je pravdepodobne komerčný program Hex-rays decompiler[2]. Ďalším novým prírastkom je program Ghidra vyvíjaná americkou bezpečnostnou agentúrou NSA. Ďalej existujú rôzne iné programy ako napríklad Hopper, Radeco či Snowman. Problémom týchto implementácií je, že väčšina z nich nie je open-source alebo ich výsledky nie sú veľmi presné. Ďalším problémom často býva obmedzenie podporovaných architektúr. Program RetDec svojím modulárnym návrhom do-

[1] https://group.softbank/en/corp/d/annual-reports/2017/future-forward/segars-interview/
[2] https://www.hex-rays.com/products/ida/index.shtml

voľuje jednoducho rozšíriť základný prekladač o nové podporované architektúry. Programy prekladá najprv do tzv. *medzikódu*, čo je jednotný zápis, na ktorom sa ďalej vykonávajú všetky ďalšie kroky spätného prekladu. Tieto kroky predstavujú optimalizáciu a pretváranie kódu do podoby, kedy je jednoduchšie spätné prekladanie. Pomocou týchto optimalizácií je možné získať zrozumiteľný výsledok aj v prípade obfuskovaného kódu. Cieľom práce je rozšíriť možnosti prekladu o 64-bitovú architektúru ARM.

Na začiatku práce popíšeme architektúru ARM ako takú, pričom sa zameriame na jej stav vykonávania AArch64. Preberieme možnosti tejto architektúry, využívané konvencie a jej aplikačno binárne rozhranie. V ďalšej kapitole rozoberieme spätný prekladač RetDec, kde popíšeme jeho architektúru, relevantné moduly a technológie, na ktorých je založený. Neskôr bude popísaný návrh, ktorý je základom pre implementáciu práce. Nasledovať bude popis samotnej implementácie práce. Ku koncu analyzujeme zvolené postupy, výsledky overíme testovaním a na záver zhodnotíme prácu a finálnu implementáciu.

Kapitola 2

Reverzné inžinierstvo

Nasledujúca kapitola slúži ako úvod do problematiky reverzného inžinierstva. Pojednáva o postupoch, technikách a nástrojoch využívaných v tomto obore. V texte uvedieme spôsoby analyzovania kódu, legálnu stránku veci a prehľad používaných nástrojov. Na koniec predstavíme a porovnáme programy, ktoré je možné v súčasnosti využiť pri práci s architektúrou ARM64. Časti tejto kapitoly sú založené na informáciách zo zdroja [10].

2.1 Definícia

Reverzné inžinierstvo (z anglického *reverse engineering*) je proces, pri ktorom sa snažíme zo všetkých dostupných informácií o danom objekte získať predstavu o tom, ako bol vyrobený alebo ako vnútorne fungujú jeho časti. Reverzné inžinierstvo je používané v mnohých vedných disciplínach a jeho výsledky často bývajú veľkým prínosom. Je ale bežné aj zneužívanie pri firemnej špionáži alebo iných nelegálnych aktivitách. Obrovské využitie má aj v oblasti hardvéru a softvéru, napríklad pri verifikácii návrhu.

2.2 Analýza programov

Analýza programu je skúmanie, čo program vykonáva počas svojho behu. Niekedy to nemusí byť triviálna úloha, napríklad v prípadoch keď medzi programom a človekom neprebiehajú žiadne interakcie. V niektorých prípadoch je dokonca možné, že aj samotní autori sa snažia zabrániť tomu, aby niekto analyzoval ich programy a pridávajú preto rôzne ochranné mechanizmy. Programy je možné podrobiť viacerým typom analýz. Najčastejšie je rozdelenie na statickú a dynamickú analýzu.

Statická analýza
: Je skúmanie kódu a programov bez toho, aby sme ich spúšťali. Toto skúmanie je relatívne bezpečné a nemusíme riskovať spúšťanie škodlivého kódu. Statickú analýzu vykonávajú programy aj ľudia, ktorí hľadajú známe vzory bezpečnostných rizík alebo programovacích chýb.

Dynamická analýza
: Je analýza programov, kedy pri ich spustení sledujeme prístup k zdrojom, čítanie a zápis do pamäti a iné ukazovatele, ktoré nastávajú počas behu. Pre analýzu nie sú potrebné informácie o tom ako aplikácia funguje ale zameriava sa len na výsledky behov

programu. Samotné spustenie kódu prebieha typicky v monitorovaných a izolovaných prostrediach.

2.3 Nástroje

Reverzné inžinierstvo je veľmi závislé od použitých nástrojov. Komplexnosť týchto programov sa neustále zvyšuje a veľa úloh sa dá v rozumnej miere automatizovať. Špecializované programy zrýchľujú analýzu a ponúkajú rozličné možnosti spracovania dát. Do reverzného inžinierstva sa stále viac a viac dostáva strojové učenie, ktoré je schopné analyzovať obrovské množstvo programov. Firmy preto investujú nemalé čiastky do vývoju týchto nástrojov.

Niektoré z nich sú vyvíjané pod open-source licenciami, ako napríklad RetDec od firmy Avast. No v iných prípadoch sa stretávame s tým, že ich cena sa pohybuje v tisícoch dolárov za licenciu.

V nasledujúcej kapitole popíšeme základné druhy nástrojov používaných reverznými inžiniermi a preberieme súčasný stav spätných prekladačov zameraných na architektúru ARM.

2.3.1 Monitorovanie systému

Operačný systém poskytuje rozhranie pre komunikáciu programu a používateľa či iných zariadení. Vždy keď chce program vykonať operáciu, ktorá nejakým spôsobom zmení stav systému, musí to vykonať prostredníctvom volaní operačného systému. Monitorovacie programy sledujú toto rozhranie a zhromažďujú štatistiky o jeho využívaní. Dokážu monitorovať operácie so súbormi, činnosť siete a iné. Typicky sa zameriavajú na jeden program, a používateľovi prezentujú informácie o jeho behu.

2.3.2 Disassembler

Hlavnou úlohou disassembleru je zobrazovanie inštrukcií vo forme binárnych dát do ich textovej reprezentácie (jazyku symbolických inštrukcií). Každá architektúra má špecifický postup a schému dekódovania inštrukcií. Tento proces je často základom analýzy programov. Disassemblery sú v celku jednoduché programy a odlišujú sa hlavne počtom podporovaných architektúr. Príkladom takého disassembleru je `Capstone`[1].

2.3.3 Debugger

Debugger je nástroj, ktorý slúži na ladenie behu programu. Programy spúšťa v monitorovanom prostredí. Dovoľuje používateľovi preskúmavať interné štruktúry sledovaného programu a spravovať jeho beh. Pri jeho používaní sa využívajú tzv. breakpointy, čo sú označené miesta kódu. Keď sa vykonávanie dostane na takéto miesto, tak sa beh programu dočasne pozastaví a riadenie sa predá používateľovi. Ten v tomto okamihu môže preskúmať pamäť, premenné, stav registrov, atď. Následne má možnosť nechať program pokračovať alebo vykonávať tzv. krokovanie, čo znamená predanie kontroly používateľovi po každej vykonanej inštrukcii.

Prekladače sú pri preklade a zostavovaní programu schopné vygenerovať ladiace informácie do výstupných súborov. Tieto informácie slúžia na zlepšenie procesu ladenia. Obsahujú

[1] http://www.capstone-engine.org/

dáta potrebné vytvoreniu mapovania medzi strojovým a zdrojovým kódom. Pri ich použití je možné automaticky zisťovať obsah premenných či krokovať riadky zdrojového kódu.

Debuggery sú často využívané programátormi pri vývoji softvéru. Za ich pomoci zisťujú chyby v programoch. Vďaka nim je možné zistiť miesto chyby a podmienky, za ktorých sa daná chyba prejaví. Debuggery sú používané taktiež aj v reverznom inžinierstve, a dovoľujú presne sledovať vykonávanie programu. Majú vstavaný disassembler a sú schopné pracovať na úrovni práve vykonávaného strojového kódu. Medzi najpoužívanejšie a najznámejšie debuggery patrí program `GDB`.

2.3.4 Spätný prekladač

Spätný prekladač podobne ako disassembler zobrazuje binárne dáta (programy) do zrozumiteľnejšej formy pre ľudí. Výsledkom je odhad zdrojového kódu použitého pri preklade. Cieľom je previesť programy na vstupe na kód vo vysoko-úrovňovom jazyku. Jazyk výsledku sa nemusí nevyhnutne zhodovať s jazykom použitým pri vývoji. Dôležité je ale zanechať význam výsledného kódu čo najpodobnejší inštrukciám, z ktorého vznikol. Rôzne spätné prekladače sa líšia počtom architektúr, ktoré podporujú. Ďalej to môže byť napríklad kvalitou spätného prekladu.

Veľká časť informácii sa pri preklade zdrojových kódov zahadzuje, pretože vo výsledku už nie sú potrebné. Sú to napríklad mená a typy premenných. To významne sťažuje prácu spätným prekladačom. Moderné spätné prekladače sú ale čoraz viac schopné tieto problémy riešiť. Používajú rôzne metódy, ktoré napomáhajú odhadnúť alebo odvodiť aspekty pôvodného kódu.

2.3.5 Súčasné nástroje

V posledných rokoch sa počet spätných prekladačov zvýšil. Jednou z príčin je pravdepodobne narastajúca dôležitosť kybernetickej bezpečnosti. V nasledujúcom texte predstavím niekoľko najvýznamnejších s ich krátkym popisom.

RetDec je spätný prekladač strojového kódu založený na LLVM. V minulosti bol dostupný ako webová služba, no v decembri 2017 boli jeho zdrojové kódy zverejnené pod open-source licenciou. Jeho návrh umožnuje pomerne jednoduché rozšírenie podporovaných architektúr. V súčasnom stave nepodporuje architektúru ARM64.

Hex-Rays Decompiler je zrejme najznámejším spätným prekladačom. Funguje ako zásuvný modul do programu IDA (Interactive disassembler). Podporuje viacero architektúr procesorov, vrátane ARM64. Jeho hlavnou nevýhodou je vysoká cena.

Ghidra je nástroj špecializovaný na reverzné inžinierstvo vyvíjaný agentúrou NSA. Pre verejnosť bol vydaný na začiatku roku 2019, pričom boli zverejnené aj jeho zdrojové kódy. Obsahuje pomerne dobrú podporu pre architektúru ARM64. Používateľom dovoľuje spolupracovať na analýze programov.

Hopper je spätný prekladač, ktorý bol pôvodne vyvinutý pre platformu MacOS. Má vstavanú podporu pre ARM64. Ponúka používateľsky prívetivé rozhranie. V porovnaní s ostatnými platenými nástrojmi je cenovo dostupný.

Radeco je spätný prekladač, práve vo vývoji. Jeho výsledky sú stále veľmi experimentálne. Je naprogramovaný v jazyku Rust.

Kapitola 3

Architektúra ARM64

Architektúra ARM je architektúra počítačov s obmedzenou sadou inštrukcií (RISC), z anglického slovného spojenia *Reduced Instruction Set Computer*. Hlavnými charakteristikami týchto procesorov sú:

- Na implementáciu je typicky potrebných menej tranzistorov, čo výrazne znižuje plochu procesora a necháva priestor pre iné komponenty.
- Pri správnych podmienkach priemerná doba spracovania jednej inštrukcie sa rovná jednému taktu procesoru.
- Procesor väčšinou pracuje s jednotným formátom inštrukcií.
- Procesor podporuje jednoduché adresovacie módy.

Táto architektúra je neustále aktívne vyvíjaná a v tejto práci sa zameriame na profil architektúry `ARMv8-A`. Ten je popísaný v referenčnom manuáli [8]. Architektúra je typu *load/store* čo znamená, že operácie ktoré spracúvajú dáta môžu pracovať len s obsahom registrov a nikdy nie priamo s pamäťou. Hlavným prínosom takéhoto prístupu je, že znižuje komplexnosť implementácie, no dáta musia byť vždy najprv načítané do registrov. Moderné prekladače sa snažia preto minimalizovať počet prístupov do pamäte, aby čo najviac znížili dopad na rýchlosť programov.

Spoločnosť ARM vo svojich manuáloch definuje `PE`, čo je tzv. *processing element*. Je to abstraktný stroj, ktorého chovanie v tejto práci popíšeme. Dôvodom je, že `Arm` licencuje firmy pre použitie ich architektúry. Implementácia výsledného čipu je už v réžii firmy, ktorá si túto špecifikáciu zakúpila.

3.1 Stav vykonávania AArch64

AArch64 je jedným z dvoch stavov vykonávania (anglicky *Execution state*). Pracuje v 64-bitovom režime, pričom na uloženie adresy sú používané 64-bitové registre. Zároveň inštrukcie v základnej sade môžu používať 64-bitové registre pri spracovávaní dát. `AArch64` implementuje inštrukčnú sadu s názvom `A64`. V práci bude ďalej používané slovné spojenie „architektúra ARM64“ čím sa myslí profil architektúry ARMv8-a, konkrétne stav vykonávania AArch64.

Druhý stav vykonávania AArch32 je prítomný v `ARMv8-A` z dôvodov spätnej kompatibility medzi jednotlivými verziami procesorov. Implementuje inštrukčné sady `A32` a `T32`.

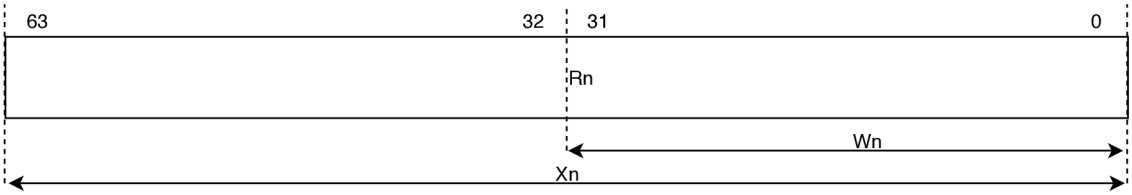
V jednej aplikácii ale nie je možné vykonávať kód z oboch stavov vykonávania. Kód napísaný pre AArch64 nie je možné spúšťať na starších verziách týchto procesorov. Naproti tomu kód pre procesory z rady `ARMv7-A` je možné spúšťať v `ARMv8-A` pomocou stavu vykonávania `AArch32`.

