VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

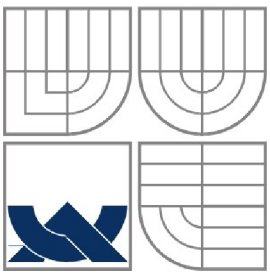
PŘEVOD BINÁRNÍHO KÓDU X86 DO VYŠŠÍHO PROGRAMOVACÍHO JAZYKA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

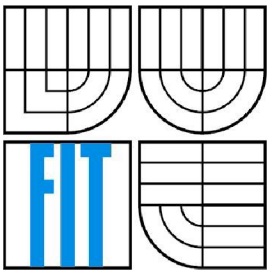
AUTOR PRÁCE
AUTHOR

Bc. Marián Jurík

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘEVOD BINÁRNÍHO KÓDU X86 DO VYŠŠÍHO PROGRAMOVACÍHO JAZYKA

TRANSLATION OF X86 BINARY CODE TO A HIGH-LEVEL LANGUAGE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Marián Jurík

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Dr. Ing. Dušan Kolář

BRNO 2008

Zadání diplomové práce/6462/2007/xjurik01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2007/2008

Zadání diplomové práce

Řešitel: **Juríik Marián, Bc.**
Obor: Informační systémy
Téma: **Převod binárního kódu x86 do vyššího programovacího jazyka**
Kategorie: Překladače

Pokyny:

1. Seznamte se s formátem PE (formát Windows 32/64). Seznamte se s instrukční sadou procesorů Intel x86 a kompatibilních.
2. Seznamte se s typickými obraty převodu programových konstrukcí v dostupných překladačích pro platformu Windows/Intel.
3. Navrhněte a implementujte pro danou architekturu zpětný překladač z binárního kódu do jazyka symbolických instrukcí (disassembler).
4. Navrhněte strohý programovací jazyk se základními konstrukcemi a jazykem symbolických instrukcí odvozeným z C/C++.
5. Rozšiřte program (bod 3), aby výstupem byl navržený jazyk (4). Pro neoptimalizované výstupy z tradičních překladačů bude program poskytovat výstup, který nebude obsahovat více jak 20% výstupu v jazyku symbolických instrukcí (kromě zavaděče apod.).
6. Otestujte program z bodu 5 na dostatečném množství (více jak 20) vzorcích.
7. Diskutujte klady a nedostatky práce, zhodnoťte její přínos, navrhněte možná rozšíření a úpravy.

Literatura:

- Dle doporučení vedoucího a konzultanta.

Při obhajobě semestrální části diplomového projektu je požadováno:

- První 4 body, včetně cca 30 stránkové zprávy.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVR-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kolář Dušan, doc. Dr. Ing.**, UIFS FIT VUT
Konzultant: Obluk Karel, Ing., Ph.D., Grisoft
Datum zadání: 24. září 2007
Datum odevzdání: 19. května 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. **Pan**

Jméno a příjmení: **Bc. Marián Juríк**
Id studenta: 89421
Bytem: Jesenského 3, 941 01 Bánov
Narozen: 27. 06. 1983, Nové Zámky
(dále jen "autor")

a

2. **Vysoké učení technické v Brně**

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
diplomová práce

Název VŠKP: Převod binárního kódu x86 do vyššího programovacího jazyka
Vedoucí/školitel VŠKP: Kolář Dušan, doc. Dr. Ing.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☒ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

....

Nabyvatel Autor

Abstrakt

Cieľom diplomovej práce je navrhnúť a implementovať program na prevod binárneho kódu do vyššieho programovacieho jazyka. Táto práca sa zameriava na binárne súbory pre operačný systém MS Windows. V práci je podrobne popísaný súborový formát PE, ktorý okrem iného definuje spôsob ukladania binárneho kódu do súboru. Taktiež je popísaná inštrukčná sada IA-32, kde hlavný dôraz bol kladený na spôsob dekódovania binárneho kódu do jazyka symbolických adries. V texte práce sú popísané typické konštrukcie používané pri preklade. Súčasťou práce je návrh vyššieho programovacieho jazyka. Návrh vychádza z existujúcich jazykov C, C++ a jazyka symbolických adries. Predposledná kapitola pojednáva o návrhu programu a samotnej implementácie. V závere práce sú zhodnotené výhody a nevýhody práce.

Kľúčové slová

binárny súbor, PE formát, inštrukcia, inštrukčná sada IA-32, prekladač, programovací jazyk

Abstract

The purpose of this MSc thesis is to create design and implementation of program for translation of x86 binary code to a high-level programming language. There is described PE file format for executables used in MS Windows operating systems in the first part of work. This document contains general information about instruction set IA-32, especially a way of decoding binary code to assembly language. There are described typical program constructions, which are being used in compilers. Design of creation high-level programming language was inspired by existing programming languages. Conclusion is made about advantages and disadvantages of approach used in this thesis.

Keywords

binary file, PE file format, instruction, instruction set IA-32, disassembler, compiler, programming language

Citácia

Jurík Marián: Převod binárního kódu x86 do vyššího programovacího jazyka. Brno, 2008, diplomová práce, FIT VUT v Brně.

Prevod binárneho kódu x86 do vyššieho programovacieho jazyka

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Doc. Dr. Ing. Dušana Kolářa. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Marián Jurík
15. 05. 2008

Poďakovanie

Touto cestou by som sa rád poďakoval vedúcemu diplomovej práce, pánovi Doc. Dr. Ing. Dušanovi Kolářovi za konzultácie a rady pri písaní diplomovej práce a za jeho pripomienky a rady pri realizácii prekladača binárneho kódu do navrhnutého jazyka. Veľké poďakovanie tiež patrí Ing. Karlovi Oblúkovi, Ph. D., ktorý mi umožnil vypracovať túto prácu.

Obsah

1 Úvod

Cieľom diplomovej práce je navrhnúť a implementovať program na preklad binárneho kódu x86 do vyššieho programovacieho jazyka. Súčasťou práce je tiež návrh vyššieho programovacieho jazyka.

Diplomová práca sa zameriava na spustiteľné súbory pre operačný systém Microsoft Windows. V druhej kapitole je podrobne popísaná štruktúra súborového formátu PE, ktorý je použitý na ukladanie informácií o spustiteľných súboroch práve v tomto operačnom systéme. Z hlavičky súboru je možné zistiť veľa informácií. Medzi základné informácie patrí typ aplikácie – spustiteľný program alebo programová knižnica, ktorú je možné používať z iných programov. Pre ďalšiu analýzu sú najdôležitejšie informácie o mieste ukladania binárneho kódu a jeho dátových zložiek. Z hlavičky nie je možné zistiť použitý programovací jazyk a prekladač, s ktorým bol program vytvorený. Tieto informácie by boli veľmi nápomocné v ďalšej analýze.

Po analýze hlavičky nasleduje detekcia samotných inštrukcií. Binárny kód inštrukcií je uložený sekvenčne v danej sekcii. Začiatok a koniec sekcie je určený v hlavičke súboru. Tretia kapitola stručne popisuje inštrukčnú sadu IA-32. Inštrukčná sada je množina všetkých inštrukcií, ktoré je možné vykonávať na daných procesoroch. Pod pojmom inštrukcia si je možné predstaviť jednoduchý príkaz, ktorý dokáže procesor vykonať. Tretia kapitola sa nezaoberá podrobným popisom jednotlivých inštrukcií. Hlavný dôraz je kladený na spôsoby dekódovania inštrukcií z binárneho kódu. V závere kapitoly sú uvedené konkrétne príklady dekódovania inštrukcií z binárneho kódu.

Štvrtú kapitolu sme venovali podrobnejšej analýze jazyka symbolických inštrukcií. Zamerali sme sa na typické konštrukcie: spôsob volania podprogramov, stav zásobníka pred a po volaní podprogramu, prístup k lokálnym aj globálnym premenným, prístup k argumentom podprogramu a vyhodnocovanie výrazov. Tiež sme sa zamerali na riadiace konštrukcie if, if – else, while, do – while a for.

V piatej kapitole sme navrhli programovací jazyk so základnými konštrukciami a jazykom symbolických inštrukcií. Ako základ jazyka sme zvolili existujúce programovacie jazyky C a C++. Výsledná aplikácia bude prekladať jazyk symbolických inštrukcií do tohto navrhnutého jazyka.

Ďalšia kapitola je venovaná realizácii samotnej aplikácii na preklad z binárneho kódu do navrhnutého programovacieho jazyka. Aplikácia by mala analyzovať vstupný súbor a detekovať sekcie použité pre binárny kód. Potom nasleduje samotná detekcia inštrukcií. Po detekcii inštrukcií nasleduje analýza inštrukcií a preklad do navrhnutého jazyka.

V závere práce sú zhodnotené dosiahnuté výsledky a určené nedostatky práce. Tiež je zhodnotený prínos práce a navrhnuté možnosti rozšírenia a úprav.

2 Súborový formát PE

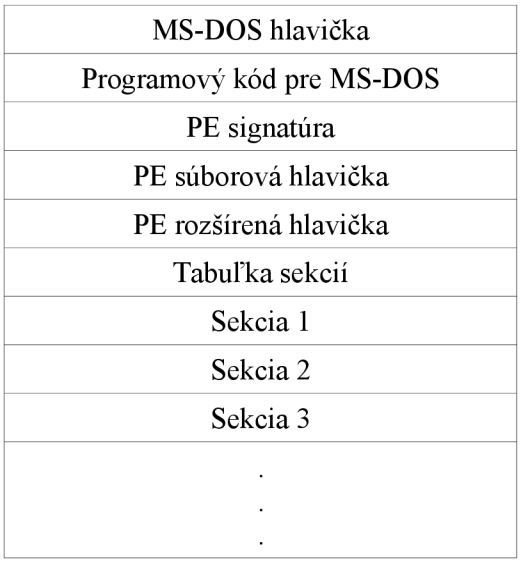
Jedným z cieľov diplomovej práce je prevod binárneho kódu do jazyka symbolických inštrukcií. Práca sa zameriava iba na spustiteľné súbory pre operačný systém Microsoft Windows 95 a vyšší a iba na 32 - bitové verzie. V tejto kapitole podrobne popíšem štruktúru súborového formátu PE[1], ktorý je použitý na ukladanie informácií o spustiteľných súborov práve v tomto operačnom systéme. Z hlavičky súboru je možné zistiť veľa informácií. Medzi základné patrí typ aplikácie – konzolová aplikácia, aplikácia s grafickým užívateľským rozhraním alebo aplikácia bez žiadneho rozhrania. Tiež je možné zistiť, či daný súbor je priamo spustiteľný program alebo iba programová knižnica, ktorú je možné používať z iných programov. Najdôležitejšie informácie sú informácie o spôsobe a mieste ukladania binárneho kódu a jeho dátových zložiek. V hlavičke sa tiež nachádzajú potrebné informácie na určenie správnych adries pri skokoch a pri volaní jednotlivých podprogramov. Po analýze hlavičky nasleduje detekcia jednotlivých inštrukcií.

Aby sme mohli analyzovať spustiteľné súbory, je potrebné sa dôkladne oboznámiť s ich štruktúrou a spôsobom ukladania binárneho kódu do súboru. Špecifikácia PE súborového formátu navrhla firma Microsoft a je súčasťou Win32 špecifikácie. Vychádza z Unixového formátu COFF (Common Object File Format). Implementovaný je na 32-bitových a 64-bitových verziách operačného systému Windows. Špecifikácia formátu bola štandardizovaná v roku 1993. Hlavným cieľom tohto formátu bolo vytvoriť univerzálny súborový formát na všetkých Win32 platformách. V súčasnosti PE formát podporujú MS Windows NT a odvodené operačné systémy, Windows 95/98 a Windows CE (operačný systém pre mobilné zariadenia). Súborový formát PE sa používa pre spustiteľné programy, knižnice poskytujúce určité funkcie a premenné, ale môže to byť napríklad ovládač zariadení. Formát PE sa tiež používa pri ActiveX komponentoch a OLE objektoch.

2.1 Štruktúra PE súborového formátu

Hlavička PE sa skladá z niekoľkých častí. Tieto časti určujú základné informácie o súbore, spôsobe načítania do pamäte, o spôsobe adresovania, o dostupných sekciách a ďalšie. Celkový prehľad štruktúry PE súborového formátu je na obrázku číslo 1.

1 PE – skratka je vytvorená z anglických slov Portable Executable



Obrázok 1: Štruktúra súborového formátu PE

2.2 MS-DOS hlavička

Na začiatku každého súboru je malá MS-DOS hlavička. V PE formáte sa používa iba na zachovanie kompatibility s MS-DOSom. Hlavička obsahuje niekoľko položiek, ktoré sú dôležité iba pri spustení programu v MS-DOS prostredí. Pre potreby práce sú dôležité iba dve hodnoty. Prvou hodnotou jednoznačný identifikátor, ktorý pozostáva zo signatúry 0x4D5A. V reprezentácii ASCII sú to písmená MZ, ktoré tvoria iniciály Marka Zbikowskeho, jedného z hlavných vývojárov architektúry MS-DOS. Preto sa občas MS-DOS hlavičke hovorí tiež MZ hlavička. Druhá dôležitá hodnota určuje pozíciu v súbore, kde začína PE hlavička. Ostatné položky sú dôležité, pokiaľ by sme analyzovali MS - DOS program. V tomto prípade ich môžeme ignorovať.

Hneď za MS-DOS hlavičkou nasleduje programový kód pre MS-DOS, zvaný tiež „MS-DOS Stub“. Pokiaľ by bol program spustený pod operačným systémom MS-DOS, vykoná sa tento kód. Pravdepodobne tam bude krátky kód, ktorý vypíše informáciu, že program nie je možné spustiť bez Windows. Vo všeobecnosti tam môže byť ľubovoľný programový kód, napríklad verzia programu pre MS-DOS.

2.3 PE hlavička

PE hlavička sa nachádza na adrese určenej jednou z položiek v MS-DOS hlavičke. Táto štvorbajtová položka obsahuje pozíciu v súbore. Prvá položka v PE hlavičke je štvorbajtové číslo 0x00004550.

Podľa tohto čísla je možné skontrolovať, či ide o PE formát. Ďalej nasleduje súborová hlavička (file header), ktorá určí základné informácie o súbore. Napríklad architektúru, pre ktorú je daný súbor určený, počet sekcií, ktoré sa tu nachádzajú. V každom súbore môže byť niekoľko sekcií. Jednotlivé sekcie sú sekvenčne uložené v súbore, preto je potrebné ukladať celkový počet sekcií. Štruktúra súborovej hlavičky je zobrazená v tabuľke číslo 1.

Veľkosť v bajtoch	Názov	Popis
2	Machine	Typ architektúry počítača, na ktorom môže byť súbor spustený. Napríklad i386, AMD64 architektúra.
2	NumberOfSections	Celkový počet sekcií nachádzajúcich sa v súbore. Jednotlivé sekcie sa nachádzajú za hlavičkami.
4	TimeDateStamp	32-bitová hodnota, ktorá určuje dátum a čas vytvorenia programu. Táto hodnota je bežne ignorovaná.
4	PointerToSymbolTable	Určuje adresu tabuľky symbolov. Pokiaľ je položka nastavená na 0, tabuľka symbolov nie je použitá.
4	NumberOfSymbols	Hodnota určuje počet symbolov v tabuľke symbolov.
2	SizeOfOptionalHeader	Hodnota určuje veľkosť rozšírenej hlavičky v bajtoch.
2	Characteristics	Hodnota určuje typ binárneho súboru.