3.2 Registre

Stav vykonávania AArch64 popisuje viacero skupín dostupných registrov. Sú to:

- Registre všeobecného využitia
- FP&SIMD registre
- Systémové registre

AArch64 ponúka 31 registrov všeobecného využitia. Registre sú označované identifikátormi od `R0` po `R30`. K celému obsahu registru, teda k 64-bitom je možné pristupovať pomocou mena registru `Xn`, kde `n` je číslo registru. Obdobne je možné pristúpiť ku spodným 32-bitom pomocou mena `Wn`, kde `n` je číslo registru. Posledné dva registre s označením `R29` a `R30` majú špeciálnu úlohu. Register `X29` sa používa ako frame pointer a typicky uchováva adresu aktuálneho zásobníkového rámca. Posledný register `X30` je používaný ako link register, pre uchovanie návratovej adresy pri volaní funkcií.



Obr. 3.1: Označenie registrov všeobecného využitia.

Osobitné postavenie má register `XZR`, ktorého používanie má pozmenenú sémantiku. Register neumožňuje uloženie ani načítanie hodnoty.

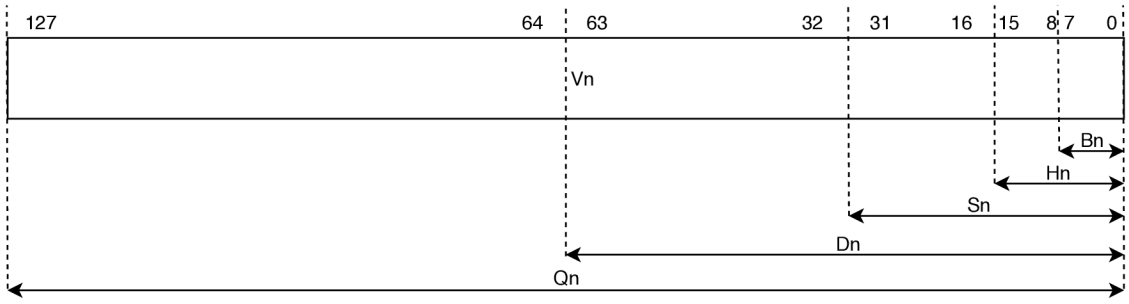
- Čítanie hodnoty vygeneruje načítanie hodnoty 0.
- Zápis hodnoty zodpovedá k zahodeniu hodnoty.

Využíva sa hlavne pri inicializovaní premenných na hodnotu 0 a často je takéto priradenie preferované pred načítaním konštanty.

AArch64 disponuje veľkým počtom systémových registrov, ktorých úlohou je napríklad konfigurácia `PE`, ladenie, prístup k časovačom alebo získanie stavu procesoru. Tieto registre sú ale často prístupné iba v privilegovaných režimoch. V mnohých prípadoch je ich meno zakončené najnižšou úrovňou privilégií, ktorá je nutná pre prístup k nim.

Ďalej AArch64 obsahuje dva samostatné 64-bitové registre, s názvom `SP` (Stack Pointer) a `PC` (Program Counter). Register `SP` sa používa na uchovanie aktuálnej adresy vrcholu zásobníku. K spodným 32-bitom `SP` je možné pristúpiť pomocou `WSP`. Následne `PC` je taktiež 64-bitový register obsahujúci adresu aktuálne vykonávanej inštrukcie. Tento register je špeciálny tým, že neumožňuje inštrukciám priamy zápis hodnoty a jediná možnosť ako ju zmeniť je pomocou vetvenia (anglicky *branching*), vstupu alebo výstupu z obsluhy výnimky.

V neposlednej rade AArch64 definuje 30 registrov využívaných pri výpočtoch s pohyblivou desatinnou čiarkou a vektorových operáciách. V procesoroch sa vyskytujú ako rozšírenia, ktoré sú v súčasnosti pomerne *časté*. Sú to registre s prefixom `Q, D, S, H, B` pričom prefix určuje časť registru, ku ktorej sa pristupuje (podobne ako tomu bolo pri registroch všeobecného využitia). Na obrázku 3.2 je možné vidieť prekrytie a mená týchto registrov.



Obr. 3.2: Označenie vektorových registrov a registrov určených pre čísla s plávajúcou desatinnou čiarkou.

AArch64 pracuje s typmi pre čísla s pohyblivou desatinnou čiarkou popísanými štandardom `IEEE 754` [4, 5]. Doplňuje jeho znenie v prípadoch, ktoré sú štandardom ponechané k vlastnej implementácii. Podporované typy sú: `Double-precision`, `Single-precision` a `Half-precision`, pričom namiesto `Half-precision` je možné využívať alternatívny formát `ARM alternative half-precision` formát. Ten vychádza z formátu `IEEE` ale upravuje jeho znenie pri hodnotách *Inf* a *NaN*. Použitý formát je možné zmeniť nastavením príslušného systémového registra. V súbore sa nachádzajú registre `FPCR` a `FPSR`, ktoré sú určené na nastavovanie chovania pri práci s desatinnými číslami.

3.3 Register príznakov

`PSTATE` je abstrakciou informácií dostupných o procese vykonávania. Obsahuje jednobitové položky, ktoré reprezentujú príznaky alebo popisujú aktuálne nastavenie `PE`. Príznaky sú používané pri podmienenom kóde. Aritmetické operácie nastavujú nasledujúce príznaky:

N (negative) Príznak negatívneho výsledku.

Z (zero) Príznak nulového výsledku.

C (carry) Príznak prenosu na poslednom ráde.

V (overflow) Príznak pretečenia pri aritmetickej operácii.

Ďalej sú v `PSTATE` prítomné informácie o aktuálnej úrovni privilégií a masky prerušení.

Prístup k položkám `PSTATE` je vykonávaný pomocou systémových registrov. Tieto registre sú načítané inštrukciou `MSR` a zapísané použitím `MRS` inštrukcie. Niektoré inštrukcie nastavujú príznaky implicitne iné vyžadujú zapnutie zápisu. Zapnutie prebieha pridaním sufixu '`s`' za operačný kód inštrukcie, ktorá to podporuje. Príkladom je inštrukcia `adds`, ktorá sčíta operandy a následne aktualizuje potrebné príznaky. Jej operačný kód pozostáva zo základu `add` ku ktorému je pridaný sufix '`s`'.

3.4 Sada inštrukcií

Stav vykonávania AArch64 podporuje iba jednu sadu inštrukcií s názvom `A64` [3, 1]. Všetky inštrukcie v tejto inštrukčnej sade majú konštantnú veľkosť 32-bitov. Je nutné aby platilo, že inštrukcie sú zarovnané na veľkosť slova inak bude vyvolaný `Pc Alignment Fault`.

3.4.1 Podmienený kód

Podmieneným kódom nazývame inštrukciu alebo sekvenciu inštrukcií, ktoré sa vykonajú len pokiaľ platí určitá podmienka. Ovplyvniť tok programu je možné na úrovni inštrukcií, kedy ku operačnému kódu inštrukcie je pridaný identifikátor podmienky (kódy jednotlivých podmienok sú popísané v prílohe A.1). Inštrukcia je vykonaná iba pokiaľ platí daná podmienka. Počas takéhoto podmienenia sa hodnota registra `PC` mení len prirodzeným pričítaním veľkosti inštrukcie, teda nie sú vykonávané žiadne skoky.

Na úrovni sekvencie inštrukcií sa tok programu ovláda pomocou inštrukcií skoku. Kód je najprv rozdelený do blokov, a po vyhodnotení podmienky sa pomocou podmienených a nepodmienených skokov predáva vykonávanie. Na riadenie toku programu sú potrebné samotné inštrukcie realizujúce skoky.

AArch32 dovoľoval podmieniť všetky inštrukcie. V AArch64 sa od toho ale upustilo a je možné podmieniť iba určité inštrukcie. V konečnom dôsledku tento návrh viedol k pridaniu väčšieho počtu dostupných registrov, keďže ich bolo možné zakódovať do inštrukcie. Podrobnejšie informácie a výkonnostné testy je možné nájsť v článku [9]. Typicky je možné podmieniť inštrukcie realizujúce skoky. Ďalšie inštrukcie, ktoré je možné podmieniť boli vybrané analýzou vzorov, ktoré vznikajú často pri programovaní. Ako napríklad podmienené priradenie/inkrementovanie alebo ternárny operátor. V niektorých prípadoch je možné ich zápis zredukovať na jednu inštrukciu.

```
1    if (x0 == x1 && x2 == 42) {
2        x2++;
3    }
4    x3 = (x5 == 123) ? x10 : x11;
```

Obr. 3.3: Ukážkový pseudo kód podmieneného príkazu.

Na obrázku 3.3 je uvedený pseudo kód, ktorý je v porovnaní 3.4 reprezentovaný v jazyku symbolických inštrukcií. Vľavo je možné vidieť riadenie toku na úrovni blokov kódu a vpravo je riadenie vykonávané na úrovni inštrukcií. Hlavným rozdielom je počet inštrukcií, čo má typicky dopad na rýchlosť programu. V prílohe B.1 sú vysvetlené inštrukcie používané v príkladoch.

3.5 Adresovacie módy

Adresovací mód je spôsob zápisu adresy ako parametra inštrukcie. Inštrukcie v sade `A64` vyžadujú 64-bitový bázový register. V tabuľke 3.1 sú uvedené dostupné adresovacie módy a ich syntax.

Na inštrukcii `ldr` je demonštrované použitie jednotlivých adresovacích módov. Inštrukcia načíta hodnotu určenú adresovacím módom do registra. Typicky je to hodnota na danej

```
    cmp x0, x1
    bne false
    cmp x2, 42
    bne false
    // telo podmieneneho prikazu
    add x2, x2, 1
false:
    // ternarny operator
    cmp x5, 123
    bne ter_false
    mov x3, x10
    b ter_end
ter_false:
    mov x3, x11
ter_end:
```

```
    cmp x0, x1
    ccmp x2, 42, 0, eq
    // telo podmieneneho prikazu
    cinc x2, x2, eq
    // ternarny operator
    cmp x5, 123
    csel x3, x10, x11, eq
```

Obr. 3.4: Porovnanie možností riadenia toku.

Adresovací mód	**Konštanta**	**Register**	**Rozšírený register**
Bázový register	[base{, #0}]	-	-
Báza + posun	[base{, #imm}]	[base, Xm{, LSL #imm}]	[base, Wm, (S|U)XTW {#imm}]
Pre-indexed	[base, #imm]!	-	-
Post-idnexed	[base], #imm	[base], Xm	
Literál (relatívne k PC)	návestie	-	-

Tabuľka 3.1: `A64` Load/Store adresovacie módy.

adrese, ale môže to byť aj relatívne posunutie k `PC`. V prípade na obrázku 3.5 bázový register určuje adresu parametra svojím obsahom.

```
    // Nacitanie 64bitovej hodnoty z~x1
    ldr x0, [x1]
```

Obr. 3.5: Príklad módu *bázový register*.

Adresovací mód *báza + posun* funguje podobne ako *bázový register* s možnosťou pripočítania voliteľného posunu k adrese v bázovom registre. Posun môže byť špecifikovaný hodnotou v registri alebo konštantou. Na obrázku 3.6 je príklad použitia tohto módu.

Pre-index a *post-index* sú obdobné adresovacie módy, ktoré umožňujú spätný zápis vypočítanej adresy späť do bázového registra. Rozdiel medzi nimi je v poradí vykonania spätného zápisu. *Pre-index* vykoná spätný zápis pred prístupom do pamäti, kým *post-index* až po ňom. Tieto módy umožňujú písanie efektívneho kódu pri prechádzaní rôznych dátových štruktúr vo vyšších programovacích jazykoch. Na obrázku 3.7 je uvedený príklad použitia aj s adresovacím módom *literál*.

3.6 Význam registrov pri volaní funkcií

Konvencia volania špecifikuje postup a podmienky, ktoré musia byť splnené pri volaní procedúr na úrovni strojového kódu. V tomto kontexte vystupujú pojmy: volaný (anglicky

```
1    // adresa = x1 + 8
2    ldr x0, [x1, 8]
3
4    // adresa = x1 + (x2 * 8)
5    ldr x0, [x1, x2, lsl 3]
6
7    // adresa = x1 + (nulove_rozsirenie(w2) * 3)
8    ldr x0, [x1, w2, UXTW #3]
```

Obr. 3.6: Príklad módu *báza + posun*.

```
1    // Pre-index
2    // adresa = x1 + 8; x1 = adresa
3    ldr x0, [x1, 8]!
4
5    // Post-index
6    // adresa = x1; x1 = adresa + 8
7    ldr x0, [x1], 8
8
9    // Literal
10   // nacitanie adresy navestia
11   adr x0, navestie
```

Obr. 3.7: Príklad módov *pre-index*, *post-index* a *literál*.

callee) kód a volajúci (anglicky *caller*) kód [7]. Konvencia slúži na zabezpečenie kompatibility medzi oboma stranami. Pre účely konvencie volania je možné rozdeliť registre na:

X0-X7, V0-V7 - registre určené na predávanie parametrov a návrat výsledku.

X8 - register sa používa na uloženie tzv. nepriameho výsledku (z angličtiny *indirect result register*). Predáva sa v ňom adresa očakávaného výsledku, napríklad pri návrate veľkých štruktúr.

X9-X15, V8-V15 - ak chce volajúci aby boli hodnoty v týchto registroch zachované musí ich uložiť na svoj zásobníkový rámec. Inak musí očakávať, že volaná funkcia ich môže zmeniť.

X16-X17, V16-V31 - špeciálne registre používané napríklad na zdieľanie hodnoty pri podobnom kóde alebo pri skoku, ktorý je mimo adresovateľný rozsah inštrukcie skoku.

X18 - register rezervovaný pre aplikačno binárne rozhranie použitej platformy.

X19-X29 - registre, ktorých hodnoty nesmú byť po návrate z volanej funkcie zmenené.

X29-X30 - registre obsahujúce adresu zásobníkového rámcu a návratovej adresy.

3.7 Systémové volania

Operačný systém sprístupňuje svoju funkcionalitu programom pomocou tzv. systémových volaní. Sú to volania vyvolané práve vykonávaným programom. Typicky sa používajú na

predanie vykonávania jadru operačného systému, ktoré pomocou nastavených parametrov vykoná požadovanú úlohu v privilegovanom režime.

V okamihu keď je vyvolaná výnimka, procesor preruší práve vykonávanú činnosť a vyhľadá správnu obslužnú rutinu (anglicky *handler*), ktorá danú výnimku obslúži. Tieto procedúry sú uložené v `exception vector table`. Každá položka v tejto tabuľke obsahuje postupnosť inštrukcií (nie adresy) s vyhradeným miestom pre 16 záznamov obsahujúcich 32 inštrukcií (16x128 bajtov). Rôzne privilegované režimy majú vlastné tabuľky s procedúrami pre obslužné rutiny. AArch64 definuje štyri úrovne výnimiek `ELn`, pričom so zvyšujúcim `n` sa zvyšuje úroveň privilégií daného režimu:

EL0 je level s najnižšou úrovňou privilégií. Určený je predovšetkým pre programy spúšťané používateľom. Vykonávanie programov v tomto režime sa nazýva aj *neprivilegované vykonávanie*.