Tabuľka 1: Popis jednotlivých položiek PE hlavičky (file header)

Za súborovou hlavičkou PE nasleduje rozšírená hlavička. V anglickej literatúre sa táto hlavička často označuje ako „optional header“. Táto štruktúra obsahuje viac ako 30 položiek a celková veľkosť je 224 bajtov. Rozšírená hlavička je rozdelená do dvoch častí. Prvá časť obsahuje podrobné informácie o veľkosti kódových a dátových sekcií, veľkosti zásobníka, verzii operačného systému. Taktiež určuje zarovnanie jednotlivých sekcií v pamäti po načítaní programu. V tabuľke číslo 2 sú popísanie najdôležitejšie položky.

Druhú časť tvorí pole štruktúr nazvaných dátové adresáre. V anglickej literatúre sa označuje ako „data directories“. Ide o pole 16 štruktúr, ktoré sú alokované na konci rozšírenej hlavičky. Niektoré z nich sa v súčasnosti nepoužívajú. Dátový adresár je definovaný štruktúrou, ktorá obsahuje dve štvorbajtové položky. Prvá položka určuje relatívnu virtuálnu adresu a druhá udáva veľkosť údajov. Pozícia v poli určuje typ dátového adresára. Najdôležitejšie sú dve položky, ktoré určujú exportované a importované funkcie z iných modulov.

Veľkosť v bajtoch	Názov	Popis
2	Magic	Hodnota určuje stav binárneho súboru.
4	SizeOfCode	Hodnota určuje veľkosť kódovej sekcie alebo sumu veľkostí všetkých kódových sekcií (sekcia .text).
4	SizeOfInitializedData	Určuje veľkosť inicializovanej dátovej sekcie alebo sumu veľkostí všetkých inicializovaných dátových sekcií (sekcia .rdata).
4	AddressOfEntryPoint	Hodnota určuje adresu vstupnej funkcie v súbore. Pre spustiteľné súbory je to štartovacia adresa. Pre ovládače zariadení je to adresa inicializačnej funkcie. Pre knižnice DLL táto hodnota nie je povinná a musí byť nastavená na 0. Pre EXE môže byť tiež nastavená na 0, potom vstupná adresa bude začiatok kódovej sekcie.
4	BaseOfCode	Adresa, ktorá určuje relatívny posun začiatku kódovej sekcie pri načítaní do pamäte.
4	BaseOfData	Adresa, ktorá určuje relatívny posun začiatku sekcie pre dáta pri načítaní do pamäte.
4	ImageBase	Uprednostnená adresa prvého bajtu programu pri načítaní do pamäte. Musí byť násobkom 65 536 bajtov.
4	SectionAlignment	Zarovnanie sekcií v pamäti v bajtoch. Musí byť väčšia alebo rovnaká ako hodnota FileAlignment. Preddefinovaná hodnota je nastavený na veľkosť stránky v systéme.
4	FileAlignment	Zarovnanie sekcií v súbore v bajtoch. Hodnota musí byť mocninou 2.
4	SizeOfImage	Položka určuje veľkosť programu načítaného v pamäti. Hodnota je v bajtoch spolu so všetkými hlavičkami.
4	SizeOfHeaders	Suma veľkostí jednotlivých hlavičiek zarovnaný na hodnotu FileAlignment.
2	Subsystem	Určuje typ subsystému, ktorý je potrebný na beh programu.
2	DllCharacteristics	Hodnota určuje podrobnosti pre DLL súbory.
4	SizeOfStackReserve	Veľkosť zásobníka. Význam len v EXE.
4	SizeOfHeapReserve	Veľkosť pamäte pre dynamickú alokáciu. Význam len v EXE.
4	NumberOfRvaAndSizes	Počet dátových adresárových položiek, ktoré nasledujú za rozšírenou hlavičkou. Určujú pozíciu a veľkosť.

Tabuľka 2: Popis vybraných položiek rozšírenej PE hlavičky (optional header)

2.4 Importovanie funkcií

V programe je možné používať funkcie, ktoré sú implementované v inom module. Pod pojmom modul si je možné predstaviť binárny súbor, ktorý poskytuje určité funkcie a premenné. Tieto funkcie je možné využívať v iných programov. Týmito modulmi najčastejšie bývajú DLL knižnice, ale môže to byť napríklad ovládač, ktorý poskytuje prístup k nejakému zariadeniu.

Aby bolo možné používať funkcie iných modulov, program sa o nich musí určitým spôsobom dozvedieť. Existujú dva spôsoby práce s externými modulmi. Prvý spôsob je automatické načítanie potrebných modulov pri štarte programu. V anglickej literatúre sa používa pojem „static linking“. Túto činnosť automaticky vykonáva systém pred začiatkom štartu programu. Program je úspešne zavedený len v prípade, že sú nájdené všetky importované funkcie zo všetkých modulov. Z toho vyplýva, že systém musí poznať názvy modulov spolu s exportovanými funkciami skôr ako sa program spustí. Tieto informácie sú uložené v PE hlavičke a príslušných sekciách. Tento spôsob načítania externých modulov sa často používa na načítanie knižníc, ktoré sú nevyhnutné na beh programu. Nevýhodou je, že pokiaľ sa nepodarilo nájsť potrebnú knižnicu, program na nespustí. V určitých prípadoch je potrebné spustiť program aj keď daná knižnica nie je dostupná. Napríklad chýbajúca knižnica poskytovala určitú činnosť, ktorá nie je kritická pre beh programu. Bolo by vhodné, aby program poskytoval dôležité funkcie, ktoré sú dostupné. Ďalšou nevýhodou je, že všetky externé moduly sa načítajú do pamäte ešte pred spustením programu aj v tom prípade, ak sa nebude využívať žiadna z jeho poskytovaných funkcií. Niektoré moderné prekladače umožňujú automatické načítanie modulu až pri jeho skutočnom použití. Teda až pred prvým volaním exportovanej funkcie. Tento spôsob je známy pod pojmom „delay loading“. Výhodou používania externých modulov je pamäťová náročnosť. Pokiaľ viac programov používa ten istý modul, tento modul sa nachádza v pamäti len jedenkrát.

Druhý spôsob načítania externých modulov je ponechaný na programátora aplikácie. Známy je pod pojmom „dynamic linking“. Programátor si musí explicitne načítať modul do pamäte a vyhľadať požadované funkcie. Win 32 API definuje funkcie na načítanie modulu na základe jeho mena. Názov funkcie je LoadLibrary. Po úspešnom načítaní modulu je potrebné vyhľadať adresy potrebných funkcií. K tomuto účelu sa používa funkcia GetProcAddress, ktorej jeden z parametrov je názov požadovanej funkcie. Po ukončení práce s modulom je potrebné uvoľniť modul z pamäte. K tomuto účelu sa používa funkcia FreeLibrary. Výhodou je, že sa program vždy spustí, aj keď externý modul nie je možné načítať. Programátor môže vhodne zareagovať na tento stav. Ďalšou výhodou je pamäťová náročnosť. Modul je načítaný v pamäti iba vtedy, keď sa používa. Pokiaľ sa modul nepoužíva, je ho možné uvoľniť. Nevýhodou je, že programátor musí explicitne načítavať modul a potrebné funkcie. V tomto prípade sa neukladajú žiadne informácie o externých moduloch do hlavičky PE.

V prvom prípade je externý modul načítaný automaticky pri štarte programu. Názov modul a volané funkcie sú uložené v PE hlavičke. Základné informácie o importovanej knižnici sa nachádzajú v štruktúre IMAGE_IMPORT_DESCRIPTOR. Každá knižnica má vlastnú štruktúru. Umiestnenie tohto pola štruktúr je možné získať z rozšírenej hlavičky. Posledná položka v rozšírenej hlavičke je pole 16 štruktúr dátových adresárov. Druhá položka v poli obsahuje relatívnu virtuálnu adresu začiatku tabuľky importovaných modulov. Pole je ukončené jednou štruktúrou vyplnenou nulami.

Pre potreby diplomovej práce je postačujúca znalosť o spôsobe importovaní funkcií z externých modulov. Nie je potrebné sa bližšie venovať spôsobu exportovania funkcií v module.

2.5 Sekcie

Sekcie sa nachádzajú bezprostredne po rozšírenej hlavičke. Počet sekcií je uložený v súborovej hlavičke PE v položke NumberOfSections. Každá sekcia obsahuje informácie o type sekcie, o umiestnení a dĺžke sekcie v súbore a po spustení programu o ich umiestnení v pamäti. Každá sekcia má hlavičku a telo. Jednotlivé položky hlavičky sekcie sú popísané v tabuľke číslo 3.

Veľkosť v bajtoch	Názov položky	Popis
8	Name	Pole obsahuje názov sekcie zakončený nulou, prípadne bez zakončenia, ak má dĺžku presne 8 znakov.
4	VirtualSize	Hodnota určuje veľkosť sekcie po načítaní programu do pamäte.
4	VirtualAddress	Adresa prvého bajtu v sekcii po načítaní programu do pamäte.
4	SizeOfRawData	Hodnota určuje veľkosť sekcie na disku v bajtoch.
4	PointerToRawData	Položka určuje adresu v súbore, kde sa začína aktuálna sekcia.
4	PointerToRelocations	Adresa v súbore, kde sa začínajú informácie pre relokáciu sekcie. Pokiaľ nie sú dostupné, hodnota musí byť 0.
4	PointerToLineNumbers	Adresa v súbore, kde sa začínajú line-number položky pre sekciu. Pokiaľ nie sú použité, hodnota položky musí byť 0.
4	NumberOfRelocations	Počet relokácií pre príslušnú sekciu.
4	NumberOfLineNumbers	Počet line-number položiek pre príslušnú sekciu.
4	Characteristics	Hodnota určuje typ sekcie. Napríklad sekcia obsahuje spustiteľný kód, inicializované alebo neinicializované premenné.

Tabuľka 3: Popis jednotlivých položiek dostupné pre každú sekciu

Názov sekcie v PE hlavičke je len ilustračný. Nie je možné odvodiť typ sekcie podľa názvu. Typ sekcie sa určuje na základe príznakov. Nasleduje popis typických sekcií. Tieto sekcie sú typicky generované MS prekladačmi.

- **Sekcia pre spustiteľný kód** – Názov sekcie je .text. V programoch sa používa iba jedna sekcia pre spustiteľný kód. V tejto sekcii sa nachádza aj vstupný bod. V anglickej literatúre označovaný ako „entry point“. Vstupný bod určuje adresu prvej inštrukcie, predstavuje začiatok vykonávania inštrukcií.

- **Dátové sekcie** – Najčastejšie sa používajú 3 dátové sekcie. Prvá má názov .bss. Obsahuje neinicializované údaje pre aplikáciu, vrátane všetkých premenných deklarovaných ako statické premenné. Ďalšia sekcia .rdata obsahuje iba údaje pre čítanie, ako napríklad textové reťazce, konštanty a informácie pre ladenie. Posledná sekcia .data obsahuje všetky ostatné premenné okrem automatických premenných, ktoré sú umiestnené na zásobníku.

- **Sekcia prostriedkov** – Názov sekcie .rsrc. Obsahuje informácie o dostupných zdrojoch pre daný modul. Samotným zdrojom môže byť kurzor, bitmapa, ikona, menu, dialóg, typ písma a podobne.

- **Sekcia exportovaných údajov** – Názov sekcie je .edata. Pokiaľ existuje táto sekcia, obsahuje funkcie, ktoré je možné zavolať z iného programu. Najčastejšie sa používa v DLL súboroch.

- **Sekcia importovaných údajov** (import data section) – Názov sekcie je .idata. Obsahuje niekoľko štruktúr, ktoré definujú zoznam importovaných funkcií. Importovaná funkcia je taká, ktorá je modulom volaná, ale nenachádza sa v danom module. Väčšinou sú to funkcie uložené v externých knižniciach. Sekcia .idata obsahuje potrebné informácie pre ich používanie.

- **Sekcia s ladiacimi informáciami** – Názov sekcie je .debug. Táto sekcia obsahuje informácie pre ladenie programov.

3 Inštrukčná sada

Inštrukčná sada je množina všetkých inštrukcií, ktoré je možné vykonávať na daných procesoroch Pod pojmom inštrukcia si je možné predstaviť jednoduchý príkaz, ktorý dokáže procesor vykonať. Cieľom tejto kapitoly je stručne popísať inštrukčnú sadu procesorov Intel x86 a kompatibilných. Zameraná je hlavne na spôsob dekódovania inštrukcií z binárneho kódu, ktorý je podstatný pre úspešné ukončenie diplomovej práce. Cieľom tejto práce nie je podrobne popísať štruktúru procesoru a činnosti, ktoré sa v ňom uskutočňujú v súvislosti vykonávaním jednotlivých inštrukcií. Tieto informácie je možné nájsť v literatúre [3]. Taktiež predpokladáme, že čitateľ má základné informácie o štruktúre procesoru, jeho registroch, spôsobe komunikácie s pamäťou, spôsobe ukladania údajov do pamäte. Pri popisovaní formátu strojových inštrukcií je uvedené základné rozdelenie inštrukcií. V prípade záujmu o túto problematiku odporúčame literatúru [1], v ktorej je popísaný základný princíp programovania v jazyku symbolických inštrukcií. V prípade hlbšieho záujmu o problematiku odporúčame literatúru [4] a [5], v ktorých je podrobne popísaný formát inštrukcií, spôsob dekódovania, podrobný popis jednotlivých inštrukcií spolu s operačným kódom.

3.1 Inštrukčná sada IA-32

Inštrukčná sada procesorov s IA-32 architektúrou pozostáva z množiny inštrukcií rôznych dĺžok. Väčšina z nich obsahuje dva operandy. Architektúra x86 procesorov sa veľmi podobá CISC[1] architektúre. Inštrukčná sada IA-32 obsahuje množinu inštrukcií pre prácu s celými číslami. Podľa typu použitia ich je možné rozdeliť do niekoľkých skupín:

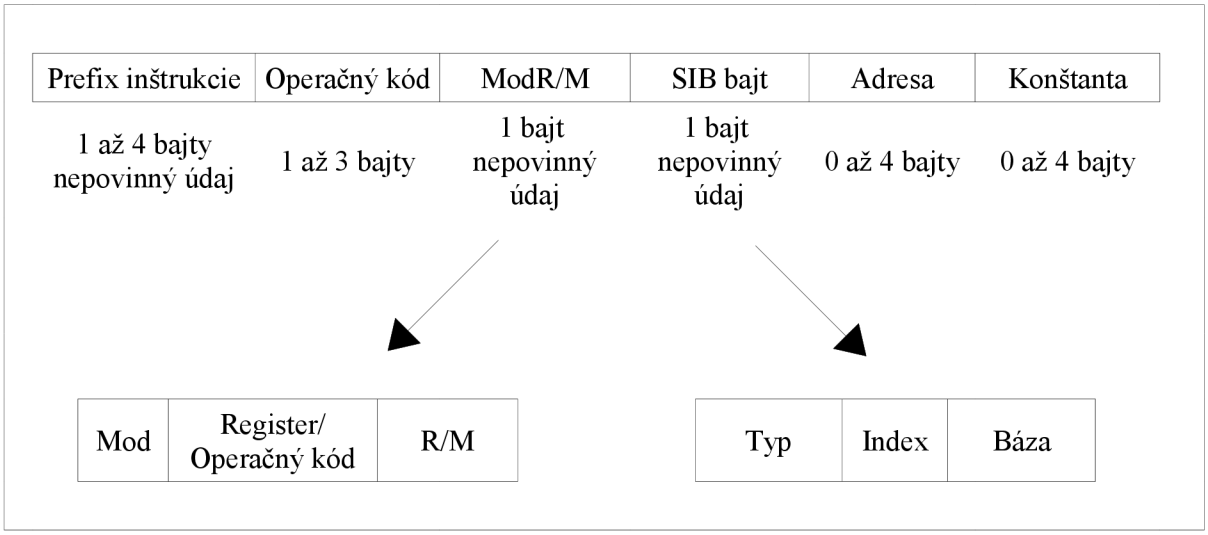
1. presun údajov – MOV, XCHG, PUSH, PUSHA, POP, POPA …
2. aritmetické inštrukcie – ADD, ADC, SUB, IMUL, MUL, IDIV, INC, DEC, NEG
3. logické inštrukcie – AND, OR, XOR, NOT
4. inštrukcie pre bitový posun – SAR, SAL, ROR, ROL, RCR, RCL
5. inštrukcie pre riadenie toku vykonávania programu – JMP, Jxx, CALL, RET, INT, LOOP
6. inštrukcie pre prácu s reťazcami – MOVS, CMPS, SCAS, LODS, STOS, REP
7. inštrukcie pre nastavovanie príznakov – STC, CLC, CMC, STD, STI, CLI
8. inštrukcie pre prácu so segmentovými registrami – LDS, LES, LFS, LGS, LSS
9. inštrukcie pre ovládanie portov – IN, OUT
10. systémové inštrukcie pre prácu s registrami CRx, DRx
11. ďalšie inštrukcie – LEA, NOP, CPUID

1 CISC – complex instructin set computer. Viac informácií o tejto problematike je možné nájsť na adrese: http://www.laynetworks.com/CISC.htm

3.2 Formát inštrukcií IA-32

Inštrukcie sú uložené v sekcii binárneho súboru PE, ktorá má zvyčajne názov .text. Tiež je určená prvá inštrukcia, ktorá sa bude vykonávať po spustení programu. Jednotlivé inštrukcie sú uložené sekvenčne v binárnej podobe. Inštrukcie môžu mať rôzne veľkosti. Medzi sebou nemajú žiadnu oddeľovaciu značku. Dĺžku inštrukcie určuje typ inštrukcie a spôsob adresovania. Preto je dôležitejšie správne určiť veľkosť inštrukcie ako jej skutočný typ spolu s operandmi. V prípade stanovenia nesprávnej veľkosti sa zle určí začiatok ďalšej inštrukcie a preklad ďalších inštrukcií sa môže porušiť.

Inštrukcia pozostáva z niekoľkých častí. Prvou časťou môžu byť prefixy, ktoré nie sú povinné. Nasleduje povinný operačný kód, ktorého dĺžka môže byť 1 až 3 bajty. Pokiaľ je potrebné, nasleduje jeden bajt, ktorý určuje spôsob adresovania pamäte a registrov. Tento bajt je označený ako ModR/M. Ďalej môže nasledovať druhý adresový bajt označený SIB[1]. Pri niektorých inštrukciách je vyžadovaná relatívna adresa alebo konštantná hodnota. Formát inštrukcií je zobrazený na obrázku číslo 2.



Obrázok 2: Formát inštrukcie

1 SIB – Skratka je vytvorená z anglických slov Scale, Index, Byte

3.2.1 Prefix inštrukcie

Prefix inštrukcie je nepovinný údaj a jeho dĺžka je jeden bajt. Prefixy inštrukcií je možné rozdeliť do 4 skupín:

1. **skupina:**
   - 0xF0 – určuje zamykanie. Na viac procesorových počítačoch umožňuje exkluzívny prístup do pamäte.
   - 0xF2, 0xF3 – používajú sa s inštrukciami pracujúcimi s textovými reťazcami (MOVS, CMPS, SCAS, LODS, INS, OUTS). Určuje opakovanie požadovanej operácie pre celý textový reťazec. Ďalej majú špeciálny význam pre inštrukcie pracujúce s MMX a XMM registrami.
2. **skupina:**
   - 0x2E – použije sa segmentový register CS
   - 0x36 – použije sa segmentový register SS
   - 0x3E – použije sa segmentový register DS
   - 0x26 – použije sa segmentový register ES
   - 0x64 – použije sa segmentový register FS
   - 0x65 – použije sa segmentový register GS
   - 0x2E – v použití s inštrukciou skoku naznačí procesoru, že s vysokou pravdepodobnosťou podmienka skoku nebude splnená. Používa sa na zvýšenie výkonu spracovania skokových inštrukcií.
   - 0x3E – v použití s inštrukciou skoku naznačí procesoru, že s vysokou pravdepodobnosťou podmienka skoku bude splnená. Používa sa na zvýšenie výkonu spracovania skokových inštrukcií. Prefixy 0x2E a 0x3E sú novinkou. Zatiaľ sú použiteľné iba pre Intel procesory.
3. **skupina:**
   - 0x66 – zmena veľkosti operandu.
4. **skupina:**
   - 0x67 – zmena veľkosti adresy.

3.2.2 Operačný kód

Veľkosť operačného kódu môže byť jeden až tri bajty. V niektorých prípadoch sú v ModR/M bajte zakódované ďalšie 3 bity operačného kódu. Operačný kód je povinný údaj, ktorý určuje typ inštrukcie a ďalšie informácie o inštrukcií. Pokiaľ inštrukcia očakáva operandy typu register alebo pamäť, nasleduje bajt, ktorý určuje spôsob adresovania registrov a pamäte – ModR/M. Niektoré bity v operačnom kóde môžu mať zvláštny význam. Napríklad určujú veľkosť operandu, veľkosť adresy,

veľkosť priamej hodnoty, určujú použitý register. V prípade skokových inštrukcií určujú typ skoku. Dekódovanie jednotlivých bitov operačného kódu silne závisí od typu inštrukcie.

3.2.2.1 Dekódovanie veľkosti operandu

V niektorých inštrukciách je zakódovaná veľkosť operandu. Preddefinovaná veľkosť operandu závisí od použitého typu operácií. Pre 16-bitový mód je veľkosť operandu 16- bitov, pre 32-bitový mód je veľkosť 32 bitov a pre 64-bitový mód je veľkosť 64 bitov. Veľkosť operandu je možné nastaviť jedným bitom. Pokiaľ je tento bit nastavený na 1, použije sa predvolená veľkosť operandu. V opačnom prípade je veľkosť operandu 8 bitov.

3.2.2.2 Dekódovanie rozšírenia znamienka

Ďalšou položkou, ktorá má jeden bit, je bit určujúci znamienko. Tento bit určuje ako sa bude prevádzať 8-bitová hodnota na 32-bitovú (prípadne 64-bitovú) hodnotu. V prípade, že bude nastavený na 0, s číslom sa bude pracovať ako s kladným. Pri konverzii sa nebude brať ohľad na najviac významový bit, ktorý určuje znamienko čísla. V prípade, že bude nastavený na 1, pri konverzii sa uvažuje o znamienku 8-bitového čísla.

3.2.2.3 Dekódovanie registra

Veľkosť tejto položky je tri bity. Je používaný priamo v operačnom kóde, ModR/M bajte a SIB bajte. Toto číslo určuje typ použitého registru. V tabuľke číslo 4 sú prehľadne zobrazené bitové hodnoty registra a výsledný typ registra.

Kód registru	**Typ registru pre 32-bitové operácie bez použitia w časti**	**Typ registru pre 32-bitové operácie s použitím veľkosti rozšíreného znamienka – časť w**	
		w = 0	**w = 1**
000	EAX	AL	EAX
001	ECX	CL	ECX
010	EDX	DL	EDX
011	EBX	BL	EBX
100	ESP	AH	ESP
101	EBP	CH	EBP
110	ESI	DH	ESI
111	EDI	BH	EDI

Tabuľka 4: Dekódovanie typu registra

3.2.2.4 Dekódovanie segmentového registra

V niektorých inštrukciách sa používajú segmentové registre. Existujú dva spôsoby zakódovania segmentového registru. Prvý spôsob používa 2-bitovú hodnotu označenú ako sreg2. Druhý spôsob je použitie 3-bitovej hodnoty označenej ako sreg3. Prehľad hodnôt sreg2 a sreg3 a príslušných typov registra je v tabuľke číslo 5.

2-bitový kód registru sreg2	Typ segmentového registru
00	ES
01	CS
10	SS
11	DS

3-bitový kód registru sreg3	Typ segmentového registru
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS
110	nepoužité
111	nepoužité

Tabuľka 5: Dekódovanie typu segmentového registra

3.2.2.5 Dekódovanie podmienky pri skoku

Pre podmienené skoky určuje spôsob spracovania podmienky. Veľkosť položky je 4 bity a je označená ako tttn. Je použitá u inštrukcií podmienených skokov (inštrukcie Jcc), u inštrukcií, ktoré nastavujú bajt na základe podmienky (inštrukcie SETcc) a u inštrukcie CMOVcc.

3.2.3 ModR/M bajt a SIB bajt

Veľa inštrukcií používajú ako operandy register alebo adresu pamäte. V tom prípade za operačným kódom nasleduje ModR/M bajt. Tento bajt určuje spôsob adresovania registrov a pamäte. Obsahuje tri položky. Bity 7 a 6 určujú použitý mód. Napríklad hodnota 11b určuje, že sa pracuje s registrami. Ďalšie bity 5 až 3 určujú použitý register alebo v niektorých prípadoch sú súčasťou operačného kódu. Posledné bity 2 až 0 určujú použitý register. V kombinácii s bitmi pre mód poskytujú 32 možností adresovania registrov a pamäte.

V niektorých prípadoch je potrebné použitie ďalšieho adresového bajtu. Tento bajt sa nazýva SIB, čo je skratka z anglických slov scale, index a base. Taktiež pozostáva z troch častí. Prvá časť bity 7 a 6 určuje stupeň alebo typ. Ďalej nasledujú 3 bity, ktoré určujú číslo indexového registru, s ktorým sa pracuje. Posledné bity 2 až 0 určujú číslo bázového registru

3.3 Príklady dekódovania

V predchádzajúcej časti je stručne popísaný všeobecný formát inštrukcií. Aj keď sa môže zdať, že formát inštrukcií je jednoduchý, opak je pravdou. Na dekódovanie jednotlivých inštrukcií je ešte potrebná tabuľka, ktorá priradí určitým operačným kódom príslušné inštrukcie. Túto tabuľku spolu s ďalšími informáciami je možné nájsť v [4] v časti B.1.5.

Najjednoduchšie sa dekódujú inštrukcie, ktoré nemajú žiadne operandy. Týchto inštrukcií je ale veľmi málo. Na nasledujúcom príklade je popísaný formát zápisu. Na prvom mieste je binárny zápis inštrukcie v súbore v hexadecimálnom formáte. Za ním nasleduje textový zápis typu inštrukcie s prípadnými operandmi. Na záver je uvedený zápis prvej časti inštrukcie v binárnej číselnej sústave, kde sú vyznačené prípadné názvy jednotlivých položiek.

```
CC          INT 3           1100 1100
90          NOP             1001 0000
60          PUSHAD          0110 0000
```

Niektoré inštrukcie pracujú iba s jedným operandom typu register alebo pamäť. Pokiaľ je parametrom register, môže existovať varianta kódovania inštrukcie do jedného bajtu. Typickým príkladom môže byť inštrukcia DEC, ktorá zníži hodnotu registru alebo hodnotu pamäte o jeden. V nasledujúcom príklade je uvedený zápis inštrukcie DEC s operandom typu register. Z príkladu je vidieť, že posledné tri bity sú určené na určenie typu registru. Typ registru je možné zistiť z tabuľky číslo 4.

```
49          DEC ecx         0100 1 reg
48          DEC eax         0100 1 reg
4E          DEC esi         0100 1 reg
```

Inštrukciu DEC s operandom typu register je možné zapísať ešte iným spôsobom. Veľkosť inštrukcie bude 2 bajty, kde prvý bajt určuje typ inštrukcie a druhý bajt určuje spôsob adresovania registru alebo pamäte – bajt ModR/M. Prvé dva bity druhého bajtu predstavujú mód adresovania. Pokiaľ je nastavený na 11b, ide o prácu s registrom. Ďalšie tri bity určujú typ registru alebo sú súčasťou operačného kódu, ktorý určuje typ inštrukcie. V tomto prípade sú súčasťou operačného kódu. Pokiaľ by tam bola iná hodnota ako 001b, išlo by o inú inštrukciu ako DEC. V príklade si je možné všimnúť posledný bit prvého bajtu označený ako w, kde je uložená informácia o veľkosti operandu. Pokiaľ je nastavený na 1, používa sa prednastavená veľkosť operandov. V tomto prípade 32 bitový register ECX. V druhej inštrukcii je nastavený na 0, takže veľkosť operandu je 8 bitov. Použije sa register CL.

```
FF C9           DEC ecx         1111 111w : 11 001 reg
FE C9           DEC cl          1111 111w : 11 001 reg
FF CA           DEC edx         1111 111w : 11 001 reg
```

Operandom inštrukcie DEC môže byť tiež adresa pamäti. Formát inštrukcie je podobný ako v predchádzajúcom príklade. V druhom bajte sa zakóduje informácia o adresovaní pamäte namiesto registra. Nasledujúci príklad má mód nastavený na 01b. V tomto prípade sa k hodnote registru pripočíta 8-bitové číslo. Výsledná hodnota určuje adresu v pamäti.

```
FF 49 08   DEC dword ptr [ecx + 8]    1111 111w : mod 001 r/m
```

Ďalším príkladom môže byť prístup na adresu zadanú priamo v inštrukcií. V príklade sú prvé dva bity druhého bajtu nastavené 00b. Posledné tri bity určujú register alebo prácu s konštantnými hodnotami. V tomto prípade sú nastavené na 101b. Za druhým bajtom nasleduje 32-bitová hodnota, ktorá určuje adresu pamäte.

```
FF 0D DC 67 41 00     DEC dword ptr [4167DC]
                                    1111 111w : mod 001 r/m
```

Inštrukcia DEC očakáva iba operand, ktorý môže byť register alebo adresa pamäte. Väčšina inštrukcií však pracuje s dvoma operandmi typu register – register, register – pamäť, pamäť – register, hodnota – register, hodnota – pamäť, register – hodnota a pamäť – hodnota. Sú umožnené všetky možné kombinácie okrem kombinácie pamäť – pamäť. Príklady inštrukcií s dvoma operandmi sú uvedené na príklade inštrukcie MOV. Táto inštrukcia kopíruje obsah druhého operandu do obsahuj prvého operandu. Najjednoduchší zápis inštrukcie je pri kopírovaní obsahu registra do iného registra. Bit w určuje veľkosť operandu. Ak je nastavený na 1, veľkosť je 32 bitov, inak 8 bitov.