EL1 je určený pre vykonávanie kódu reprezentujúceho jadro operačného systému. Typicky to okrem samotného jadra bývajú ovládače.

EL2 je zamýšľaný pre beh tzv. `hypervisoru`, čo je program monitorujúci virtuálne počítače.

EL3 je privilegovaný režim s najvyššou úrovňou oprávnení. Procesor po reštarte sa nachádza v tomto režime a je na programátorovi aby inicializoval a predal kontrolu programom v ostatných privilegovaných režimoch.

Úroveň výnimiek je možné zmeniť jedine pri vstupe alebo návrate z obsluhy výnimky. Zároveň ale platí obmedzenie, že pri vstupe do výnimky je možné úroveň zvýšiť a pri výstupe znížiť. Oba prípady ale umožňujú taktiež ponechať úroveň nezmenenú.

Nasledujúci odstavec popisuje konvenciu platnú pre operačný systém Linux. Kód určený na vyvolanie výnimky je koncipovaný na pripravenie parametrov volania do správnych registrov a samotné vyvolanie výnimky. Register `X8` musí obsahovať číslo systémového volania (identifikátor). Tieto čísla sa v závislosti od použitého operačného systému môžu meniť. Pre systém Linux je možné získať informácie o tabuľke systémových volaní napríklad na internetovej adrese[1]. Ďalej je potrebné nastaviť argumenty. To prebehne naplnením registrov `X0, X1, X2, X3, X4, X5` na zvolenú hodnotu. Každé volanie môže požadovať iný počet argumentov a sú zoradené v poradí od registru `X0` po register `X5`. Posledným krokom je vykonať inštrukciu `svc`, ktorá vyvolá systémové volanie. Na obrázku 3.8 je príklad systémového volania `exit`, ktoré v registri `X0` prijíma hodnotu, s ktorou bude program ukončený.

```
1    mov x0, #0
2    mov x8, #93
3    svc #0
```

Obr. 3.8: Príklad systémového volania `exit`.

[1] https://thog.github.io/syscalls-table-aarch64/latest.html

Kapitola 4

Spätný prekladač RetDec

RetDec je spätný prekladač od firmy Avast založený na technológii LLVM[1]. Podporuje preklad viacerých architektúr a dokáže spracovať rôzne súborové formáty. V nasledujúcich sekciách rozoberieme architektúru RetDecu a niektoré technológie, ktoré budeme ďalej v práci používať.

4.1 LLVM

LLVM je súbor viacerých nástrojov, ktoré sú zamerané na infraštruktúru prekladačov. V tejto práci sa zameriame na jadro `LLVM` (core), obsahujúce popis jazyka pre medzikód. Tento jazyk sa nazýva `LLVM assembly language` a je určený pre čistú reprezentáciu vysoko-úrovňových jazykov. Používa sa na popis kódu jazykov ako napríklad C++ alebo C. Tento kód je vytváraný prekladačom pri preklade za účelom optimalizácie a je používaný vo všetkých častiach prekladu. V nasledujúcich sekciách rozoberieme základy tohto jazyka.

4.2 Jazyk LLVM assembly language

`LLVM assembly language` alebo aj LLVM IR (skratka *LLVM Intermediate Representation*) je typu *Static Single Assignment (SSA)*, čo v praxi znamená, že k definovaným lokálnym premenným je možné priradiť hodnotu iba jeden krát. Toto sa využíva najmä z dôvodu neskoršej jednoduchšej analýzy. Nasledujúce kapitoly slúžia na uvedenie základov tohto jazyka nutných pre porozumenie práci. Celú dokumentáciu je možné nájsť na oficiálnych internetových stránkach projektu LLVM[2].

4.2.1 Identifikátory

V LLVM IR sú používané dva typy identifikátorov. Sú to globálne identifikátory (pre globálne premenné a funkcie), ktoré začínajú prefixom '@'. Druhým typom sú lokálne identifikátory (pre lokálne premenné a typy), ktoré začínajú prefixom '%'. Štandardne sa používajú tri nasledujúce formáty identifikátorov:

1. Pomenované hodnoty reprezentované ako reťazec znakov s ich prefixom. Tento typ identifikátorov je možné popísať pomocou regulárneho výrazu `[%@][-a-zA-Z$._][-a-zA-Z$._0-9]*`.

[1] https://llvm.org/
[2] https://llvm.org/docs/LangRef.html

2. Nepomenované hodnoty sú reprezentované bezznamienkovým číslom s prefixom. Sú využívané hlavne ako mená premenných pri pomocných výpočtoch. Príkladom môže byť napríklad '@1' či '%42'.

3. Konštanty, ktoré budú bližšie vysvetlené v 4.2.6.

4.2.2 Globálne premenné

Programy v jazyku LLVM IR sú zložené z modulov, kde každý modul je samostatnou prekladovou jednotkou. Moduly sa skladajú z funkcií, globálnych premenných a zo záznamov do tabuľky symbolov. Jednotlivé moduly je možné pomocou nástroja `LLVM linker` previazať medzi sebou.

Globálna premenná je miesto v pamäti vyhradené počas prekladu. Globálne premenné musia mať pri vytvorení priradenú hodnotu. Výnimku tvoria globálne premenné z iných modulov, ktoré môžu byť deklarované bez počiatočnej hodnoty. V kóde sú deklarované pomocou kľúčového slova `global`, pričom môžu byť špecifikované aj ďalšie parametre ako napríklad umiestnenie premennej v moduloch, konštantnosť premennej, zarovnanie, atď.

4.2.3 Funkcie

Funkcia je pomenovaná časť kódu, ktorá prijíma určité parametre a môže vracať hodnotu v podobe výsledku. Skladá sa z hlavičky a tela, ktoré je možné ďalej rozdeliť na postupnosť jednoduchých blokov obsahujúcich kód. Bloky definujú graf toku vykonávania funkcie. Voliteľne začínajú návestím, ktoré pridelí danému bloku záznam do tabuľky symbolov. Telo bloku sa skladá zo zoznamu inštrukcií a na konci má uvedenú ukončovaciu inštrukciu (napríklad nepodmienený skok alebo návrat z funkcie). Definícia funkcie je uvedená kľúčovým slovom `define` a deklarácia podobne kľúčovým slovom `declare`.

Funkcie môžu obsahovať rôzne nastavenia ako napríklad konvenciu volania, názov správy pamäte (anglicky *garbage collector*), prológ funkcie, atď.

4.2.4 Dátová schéma

V module je možné špecifikovať reťazec dátovej schémy (anglicky *data layout string*), ktorý hovorí o tom, ako majú byť dáta rozložené v pamäti. Reťazec sa skladá zo špecifikácií oddelených znakom '-'. Každá špecifikácia začína znakom, ktorý určuje jej typ. Za uvádzacím znakom nasleduje daná hodnota.

V dátovej schéme je možné špecifikovať napríklad typ endianity, zarovnanie zásobníka, predvolenú bitovú šírku celočíselných typov, veľkosť ukazateľa a pod.

4.2.5 Dátové typy

LLVM IR je silne typovaný jazyk, vďaka čomu má veľké možnosti optimalizácií, bez nutnosti vykonávať zložité analýzy. Typy sa delia na jednoduché (anglicky *first class types*) a zložené (anglicky *aggregate types*).

Typ void nereprezentuje žiadnu hodnotu a nemá veľkosť.

Typ funkcia reprezentuje dvojicu návratovej hodnoty a zoznamu parametrov funkcie. Zástupcom tohto typu je napríklad typ `{i32, i32} (i32)`, teda funkcia prijímajúca `i32`, ktorá vracia štruktúru obsahujúcu dve položky `i32`.

Jednoduché typy sú typmi, ktoré môžu byť výsledkom inštrukcie.

- Typ `iN`, kde `N` je z rozsahu 1 až $2^{23} - 1$ reprezentuje celočíselný typ s bitovou šírkou `N`. Príkladom takýchto typov je napríklad 32 bitové celé číslo popísané typom `i32`.
- Typy `half, float, double, fp128, x86_fp80, ppc_fp128` reprezentujú desatinné čísla s rozličnou bitovou šírkou.
- Typ `x86_mmx` reprezentuje hodnotu v registroch `MMX` na architektúre x86.
- Typ `*` reprezentuje ukazateľ na typ predchádzajúci symbolu '*'. Je používaný na špecifikovanie miesta v pamäti. Typicky sa používajú ako odkaz na objekty v pamäti. Príkladom je typ `i32 *`, ktorý reprezentuje ukazateľ odkazujúci na 32 bitové číslo v pamäti.
- Typ `vektor` reprezentuje niekoľko primitívnych dátových typov používaných pri type inštrukcií `SIMD`.
- Typ `label` reprezentuje návestie kódu.
- Typ `metadata` reprezentuje doplňujúce informácie.

Zložené typy sú podmnožinou odvodených typov, ktoré môžu obsahovať viaceré typy.

- Typ `pole` je zložený z viacerých hodnôt rovnakého typu. Jednotlivé elementy sú uložené v pamäti sekvenčne. Príkladom je `[20 x i64]`, čo je pole o dvadsiatich prvkoch typu `i64`.
- Typ `štruktúra` reprezentuje kolekciu dátových členov spolu v pamäti. Funguje podobne ako typ `pole` ale jej prvky nemusia byť rovnakého typu. Štruktúry môžu byť typu „packed“, čo znamená, že zarovnanie takejto štruktúry je jeden bajt a neobsahuje medzery medzi jednotlivými položkami.

4.2.6 Konštanty

Konštanty špecifikujú zápis literálov pre každý dátový typ. Delia sa na jednoduché a komplexné. Dostupné jednoduché konštanty sú: boolovské ('true' alebo 'false'), celočíselné, desatinné a nulový ukazateľ. Komplexné konštanty sú (potencionálne rekurzívnou) kombináciou jednoduchých konštánt a menších komplexných konštánt. Do tejto kategórie patria konštanty: štruktúr, polí, vektorov, nulová inicializácia a uzly metadát.

4.2.7 Metadáta

Metadáta sú spojené priamo s inštrukciami v danom programe a môžu poskytovať doplňujúce informácie pre ďalšie nástroje pracujúce s týmto kódom. Príkladom takýchto metadát sú ladiace informácie. Metadáta nemajú typ a nie sú ani hodnota. Metadáta používajú ako prefix znak '!'.

4.2.8 Inštrukcie

Jazyk LLVM IR definuje veľké množstvo inštrukcií, ktoré je možné použiť na jednoduché a jasné popísanie vysoko-úrovňových jazykov. Tieto inštrukcie je možné klasifikovať do nasledujúcich kategórií:

Inštrukcie meniace tok programu slúžia na ukončovanie blokov kódu programu. Tieto inštrukcie špecifikujú, ktorý ďalší blok bude vykonaný. Príkladom sú inštrukcie `ret`, `br`, `switch`, `resume`.

Binárne operácie sú operácie pracujúce s dvoma operandmi rovnakého typu. Po vykonaní operácie vrátia jednu hodnotu. Príkladom sú inštrukcie `add`, `sub`, `fdiv`, `mul`, `frem`.

Bitové inštrukcie vyžadujú taktiež dva operandy rovnakého typu. Pracujú na bitovej úrovni a ich vykonanie býva často veľmi efektívne. Po vykonaní operácie vrátia výslednú hodnotu. Príkladom sú inštrukcie `and`, `or`, `xor`, `shl`, `lshr`, `ashr`.

Pamäťové inštrukcie sú určené na správu a operácie s hodnotami uloženými v pamäti. LLVM nereprezentuje pamäť pomocou *SSA* a používa architektúru *load/store*. Príkladom sú inštrukcie `alloca`, `load`, `store`.

Inštrukcie porovnania slúžia pri zmenách toku programu, kde sa používajú na porovnanie dvoch hodnôt. Inštrukcie vyžadujú dva operandy, ktoré pri vykonaní určitým spôsobom porovnajú a vrátia výsledok porovnania. Na základe tohto porovnania môžu ďalšie inštrukcie zmeniť tok vykonávania programu. Príkladom sú inštrukcie `icmp`, `fcmp`.

Inštrukcie prevodu sa využívajú na prevod hodnoty jedného typu na hodnotu typu druhého. Každá inštrukcia špecifikuje pravidlá použité pri prevode a po vykonaní vráti hodnotu prevodu. Príkladom sú inštrukcie `trunc`, `zext`, `sext`.

4.3 Capstone – disassembler

Capstone je disassemblovací framework. Podporuje veľké množstvo architektúr vrátane architektúry ARM64. Je dobre testovaný a taktiež uspôsobený pre prácu s malvérom. V RetDecu sa používa na disassemblovanie strojového kódu, pričom poskytuje kľúčové informácie využívané pri spätnom preklade.

Jeho vstupom sú binárne dáta (rôznych veľkostí) typicky malé časti programov, ktoré transformuje do ľudsky zrozumiteľnej podoby vo forme inštrukcií. Výstupom nie je len textová reprezentácia, ale aj *komplexné* informácie o jednotlivých častiach inštrukcie. Príkladom sú informácie o čítaní alebo zápise jednotlivých operandov. Ponúka rozhranie pre programovanie aplikácií nezávislé na zvolenej architektúre. Je implementovaný v jazyku C, pričom existujú bindingy pre mnohé v súčasnosti používané jazyky. Spracované inštrukcie zo sady `A64` sú uložené v štruktúre, ktorá obsahuje nasledujúcich členov.

`cc` udáva či je inštrukcia podmienená. V prípade, že je podmienená tak špecifikuje aká podmienka musí platiť v dobe vykonávania aby daná inštrukcia bola sputená.

`update_flags` signalizuje či daná inštrukcia mení príznaky `PSTATE`.

`writeback` značí, že mód adresovania vyžaduje spätný zápis vypočítanej hodnoty do registru. Toto platí pre adresovacie módy `pre-index` a `post-index`.

`op_count` obsahuje počet operandov spracovanej inštrukcie.

Operandy sú uložené samostatne v poli `operands`. Do tohto poľa je možné uložiť viacero druhov operandov, napríklad register alebo systémový register, pamäťový operand, atď. Štruktúra popisujúca operandy inštrukcií je na nasledovná:

`type` obsahuje informáciu o type daného operandu. Možné typy operandov sú: register, konštanta, adresa v pamäti, desatinné číslo a pod.