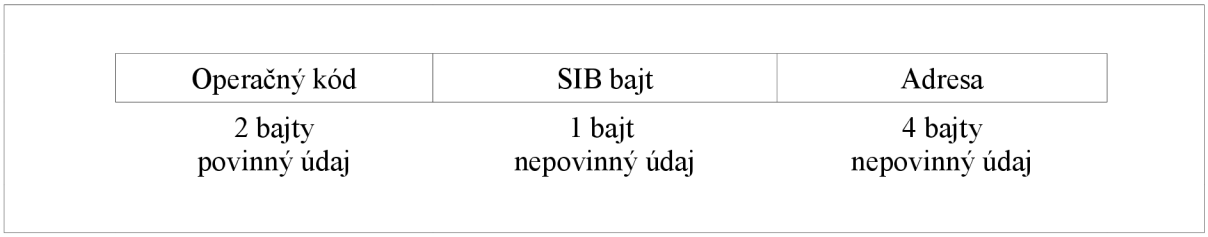
```
8B C5           MOV eax, ebp        1000 101w : 11 reg1 reg2
8A D9           MOV bl, cl          1000 101w : 11 reg1 reg2
```

V tejto časti sme sa prakticky pokúsili priblížiť spôsob dekódovania inštrukcií z binárneho súboru. Medzi príkladmi sa nachádzajú typické formáty inštrukcií spolu s popisom ich dekódovania. Z príkladov je vidieť, že niektoré inštrukcie majú viac spôsobov zápisu do binárneho súboru. Príkladom je inštrukcia DEC s operandom typu register.

3.4 Inštrukčná sada x87

Inštrukčná sada x87 obsahuje inštrukcie pre prácu s reálnymi číslami. Tieto inštrukcie používajú 8 špeciálnych registrov. Označenie registrov je st(0), st(1) až st(7). Veľkosť jedného registra je 80 bitov. Najväčší rozdiel medzi registrami pre prácu s reálnymi číslami a registrami zo sady IA-32 je, že registre st(i) sú organizované do zásobníka. Register st(0) je na vrchole zásobníka, register st(1) nasleduje za ním a podobne pre ďalšie registre. Veľa inštrukcií ukladá alebo odoberá hodnotu zo zásobníka. Pri týchto operáciách sa menia hodnoty vo všetkých registroch. Viac informácií o inštrukčnej sade x87 je možné nájsť v literatúre [3], [4] a [5].

Formát inštrukcií x87 je oproti formátu inštrukcií IA-32 jednoduchší. Formát je zobrazený na obrázku číslo 3. Operačný kód je veľký 2 bajty. Prvých 5 bitov je 11011. Súčasťou operačného kódu môže byť aj určenie použitého registra a spôsob adresovania.



Obrázok 3: Formát inštrukcie x87

Dekódovanie inštrukcií x87 je jednoduchšie ako inštrukcií z inštrukčnej sady IA-32. Veľkosť inštrukcií je minimálne 2 bajty. Väčšina inštrukcií obsahuje jeden alebo dva parametre typu register. Typ registru je súčasťou operačného kódu. Nasledujúci príklad ukazuje inštrukciu FADD, ktorá spočíta hodnotu registru st(0) so zadaným registrom. Výsledok operácie je uložený na vrchole zásobníka.

```
D8 C0+i     FADD st(0), st(i)          1101 1000    1100 0reg
```

Súčasťou druhého bajtu operačného kódu môžu byť položky mod a r/m. Tieto položky majú rovnakú interpretáciu ako pri inštrukčnej sade IA-32. Nasledujúci príklad ukazuje zložitejší formát inštrukcie FADD. Inštrukcia spočíta register st(0) so 64-bitovým reálnym číslom, ktoré je uložené na zadanej adrese.

```
DC 05 50 57 41 00      FADD st(0), qword ptr[415750]
                                   1101 1100   mod 000 r/m
```

4 Typické konštrukcie pri preklade

Táto kapitola je zameraná na spôsob prekladu vyššieho programovacieho jazyka do jazyka symbolických inštrukcií. V príkladoch je použitý programovací jazyk C. Programový kód v jazyku symbolických inštrukcií je možné rozdeliť na samostatné časti – funkcie. Preto sme hlavnú časť kapitoly venovali analýze volania funkcií. Zamerali sme sa na spôsob volania funkcie, predávanie parametrov a určenie lokálnych premenných. Lokálne premenné sú ukladané na zásobník. V ďalšej časti je popísaný spôsob prístupu k lokálnym a globálnym premenným. Medzi jednotlivými premennými je možné vykonávať určité operácie a vytvárať výrazy. V závere kapitoly sú popísané podmienené príkazy a cykly.

4.1 Volanie funkcie

V jazyku symbolických inštrukcií je volanie funkcie implementované pomocou inštrukcie CALL. Inštrukcia môže byť použitá na dva typy volania. Prvým je blízke volanie. V prípade blízkeho volania sa zavolá funkcia v aktuálnom kódovom segmente. Ten je určený registrom CS. Druhým typom je vzdialené volanie. V tomto prípade sa zavolá funkcia, ktorá sa nachádza v inom kódovom segmente.

Pri blízkom volaní inštrukcia CALL ukladá na zásobník hodnotu registra EIP, ktorá je dôležitá pre návrat z funkcie. Potom načíta adresu prvej inštrukcie do registra EIP, ktorá je získaná z operandu inštrukcie. Operandom môže byť konštanta, register alebo adresa pamäte. Ďalej nasleduje vykonávanie inštrukcií volanej funkcie. Funkcii je možné predávať parametre. Existujú dva spôsoby. Prvý spôsob je predávanie parametrov cez registre procesora. Pri volaní funkcii sa ponecháva pôvodná hodnota registrov. Druhou možnosťou je predávanie parametrov cez pamäť zásobníka. Týmto spôsobom je možné predať veľké množstvo parametrov. Vrchol zásobníka je určený hodnotou v registru ESP. Cez tento register sa pristupuje k lokálnym premenným. Pri vložení alebo odobraní premennej zo zásobníka sa zmení jeho hodnota. Preto sa pre jednoduchosť prístupu najčastejšie vrchol zásobníka uloží do registra EBP, ktorý sa vo funkcii nemení. Adresovanie argumentov a lokálnych premenných je potom jednoduchšie.

Volaná funkcia môže explicitne uložiť hodnoty registrov na zásobník a potom ich pred návratom obnoviť do pôvodného stavu. Tieto hodnoty môžu byť uložené na zásobník. Na uloženie hodnoty registru na vrchol zásobníka je možné použiť inštrukciu PUSH. Inštrukcia POP dokáže obnoviť hodnotu z vrcholu zásobníka. Na uloženie a obnovenie hodnôt všetkých registrov je možné použiť inštrukcie PUSHA a POPA.

Nasledujúci príklad ukazuje volanie funkcie s dvoma parametrami typu int. Volanie funkcie v jazyku C môže vyzerať nasledovne:

```
/* deklarácia funkcie       */
int volanaFunkcia(int a, int b);

/* lokálna premenná         */
int hodnota1 = 5;

/* miesto volania funkcie   */
int retval = volanaFunkcia(hodnota1, 6);
```

Príslušné volanie funkcie v jazyku symbolických inštrukcií môže vyzerať nasledovne:

```
00411B1D      push        6
00411B1F      mov         eax, dword ptr [hodnota1]
00411B22      push        eax
00411B23      call        volanaFunkcia (411523h)
00411B28      add         esp, 08h
00411B2B      mov         dword ptr [retval], eax
```

Z príkladu je vidieť, že parametre sa predávajú prostredníctvom zásobníka. Na zásobník sa najprv uloží hodnota druhého parametru – číslo 6. Potom sa uloží hodnota premennej hodnota1. Na zásobník sa premenné ukladajú v opačnom poradí ako sú zapísané v jazyku C. Pri analýze volania funkcie je potrebné zohľadniť premenné, ktoré sa ukladajú na zásobník. Premenná môže byť uložená na zásobník na ľubovoľnom mieste pred volaním funkcie. Aj preto nie je možné určiť počet parametrov funkcie pri volaní funkcie. Taktiež nie je možné určiť, ktoré registre sú použité ako parametre funkcie. Túto skutočnosť je potrebné vziať do úvahy pri analýze parametrov pre volané funkcie.

Na začiatku volanej funkcie sa typicky vykonávajú určité činnosti. Na zásobník sa ukladá hodnota registrov, ktoré sa budú používať. Na konci funkcie sa ich hodnota obnoví. Ďalej sa nastavuje hodnota registra ESP, ktorá určuje vrchol zásobníka. Od nej sa odpočíta veľkosť pamäte potrebných pre lokálne premenné (posunie sa vrchol zásobníka smerom hore). Niektoré prekladače pri určitých nastaveniach inicializujú lokálne premenné na určitú hodnotu. Napríklad prekladač Microsoft Visual Studio 2005 pri určitých nastaveniach inicializuje lokálne premenné na hodnotu 0xCC. Podľa zmien na začiatku je možné určiť adresu a veľkosť pamäte pre lokálne premenné. Potom je jednoduché sledovať prístupy do tejto časti a vytvárať zoznam lokálnych premenných.

Pri volaní funkcie je možné predávať argumenty prostredníctvom zásobníka a registrov. Argumenty sa ukladajú na vrchol zásobníka. Pokiaľ sa pristupuje k nejakej hodnote na zásobníku, je to argument. Je vysoký predpoklad, že kód vo funkcii bude pristupovať iba k svojím argumentom. V skutočnosti môže pristupovať k obsahu celého zásobníka. Toto nastáva napríklad pri chybe v kóde, ktorá môže prepísať určité hodnoty. Napríklad sa prepíše adresa miesta volania funkcie potrebnej na návrat z funkcie pri inštrukcii RET.

Nasledujúci príklad zobrazuje začiatok volanej funkcie v jazyku symbolických inštrukcií. Prvé 3 inštrukcie nastavia nový vrchol zásobníka. Posunutím vrcholu zásobníka sa vytvorí miesto pre lokálne premenné. Nasleduje uloženie hodnôt niektorých registrov. Posledné 4 inštrukcie nastavia hodnotu lokálnych premenných na hodnotu 0xCCCCCCCC.

```
00413110    push        ebp
00413111    mov         ebp, esp
00413113    sub         esp, 0E4h
00413119    push        ebx
0041311A    push        esi
0041311B    push        edi
0041311C    lea         edi, [ebp-0E4h]
00413122    mov         ecx, 39h
00413127    mov         eax, 0CCCCCCCCh
0041312C    rep stos    dword ptr [edi]
```

Zmena vrcholu zásobníka. Pre lokálne premenné je určené 0xE4 bajtov.

Uloženie hodnoty registrov na zásobník

Nastavenie hodnoty lokálnych premenných na 0xCCCCCCCC.

Na konci funkcie sa najčastejšie nastavuje návratová hodnota. Návratová hodnota sa môže ukladať do niektorého z registrov alebo sa zmení niektorá hodnota na zásobníku. V príklade sa návratová hodnota ukladá do registru EAX. Potom nasleduje obnovenie pôvodných hodnôt registra zo zásobníka (v prípade ak boli na začiatku funkcie uložené na zásobník). Nastavuje sa pôvodná hodnota vrcholu zásobníka – register ESP. A na záver nasleduje inštrukcia RET, ktorá predá riadenie programu hneď za miesto volania funkcie. Táto adresa bola uložená v zásobníku pri vykonávaní inštrukcie CALL.

```
004131EB    mov         eax, 1
004131EE    pop         edi
004131EF    pop         esi
004131F0    pop         ebx
004131F1    mov         esp, ebp
004131F3    pop         ebp
004131F4    ret
```

Nastavenie návratovej hodnoty do registra EAX.

Obnovenie niektorých hodnôt registrov.

Nastavenie zásobníka do pôvodnej podoby ako na začiatku funkcie.

Návrat z funkcie.

4.2 Vyhodnocovanie výrazov

Hodnotu premenných je možné ukladať na viacerých miestach. Spôsob detekcie lokálnych premenných je popísaný v časti 4.1. K lokálnym premenným je možné pristupovať iba v rámci jednej funkcie a sú uložené v pamäti zásobníka. Pristupuje sa k nim cez adresu vrcholu zásobníka. Druhou možnosťou je ukladať hodnotu premennej v registri. Ďalšou možnosťou je používanie globálnych premenných. Globálne premenné je možné používať na ľubovoľnom mieste v programe. Informácia o adrese pamäte pre globálne premenné sa nachádza v hlavičke PE. Ku globálnym premenným sa pristupuje prostredníctvom jednoznačnej adresy.

Matematické operácie sa vykonávajú pomocou aritmetických inštrukcií (ADD, SUB, DIV, MUL...), logických inštrukcií (AND, OR, XOR...) a inštrukcií pre bitový posun (RCL, RCR, ROL, ROR). Pre prácu s reálnymi číslami sa používajú inštrukcie z inštrukčnej sady x87 (FADD, FSUB, FMUL, FDIV...).

Tieto inštrukcie menia hodnotu registrov alebo pamäte. Skoro všetky inštrukcie majú dva parametre. Parametre určujú operandy matematickej operácie. Výsledok operácie je uložený na adresu prvého parametru. Napríklad inštrukcia ADD EAX, EBX spočíta hodnotu registrov EAX a EBX a výsledok uloží do prvého parametru.

Jazyk symbolických inštrukcií umožňuje vytvárať iba jednoduché výrazy. Pre výpočet zložitejších výrazov je potrebné použiť niekoľko inštrukcií. Nasledujúci príklad demonštruje zápis zložitejšieho výrazu v jazyku C a zápis v jazyku symbolických inštrukcií.

```
int vysledok, a, b, c;
...
/* zápis výrazu v jazyku C */
vysledok = (a + b - c * b) / 5;
```

Príklad zápisu výrazu v jazyku symbolických inštrukcií.

```
mov         eax, dword ptr [a]          ; eax = a
add         eax, dword ptr [b]          ; eax = eax + b
mov         ecx, dword ptr [c]          ; ecx = c
imul        ecx, dword ptr [b]          ; edx:eax = eax * b
sub         eax, ecx                    ; eax = eax - ecx
cdq                                     ; edx:eax = eax
mov         ecx, 0x5                    ; ecx = 0x5
idiv        ecx                         ; eax = edx:eax / ecx
mov         dword ptr [vysledok], eax   ; vysledok = eax
```

Nasledujúci príklad ukazuje prácu s reálnymi číslami.

```
float vysledok, a, b, c;
...
/* zápis výrazu v jazyku C */
vysledok = (a + b - c * b) / 5.0;

fld         dword ptr [a]           ; st(0) = a
fadd        dword ptr [b]           ; st(0) = st(0) + b
fld         dword ptr [c]           ; st(1) = st(0), st(0) = c
fmul        dword ptr [b]           ; st(0) = st(0) * b
fsubp       st(1), st(0)            ; st(0) = st(1) - st(0)
fdiv        5.0                     ; st(0) = st(0) / 5.0
fstp        dword ptr [vysledok]    ; vysledok = st(0)
```

4.3 Podmienené vetvenie

Programovací jazyk C poskytuje niekoľko typov príkazov pre podmienené vetvenie. Dostupné sú if, if – else a switch. Vetvenie programu sa vykonáva pomocou inštrukcií Jcc a JMP.

Nasleduje príklad zápisu príkazu if – else v jazyku C a príslušný zápis v jazyku symbolických inštrukcií.

```
if (value > 1) {                cmp     dword ptr [value], 1
                                jle     Label_1

    // príkazy 1                ; príkazy 1
                                jmp     Label_2
} else {                    Label_1:
    // príkazy 2                 ; príkazy 2
)
                            Label_2:
```

4.4 Cykly

Jazyk C poskytuje 3 druhy cyklov: while, do – while a for. Cyklus while vyhodnocuje podmienku na začiatku. Pokiaľ je podmienka splnená, vykoná sa telo príkazu. Cyklus do – while vyhodnocuje podmienku až na konci. Tento cyklus sa vykoná minimálne raz. Oproti cyklu while, ktorý sa nemusí vykonať ani raz. V cykloch je možné používať príkazy break a continue. Príkaz break ukončí vykonávanie cyklu a tok programu sa presunie na príkaz za cyklom. Príkaz continue preruší vykonávanie cyklu a spôsobí prechod na ďalší krok cyklu bez toho, aby sa predchádzajúci krok cyklu ukončil. Nespôsobuje skok z tela cyklu von. Nasleduje príklad cyklu while spolu s príkazom break.

```
int i, value;
...

                                Label_1:
while (i > 0) {                       cmp dword ptr [i], 0
                                      jle Label_2

    // príkazy 1                      ; príkazy 1

    if ( value > 8) {                 cmp   dword ptr [value], 8
                                      jle   0x411451

            break;                    jmp   Label_3
    }
                                Label_2:
    // príkazy 2                      ; príkazy 2
 }                                    jmp   Label_1

                                Label_3:
```

5 Návrh programovacieho jazyka

V tejto kapitole je navrhnutý programovací jazyk, do ktorého bude prekladaný spustiteľný binárny kód. Vstupom bude jazyk symbolických inštrukcií, ktorý je získaný z binárneho súboru. Keďže vstupom je jeden súbor, výstupom bude tiež jeden textový súbor s výstupným prekladom.

Pri tvorbe jazyka sme sa inšpirovali existujúcimi jazyky, najmä jazykom ANSI-C. Výsledný jazyk nie je určený na strojové spracovanie. Nepredpokladáme, že by sa k nemu vytvoril prekladač do ďalšieho jazyka. Naším cieľom je vytvoriť jazyk, ktorý bude pre človeka čitateľný. Preto sme sa zamerali hlavne na jeho vizuálnu stránku.

V prípade akejkoľvek nejasnosti v špecifikácii jazyka sa používajú pravidlá z jazyka ANSI-C. Pokiaľ tu nie je explicitne uvedená zmena oproti jazyku C, taktiež sa používajú pravidlá z jazyka ANSI-C.

Príklad zápisu jednoduchého programu v tomto jazyku sa nachádza v prílohe číslo 3.

5.1 Základné elementy

Táto sekcia popisuje základné elementy používané v programovacom jazyku. Zaoberá sa komentármi, tvorbou identifikátorov, zápisom číselných a textových konštánt a obsahuje zoznam kľúčových slov.

5.1.1 Komentáre

Pod komentárom si je možné predstaviť ľubovoľný text. Tento text nie je súčasťou príkazov a výrazov. Používa sa napríklad na slovný popis podprogramu, popis premennej alebo zaujímavej skutočnosti. V podstate to môže byť ľubovoľný text. V jazyku je možné používať iba jednoriadkové komentáre. Každý riadok komentáru musí začínať znakom ';'. Komentár pokračuje do konca riadka. Za komentárom nemôže nasledovať príkaz. Komentár môže byť umiestený za príkazom. Príklad komentáru:

```
var_1256 = 56     ; toto je komentár
```

5.1.2 Kľúčové slová

Kľúčové slová sú slová, ktoré majú špeciálny význam v jazyku. Kľúčové slovo nesmie byť použité ako napríklad identifikátor. Zoznam všetkých kľúčových slov: Begin, End, Procedure, Arguments, Variables, int8, int16, int32, int64, int80, int 128, byte, word, dword, qword, float, double,

Begin_asm, End_asm, if, then, else if, endif, do, while, endwhile, repeat, until, for, to, step, endfor, switch, case, default, goto, break, continue, call, register, EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, AL, CL, DL, BL, AH, CH, DH, BH, AX, CX, DX, BX, SP, BP, SI, DI, ES, CS, SS, DS, FS, GS, ST0, ST1, ST2, ST3, ST4, ST5, ST6, ST7, MMX0, MMX1, MMX2, MMX3, MMX4, MMX5, MMX6, MMX7, XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6, XMM7, eax, ecx, edx, ebx, esp, ebp, esi, edi, al, cl, dl, bl, ah, ch, dh, bh, ax, cx, dx, bx, sp, bp, si, di, es, cs, ss, ds, fs, gs, st0, st1, st2, st3, st4, st5, st6, st7, mmx0, mmx1, mmx2, mmx3, mmx4, mmx5, mmx6, mmx7, xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, xmm7, return, void, signed, unsigned, char, string, const. Niektoré slová v jazyku nie sú použité, ale sú v zozname, aby ich nebolo možné používať ako napríklad názov identifikátorov.

5.1.3 Identifikátory

Identifikátory sú názvy premenných, podprogramov a návestí v programe. Ako názov identifikátoru nie je možné použiť kľúčové slovo, ktoré je použité na špeciálne účely. V názve identifikátora sa rozlišujú malé a veľké písmená. Identifikátory „premenna“ a „Premenna“ sú rozdielne. Odporúčame nepoužívať veľmi podobné názvy identifikátorov v jednom podprograme.

V názve identifikátorov je možné používať nasledovné symboly:

Písmená: a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Číslice: 0 1 2 3 4 5 6 7 8 9
Špeciálne znaky: _

Všetky identifikátory musia začínať písmenom. Za ním môže nasledovať niekoľko písmen, číslic alebo špeciálnych znakov v ľubovoľnom poradí. Dĺžka identifikátoru nie je obmedzená. Pri tvorbe určitých typov identifikátorov existujú nasledovné pravidlá. Pokiaľ ide o identifikátor návestia, musí začínať reťazcom „label_“. Identifikátor podprogramu musí začínať „subroutine_“. Lokálne premenné musia začínať reťazcom „loc_var_“ a argumenty podprogramu reťazcom „arg_“. Globálne premenné majú prefix „global_“.

5.1.4 Konštanty

Konštanta predstavuje číslo, znak alebo textový reťazec. Všetky celé čísla v programe sú v desiatkovej alebo hexadecimálnej číselnej sústave. Čísla v hexadecimálnej sústave majú prefix „0x“ alebo „0X“. V jazyku nie je možné zapísať číselné konštanty v binárnej alebo v osmičkovej

sústave. Čísla pozostávajú z nasledovných symbolov: 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F. Pri tvorbe čísiel nie je žiadne obmedzenie. Veľkosť čísla je obmedzená veľkosťou premennej, do ktorej sa ukladá. V prípade ukladania väčšieho čísla do premennej menšieho typu, nastane orezanie hodnoty. V nasledujúcom príklade je ukázaný formát zápisu čísiel. V príklade sú všetky čísla zapísané v hexadecimálnej sústave a majú rovnakú hodnotu.

```
0x12abcd
0X12ABCD
```

Reálne čísla je možné zapísať iba v desiatkovej sústave. Sú povolené všetky bežné formáty čísiel. Číslo môže byť vytvorené celou časťou a desatinnou časťou alebo celou časťou a exponentom alebo celou časťou, desatinnou časťou a exponentom. Celá časť je od desatinnej časti je oddelená znakom '.' (bodka). Exponent má pred celou postupnosťou nepovinné znamienko '+' (plus) alebo '-' (mínus) a začína znakom 'e' prípadne 'E'.

Znakové konštanty v skutočnosti predstavujú číslo. Hodnota znaku zodpovedá pozícii v ASCII tabuľke. Zápis znakovej konštanty je jednoduchý. Znak je ohraničený symbolmi ' (apostrof). Textový reťazec je sekvencia znakov ohraničená znakom “. Textové reťazce sa používajú na reprezentáciu sekvencie znakov. Pokiaľ nie je možné reprezentovať znak pomocou symbolu, je možné použiť nasledovný zápis: \AA, kde AA predstavuje jeho hodnotu v hexadecimálnom zápise. Tento zápis je možný používať pri znakoch aj pri reťazcoch.

5.2 Dátové typy

V jazyku je možné používať dátové typy podporujúce celé aj reálne čísla. Jazyk symbolických inštrukcií má definovanú presnú veľkosť primitívnych dát, preto bolo potrebné presne určiť veľkosť typov. Všetky celočíselné typy majú definované varianty so znamienkom a bez znamienka. Typy podporujúce reálne čísla, vždy umožňujú uložiť aj zápornú hodnotu. Popis všetkých primitívnych dátových typov je v tabuľke číslo 6. Jazyk neposkytuje typ typu boolean.

Názov typu	Počet bitov	Počet bajtov	Minimálna hodnota	Maximálna hodnota
int8	8	1	-128	127
int16	16	2	-32 768	32 767
int32	32	4	- 2 147 483 648	2 147 483 647
int64	64	8	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
int80	80	10	-604 462 909 807 314 587 353 088	604 462 909 807 314 587 353 087
byte	8	1	0	255
word	26	2	0	65 535
dword	32	4	0	4 294 967 295
qword	64	8	0	18 446 744 073 709 551 615
byte80	80	10	0	1 208 925 819 614 629 174 706 175
float	32	4	1.18e - 38	3.40e 38
double	64	8	2.23e - 308	1.79e 308
float80	80	10	3.37e - 4932	1.18e 4932

Tabuľka 6: Prehľad základných dátových typov

5.3 Ukazovatele a polia

Deklarácia lokálnej premennej je odlišná od jazyka C a C++. Na jednom riadku je možné deklarovať iba jednu premennú. Na prvom mieste je názov premennej. Názov premennej musí začínať reťazcom „loc_var_“, za ktorým nasleduje jednoznačný identifikátor. Identifikátor musí byť jednoznačný iba v príslušnom podprograme. Za názvom nasleduje znak '=', za ktorým nasleduje dátový typ. Prehľad dátových typov je v tabuľke číslo 6. Príklad deklarácie lokálnej premennej typu int32.

```
loc_var_10 = int32
```

Jazyk podporuje ukazovatele v takom istom rozsahu ako sú definované v jazyku C. Na deklaráciu ukazovateľa je potrebné pridať znak '*' pred dátový typ. Adresu premennej je možné získať pomocou znaku '&' pred názvom premennej. Pomocou ukazovateľa je možné pristupovať k polu. Prístup k hodnote pola na pozícii 2 umožňuje nasledovný zápis: loc_var_5[2].

Jazyk podporuje ďalší spôsob prístupu k polu. Na prvom mieste sa uvedie dátový typ pola. Za ním sa určí adresa premennej pomocou výrazu. Vo výraze je možné používať číselné konštanty a premenné. Adresa premennej nemusí byť známa. Jej hodnota sa určí až pri behu programu. Nasledujúci príklad ukazuje prístup k druhému prvku pola na adrese 0x411420. Typ pola je dword:

```
dword[0x411420 + 2]
```

5.4 Operátory a výrazy

Operátory sú veľmi podobné aritmetickým operátorom v jazyku ANSI-C. V tabuľke číslo 7 je stručný prehľad operátorov spolu s popisom. Výsledkom aritmetickej operácie je jej skutočná hodnota. V prípade ukladania väčšieho čísla ako je dostupná veľkosť, sa uloží iba jeho časť. Platia tu rovnaké pravidlá ako v jazyku C. Výsledkom pre porovnávacie a logické operácie je celočíselná hodnota, kde hodnota 0 určuje nepravda a iná hodnota určuje pravda. Bitové operácie umožňujú manipuláciu s jednotlivými bitmi celočíselných typov.

Symbol	Popis	Príklad použitia
++	zvýšenie hodnoty premennej o 1	a++
--	zníženie hodnoty premennej o 1	a--
()	volanie funkcií	nazovFunkcie()
! ~	logický not, bitový not	~a, !a
* / %	násobenie, delenie a zvyšok po delení	a * b, a / b, a % b
+ -	sčítanie a odčítanie	a + b, a - b
<<	bitový posun vľavo	a << b
>>	bitový posun vpravo	a >> b
< > <= >=	menší ako, väčší ako, menší alebo rovný ako, väčší alebo rovný ako	a < b, a > b, a <= b, a >= b
== !=	rovná sa, nerovná sa	a == b, a != b
& ^ |	bitový and, bitový xor, bitový or	a & b, a ^ b, a | b
&& ||	logický and, logický or	a && b, a || b
=	priradenie hodnoty	a = 5

Tabuľka 7: Prehľad operátorov

Operátory sa používajú spolu s operandmi na stavbu výrazov. Je možné použiť v jednom výraze viac operátorov a operandov. Na zabezpečenie správneho poradia vyhodnocovania výrazu je možné použiť zátvorky. Výsledkom výrazu je jeho hodnota, ktorá môže byť uložená do premennej, alebo použitá ako vyhodnocovacia podmienka v podmienených skokoch alebo použitá ako parameter do podprogramu.

Operátor priradenia je reprezentovaný jedným znakom '='. Používa sa na priradenie hodnoty z pravej strany operátora do premennej na ľavej strane. Operátor neumožňuje inicializáciu premennej pri deklarácii. Do premennej je možné priradiť konštantu, hodnotu inej premennej alebo hodnotu výrazu. Taktiež je možné priradiť textový reťazec do premennej typu ukazovateľ na byte alebo na int8. Do premennej sa v skutočnosti priradí iba adresa reťazca. Výsledok priradenia je priradená hodnota. Týmto je umožnené uložiť výsledok priradenia do ďalšej premennej. Napríklad: var1 = var = 8.

Jazyk poskytuje automatické dátové konverzie. Pri konverzii celočíselných znamienkových typov z menšieho rozsahu na väčší rozsah sa zachováva znamienko. Pri konverzii bezznamienkových čísiel sa nezaplnená časť vyplní nulami. Pri konverzii typu väčšieho rozsahu do typu menšieho rozsahu sa zachová iba príslušná časť. V prípade vyšších čísiel ako maximálna hodnota menšieho typu sa číslo automaticky odstráni vyššia časť, ktorú nie je možné uložiť do premennej. V jazyku je umožnená explicitná konverzia. Používajú sa rovnaké pravidlá ako pri implicitnej konverzii. Syntax je rovnaká ako v jazyku C. Napríklad: var1 = (byte) var2.

Priorita jednotlivých operátorov je podobná ako v jazyku ANSI-C. Prehľad všetkých operátorov zoradených podľa priority je v tabuľke číslo 7. Najvyššia priorita je na začiatku tabuľky.