`shift` špecifikuje operáciu posunu a jeho hodnotu pre daný operand. Pre AArch64 je veľmi častý vzor, kedy sa počas vykonávania inštrukcie vykoná taktiež logický posun niektorého z operandov. Príkladom je inštrukcia `add x0, x1, x2, LSL #2`, ktorá pred sčítaním vykoná logický posun hodnoty v registri `x2` o 2 bity doľava.

`reg, imm, fp, mem, pstate, sys, prefetch, barier` sú členmi typu `union` reprezentujúce hlavnú hodnotu operandu. Obsah premennej `type` určuje, ako má byť daná hodnota interpretovaná. Možnými interpretáciami je napríklad identifikátor registru, 64-bitová konštanta alebo adresa pamäte.

`access` popisuje spôsob pracovania s operandom. Obsahuje informáciu o prípadom zápise alebo čítaní z operandu.

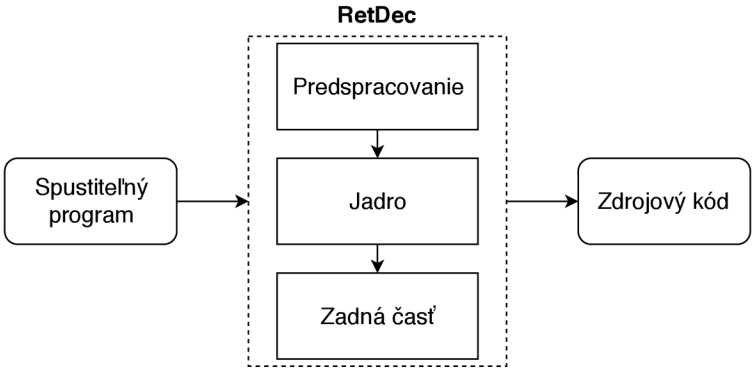
`ext` popisuje akým spôsobom má byť operand rozšírený pred prácou s ním. Táto hodnota je použitá iba pri registrových operandoch.

`vas, vess` špecifikujú prístup k vektorovým registrom. Môžu udávať rozloženie a dátové typy v registri. Táto hodnota je valídna iba pri vektorových operandoch.

`vector_index` pri prístupe k jednému prvku vektorového registra, určuje index tohto prvku.

4.4 Architektúra programu RetDec

Spätný prekladač RetDec je komplexný program zložený z viacerých samostatných modulov, ktoré sú zodpovedné za jednotlivé časti spätného prekladu. Proces spätného prekladu je znázornený na obrázku 4.1. Skladá sa z viacerých úrovní, kde prvým krokom je predspracovanie (anglicky *preprocessing*), následne je spustené jadro (anglicky *core*). Na koniec je spustená zadná časť (anglicky *backend*). Nasledujúca sekcia o fungovaní programu RetDec čerpá informácie z [11, 12].

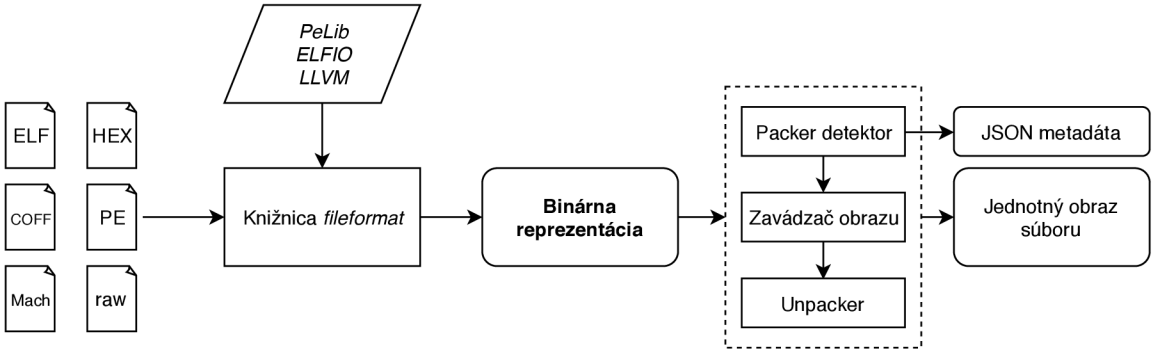


Obr. 4.1: Schéma nástroja RetDec. Prebrané z [12].

4.4.1 Predspracovanie

Prvá časť spätného prekladu má za úlohu pripraviť binárny súbor na spracovanie. Keďže existujú mnohé formáty spustiteľných súborov pre rôzne platformy, je nutné ich konvertovať na jednotnú univerzálnu reprezentáciu, ktorú je možné ďalej spracovať.

Táto reprezentácia je skontrolovaná na použitie packerov. Packer je nástroj slúžiaci na zbalenie spustiteľných programov, s cieľom zmenšiť jeho veľkosť alebo sťažiť jeho analýzu. Použitie takého programu komplikuje spätný preklad, keďže inštrukcie už nie sú uložené v rozpoznateľnej podobne. Kontrola prebieha pomocou nástroja YARA[3], pričom vznikne súbor vo formáte `JSON`[4] s informáciami o prípadne použitom packery.



Obr. 4.2: Schéma predspracovania. Prebrané z [12].

Existujú programy (typicky *malvér*), ktoré sú závislé na použití zavádzača. Je preto použitý nástroj, ktorý simuluje zavedenie programu do pamäte.

Na konci predspracovania je binárna reprezentácia v prípade nutnosti rozbalená a vzniká jednotný obraz spustiteľného programu. V tejto fáze sú taktiež spracovávané súbory, ktoré obsahujú informácie o ladení. Tieto súbory vo formáte `DWARF` alebo `PDB` sú analyzované a extrahované informácie slúžia k zlepšeniu prekladu.

4.4.2 Jadro

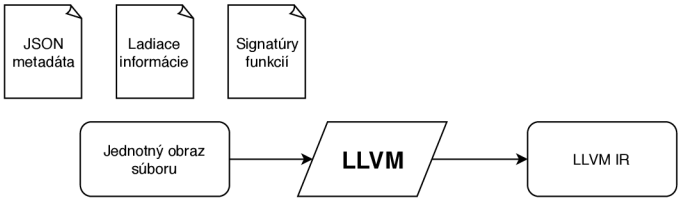
Táto fáza má za úlohu previesť kód v obraze do LLVM IR a previesť optimalizácie. Prevod funguje ako sekvenčné spúšťanie LLVM priechodov (anglicky *LLVM pass*). *Priechod* je jedno ucelené spracovanie `LLVM IR` kódu. Výsledkom typicky býva pozmenená reprezentácia alebo informácie o nej. Ďalšie informácie môžu byť pripojené formou metadát vo formáte `JSON`, ladiacich informácií alebo zoznamu signatúr funkcií. Po priechode, ktorý inicializuje prostredie je spustené dekódovanie inštrukcií. Schéma jadra je zobrazená na obrázku 4.3.

Dekóder

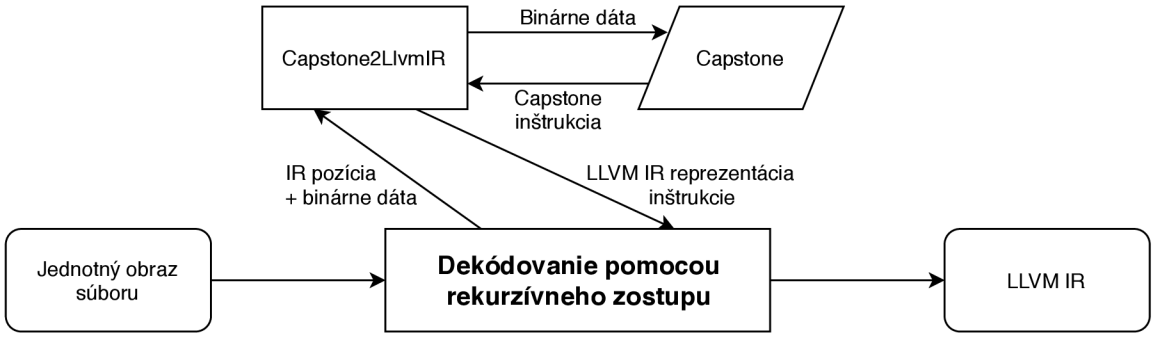
Dekóder je časť jadra, ktorá je zodpovedná za prevod inštrukcií na ich zodpovedajúcu reprezentáciu v LLVM IR. Taktiež riadi, ktoré inštrukcie sú spracované. Na obrázku 4.4 je znázornená schéma tohto procesu.

Pri dekódovaní je použitý takzvaný *rekurzívny zostup*. V praxi to znamená, že inštrukcie sú dekódované z určitého miesta (typicky vstupný bod programu), pričom dekódovanie sleduje tok programu. Riadenie je implementované pomocou prioritnej rady (anglicky *priority*

[3] http://virustotal.github.io/yara/
[4] https://www.json.org/



Obr. 4.3: Schéma jadra. Prebrané z [12].



Obr. 4.4: Schéma dekóderu. Prebrané z [12].

queue), ktoré na počiatku obsahuje ciele vetvenia kódu, vstupné body, ladiace časti, symboly, atď. Lineárne dekódovanie sa nepoužíva, pretože pri procese dekódovania nie je možné odlíšiť dáta od kódu. To by mohlo viesť k nezmyselným výsledkom spätného prekladu. Ďalšou jeho nevýhodou je skutočnosť, že pomerne často je touto metódou spracovávaný aj tzv. mŕtvy kód, ktorý nie je počas vykonávania programu nikdy spustený.

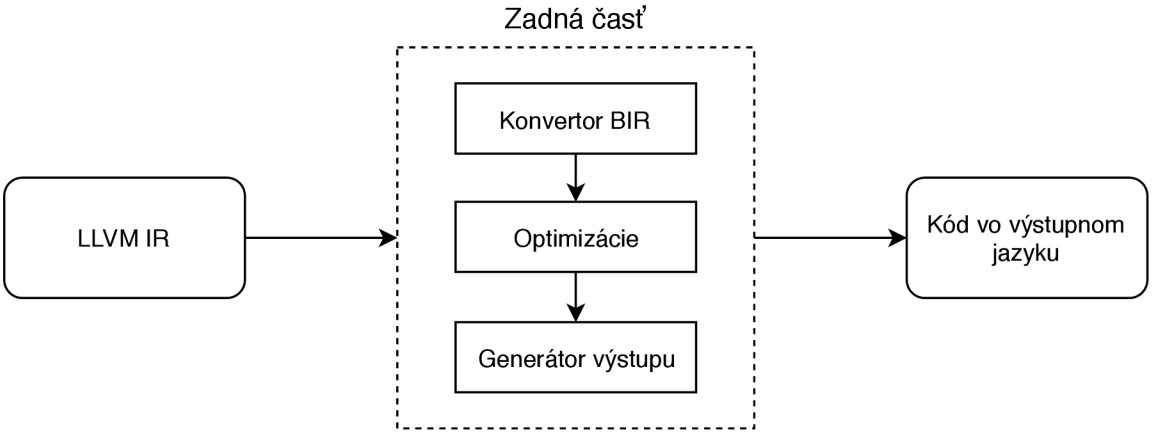
Pomocou rekurzívneho zostupu sú vybrané inštrukcie na dekódovanie. Tieto inštrukcie sú disassemblované pomocou Capstone 4.3. Na základe operačného kódu inštrukcie je vyhľadaná a vykonaná funkcia zodpovedná za dekódovanie tejto inštrukcie. LLVM IR je zapísaný do bloku podľa miesta, kde bola inštrukcia nájdená. V prípade, že nájdenie funkcie zlyhá, do výstupu je vygenerovaná *pseudo* inštrukcia.

Po dokončení dekódovania nasleduje fáza optimalizácií. Pozostáva z rozličných LLVM priechodov, ktorých úlohou je optimalizovanie aktuálnej reprezentácie. Ide napríklad o priechody spracúvajúce konštantné hodnoty, prácu so zásobníkom alebo systémové volania.

4.4.3 Zadná časť

Hlavnou úlohou zadnej časti je konvertovať reprezentáciu kódu LLVM IR do výstupného jazyka. RetDec štandardne generuje výstupný kód v jazyku C. Návrh dovoľuje rozšíriť prekladač o ďalšie výstupné jazyky. Zadná časť je schopná generovať taktiež aj grafy volaní funkcií (anglicky *call graphs*) a grafy kontroly toku (anglicky *control flow graphs*).

Najprv táto časť konvertuje kód v LLVM IR do reprezentácie `BIR` (skratka *backend intermediate representation*), ktorá má podobu *abstraktného syntaktického stromu*. Táto forma je efektívna na spracovanie v zadnej časti. Prebiehajú na nej finálne optimalizácie a následné rozpoznávanie vzorov toku programu. Spätný preklad je ukončený prevodom stromu na zodpovedajúci kód vo výstupnom jazyku. Na obrázku 4.5 je znázornená schéma zadnej časti.



Obr. 4.5: Schéma zadnej časti. Prebrané z [12].

4.5 Keystone – assembler

Keystone je opakom Capstone popísaným v sekcii 4.3. Je určený na prekladanie inštrukcií do ich binárnej reprezentácie. V RetDecu nie je využívaný pri spätnom preklade ale najmä pri testovaní. Používa sa pri jednotkových testoch, pri testovaní výstupu a správnej interpretácie inštrukcií.

Kapitola 5

Návrh implementácie

V nasledujúcej kapitole identifikujeme moduly, ktoré je nutné rozšíriť. Ďalej analyzujeme nové programy, ktoré potencionálne uľahčia vývoj a testovanie. Neskôr v kapitole bude uvedený návrh implementácie nového modulu pre spätný prekladač RetDec a spôsob reprezentácie novej architektúry v LLVM IR.

V súčasnosti v programe RetDec nie je možný spätný preklad architektúry ARM64. Pri pokuse o spracovanie súborov pre túto architektúru je preklad ukončený chybovým hlásením. Je nutné preto vytvoriť nový samostatný modul schopný prekladu inštrukcií do LLVM IR. Nejedná sa o rozšírenie prítomného modulu určeného pre preklad architektúry ARM32, pretože stavy vykonávania AArch64 a AArch32 nie sú medzi sebou kompatibilné.

5.1 Identifikovanie nutných rozšírení

Najskôr je potrebné rozšíriť jednoduchý program CapstoneDumper[1] o pridávanú architektúru ARM64. Tento program slúži na získavanie informácií o danej inštrukcii v kontexte disassembleru Capstone. Zobrazuje štruktúry reprezentujúce inštrukciu v textovej podobe. Získané informácie sú kľúčové pre správne pochopenie schémy inštrukcie ako aj jej sémantiky. Táto reprezentácia je v ďalších programoch používaná ako hlavný zdroj informácií o spracovávanej inštrukcii.