5.5 Príkazy

Každý príkaz musí byť na samostatnom riadku. Nie je možné, aby boli dva výrazy na jednom riadku aj keby boli vizuálne oddelené. Príkaz nie je ukončený žiadnym znakom.

Jazyk podporuje niekoľko typov príkazov. Prvým typom je výraz. Výsledkom výrazu je jeho hodnota. Táto hodnota môže byť uložená do premennej alebo použitá ako podmienka, v parametre funkcie.

Blok niekoľkých príkazov nemusí byť vizuálne oddelený prázdnym riadkom. Vnorený blok v inom bloku taktiež nemusí byť vizuálne zvýraznený. Pre prehľadnosť a čitateľnosť programu odporúčame zvýrazniť vnorený blok odsadením o určitý počet bielych znakov zľava. Začiatok a koniec bloku nie je ohraničený žiadnymi oddeľovačmi, napríklad zátvorkami.

5.6 Skoky

Jazyk poskytuje dva typy skokov: podmienený a nepodmienený. Nepodmienený skok presunie vykonávanie programu na miesto určené parametrom. Používa sa kľúčové slovo goto, za ktorým nasleduje identifikátor návestia. V jazyku sú povolené skoky dopredu aj dozadu. Návestie sa definuje na začiatok nového riadku. Začína textovým reťazcom „label_“, za ktorým nasleduje jednoznačný identifikátor. Platnosť návestia je v danom podprograme. Identifikátor môže byť vytvorený z adresy kódu v pamäti. Návestie je ukončené znakom ':'. Za návestím nemôže nasledovať žiadny príkaz.

```
goto <navestie>
```

Syntax podmienených skokov je podobná syntaxy podmieneným skokov v jazyku C. Jazyk podporuje podmienené skoky if, if – else, if – else – if a switch. Základná forma je nasledovná:

```
if <vyraz> then
      <prikazy>
endif

if <vyraz> then
      <prikazy>
else if < vyraz> then
      <prikazy>
else
      <prikazy>
endif

 switch (<vyraz>)
      case <label1> :
            < prikazy 1>
            break           ; nepovinný príkaz
      case <label2> :
            < prikazy 2>
            break           ; nepovinný príkaz
      default:
            < prikazy 3>
 endswitch
```

Podmienený skok if musí byť ukončený kľúčovým slovom endif. Podobne príkaz switch musí byť ukončený endswitch. Nie je potrebné vizuálne oddelenie príkazov patriacich do bloku od zvyšnej časti programového kódu.

5.7 Cykly

Jazyk podporuje všetky bežné cykly z jazyka C a C++. Syntax cyklu while, do while a for je nasledovná:

```
while <vyraz>
      <prikazy>
 endwhile

 repeat                ; do
      <prikazy>
 until <vyraz>         ; while


 for       <vyraz>
     to    <vyraz>
     step  <prikazy>
     do
           <prikazy>
 endfor
```

Cyklus while musí byť ukončený endwhile. Obdobne cyklus for musí byť ukončený endfor. Na ľubovoľnom mieste v cykle je možné používať kľúčové slová continue a break. Majú rovnaký význam ako v jazyku C. Príkaz break ukončí vykonávanie cyklu a tok programu sa presunie na príkaz za cyklom. Príkaz continue preruší vykonávanie cyklu a spôsobí prechod na ďalší krok cyklu bez toho, aby sa predchádzajúci krok cyklu ukončil. Nespôsobuje skok z tela cyklu von.

5.8 Podprogramy

Syntax podprogramov je veľmi jednoduchá. Prvý riadok obsahuje nasledovné údaje: kľúčové slovo „Procedure: “, za ktorým nasleduje názov podprogramu. Na tomto riadku už nemôže byť žiadna ďalšia informácia okrem komentára.

V prípade, že podprogram používa argumenty, tieto je potrebné deklarovať. Použije sa kľúčové slovo „Arguments:“. Každý argument musí byť na samostatnom riadku. Názov argumentu musí začínať reťazcom „arg_“, kde nasleduje jeho jednoznačný identifikátor. Ďalej nasleduje znak '=', za ktorým nasleduje dátový typ. V prípade, že ide o ukazovateľ, je potrebné pred dátový typ umiestniť znak '*'. Pokiaľ sa predáva referencia, použije sa znak '&'.

Podprogram môže využívať lokálne premenné. Na začiatku deklarácie premenných je použité kľúčové slovo „Variables:“. Každá premenná musí byť na samostatnom riadku. Formát je rovnaký ako u argumentov, ale názov premennej musí začínať reťazcom „loc_var_“ namiesto „arg_“.

Začiatok podprogramu je určený kľúčovým slovom „Begin“ a koniec „End“. Medzi týmito kľúčovými slovami sa nachádza telo podprogramu. Podprogram môže byť vytvorený jedným alebo viacerými príkazmi alebo výrazmi. V tele podprogramu je možné používať iba zadefinované argumenty a lokálne premenné. Tiež je možné používať globálne premenné. Názov premenných je tvorený prefixom „global_“, za ktorým nasleduje typ premennej a jednoznačný identifikátor.

Vykonávanie podprogramu je možné ukončiť na ľubovoľnom mieste. K tomuto účelu sa používa kľúčové slovo return. Pokiaľ by bol tento príkaz úplne posledným príkazom v podprograme, nie je ho potrebné uvádzať. Súčasťou príkazu return nemôže byť výraz. Podprogram nevracia žiadnu hodnotu.

5.9 Zápis inštrukcií

Tento jazyk je primárne určený ako cieľový jazyk na preklad z jazyka symbolických inštrukcií. Preto je potrebné určitým spôsobom zapísať inštrukcie, ktoré sa nepodarilo preložiť. K tomuto účelu sú použité dve kľúčové slová. Prvé slovo „Begin_asm:“ určuje začiatok a druhé slovo „End_asm:“ určuje koniec zápisu inštrukcií. Medzi týmito kľúčovými slovami môže byť jedna alebo viac inštrukcií. Tento zápis je možné použiť iba v tele podprogramu.

6 Návrh a implementácia aplikácie

Cieľom diplomovej práce je vytvoriť funkčnú aplikáciu na prevod binárneho kódu do vyššieho programovacieho jazyka. Aplikácia sa zameriava na spustiteľné súbory pre operačný systém MS Windows. V tejto kapitole sú podrobne popísané základné požiadavky na aplikáciu. Na základe požiadaviek a dostupných informácií je určený konkrétny programovací jazyk a vývojové prostredie.

V ďalších podkapitolách je podrobne uvedený postup riešenia. Riešenie je možné rozdeliť do troch častí: analýza PE hlavičky, preklad z binárneho kódu do jazyka symbolických inštrukcií a preklad z jazyka symbolických inštrukcií do navrhnutého jazyka.

Zdrojové súbory výslednej aplikácii sa nachádzajú na priloženom CD v adresári src. V adresári bin sa nachádza spustiteľná aplikácia, ktorú je možné použiť v operačnom systéme MS Windows 2000, XP, alebo Vista. Na CD sa tiež nachádzajú vzorové príklady spolu s výstupmi, ktoré poskytuje aplikácia. Celkovo sa tam nachádza viac ako 30 rôznych príkladov. Každý príklad je preložený 4 prekladačmi. Sú to prekladače MS Visual Studio 2003, 2005, 2008 a Gcc vo verzii 3.4.4. Prekladač Gcc bol použitý z Cygwin[1] vo verzii 1.5.25. Každý z príkladov sa zameriava na otestovanie určitej časti.

6.1 Požiadavky na aplikáciu

Cieľom diplomovej práce je vytvoriť aplikáciu, ktorá dokáže analyzovať spustiteľné súbory pre operačný systém MS Windows, konkrétne 32-bitová verzia. Program analyzuje PE hlavičku, ktorá obsahuje potrebné informácie o súbore. Program získa zoznam všetkých importovaných modulov (knižníc). Pre každý modul je potrebné zistiť zoznam funkcií, ktoré sú importované. Pre ďalšiu analýzu sú najdôležitejšie informácie o jednotlivých sekciách. Každá sekcia obsahuje informácie o type sekcie, o umiestnení a dĺžke sekcie a po spustení programu o ich umiestnení v pamäti. Každá sekcia má hlavičku a telo. Jednotlivé položky hlavičky sekcie sú popísané v kapitole číslo 2. Najdôležitejšia sekcia je sekcia s programovým kódom. Ďalej nasleduje samotná detekcia jednotlivých inštrukcií. Hlavným cieľom je správne určiť všetky inštrukcie z inštrukčnej sady IA-32. Toto sú najviac používané inštrukcie. Keďže dĺžka inštrukcií je rôzna a informácia o dĺžke inštrukcie nie je nikde uložená, je potrebné správne určiť všetky inštrukcie aj z iných inštrukčných sád ako napríklad MMX, SSE, SSE2, SSE3. Veľkosť operačného kódu inštrukcie určuje typ inštrukcie a použitý spôsob adresovania pamäte a registrov. Viac o spôsobe dekódovania inštrukcií z binárneho súboru sa nachádza v kapitole 3.

1 Cygwin je možné získať z internetovej adresy: http://www.cygwin.com/

Hlavnou úlohou aplikácie je preložiť zistené inštrukcie do navrhnutého jazyka. Je kladený dôraz na korektnú analýzu všetkých inštrukcií z inštrukčnej sady IA-32. V prípade, že nie je možné korektne rozpoznať niektoré inštrukcie, je potrebné túto skutočnosť zaznamenať. Výstupný program je možné uložiť do textového súboru. Na záver aplikácia zobrazí štatistické informácie o analyzovanom súbore, napríklad koľko percent z programu sa podarilo korektne analyzovať a koľko percent z celkového počtu inštrukcií je relevantných inštrukcií. Pre potreby diplomovej práce je potrebné korektne analyzovať minimálne 80 % relevantných inštrukcií pre neoptimalizované výstupy z bežných prekladačov.

Aplikácia je naprogramovaná ako konzolová aplikácia. Ovládanie je pomocou príkazovej riadky, kde sa zadávajú 3 povinné údaje: vstupný súbor, výstupný súbor a požadovaná činnosť. Je možné zadávať ďalšie parametre, ktoré môžu upresniť činnosť aplikácie. Tieto parametre ale nie sú potrebné pre korektný beh. Aplikáciu je možné spustiť pod operačným systémom Microsoft Windows 2000, XP alebo Vista. Aplikácia je naprogramovaná v programovacom jazyku C++. Pre spustenie nie sú potrebné žiadne špeciálne knižnice. Na preklad aplikácie je možné použiť vývojové prostredie MS Visual studio 2003, 2005 alebo 2008. Všetky potrebné súbory pre preklad sú uložené na CD v adresári src.

6.2 Návrh aplikácie

V nasledujúcej časti je stručne popísaný návrh aplikácie. Pre jednoduchosť a prehľadnosť sú zobrazené iba najdôležitejšie časti návrhu. V skutočnosti je návrh a implementácia aplikácie podstatne zložitejšia.

Aplikáciu je možné rozdeliť do troch samostatných častí:

1. analýza PE hlavičky,
2. prekladač z binárneho kódu do jazyka symbolických inštrukcií a
3. prekladač z jazyka symbolických inštrukcií do navrhnutého jazyka.

6.2.1 Analýza PE hlavičky

Prvým krokom pri analýze vstupného súboru je analýza PE hlavičky. Z hlavičky je potrebné zistiť, či ide o spustiteľný súbor pre operačný systém MS Windows. Niektoré položky z PE hlavičky musia mať určitú hodnotu. Podľa týchto hodnôt je možné zistiť, či ide o PE formát. Okrem týchto preddefinovaných hodnôt je potrebné kontrolovať hodnoty niektorých položiek. Nie všetky hodnoty môžu byť povolené. Podrobný popis štruktúry PE hlavičky je v kapitole 2.

Trieda PeParser tvorí základ pre analýzu hlavičky binárneho súboru. Definíciu tejto triedy je možné nájsť v hlavičkovom súbore /src/instAnalyzer/PeParser.h. Na reprezentáciu binárneho súboru sa používa trieda BinaryFile. Táto trieda umožňuje efektívne načítanie potrebných údajov zo súboru.

Trieda PeParser poskytuje základné metódy pre načítanie jednotlivých položiek PE hlavičky. Okrem základných metód pre prístup k položkám, trieda poskytuje pokročilejšie funkcie. Dôležitou súčasťou v súbore sú sekcie. Trieda umožňuje jednoduchý prístup k informáciám o sekciách a k samotným údajom v sekciách. K tomuto účelu sa používa metóda readFromVirtualAddress. Pri prístupe nie je potrebné používať reálnu adresu v súbore, stačí používať virtuálnu adresu. Táto metóda je použitá hlavne pri prístupe k číselným a textovým konštantám.

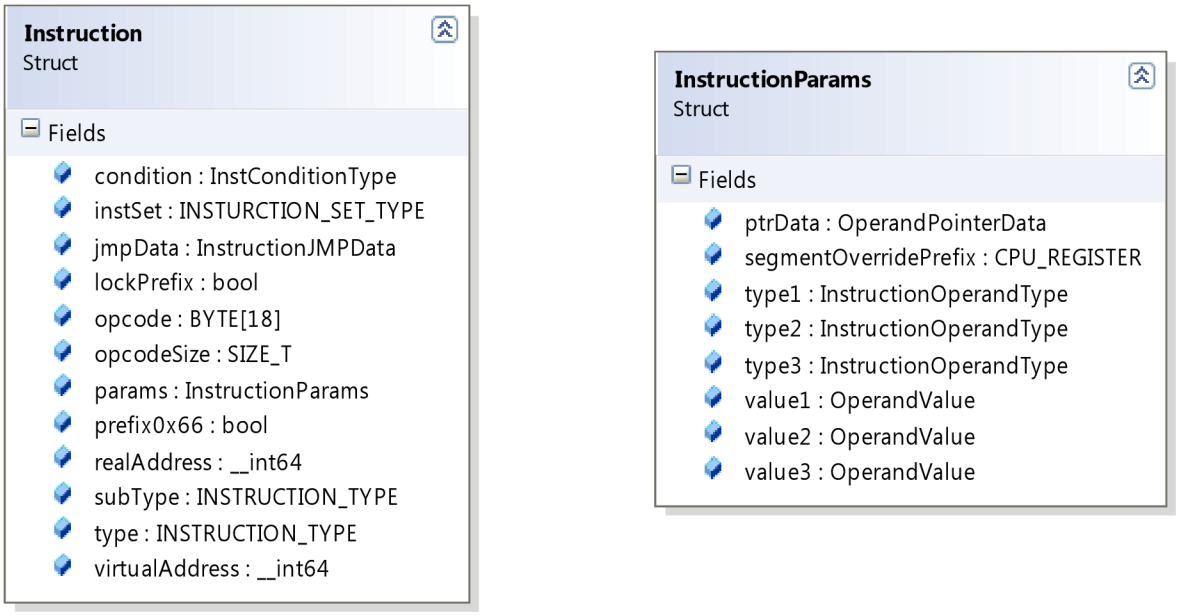
Ďalšou dôležitou funkciou je načítanie zoznamu importovaných knižníc. Pre každú knižnicu je potrebné zistiť jej názov a zoznam všetkých importovaných funkcií. Pre každú funkciu je potrebné zistiť jej názov[1] a adresu. Pomocou adresy sa uskutočňuje volanie funkcie z programu.

K triede PeParser neodmysliteľne patrí trieda PeWriter. Jej cieľom je uložiť hodnoty jednotlivých položiek PE hlavičky do textového súboru v čitateľnej súbore. Taktiež je možné uložiť zoznam importovaných funkcií z iných knižníc. Viac informácií o tejto triede je možné nájsť v hlavičkovom súbore /src/instAnalyzer/PeWriter.h.

6.2.2 Prekladač z binárneho kódu do jazyka symbolických inštrukcií

Po analýze PE hlavičky nasleduje prevod operačného kódu do jazyka symbolických inštrukcií. Teoretický postup prevodu je podrobne popísaný v kapitole 3. Potrebné informácie o jednotlivých inštrukciách sú uložené v štruktúre Instruction. Najdôležitejší údaj je typ inštrukcie a typ inštrukčnej sady. Väčšina inštrukcií obsahuje parametre. Typ parametrov a ich hodnoty sa tiež zaznamenávajú. V štruktúre je uložených viac informácií. Niektoré z nich sú potrebné iba pre určité typy inštrukcií. Zoznam jednotlivých položiek štruktúry Instruction spolu so štruktúrou pre parametre je zobrazený na obrázku číslo 4. Podrobný popis jednotlivých položiek štruktúry je možné nájsť v súbore /src/instAnalyzer/instructionDefs.h. V tomto súbore sú tiež definované všetky typy inštrukcií, inštrukčné sady, parametre a ostatné potrebné informácie pre prácu s inštrukciami.

1 V niektorých prípadoch nie je možné zistiť názov funkcie. Identifikácia funkcie sa vykonáva na základe 32 bitového čísla.



Obrázok 4: Štruktúra Instruction a štruktúra InstructionParams

Najdôležitejšou triedou je InstructionAnalyzer. Na základe vstupného operačného kódu vypĺňa štruktúru Instruction spolu s parametrami. Pre jednoduchý prístup k jednotlivým inštrukciám sa používa trieda InstuctionLoader. Trieda umožňuje sekvenčný prístup k jednotlivým inštrukciám bez nutnosti uchovávania aktuálnej adresy. V prípade potreby je samozrejme možné získať inštrukciu aj na určitej adrese. Trieda poskytuje dva módy. V prvom móde analyzuje všetky inštrukcie vo vstupnom súbore hneď na začiatku. V druhom móde analyzuje inštrukcie až pri ich používaní. Druhý mód má výhodu, že v pamäti je vždy iba jedna inštrukcia a tým dochádza k šetreniu operačnej pamäte. Výhodou prvého módu je rýchlosť v prípade opakovaných prístupov k tým istým inštrukciám, pretože nie je potrebná opakovaná analýza operačného kódu inštrukcie. Pri preklade z jazyka symbolických inštrukcií do navrhnutého jazyka je použitý druhý mód. Pri testoch nebolo zaznamenané výraznejšie spomalenie behu aplikácie oproti prvému módu. Bola zaznamenaná výrazná úspora operačnej pamäte najmä pri analýze väčšieho počtu inštrukcií.

Zápis jednotlivých inštrukcií do textového súboru umožňuje trieda InstructionWriter. Pri svojej činnosti využíva triedu Instruction2Text, ktorá vytvára textovú reprezentáciu inštrukcie. Je možné zadať niekoľko formátov textového výstupu, napríklad či sa má zobrazovať operačný kód inštrukcie, prípadne virtuálna adresa.

6.2.3 Spôsob ukladania výstupného jazyka v pamäti

Výstupný jazyk podporuje rôzne typy príkazov. Najjednoduchší príkaz je výraz, príkaz goto, break alebo continue. Tieto príkazy neobsahujú ďalšie príkazy. Niektoré príkazy môžu obsahovať ďalšie príkazy. Typickým príkazom je if, ktorý obsahuje jeden výraz ako podmienku a blok príkazov ako telo príkazu if.

Pre jednoduchosť je potrebné ku všetkým typom príkazov pristupovať jednotným spôsobom. Preto existuje abstraktná trieda Statement, z ktorej musia byť odvodené všetky triedy predstavujúce akýkoľvek typ príkazu vo výstupnom jazyku. Podrobný popis tejto triedy je možné nájsť v zdrojovom súbore /src/progAnalyzer/statement.h.

Celkovo je dostupných 14 typov príkazov. Prehľadne sú zobrazené na obrázku číslo 5. Nasleduje stručný popis príkazov:

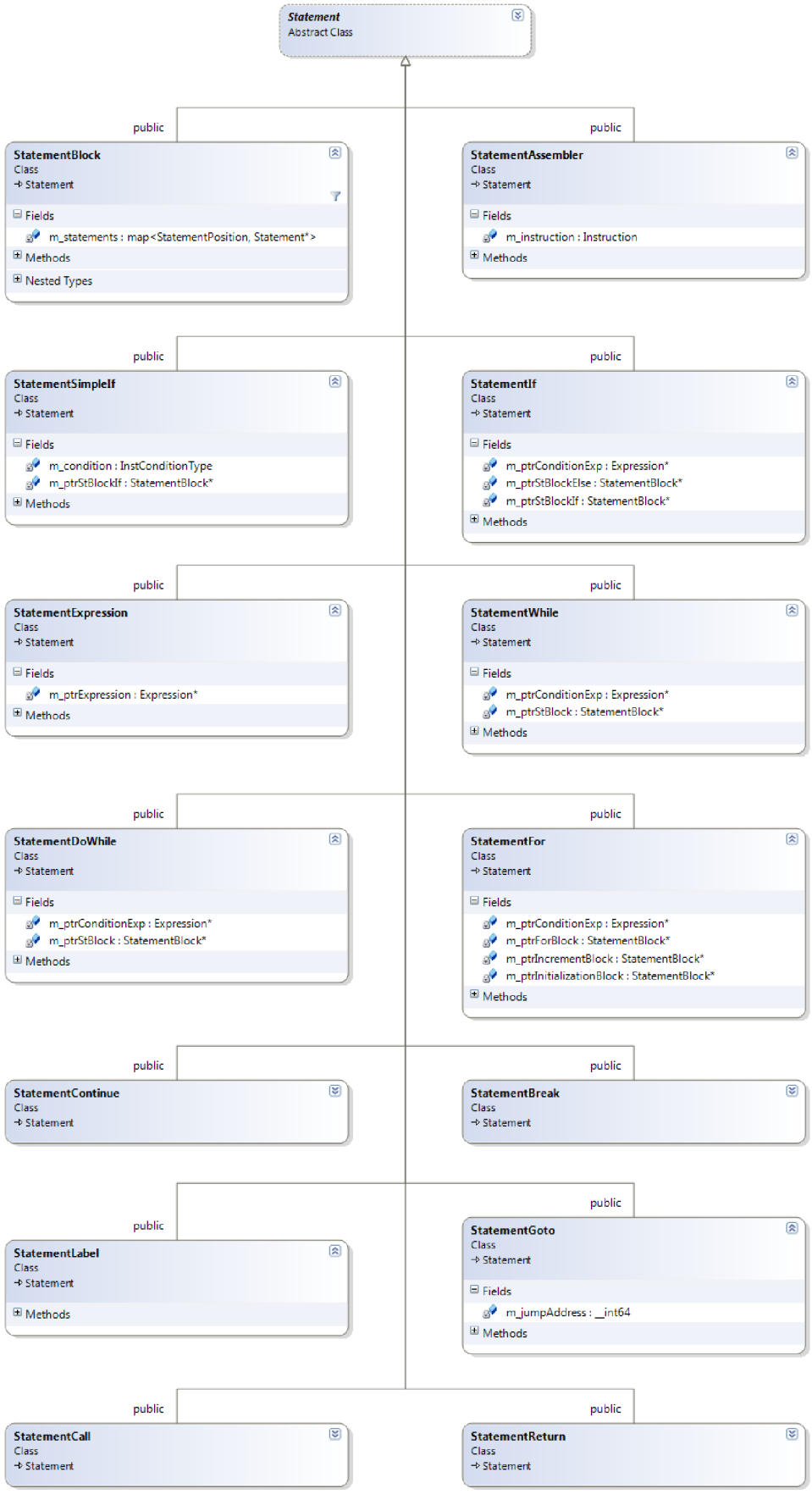
StatementBlock – príkaz, ktorý môže obsahovať ďalšie príkazy. K tomuto typu príkazu sa pristupuje ako k jednému príkazu. Preto sa používa ako príkaz v iných príkazoch, napríklad v príkaze if, while, do while. Každý podprogram je vytvorený z jedeného príkazu tohto typu. StatementBlock obsahuje metódy na prechádzanie medzi jednotlivými vnorenými príkazmi.

StatementAssembler – tento príkaz sa používa na uloženie inštrukcie. Na začiatku analýzy sú všetky inštrukcie vo funkcii uložené pomocou tohto príkazu. V priebehu analýzy sú postupne nahradzované inými typmi príkazmi. Na záver analýzy by sa vo výslednom programu v ideálnom prípade nemal nachádzať ani jeden príkaz tohto typu.

StatementExpression – typ príkazu, ktorý obsahuje výraz. Výraz predstavuje trieda Expression. Výraz je vytváraný hlavne z aritmetických a logických inštrukcií.

StatementSimpleIf – je to zjednodušená obdoba príkazu if. Obsahuje typ podmienky ako v jazyku symbolických inštrukcií a jeden príkaz StatementBlock, ktorý tvorí telo príkazu if. Príkaz neobsahuje časť else. Z tohto príkazu je pri ďalšej analýze odvodený príkaz StatementIf.

StatementIf – je plnohodnotný príkaz if. Oproti príkazu StatementSimpleIf sa líši v podmienke a existencie else bloku. Podmienka je tvorená pomocou výrazu. Pomocou vnorenia príkazov je možné vytvoriť ľubovoľné zanorenie rôznych typov if príkazov, vrátane príkazu typu if – else – if.



Obrázok 5: Diagram tried zobrazujúci typy príkazov

StatementWhile – používa sa na zachytenie potrebných informácií o cykle while. Obsahuje podmienku a jeden blok, ktorý tvorí telo cyklu.

StatementDoWhile – používa sa na uloženie informácií o cykle typu do while. Je podobný príkazu typu StatementWhile. Jediný rozdiel je v podmienke, ktorá sa vyhodnocuje až na konci príkazu.

StatementFor – reprezentuje cyklus for. Okrem tela cyklu obsahuje 2 bloky príkazov a jeden výraz. Prvý blok je inicializačný a druhý blok je inkrementačný. Tieto dva bloky môžu obsahovať ľubovoľné príkazy. Výraz obsahuje podmienku cyklu for.

StatementContinue – predstavuje príkaz continue. Príkaz je možné použiť iba v tele cyklu.

StatementBreak – predstavuje príkaz break, ktorý znamená prerušenie vykonávania aktuálneho cyklu. Môže sa použiť iba v tele cyklu.

StatementGoto – zachytáva nepodmienený skok. Obsahuje adresu skoku. Na danej adrese musí existovať StatementLabel.

StatementLabel – označuje návestie. Tento príkaz určuje adresu skoku.

StatementCall – príkaz predstavuje volanie podprogramu. Zaznamenáva názov volaného podprogramu, jeho adresu a predávané parametre. V niektorých prípadoch nie je možné zistiť adresu volaného podprogramu. V tomto prípade sa zaznamená výraz, ktorý určuje adresu. Keďže podprogramy nevracajú žiadnu návratovú hodnotu, nie je potrebné túto informáciu uchovávať.

StatementReturn – príkaz predstavuje ukončenie aktuálneho podprogramu. Môže byť umiestnený na ľubovoľnom mieste v podprograme.

V predchádzajúcom texte sú stručne popísané triedy, ktoré reprezentujú výstupný jazyk v pamäti. Pomocou týchto tried je možné zostaviť obsah ľubovoľného podprogramu v navrhnutom jazyku. Samotný podprogram reprezentuje trieda Procedure. Obsahuje základné informácie o podprograme – začiatočná a koncová adresa, argumenty a lokálne premenné. Tiež obsahuje informácie o miestach volania daného podprogramu. Vďaka tomu by bolo možné vytvoriť graf volania podprogramov. Zoznam všetkých podprogramov je uložený v triede TableProcedures.

6.2.4 Základná štruktúra prekladača

Samotný preklad sa vykonáva v niekoľkých iteráciách. Pre každú iteráciu existuje samostatný „analyzátor“. Pod pojmom analyzátor je myslená trieda, ktorá dokáže určitým spôsobom previesť vstupný jazyk na výstupný jazyk. Výsledkom analyzátora je medzivýsledok, ktorý je vstupom do ďalšieho analyzátora. Výsledok každého analyzátora by sa mal zlepšovať a výsledok posledného analyzátora by mal byť výstupný jazyk. Na poradí vykonávania jednotlivých analyzátorov záleží.

Každý analyzátor sa zameriava iba na určitý problém. Týmto sa zjednoduší implementácia samotného analyzátora. Využíva sa prístup rozdeľuj a panuj. Každý analyzátor je tvorený triedou, ktorá je odvodená z abstraktnej triedy Analyzer. Táto trieda je definovaná v súbore /src/progAnalyzer/analyzer.h.

Samotná analýza prebieha na úrovni celých blokov príkazov ale aj na úrovni jednotlivých príkazov. Všetky príkazy, ktoré obsahujú ďalšie príkazy, sa musia postarať o správne zavolanie analyzátora aj pre vnorené príkazy. Napríklad príkaz typu StatementIf zavolá analyzátor pre príkaz StatementIf, ale aj na vnorený blok príkazov (príkaz StatementBlock). V prípade, že by príkaz StatementIf obsahoval aj časť „else“, tak sa zavolá aj pre túto časť. Podobne StatementBlock zavolá analyzátor pre všetky vnorené príkazy. Týmto spôsobom sa zabezpečí analýza celého podprogramu.

Výsledkom analýzy môže byť odstránenie niektorých príkazov, pridanie nových príkazov a nahradenie príkazov za iné príkazy. Tieto úpravy nie je možné vykonávať priamo v analýze. Výsledok je potrebné uložiť do objektu triedy AnalyzeResults, ktorá sa postará o aktualizáciu podprogramu v správnom okamihu. Týmto spôsobom sa zjednoduší implementácia analyzátora a predchádza sa možným komplikáciám pri analýze podprogramu.

Jedným z hlavných požiadaviek na návrh aplikácie bola možnosť jednoduchého rozšírenia činnosti programu. Cieľom je bez veľkých zmien vylepšiť aktuálne prekladové schopnosti programu. Nový analyzátor je pomerne jednoduché vložiť na vhodné miesto medzi ostatné analyzátory. Analyzátor je možné vložiť v súbore /src/progAnalyzer/procedure.h, konkrétne v metóde analyze v triede Procedure. V tejto metóde sú volané skoro všetky existujúce analyzátory.