Najväčšie zmeny je nutné vykonať v knižnici `Capstone2LlvmIr`. Knižnica je určená na prekladanie binárnych dát do ich LLVM IR reprezentácie. V súčasnom stave nepodporuje architektúru ARM64. Pre pridanie podpory AArch64 je nutné vytvoriť nový modul. Daný modul musí obsahovať prekladové rutiny pre jednotlivé inštrukcie a inicializáciu LLVM prostredia.

Ďalším krokom je popísanie ABI a konvencie volania funkcií. Informácie z týchto rozšírení sú následne používané v analýzach nad LLVM IR inštrukciami, ktoré majú za úlohu zlepšiť výsledok spätného prekladu. Kvôli analýze systémových volaní je nutné implementovať tabuľku mapujúcu čísla volaní k ich správnemu názvu.

V neposlednom rade je potrebné povoliť spracovávanie ARM64 súborov. To je riešené v hlavnom skripte `retdec-decompiler.py`, kde je nutné nastaviť správne hodnoty prostredia pre korektný spätný preklad.

Vylepšenie podpory prekladu objektových súborov je vykonané v časti, ktorá simuluje zavádzač binárnych súborov. Je potrebné rozšíriť analýzu importovaných funkcií a symbolov.

[1] https://github.com/avast-tl/capstone-dumper

5.2 Modelovanie registrov

Modelovanie registrov je úzko späté s ich reprezentáciou v Capstone. Ten ale niektoré registre nedefinuje, pretože ich nie je možné zakódovať v inštrukciách. Príkladom takéhoto registru je register `PC`, ktorý neumožňuje priamy zápis ani čítanie a preto ho nie je možné použiť ako operand žiadnej inštrukcie. V takýto prípadoch rozširujeme definície a mapovanie Capstone o ďalšie registre, aby bolo možné reprezentovať stav `PE`.

V sekcii 5.7 je možné nájsť vysvetlené príklady reprezentácie inštrukcií v LLVM IR. Všetky 64 bitové celočíselné registre podporujú aj prístup k spodným 32 bitom daných registrov. Toto obmedzenie bude riešené vytvorením mapovania medzi registrami. Podradené registre sa budú odkazovať na ich nadradené registre. Príkladom je pridelenie registru `w0` k registru `x0`. Pri prístupe k registru bude nutné vždy skontrolovať či identifikátor registru nemá pridelený nadradený register a ak áno, tak pristúpiť k nemu.

Registre budú reprezentované celočíselnými typmi LLVM IR podľa ich bitovej šírky.

64 bitové registre budú reprezentované LLVM IR typom `i64`.

32 bitové registre budú reprezentované LLVM IR typom `i32` ale pri prístupe k nim sa pracuje s odpovedajúcou častou nadradeného registra.

registre príznakov budú reprezentované jedno-bitovým LLVM IR typom `i1`.

Aj napriek tomu, že `PSTATE` nie je popísaný ako samostatný systémový register, bude modelovaný ako štyri jedno-bitové registre príznakov. Tieto registre budú mať pridelený LLVM typ `i1` a názov podľa príznaku, ktorý signalizujú.

Modelovanie vektorových registrov je zložitejšie. Textový zápis operandu špecifikuje typ prístupu k danému registru. Podobne ale ako pri celočíselných registroch je ale možný prístup k podmnožine hlavného registra. Tieto menšie časti registrov sú ale reprezentované ako čísla s pohyblivou desatinnou čiarkou. Ak by boli hlavné registre modelované 128 bitové celočíselné typy, prístup k podregistru by vyzeral nasledovne:

1. Načítanie hlavného registra, ktoré by malo typ `i128`.
2. Pretypovanie orezaním na typ podregistru. Predpokladajme napríklad `i64`.
3. Bitové pretypovanie na požadovanú výslednú hodnotu `f64`.

Uloženie hodnoty by prebiehalo analogicky. Takéto chovanie by ale do výsledného LLVM IR vnieslo nejasnosti aj pri relatívne jednoduchých operáciách. Dôsledkom toho sa vo výstupe vyskytovali výrazy obsahujúce veľké množstvo pretypovaní.

Nový návrh stavia na znalosti že inštrukcie, ktoré pracujú s podmnožinami vektorových registrov, vždy operujú s rovnakými typmi. V prípadoch kedy je žiadané pracovať s inými typmi je najprv nutná konverzia. Najjednoduchšie je demonštrovať toto správanie na príklade. Inštrukcia `fadd d0, d1, d2` sčíta dve 64-bitové hodnoty, ktoré sú uložené ako čísla s pohyblivou desatinnou čiarkou. Výsledok tejto operácie je uložený do registru `v0`, teda konkrétne do spodných 64 bitov reprezentovaných menom `d0`. Ak sa niekto následne rozhodne pracovať s 32-bitovou reprezentáciou tohto čísla, je nútený vykonať inštrukciu realizujúcu príslušnú konverziu. Pri práci s číslami s pohyblivou desatinnou čiarkou je tento postup vynútený, kvôli spôsobu uloženia hodnoty v registri. V tomto prípade nie je možné

Registre	Vn	Qn	Dn	Sn	Hn	Bn
Pridelený typ	i128	f128	f64	f32	i16	i8

Tabuľka 5.1: Typy pridelené registrom. Znak n reprezentuje dostupné čísla registrov.

vykonať typovú konverziu medzi bitovými šírkami jednoduchým orezaním alebo znamienkovým rozšírením hodnoty. Podregistre budú teda modelované ako samostatné registre, s vlastnou hodnotou podľa tabuľky 5.1.

Významné sú registre Hn a Bn, ktoré sú modelované ako celé čísla. Ich výskyt v programoch je podmienený rozšíreniami procesora. V prípade Hn sú nutné špeciálne knižnice umožňujúce prácu s 16-bitovými číslami s pohyblivou desatinnou čiarkou. Bn reprezentujú celé čísla o veľkosti jedného bajtu.

Takýmto návrhom je možné zjednodušiť výsledné LLVM IR a aj výstup spätného prekladu. Pri preklade inštrukcií konverzie bude nutné vykonať čítanie a zápis konvertovanej hodnoty do správneho registra. Pre zaručenie prístupu k registrom z každého miesta v kóde je potrené ich modelovať ako globálne premenné v jazyku LLVM IR.

5.3 Modelovanie zmien toku vykonávania

Pri inicializácii prekladového prostredia LLVM IR sú vygenerované pseudo funkcie spojené s riadením toku vykonávania programu. Tieto funkcie korešpondujú s všeobecnými inštrukciami typickými pre každú architektúru ako sú: `branch, call, return` a pod. Aj napriek tomu, že `LLVM IR` existujú inštrukcie, ktoré sú schopné ich reprezentovať, nevolíme ich používanie. Dôvodom je, že pri ich generovaní si vyžadujú znalosť cieľového návestia, na ktoré bude predaná kontrola toku programu. Získanie správnych informácií by ale vyžadovalo modifikáciu výstupu mimo spracovávanú sekvenciu. Takéto chovanie by ale bolo nežiadúce, preto sú pseudo funkcie preferovaným riešením. Na obrázku 5.1 sú uvedené deklarácie týchto funkcií.

```
1  ; Parameter i64 reprezentuje adresu skoku
2  ; Pseudo volanie funkcie
3  declare void @__pseudo_call(i64)
4  ; Pseudo navrat z funkcie
5  declare void @__pseudo_return(i64)
6  ; Pseudo nepodmieneny skok
7  declare void @__pseudo_branch(i64)
8  ; Pseudo podmieneny skok, ktory sa vykona iba ak plati podmienka
9  declare void @__pseudo_cond_branch(i1, i64)
```

Obr. 5.1: Deklarácie pseudo funkcií používaných na reprezentáciu riadenia toku.

5.4 Modelovanie inštrukcií

Cieľom je aby program dokázal spracovať akúkoľvek inštrukciu pre danú architektúru. Existujú ale prípady kedy je užitočnejšie preložiť inštrukciu do podoby tzv. pseudo inštrukcie. Inštrukcia je v tejto forme podobná volaniu funkcie, ktorá v mene obsahuje názov pôvodnej inštrukcie. Argumenty pre túto funkciu sú generované automaticky, podľa informácií získa-

ných z Capstone reprezentácie o zápise a čítaní z operandov. Príkladmi takých inštrukcií sú napríklad:

1. Kryptografické inštrukcie, ktoré vykonávajú kryptografickú funkciu nad obsahom registru.
2. Inštrukcie nesúce sémantiku, ktorú nie je možné reprezentovať na úrovni LLVM IR. Napríklad inštrukcia `prfm`, ktorá signalizuje pamäťovému systému, aby v budúcnosti očakával prístup k lokalite v pamäti. Táto inštrukcia neovplyvňuje sémantiku programu, pričom `PE` ju môže považovať za `nop` inštrukciu.
3. Vektorové operácie nad registrami.
4. V niektorých prípadoch inštrukcie, ktoré nie je možno popísať čisto v jazyku C a viedli by k vygenerovaniu veľkého množstva kódu.

Plné modelovanie sémantiky týchto inštrukcií by dospelo ku generovaniu veľkého množstva kódu, ktorý by používateľ ťažšie pochopil. Zavedenie pseudo inštrukcií vo významnej miere zvyšuje čitateľnosť a prehľadnosť výsledného kódu.

Automatické generovanie pseudo inštrukcie je preferovaný spôsob zotavenia sa v prípade, že nie je nájdená zodpovedajúca prekladová rutina danej inštrukcie. Nie všetky inštrukcie je možné generovať automaticky, preto je nutné vytvoriť aj manuálne spôsoby generovania pseudo inštrukcií. Pri návrhu je možné zanedbať určité detaily popisu architektúry ARM64, pretože nie sú pri spätnom preklade relevantné. Ide najmä o pamäťový model a časovanie.

5.5 Spracovanie operandov

Stav vykonávania AArch64 dovoľuje vykonávať rôzne operácie s operandmi. Možnosti týchto operácií sú priamo obmedzené typom inštrukcie a povolenými adresovacími módmi, takže nie vždy dovoľujú špecifikovať ľubovoľné konštanty.

5.5.1 Operandy s bitovým posunom

Pri určitých inštrukciách je možné pridať bitové posunutie k druhému operandu inštrukcie. Toto posunutie je špecifikované pomocou kľúčového slova a hodnoty posunutia. Možné posuny sú uvedené v tabuľke 5.2.

Kľúčové slovo	**Význam**
`LSL`	Logický posun doľava (dopĺňanie bitmi 0)
`MLS`	Logický posun doľava (dopĺňanie bitmi 1)
`LSR`	Logický posun doprava
`ASR`	Aritmetický posun doprava
`ROR`	Rotácia bitov doprava

Tabuľka 5.2: Možné posuny operandov.

Informácie o použitom posune operandu je možné získať v štruktúre, ktorú naplní Capstone pri spracovaní inštrukcie. V prípade, že je posunutým operandom *vektorový* register je toto posunutie aplikované na každom prvku, podľa príslušného rozloženia. Inštrukcie typicky podporujú podmnožinu validných posunov, či už sú obmedzené operácie alebo

povolené hodnoty posunu. Dôvodom je obmedzené miesto na zakódovanie inštrukcie. Pri preklade inštrukcie v LLVM IR sa vygeneruje načítanie hodnoty a následne sa aplikuje požadovaný posun pomocou bitových inštrukcií `LLVM IR`.

Napríklad inštrukcia `add x0, x1, x2, LSL #3` pred sčítaním registrov `x1` a `x2` posunie hodnotu v registri `x2` o 3 bity doľava.

5.5.2 Rozšírenie operandov

Ďalšou možnosťou, ako ovplyvniť interpretáciu operandu, je voliteľné znamienkové alebo nulové rozšírenie časti druhého operandu. Určité inštrukcie dovoľujú kombinovať rozšírenie operandu s bitovým posunom `LSL`. V takom prípade má rozšírenie operandu prednosť pred posunom. Rozšírenie hodnoty registra sa používa pri inštrukciách, ktoré umožňujú ako operandy špecifikovať registre s rozličnými bitovými šírkami. Napríklad pri sčítaní 32-bitového čísla s číslom 64-bitovým.

Kľúčové slovo	**Význam**
`{U/S}XTB`	Nulové/Znamienkové rozšírenie bajtu
`{U/S}XTH`	Nulové/Znamienkové rozšírenie pol slova
`{U/S}XTW`	Nulové/Znamienkové rozšírenie slova
`{U/S}XTX`	Nulové/Znamienkové rozšírenie dvoj slova

Tabuľka 5.3: Možné rozšírenie operandov.

5.6 Modelovanie systémových volaní

Systémové volania sú vyvolané inštrukciou `svc`. Aplikačno binárne rozhranie systému Linux udáva[2], že číslo obsluhovaného volania bude uložené v registri `x8`. Ďalšie parametre sú predávané pomocou registrov `x0, x1, x2, x3, x4, x5` v poradí od prvého k poslednému. Výsledok daného volania je vrátený pomocou registru `x0`.

5.7 Príklady prekladu inštrukcií

V nasledujúcej časti budú uvedené a popísané príklady možného prekladu niektorých ukážkových inštrukcií. Prvou inštrukciou je `mov x0, x1`. V prvom kroku je načítaná hodnota globálnej premennej `@x0` do novej lokálnej premennej `%0`. Táto hodnota sa ďalej zapíše do globálnej premennej `@x1`, ktorá reprezentuje register `x1`.

```
1   %0 = load i64, i64* @x1
2   store i64 %0, i64* @x0
```

Obr. 5.2: Príklad sémantiky jednoduchej inštrukcie.

Ukážka kódu 5.3 s prekladom inštrukcie `add x0, x1, w2, SXTH` demonštruje znamienkové rozšírenie posledného operandu. K hodnote registru `w2` je pristúpené pomocou jeho nadradeného registra `x2`. Hodnota je preto zmenšená orezaním na 32 bitov. Ďalšie zmenšenie je spôsobené použitím rozšírenia operandu z pol slova `SXTH`. Hodnota, ktorá vznikla

[2] http://man7.org/linux/man-pages/man2/syscall.2.html

orezaním je teraz znamienkovo rozšírená na požadovanú bitovú šírku a je vykonaná operácia sčítania.

```
%0 = load i64, i64* @x1
%1 = load i64, i64* @x2
%2 = trunc i64 %1 to i32
%3 = trunc i32 %2 to i16
%4 = sext i16 %3 to i64
%5 = add i64 %0, %4
store i64 %5, i64* @x0
```

Obr. 5.3: Príklad znamienkového rozšírenia registru.

Poslednou inštrukciou je `ldr x0, [x1, #8]!`, ktorá načíta do registru špecifikovaného prvým operandom hodnotu uloženú v pamäti na mieste odkazovanom druhým operandom. Miesto v pamäti je v adresovacom móde *pre-incremenet* čo znamená, že hodnota `#8` bude pripočítaná pred samotným prístupom do pamäti. Sufix '!' signalizuje, že výsledná hodnota druhého operandu bude po vykonaní inštrukcie zapísaná späť do bázového registru druhého operandu. Inštrukcia LLVM IR `inttoptr` konvertuje vypočítanú hodnotu na typ ukazateľ do pamäti. Z tohoto miesta je následne načítaná hodnota. Preklad inštrukcie je navrhnutý na ukážke 5.4.

```
%0 = load i64, i64* @x1
%1 = add i64 %0, 8
%2 = inttoptr i64 %1 to i64*
%3 = load i64, i64* %2
store i64 %3, i64* @x0
store i64 %1, i64* @x1
```

Obr. 5.4: Príklad adresovacieho módu *pre-increment*.

5.8 Insrunner

Capstone je schopný rozpoznať približne 1000 inštrukcií inštrukčnej sady `A64`. Je preto výhodné do určitej miery automatizovať testovanie a overovanie sémantiky týchto inštrukcií. Pôvodne boli tieto úlohy vykonávané ako písanie malých kusov kódu v assemblery. Nasledovalo preloženie a zlikovanie. Potom bolo nutné výsledný program spustiť v ladiacom nástroji a jednotlivo vykonávať inštrukcie, pričom si poznamenávať získané výsledky. V prípade chyby bolo potrebné zmeniť alebo upraviť testované inštrukcie, kvôli čomu bolo potrebné upraviť zdrojové kódy a následne vykonať všetky kroky od prekladu po ladenie odznovu. Toto riešenie veľmi rýchlo prestalo byť optimálne a vyžadovalo si zmeny.

Program Insrunner je navrhnutý aby riešil tieto problémy. Poskytne používateľovi rozhranie pre jednoduché spúšťanie jednej alebo viacerých inštrukcií. Po vykonaní je jeho výstupom rozdiel prostredia pred a po vykonaní inštrukcie. Základná funkcionalita zahrňuje:

- Automatizovaný preklad, linkovanie a ladenie testovaných inštrukcií.
- Informácie o zmene stavu procesoru pred a po vykonaní inštrukcie.

- Generovanie kostry jednotkových testov pre vykonanú inštrukciu.

5.9 ELF importované funkcie

Väčšina programov preložených pre architektúru ARM64 je typicky vo formáte ELF. Z toho dôvodu aj testovanie návrhu prebiehalo na platforme Linux s formátom binárnych súborov ELF. Pri viacerých fázach vývoja boli objavené nedostatky pri načítaní tohto formátu, menovite pri práci s objektovými súbormi (nezlinkované programy). Na obrázku 5.5 je zobrazený aktuálny výstup spätného prekladu (vpravo) pre zdrojový kód (vľavo). Ako je možné vidieť výstup nie je správny a obsahuje nekompletné funkcie.

```c
#include <stdio.h>
#include <math.h>

const char* sfoo = "foo";
const char* sbar = "bar";

int foo(int x)
{
  puts(sfoo);
  return sqrt(x);
}

int bar(int x)
{
  puts(sbar);
  return x + x;
}

int main()
{
  puts("Hello World!");
  printf("%d %d\n", foo(5), bar(5));
  return 0;
}
```

```c
#include <stdint.h>
#include <stdlib.h>

int64_t bar(void);
int64_t foo(void);

// Address range: 0x0 - 0x1
int64_t foo(void) {
  // 0x0
  int64_t result; // x0
  return result;
}

// Address range: 0x34 - 0x4c
int64_t bar(void) {
  // 0x34
  abort();
  // UNREACHABLE
}

// Address range: 0x60 - 0x74
int main(int argc, char ** argv) {
  // 0x60
  return 0;
}
```

Obr. 5.5: Súčasný stav prekladu objektových súborov.

Objektové súbory vznikajú pri preklade bez následného linkovania modulov. Zdrojové súbory sú preložené do objektových súborov obsahujúcich strojový kód. Ten nie je v spustiteľnom formáte pretože, sa v ňom vyskytujú nevyriešené relokácie. Nastáva to napríklad pri volaní funkcie `printf` zo štandardnej knižnice jazyka C. Na tomto mieste v kóde sa síce vygeneruje inštrukcia skoku ale jej argumentom je typicky hodnota 0, čo v konečnom dôsledku reprezentuje relatívny skok na samého seba. V relokačných tabuľkách je potom uložený záznam o tejto relokácii. Spolu s hodnotou sa odkazuje na symbol, ktorý reprezentuje meno funkcie v pripájanej knižnici.

Takýto kód typicky zhoršuje výsledky spätného prekladu. Často krát sa vo výsledkoch vyskytovali nekompletné funkcie, chýbajúce funkcie alebo v niektorých extrémnych prípadoch nebola rozpoznaná ani funkcia `main`.

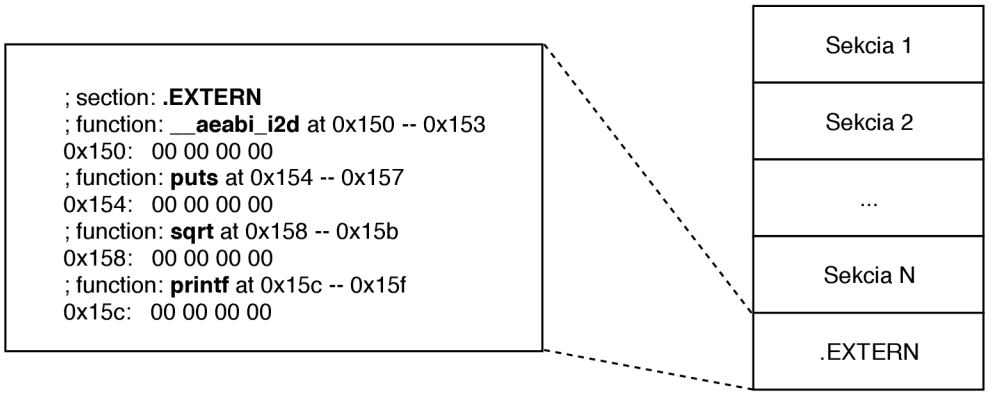
Navrhovaným riešením je pri načítaní súboru simulovať prácu *statického linkeru*. Pre importované funkcie bude vytvorený virtuálny segment. Segment je označený ako virtu-

álny, pretože sa nenachádza v načítavanom súbore. Je vytvorený umelo a obsahuje práve dostatok miesta pre relokované funkcie. Veľkosti záznamov ako aj celého segmentu závisia od načítaného programu.

1. Veľkosť záznamu je určená triedou súboru ELF, teda 64 bitov pre `CLASS64` a 32bitov pre `CLASS32`.

2. Celková veľkosť segmentu je určená počtom importovaných funkcií vynásobených veľkosťou jedného záznamu. Funkcie, ktoré sa vyskytujú viacnásobne sú ignorované a do úvahy sa berie iba ich prvý výskyt.

Na nasledujúcom diagrame 5.6 je zobrazená navrhnutá reprezentácia ELF súboru v pamäti.



Obr. 5.6: Reprezentácia súboru typu ELF v pamäti.

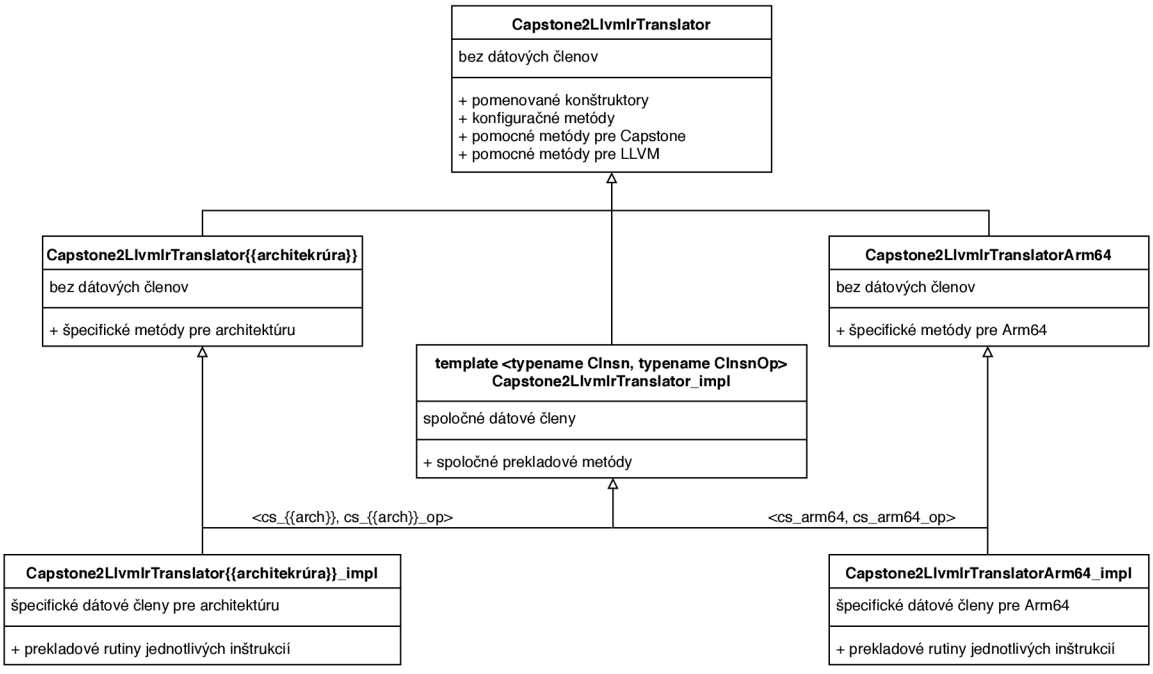
Kapitola 6

Implementácia

V tejto kapitole sa zameriame na popis jednotlivých rozšírení resp. nových programov, ktoré boli počas práce implementované. Program RetDec je implementovaný v jazyku C++ (štandard C++11), preto všetky rozšírenia sú taktiež v tomto jazyku. Ďalej bol pre rôzne pomocné programy používaný jazyk `Python` vo verzii 3.7. Zmeny sa riadia návrhom, popísaným v kapitole 5.

6.1 Knižnica Capstone2LlvmIr

Modul `capstone2llvmir`, obsahuje pre každú architektúru osobitný prekladač do inštrukcií LLVM IR. Rozhranie pre jednotlivé prekladače je pevne stanovené. Na to aby bol program RetDec schopný pracovať s novou architektúrou je nutné dodržať toto rozhranie. Na diagrame 6.1 je zobrazená architektúra modulu `Capstone2LlvmIr`. Reťazce `{{architektúra}}` a `{{arch}}` reprezentujú meno architektúry implementovanej daným prekladačom.



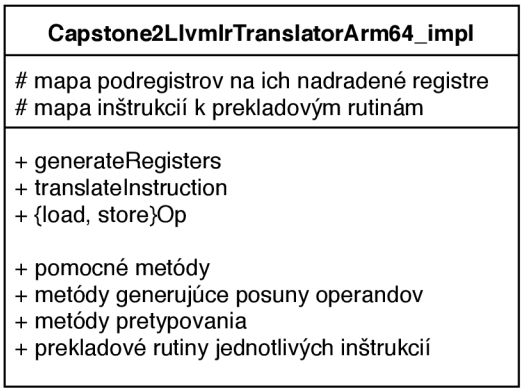
Obr. 6.1: Zjednodušený diagram tried pre prekladový modul ARM64.

Popisovanú štruktúru je možné rozlíšiť na *verejné* a *súkromné* časti. Vo verejných hlavičkových súboroch (štandardne umiestnených v priečinku `include/`) sa nachádzajú iba deklarácie metód a dátových členov nevyhnutne nutných pre využívanie knižnice. Hlavičkové súbory obsahujúce ostatné deklarácie sú „skryté“ v adresárovej štruktúre pri súboroch s implementáciou spomínaných tried. Motiváciou za týmto rozdelením je odtienenie používateľa knižnice od preňho nepodstatných implementačných detailov.

Trieda `Capstone2LlvmIrTranslator` definuje verejné rozhranie spoločné pre všetky prekladače. Toto rozhranie ďalej implementuje trieda `Capstone2LlvmIrTranslator_impl` špecializovaná Capstone typmi reprezentujúcimi inštrukciu a operand pre použitú architektúru. Napríklad pre architektúru ARM64 sú to štruktúry `cs_arm64` a `cs_arm64_op`, bližšie popísané v sekcii 4.3.

6.1.1 Modul prekladača architektúry ARM64

Jadro práce je implementované v triede `Capstone2LlvmIrTranslatorArm64_impl`. Na obrázku 6.2 je zobrazený jej diagram.



Obr. 6.2: Zjednodušený diagram tried pre triedu `Capstone2LlvmIrTranslatorArm64_impl`.

Metóda `translateInstruction` je zavolaná vždy pri zahájení prekladu inštrukcie. Jej úlohou je vyhľadať správnu prekladovú rutinu. Vyhľadávanie prebieha v mape spájajúcej `ID` inštrukcie s prekladovou metódou. Ak je vyhľadávanie úspešné, metóda je zavolaná s príslušnými parametrami. Ak ale nie je možné nájsť správnu rutinu postupuje sa nasledovne:

1. Prebehne kontrola, či je daná inštrukcia podmienená. Informácie sú dostupné v štruktúre `cs_insn* i` a podmienené inštrukcie sú bližšie popísané v kapitole 3.4.1.

   (a) Ak je inštrukcia podmienená, vygeneruje sa riadiaci príkaz `IF` zodpovedajúci podmienke.

2. Preklad sa dokončí automatickým generovaním pseudo inštrukcie.

Tento postup je zvolený, pretože ARM64 je jednou zo špecifických architektúr, ktoré podporujú podmienené vykonávanie na úrovni jednotlivých inštrukcií. Ak by bola vygenerovaná iba pseudo inštrukcia, stratila by sa informácia o toku vykonávania programu, čo by mohlo negatívne ovplyvniť spätný preklad.

Trieda obsahuje samotné definície metód realizujúcich generovanie LLVM IR kódu. Generovanie prebieha volaním metód z knižnice LLVM, podľa sémantiky práve spracovávanej

inštrukcie. Typicky prekladové rutiny sú schopné preložiť viacero pridružených inštrukcií s podobnou sémantikou. Všetky prekladové rutiny prijímajú rovnaké parametre. Sú to:

`cs_insn* i` – ukazateľ na všeobecnú reprezentáciu Capstone inštrukcie. Tá obsahuje `ID` spracovávanej inštrukcie, ako aj jej textovú reprezentáciu.

`cs_arm64* ai` – ukazateľ na Capstone reprezentáciu ARM64 inštrukcie, bližšie popísanej v kapitole 4.3. Táto štruktúra sa najčastejšie využíva na prístup k informáciám o operandoch.

`llvm::IRBuilder<>& irb` – referencia na objekt poskytujúci jednotné rozhranie pre vytváranie a vkladanie inštrukcií. Počas prekladu sú prostredníctvom tohto objektu generované inštrukcie odpovedajúce sémantike spracovávanej inštrukcie.

6.2 Systémové volania

Prvým krokom implementácie tejto optimalizácie bolo nájdenie informácií o systémových volaniach pre systém Linux. Tieto informácie boli získané z rôznych zdrojov[1,2] pričom medzi nimi bola overovaná správnosť. Z informácií bolo vytvorené mapovanie medzi hodnotou špecifikujúcou systémové volanie a jej odpovedajúcim menom.

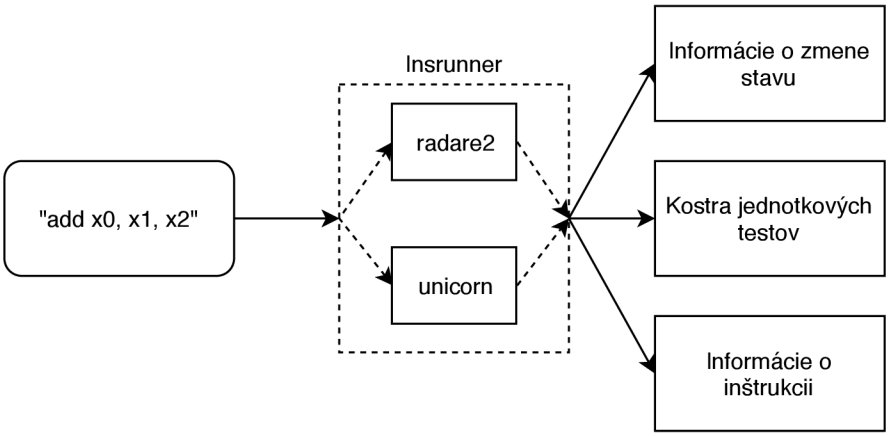
Analýza pozostáva z lineárneho priechodu LLVM IR inštrukcií. Počas priechodu je vyhľadávaná inštrukcia `svc #0`, ktorá iniciuje systémové volanie. V prípade výskytu sa analýza pokúsi zrekonštruovať hodnotu v registri. Meno registra je získané z informácií o `ABI` pre danú architektúru, čo je v tomto prípade register `X8`. Po odhade hodnoty je v tabuľke vyhľadané systémové volanie korešpondujúce k danej hodnote. Posledným krokom je transformovanie týchto inštrukcií na volanie funkcie s názvom systémového volania.

6.3 Insrunner

Veľký dôraz je kladený na spôsob vykonávania inštrukcií. Prvotný prototyp je založený na používaní `radare2`. Radare2 je framework určený pre reverzné inžinierstvo, ktorý poskytuje vhodné programové rozhranie pre ladenie programov. Výhodou je, že inštrukcie sú spúšťané priamo na stroji s cieľovou architektúrou a teda výsledky sú veľmi presné. Tento prístup ale prináša obmedzenie na akom počítači je možné inštrukcie testovať. Ak chceme napríklad testovať inštrukcie z inštrukčnej sady `A64` je nutné spúšťať Insrunner na systéme s architektúrou ARM64. Aj napriek nevýhodám je toto riešenie preferované pred emuláciou, ktorá pridáva medzi výsledok inštrukcie ďalšiu vrstvu spracovávania, kde sa môžu vyskytnúť chyby.

Program je implementovaný v jazyku Python a jeho návrh umožňuje pridávanie nových modulov vykonávajúcich inštrukcie. Existuje možnosť pridať debugger založený na emulátore `Unicorn`[3]. Toto rozšírenie by umožnilo používateľovi spúšťať inštrukcie inej architektúry ako tej, na ktorej beží hosťovský systém. Na obrázku 6.3 je možné vidieť prehľad funkcionality programu.

[1] https://thog.github.io/syscalls-table-aarch64/latest.html
[2] https://github.com/hugsy/cemu/blob/master/cemu/syscalls/aarch64.csv
[3] https://www.unicorn-engine.org/



Obr. 6.3: Schéma použitia programu Insrunner.

6.4 Importované funkcie vo formáte ELF

Rozšírenie je implementované vo funkcii `ElfImage::createExternSegment`, ktorá sa nachádza v module *loader*. Tento modul simuluje načítanie a zavádzanie programov do pamäti. Implementácia sa riadi návrhom diskutovaným v kapitole 5.9. Funkcia má za úlohu:

1. Prejsť segmenty spracovávaného súboru a vypočítať počiatočnú adresu nového segmentu.
2. Zistiť veľkosť jedného záznamu.
3. Spracovať všetky symboly v tabuľke importov, pričom vyberá unikátne záznamy.
4. Inicializuje nový objekt reprezentujúci segment a následne ho zaradí k pôvodným segmentom.

Trieda `ElfImage` bola rozšírená o dátové členy, ktoré reprezentujú nový segment. Spracované informácie o unikátnych importovaných funkciách sú uložené v *mape*, ktorá spája symbol s adresou v novom segmente. Zároveň boli rozšírené podporované typy relokácii pre 64-bitové architektúry `x86` a ARM64 podľa špecifikácie súborov `ELF` [6, 2].

Kapitola 7

Testovanie

Cieľom testovania bolo určiť kvalitu spätného prekladu na rôznych ukážkových príkladoch, overenie sémantického významu jednotlivých inštrukcií, ale aj odolnosť a schopnosť zotaviť sa z chýb pri spätnom preklade. Jednoduché testovanie prebiehalo samostatným prekladom zdrojových súborov do programov pre cieľovú architektúru a následný spätný preklad s porovnaním výsledkov. Tento prístup ale pri veľkosti a zložitosti spätného prekladača nebol dostatočný. Toto podmienilo vznik viacerých testovacích metód na rôznych úrovniach.

1. Testovanie sémantiky inštrukcií.
2. Regresné testovanie.
3. Testovanie výsledkov spätného prekladu.

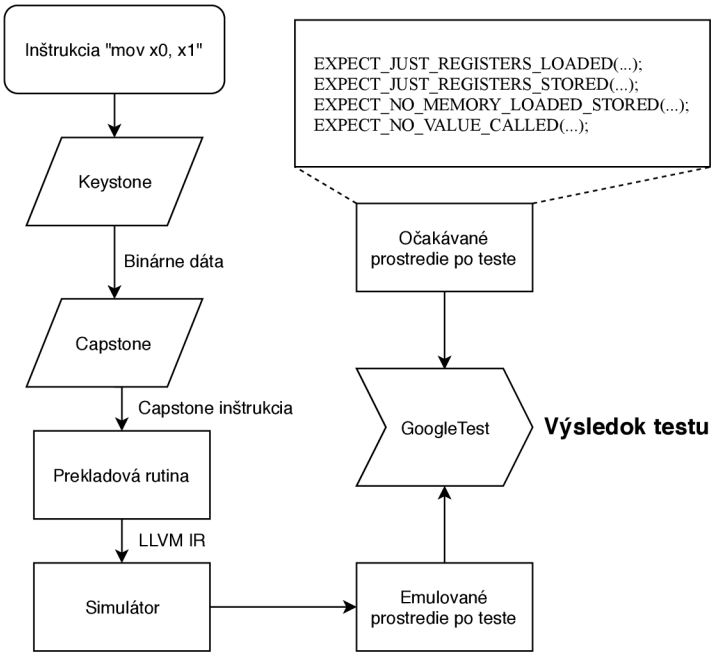
7.1 Jednotkové testy

Testovanie prekladu inštrukcií do LLVM IR prebiehalo najmä za pomoci jednotkových testov. Každá implementovaná inštrukcia má vždy pridelený aspoň jeden test, ktorý overuje správnosť viacerých aspektov prekladu danej inštrukcie. Bližšie informácie o počte testov a rozsahu testovaných inštrukcií je možné nájsť v prílohe C.1.

Implementácia inštrukcií prebiehala nasledovne. Z manuálu architektúry ARM bolo zistené očakávané fungovanie danej inštrukcie. Správanie inštrukcií bolo často do veľkej miery ovplyvnené počtom a typom operandov. Toto chovanie bolo následne overené na stoji s procesorom `ARM`. Z tohoto overenia boli neskôr vybrané testovacie príklady, ktoré boli prevedené do jednotkových testov. Nakoniec po implementovaní prekladovej rutiny danej inštrukcie bola pomocou daného jednotkového testu overená správnosť prekladu.

Framework pre jednotkové testy (na obrázku 7.1) je navrhnutý tak, že na začiatku sa nastaví prostredie testu. To zahrňuje nastavenie hodnoty registrov resp. pamäte pred začatím testu. Následne je emulovaná inštrukcia voči tomuto prostrediu. Výsledky je možné overiť priamym porovnaním hodnoty v registroch alebo aj overením operácií čítania a zapisovania nad určitými registrami. Nasledujúci zoznam popisuje všetky možnosti testovania emulácie inštrukcie.

- Z ktorých registrov bolo vykonané čítanie.
- Do ktorých registrov bol vykonaný zápis a aký obsah ma register.
- Z akej adresy v pamäti bolo vykonané čítanie.



Obr. 7.1: Schéma vyhodnotenia jednotkového testu.

- Na aké adresy v pamäti bol vykonaný zápis a aká hodnota bola zapísaná.
- Prípadné volanie inej LLVM funkcie aj s danými parametrami.

Na obrázku 7.2 je možné vidieť príklad jednotkového testu. Testy sú písané vo frameworku GoogleTest[1]. Názov testu je zostavený tak aby čo najpresnejšie odrážal testovaný aspekt inštrukcie. Pred testovaním inštrukcie je možné voliteľne nastaviť prostredie testu (obsah pamäte a registrov). Volanie funkcie `emulate` spúšťa emuláciu inštrukcie. Na riadku 10 začínajú kontrolné príkazy, ktoré kontrolujú stav po vykonaní inštrukcie s očakávaným stavom.

7.2 Generovanie programov pre ARM64

Neodlučiteľnou súčasťou testovania bolo generovanie testovacích programov. Typicky prekladače generujú výsledné programy pre cieľovú architektúru zhodnú s hosťovskou. Na osobných počítačoch v súčasnosti prevláda architektúra x86 alebo jej deriváty, preto je nutné používať prekladač s cieľovou architektúrou ARM64. Najjednoduchšou cestou ako získať vhodný prekladač je pomocou balíčkovacích systémov Linuxových distribúcií[2,3].

7.3 Regresné testy

Regresné testy slúžia k overeniu správnosti aplikácie po pridaní novej funkcionality alebo zmene existujúcich vlastností programu. Po pridaní nových možností prekladu bolo nutné

[1] https://github.com/google/googletest
[2] https://www.archlinux.org/packages/community/x86_64/aarch64-linux-gnu-gcc/
[3] https://packages.ubuntu.com/trusty/devel/gcc-aarch64-linux-gnu

```
TEST_P(Capstone2LlvmIrTranslatorArm64Tests, ARM64_INS_ADD_s_negative_r_r_r)
{
    setRegisters({
        {ARM64_REG_X1, 0xffff000000000000},
        {ARM64_REG_X2, 0x1234},
    });

    emulate("adds x0, x1, x2");

    EXPECT_JUST_REGISTERS_LOADED({ARM64_REG_X1, ARM64_REG_X2});
    EXPECT_JUST_REGISTERS_STORED({
        {ARM64_REG_X0, 0xffff000000001234},
        {ARM64_REG_CPSR_N, true},
        {ARM64_REG_CPSR_Z, false},
        {ARM64_REG_CPSR_C, false},
        {ARM64_REG_CPSR_V, false},
    });
    EXPECT_NO_MEMORY_LOADED_STORED();
    EXPECT_NO_VALUE_CALLED();
}
```

Obr. 7.2: Príklad jednotkového testu.

aby všetky regresné testy prešli. Ak by sa tak nestalo, znamenalo by to, že niektoré nami vykonané zmeny upravili doteraz správne fungovanie programu. Keďže ale väčšina vývoja prebiehala v nových resp. oddelených moduloch, výsledky testov nevykazovali žiadne väžnejšie chyby. Bolo teda potrebné ich len rozšíriť o testovacie príklady zamerané na overenie funkčnosti novej funkcionality. Zdrojové kódy týchto súborov pochádzali zo spoločných testov pre všetky architektúry. Testovacie programy boli vygenerované cross-prekladačom pre architektúru AArch64. Bolo overené, že v týchto prípadoch je možný spätný preklad minimálne na rovnakej úrovni ako to zvládli ostatné architektúry. Testované aspekty boli napríklad:

- Preklad riadiacich výrazov `for`, `while`, `do while`, a pod.
- Testovanie prekladu práce so zásobníkom.
- Operácie s číslami s plávajúcou desatinnou čiarkou.
- Preklad funkcií ako `strlen` alebo `bitcnt`.
- Práca s globálnymi premennými.

7.4 Nočné testy

Pred integráciou zmien do hlavnej vetvy bolo vykonané podrobné testovanie. Pozostávalo z viacerých fáz. Prekladač sa testoval na množine binárnych súborov. Následne sa testovacie súbory generovali zo zdrojových kódov preložených s rôznymi nastaveniami prekladača.

1. `-O{0-3}` Rôzne úrovne optimalizácie kódu
2. `--strip` Odstránenie tabuľky symbolov

3. `-g` Preklad s ladiacimi informáciami

Na snímke obrazovky je vidieť výsledky nočných testov. V prvom riadku je zobrazené v koľkých percentách bol spätný preklad úspešný a v druhom riadku je vidieť v koľkých prípadoch bolo možné znovu preložiť výsledok. Farba signalizuje rozdiel medzi základom testu a aktuálnym výsledkom. Je sýto zelená pretože v minulých verziách nebol možný spätný preklad.

arm64/elf	gcc/O0	gcc/O1	gcc/O2	gcc/O3
C generation result (%)	100.0	100.0	100.0	100.0
C syntax result (%)	72.6	83.4	83.4	82.8

Obr. 7.3: Výsledky posledných nočných testov.

7.5 QEMU virtuálne stroje

Z počiatku jednoduché testy prebiehali vo virtuálnom stroji za pomoci nástroja QEMU[4]. Emuláciou systémov bolo možné sledovať ako sa jednotlivé inštrukcie správajú za daných podmienok. Toto riešenie ale veľmi rýchlo prestalo byť relevantné kvôli rýchlosti emulovaných strojov.

7.6 Počítače Raspberry PI

Postupne vznikala potreba testovať programy a jednotlivé inštrukcie na reálnom stroji s integrovaným procesorom ARM. Testovanie prebiehalo na mikropočítači Raspberry PI, ktorý obsahuje procesor Cortex-A53 (ARMv8) so stavom vykonávania AArch64.

Problémom bolo, že väčšina predpripravených operačných systémov pre toto zariadenie obsahuje Linux s 32-bitovým jadrom. Takéto systémy síce bežia na architektúre ARM ale je možné využívať iba stav vykonávania AArch32. Dôsledkom toho neumožňujú spúšťanie 64-bitových programov. Riešením je zostaviť si vlastné 64-bitové jadro alebo použiť operačný systém s priloženým 64-bitovým jadrom. Na základe týchto požiadavkou bola zvolená varianta distribúcie Kali Linuxu zameraná na počítače Raspberry Pi s 64-bitovými procesormi[5]. Vybraný operačný systém obsahoval všetky potrebné nástroje. Menovite to boli assembler `GAS`, prekladač `GCC`, linker a rôzne ladiace nástroje.

7.7 Výsledky testov

Určovanie správnosti spätného prekladu prebiehalo hodnotením syntaktickej a sémantickej stránky výsledných súborov. Množina zdrojových kódov v jazyku C bola prekladaná pre architektúru ARM64 s rôznymi úrovňami optimalizácie. Následne bol vykonaný spätný preklad pričom sa porovnávala podobnosť výsledkov. Práca bola testovaná na rôznych úrovniach aby bola zaručená čo najväčšia presnosť spätného prekladu.

[4] https://www.qemu.org/
[5] https://www.offensive-security.com/kali-linux-arm-images/#1536677610546-d94f090d-c5ee

Kapitola 8

Výsledky implementácie

V nasledujúcej kapitole sú prezentované výsledky spätného prekladu. Predvedené budú na základných konštrukciách jazyka C. Ukážky sú vo forme porovnaní, kde vždy na pravej strane je výsledok. Na ľavej strane je uvedený buď pôvodný zdrojový kód alebo sekvencia inštrukcií, ktorá je predmetom spätného prekladu. V prípadoch kde to dáva zmysel bude uvedený komentár k porovnaniu. V prílohe C.1 a C.2 sú uvedené výsledné metriky kódu.

```
int v;
scanf("%d", &v);

if (v~== 5) {
    printf("Rovna sa 5\n");
} else {
    printf("Nerovna sa 5\n");
}
```

```
int32_t v1; // bp-4
scanf("%d", &v1);
if (v1 == 5) {
    // 0x4006d0
    puts("Rovna sa 5");
} else {
    // 0x4006e0
    puts("Nerovna sa 5");
}
```

Obr. 8.1: Spätný preklad úplného podmieneného príkazu.

```
iteration:
    str wzr, [sp + 0x2c]
    b condition
    ldr w1, [sp + 0x2c]
    adrp x0, segment.ehdr
    add x0, x0, 0x690
    bl sym.imp.printf
    ldr w0, [sp + 0x2c]
    add w0, w0, 1
    str w0, [sp + 0x2c]
condition:
    ldr w0, [sp + 0x2c]
    cmp w0, 4
    ble 0x4005ac
```

```
for (int64_t i = 0; i < 5; i++) {
    // 0x4005ac
    printf("iteracia: %d\n", i);
}
```

Obr. 8.2: Spätný preklad riadiaceho príkazu `for`.

Rozpoznávanie parametrov variadickej funkcie, je pri konštantných hodnotách totožné s pôvodným kódom.

```
printf("%*s", 4, "abcd");
printf("%*d", 4, 1);
printf("% d", 1);
printf("%-d", 1);
printf("%+d", 1);
printf("%*.*s\n", 55, 66, "abcd");
```

```
printf("%*s", 4, "abcd");
printf("%*d", 4, 1);
printf("% d", 1);
printf("%-d", 1);
printf("%+d", 1);
printf("%*.*s\n", 55, 66, "abcd");
```

Obr. 8.3: Spätný preklad volaní funkcie `printf`.

V niektorých prípadoch nie sú správne zrekonštruované typy parametrov alebo návratový typ funkcie. To vedie k zavedeniu nepresností do výsledného kódu. Význam kódu nie je ovplyvnený, pričom ale musia byť vygenerované pomocné operácie aby boli rešpektované detekované typy.

```
// (fcn) sym.add_sub 44
add_sub:
    sub sp, sp, 0x10
    str w0, [sp + 0xc]
    str w1, [sp + 0x8]
    str w2, [sp + 0x4]
    ldr w1, [sp + 0x8]
    ldr w0, [sp + 0x4]
    mul w1, w1, w0
    ldr w0, [sp + 0xc]
    add w0, w1, w0
    add sp, sp, 0x10
    ret
```

```
int64_t add_sub(int64_t a1, int64_t a2, int64_t ↩
    a3) {
    return a3 * a2 + a1 & 0xffffffff;
}
```

Obr. 8.4: Spätný preklad jednoduchej funkcie.

Na nasledujúcej ukážke je možné vidieť, že výsledok nie je 100% zhodný s pôvodným kódom. No v konečnom dôsledku sa zhodujú významom. Takéto prípady nastávajú napríklad z dôvodov optimalizácie kódu. Pokiaľ sú ale zhodné významom, je aj takýto výsledok spätného prekladu považovaný za správny.

```
int factorial(int n) {
    if (n == 0)
        return 1;
    return n*factorial(n-1);
}
```

```
int64_t factorial(int32_t a1) {
    int64_t result;
    if (a1 != 0) {
        // 0x4005e0
        result = factorial(a1 - 1) * (↩
            int64_t)a1 & 0xffffffff;
    } else {
        result = 1;
    }
    // 0x4005f8
    return result;
}
```

Obr. 8.5: Spätný preklad funkcie `factorial`.

Kapitola 9

Záver

V práci boli vytýčené základy pre spätný preklad programov pre architektúru ARM64. Teoretické poznatky pozostávali z úvodu do reverzného inžinierstva, popisu architektúry `ARM64`, rozobratia častí programu RetDec a základov reprezentácie LLVM IR.

V návrhu sú prediskutované potrebné úkony pre vytvorenie nového modulu spätného prekladača. Je v ňom uvedené prepojenie znalostí o architektúre ARM s reprezentáciou LLVM IR a následné zakomponovanie do spätného prekladača RetDec. Diskutované je aj rozšírenie vylepšujúce spätný preklad objektových súborov formátu `ELF`.

Výsledkom práce je rozšírenie knižnice `Capstone2LlvmIr` schopné spätného prekladu inštrukcií zo sady `A64` do LLVM IR. Implementované sú rozšírenia niektorých optimalizačných priechodov. Ku každej inštrukcii bola vytvorená sada jednotkových testov, ktoré overujú správnosť sémantického významu danej inštrukcie. Ďalej bola definovaná množina regresných testov, pre pridané rozšírenia. Ako rozšírenie práce bola vylepšená podpora spätného prekladu objektových súborov. Pre účely testovania vznikla pomocná utilita s názvom `Insrunner`. Výsledky spätného prekladu boli overené automatickými testami určujúcimi mieru podobnosti medzi vstupným a výstupným kódom.

Architektúra ARM je neustále aktívne vyvíjaná a je pravdepodobné, že v budúcnosti budú nutné rozšírenia tejto práce. Taktiež projekt Capstone, postupne zlepšuje podporu tejto architektúry, čo môže ovplyvniť projekt v budúcnosti. Prácu je možné v súčasnej podobe vylepšiť rozšírením podpory vektorových inštrukcií, ktoré sú generované ako pseudo inštrukcie.

Literatúra

[1] The A64 instruction set. Version 1,0. [vid. 14 Okt 2018]. Dostupné z: https://static.docs.arm.com/100898/0100/the_a64_Instruction_set_100898_0100.pdf.

[2] Executable and Linkable Format (ELF) [online]. Portable Formats Specification, Version 1,1. [vid. 23 Mar 2019]. Dostupné z: http://www.skyfree.org/linux/references/ELF_Format.pdf.

[3] Fundamentals of ARMv8-A. Version 1,0. [vid. 10 Jan 2018]. Dostupné z: https://static.docs.arm.com/100878/0100/fundamentals_of_armv8_a_100878_0100_en.pdf.

[4] *IEEE standard for binary floating-point arithmetic*. New York: Institute of Electrical and Electronics Engineers, 1985.

[5] *IEEE standard for binary floating-point arithmetic*. New York: Institute of Electrical and Electronics Engineers, 2008, version: IEEE 754-2008 revision.

[6] ARM Limited: *ELF for the ARM® 64-bit Architecture (AArch64)*. 5 2013, version: 1.0.

[7] ARM Limited: *Procedure Call Standard for the ARM 64-bit Architecture(AArch64)*. 5 2013, [vid. 20 Feb 2019]. Dostupné z: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf.

[8] ARM Limited: *ARM Architecture Reference Manual® ARMv8, for ARMv8-A architecture profile*. 6 2016, [vid. 10 Okt 2018]. Dostupné z: https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile.

[9] Cheng, H.-J.; Hwang, Y.-S.; Chang, R.-G.; aj.: Trading Conditional Execution for More Registers on ARM Processors. Január 2011, doi:10.1109/EUC.2010.18. URL https://ieeexplore.ieee.org/document/5703498

[10] Eilam, E.: *Reversing: Secrets of Reverse Engineering*. Wiley, prvé vydanie, 2005, ISBN 978-0764574818.

[11] Křoustek, J.; Matula, P.; Zemek, P.: RetDec: An Open-Source Machine-Code Decompiler. [talk], December 2017, botconf 2017, Montpellier, FR. Dostupné z: https://retdec.com/static/publications/retdec-slides-botconf-2017.pdf.

[12] Matula, P.; Milkovič, M.: An Open-Source Machine-Code Decompiler. Technická správa, June 2018, prezentované na RECon 2018, Montreal, CA. Dostupné z: https://retdec.com/static/publications/retdec-slides-recon-2018.pdf.

Príloha A

Podmienené vykonávanie na AArch64

Identifikátor	**Význam**	**Podmienka**
EQ	Rovné	Z == 1
NE	Nerovné	Z == 0
CS alebo HS	Príznak prenosu nastavený	C == 1
CC alebo LO	Príznak prenosu nenastavený	C == 0
MI	Mínus, negatívne	N == 1
PL	Plus, pozitívne alebo nula	N == 0
VS	Príznak pretečenia nastavený	V == 1
VC	Príznak pretečenia nenastavený	V == 0
HI	Bezznamienkové: vačšie	C == 1 && Z == 0
LS	Bezznamienkové: menšie alebo rovné	!(C == 1 && Z == 0)
GE	Znamienkové: vačšie alebo rovné	N == V
LT	Znamienkové: menšie	N != V
GT	Znamienkové: vačšie	Z == 0 && N == V
LE	Znamienkové: menšie alebo rovné	!(Z == 0 && N == V)
AL	Vždy	Nezáleží
NV	Vždy	Nezáleží

Tabuľka A.1: Kódy pre podmienené vykonávanie inštrukcií.

Príloha B

Význam inštrukcií používaných v texte

Inštrukcia	**Význam**
`add`	Sčíta operandy a výsledok uloží do registra.
`adr`	Načíta adresu návestia do registra.
`adrp`	Načíta adresu návestia zarovnanú na 4KB stránku do registra.
`b`	Nepodmienený skok na návestie.
`bl`	Volanie funkcie. Inštrukcia vykoná skok a uloží hodnotu `PC` do `LR`.
`b<cond>`	Podmienený skok na návestie. Vykoná sa len prípade, že platí `<cond>`.
`ccmp`	Ak je platná podmienka (posledný operand), inštrukcia porovná obsah registrov a podľa výsledku porovnania nastaví príznaky, inak nastaví hodnotu príznakov na hodnotu predposledného operandu.
`cinc`	Uloží hodnotu do registra, pričom ak je platná podmienka (posledný operand) hodnota je predtým inkrementovaná.
`cmp`	Porovná obsah registrov a podľa výsledku porovnania nastaví príznaky.
`csel`	Ternárny operátor.
`eret`	Návrat z obsluhy výnimky.
`ldr`	Načíta hodnotu z pamäte do registra.
`mov`	Presunie hodnotu druhého operandu do prvého.
`mul`	Vynásobí operandy a výsledok uloží do registra.
`nop`	Inštrukcia, ktorá nevykoná žiadnu operáciu.
`ret`	Nepodmienený skok na `LR` register, návrat z funkcie.
`str`	Uloží hodnotu registra do pamäte.
`sub`	Odčíta operandy a výsledok uloží do registra.

Tabuľka B.1: Vysvetlenie významu inštrukcií používaných v texte.

Príloha C

Metriky projektu

Metrika	Hodnota
Počet riadkov kódu	~14000
Počet riadkov kódu (bez testov)	~5000
Počet implementovaných inštrukcií (podľa Capstone ID)	143
Počet prekladových rutín	55
Počet pseudo inštrukcií	~300
Počet jednotkových testov	419
Počet implementovaných ale netestovaných inštrukcií	0
Počet regresných testov	24

Tabuľka C.1: Štatistiky na projekte RetDec.

Metrika	Hodnota
Počet riadkov kódu	~500

Tabuľka C.2: Štatistiky programu Insrunner.

Príloha D

Obsah pamäťového média

Súbor	Popis
`README.txt`	Dodatočné informácie o obsahu a preklade projektu.
`doc/`	Elektronická verzia tejto správy.
`doc_src/`	Zdrojové súbory tejto správy.
`insrunner/`	Zdrojové kódy programu insrunner.
`retdec/`	Aktuálna verzia RetDec s podporou prekladu ARM64.
`retdec_install/`	Preložený program (info v `README.txt`).
`test_bins/`	Testovacie súbory.

Tabuľka D.1: Popis jednotlivých súborov na priloženom pamäťovom médiu.