Pridať nový analyzátor do programu je jednoduché, zložitejšie je ho vytvoriť. Celkovo je vytvorených 10 analyzátorov. Každý analyzátor sa zameriava na určitý typ problému. Nasleduje stručný popis jednotlivých analyzátorov v poradí ako sa vykonávajú.

6.2.4.1 Trieda AnalyzerSimpleIf

Tento analyzátor vyhľadáva inštrukcie typu JMP a Jcc. Z inštrukcie JMP sa vytvára príkaz StatementGoto, ktorý predstavuje nepodmienený skok. Na adrese skoku je potrebné vytvoriť návestie – príkaz typu StatementLabel.

Pre podmienené inštrukcie Jcc sa vytvára príkaz typu StatementSimpleIf. Môžu nastať dva prípady. V prvom prípade je adresa skoku väčšia ako adresa inštrukcie. Telo príkazu if je vytvorené zo všetkých inštrukcií od inštrukcie skoku až po inštrukciu na adrese skoku. V opačnom prípade je telo príkazu if vytvorené iba z príkazu StatementGoto.

```
jne Label_1                    if (jne)
    ; príkazy                      ; príkazy
Label_1:                       endif
```

6.2.4.2 Trieda AnalyzerExpressionCreate

Hlavnou úlohou analyzátora je vyhľadať všetky inštrukcie, z ktorých je možné vytvoriť matematický výraz. Do výrazu je možné previesť všetky aritmetické inštrukcie (ADD, SUB, DIV...), logické inštrukcie (AND, OR, XOR, NOT...) a inštrukcie pre bitový posun (RCL, RCR, ROL, ROR). Taktiež je implementované vytváranie výrazov z niektorých inštrukcií z inštrukčnej sady x87 (FADD, FSUB, FMUL, FDIV...).

Väčšina z týchto inštrukcií používa dva parametre. Výsledok operácie je uložený na miesto určené prvým parametrom. Napríklad inštrukcia ADD EAX, EBX spočíta hodnotu registrov EAX a EBX a výsledok uloží do registra EAX. Z tejto inštrukcie je možné vytvoriť matematický výraz EAX = EAX + EBX. Podobným spôsobom je možné vytvoriť výrazy aj z iných inštrukcií.

Pri vytváraní výrazov sa vypĺňa tabuľka symbolov. Tabuľka symbolov tvorí podstatnú časť najmä pri ďalšej úprave výrazov ako je spájanie viacerých výrazov do jedného. Ďalej sa používa pri určovaní lokálnych a globálnych premenných. V tabuľke sa uchovávajú informácie o použitých premenných, registroch a číselných konštantách. Ďalej sa uchovávajú pomocné premenné vytvorené pre potreby aplikácie.

Vnútorná štruktúra tabuľky je implementovaná pomocou zoznamu premenných. Pre každú premennú sa uchovávajú podrobnejšie informácie: typ premennej, jednoznačný identifikátor, názov a dátový typ. Každá premenná obsahuje ďalší zoznam, v ktorom sú zaznamenávané jej hodnoty vzhľadom k určitému miestu.

Implementácia tabuľky symbolov je uskutočnená pomocou niekoľkých tried. Najdôležitejšou triedou je TableVariables, ktorá poskytuje jednotný prístup k premenným. Z tabuľky symbolov je možné získať pokročilejšie informácie. Napríklad je možné zistiť hodnotu premennej na určitej pozícii v programe. Taktiež je možné získať zoznam lokálnych a globálnych premenných a zoznam argumentov, ktoré boli predané podprogramu.

Výstupný jazyk obsahuje blokové štruktúry. V tabuľke symbolov je potrebné zaznamenať platnosť premennej vzhľadom k určitému miestu. Premenná môže mať na určitom mieste známu hodnotu, ale na inom mieste už nemusí byť známa. Pre každú premennú sa preto zaznamenávajú všetky platné hodnoty pre príslušnú časť podprogramu.

Analyzátor AnalyzerExpressionCreate ukladá do tabuľky symbolov iba číselné konštanty a použité registre. Pri určovaní hodnoty registru je potrebné vziať do úvahy viacero faktorov. Jedným z cieľov tabuľky symbolov je určiť rozsah platnosti hodnoty registrov. Na nasledujúcom príklade je možné vidieť rozsah platnosti hodnoty registru EAX.

```
EAX = 0x10              ; nastavenie hodnoty registru na 0x10

; hodnota registru EBX je neznáma, nie je možné určiť, či sa
; vykoná telo podmienky

if (EBX == 0x12)
                        ; hodnota registru EAX je 0x10
    EAX = 0x11;
                        ; hodnota registru EAX je 0x11
endif

; na konci príkazu if je hodnota registru EAX nedefinovaná
EBX = EAX          ; hodnota registru ebx je tiež nedefinovaná
```

6.2.4.3 Trieda AnalyzerExpressionPointer

Tento analyzátor priamo naväzuje na predchádzajúci analyzátor. Jeho cieľom je podrobnejšie skúmať výrazy a pokúsiť sa v nich vyhľadať ukazovatele, lokálne a globálne premenné. Nie je možné ho spojiť s predchádzajúcim analyzátorom, lebo v tej dobe ešte nie sú známe hodnoty registrov a ich rozsah platnosti. Rozsah platnosti je známy až po skončení predchádzajúceho analyzátora.

Určenie lokálnych premenných je v podstate problém určenia všetkých prístupov na zásobník. Na začiatku podprogramu je vrchol zásobníka určený hodnotou registra ESP. Pri prístupe do pamäte sa určuje, či adresa pamäte je vyjadrená pomocou hodnoty registra ESP. Napríklad prístup do pamäte na adrese ESP + 4 určuje lokálnu premennú, ktorá je vzdialená 4 bajty od vrcholu zásobníka. Pokiaľ by vzdialenosť bola záporná, je to argument predávaný podprogramu. Globálne premenné sa určujú podobným spôsobom. Sú určené absolútnou adresou, neexistuje tu žiadny vzťah k vrcholu zásobníka.

6.2.4.4 Trieda AnalyzerExpressionG

Táto trieda sa zameriava na úpravu výrazov. Cieľom je spájať jednoduché výrazy do jedného zložitejšieho. Pri spájaní sa snaží vytvoriť najmenší možný počet výrazov. Nasledujúci príklad ukazuje spojenie 8 jednoduchých výrazov do jedného zložitejšieho.

```
reg_eax = arg_1a
reg_eax = reg_eax + arg_1b
reg_eax = reg_eax - arg_1c
reg_ecx = arg_1b
reg_ecx = reg_ecx * arg_1c
reg_ecx = reg_ecx + 0x8
reg_eax = reg_eax * reg_ecx
loc_var_18 = reg_eax
```

Táto postupnosť výrazov je ekvivalentná s nasledujúcim výrazom:

```
loc_var_18 = (arg_1a + arg_1b - arg_1c) * (arg_1b * arg_1c + 0x8)
```

6.2.4.5 Trieda AnalyzerIf

V tomto momente sú už vytvorené výrazy. Analyzátor AnalyzerIf spája výrazy s príkazom StatementSimpleIf a vytvára plnohodnotný príkaz if. Výraz, ktorý bude použitý v podmienke sa získa z tabuľky príznakov (trieda TableFlags).

Trieda TableFlags zaznamenáva všetky prístupy k registrom príznakov. V analyzátore AnalyzerExpressionCreate sa pre príslušné inštrukcie nastaví zmena príslušných bitov registru príznakov a výraz, ktorý ich spôsobil. V AnalyzerIf sa zisťuje, ktorý výraz spôsobil modifikáciu daného bitu príznakového registra.

6.2.4.6 Trieda AnalyzerIfG

Tento analyzátor pracuje priamo s príkazmi typu StatementIf. Preto ho nie je možné spojiť priamo s analyzátorom AnalyzerIf. Cieľom analyzátora je spojiť viac príkazov if do jedného príkazu if so zložitejšou podmienkou. Príklad spájania dvoch príkazov if do jedného:

```
if (loc_var_5 == 0x12)
    if (loc_var_6 == 0x5)

        ;  príkazy

    endif
endif


if (loc_var_5 == 0x12 && loc_var_6 == 0x5)

    ;  príkazy

endif
```

Ďalšou úlohou analyzátora je vytváranie časti else pre príkaz if. Príklad:

```
if (loc_var_5 == 0x12)
    ; príkazy 1
    goto Label_1:

endif

    ; príkazy 2

Label_1:
```

```
if (loc_var_5 == 0x12)
    ; príkazy 1

else

    ; príkazy 2

endif
```

6.2.4.7 Trieda AnalyzerCycles

Tento analyzátor je zameraný na detekciu cyklov typu while, do – while a for a neštruktúrovaných cyklov. Najjednoduchšia analýza je cyklu while. Vyhľadáva sa príkaz if, kde posledný príkaz v tele príkazu if je príkaz goto s adresou na začiatok príkazu if. Napríklad:

```
Label_1:
if (reg_eax == 0x12)
    ; príkazy
    goto Label_1
endif
```

```
while (reg_eax == 0x12)
    ; príkazy
endwhile
```

Podobným spôsobom sa určuje cyklus do – while. Rozdiel je iba v podmienke, ktorá sa nachádza až na konci cyklu. V príklade je cyklus do – while zapísaný pomocou kľúčových slov repeat until.

```
Label_1:
    ; príkazy
if (reg_eax == 0x12)
    goto Label_1
endif
```

```
repeat
    ;  príkazy
until (reg_eax == 0x12)
```

Aplikácia dokáže detekovať cyklus for v nasledujúcom tvare:

```
var_i = 0
goto Label_1

Label_2:
var_i = var_i + 1

Label_1:
if (var_i < 8)

    ; príkazy
    goto Label_2
endif
```

Výsledný cyklus for:

```
for (var_i = 0, var_i < 8, var_i = var_i + 1)
    ; príkazy
endfor
```

Posledným typom cyklu je neštruktúrovaný cyklus. Tento cyklus má podmienku uprostred tela. Nasledujúci príklad ukazuje neštruktúrovaný cyklus, ktorý je zapísaný pomocou cyklu while. Cyklus je možné opustiť iba pomocou príkazu break.

```
Label_1:

; príkazy 1

if (i > 5)
    goto Label_2
endif

; príkazy 2
goto Label_1

Label_2:
```

```
while (1)

    ; príkazy 1

    if (i > 5)
        break
    endif

    ; príkazy 2

endwhile
```

6.2.4.8 AnalyzerBreakContinue

V predchádzajúcom analyzátore boli detekované cykly while, do – while a for. Teraz je možné určiť príkazy break a continue. Nasleduje príklad detekcie príkazu break v cykle while.

```
while (loc_var_28 > 0x0)
    ; príkazy 1
    if (loc_var_26 > 0x8)
        goto Label_1
    endif
    ; príkazy 2
endwhile
Label_1:
```

```
while (loc_var_28 > 0x0)
    ; príkazy 1
    if (loc_var_26 > 0x8)
        break
    endif
    ; príkazy 2
endwhile
```

Príklad detekcie príkazu continue v cykle do – while:

```
repeat
    ; príkazy 1
    if (loc_var_26 > 0x8)
        goto Label_1
    endif
    ; príkazy 2
Label_1:
until (loc_var_28 > 0x0)
```

```
repeat
    ; príkazy 1
    if (loc_var_26 > 0x8)
        continue
    endif
    ; príkazy 2
until (loc_var_28 > 0x0)
```

6.2.4.9 Trieda AnalyzerSimpleCall

Cieľom analyzátora je určiť adresu volaného podprogramu. Adresa podprogramu môže byť zadaná pomocou číselnej konštanty. V tomto prípade nie je žiadny problém určiť adresu. V ďalšom prípade môže byť adresa volania zadaná pomocou výrazu. Je potrebné určiť adresu aj v tomto prípade. Nasledujúci príklad zobrazuje volanie podprogramu pomocou registra. Program musí korektne určiť adresu volaného podprogramu v obidvoch prípadoch.

```
edx  = 0x411410                    edx  = 0x411410
call edx                           call subroutine_411410()

; príkazy, ktoré nemenia           ; príkazy
; hodnotu registru edx

call edx                           call subroutine_411410()
```

6.2.4.10 Trieda AnalyzerCall

Záverečný analyzátor, ktorý sa vykonáva až po analyzovaní všetkých podprogramov v programe. Na základe volaného podprogramu priradí predávané argumenty. Zoznam všetkých predávaných argumentov je možné získať až po analyzovaní volaného podprogramu. Preto sa vykonáva až po analyzovaní všetkých podprogramov. V niektorých prípadoch nie je možné určiť všetky predávané argumenty. Nasledujúci príklad v jazyku symbolických inštrukcií ukazuje nemožnosť zistenia adresy pri prístupe k pamäti na zásobníku. Je možné, že v prípade spustenia programu sa pristupuje k niektorému z argumentov.

```
; Hodnota registru ebp nie je známa. V prípade spustenia
; programu, pravdepodobne určuje nejakú pozíciu na zásobníku.
mov eax, dword ptr [ebp - 0x10]
```

6.2.5 Zápis výsledného programu

Na záver, keď už sú analyzované všetky podprogramy, je možné vytvoriť textovú reprezentáciu výsledného programu. K tomuto účelu sú vytvorené triedy ProgramWriter a ProcedureWriter. Zápis programu prebieha priamo do textového súboru.

6.2.5.1 Trieda ProcedureWriter

Trieda ProcedureWriter umožňuje prevod podprogramu do textového formátu. Trieda umožňuje nastavenie jednoduchého formátovania. Napríklad nastavenie veľkosti odsadenia príkazov pri vnorenom bloku. Pri zápise sa dodržiava požadovaná syntax výstupného jazyka ako bola definovaná v kapitole 5.

6.2.5.2 Trieda OperatorPrecedenceTable

Táto trieda sa používa pri prevode výrazu do textu. Výraz v pamäti je reprezentovaný triedou Expression. Expression zaznamenáva správne poradie vykonávania jednotlivých operátorov. Problém nastáva až pri linearizácii výrazu do textového reťazca, kde sa priorita operátorov určuje pomocou zátvoriek. Trieda OperatorPrecedenceTable určuje, ktoré zátvorky je potrebné zapísať pri zachovaní správnej sémantiky výrazu. Priorita operátorov je určená v tabuľke číslo 7.

6.2.5.3 Trieda ProgramWriter

Trieda umožňuje automatický zápis všetkých podprogramov do výstupného súboru. Zoznam všetkých podprogramov získava z triedy TableProcedures.

6.3 Ovládanie aplikácie

Ovládanie aplikácie je veľmi jednoduché. Všetky potrebné informácie pre aplikáciu sa zadávajú pomocou príkazovej riadky. Formát vstupných parametrov je nasledovný:

-i <názov vstupného súboru> -o <názov výstupného súboru> -a <akcia>

Za parametrom „-i“ nasleduje názov vstupného súboru. Predpokladá sa, že je to binárny súbor spustiteľný pod operačným systémom Windows. V prípade iného súboru sa nepokračuje v ďalšej analýze. Názov výstupného súboru sa zadáva za parameter „-o“. Aplikácia kontroluje, či výstupný súbor existuje. V prípade, že existuje, činnosť aplikácie je prerušená. Túto vlastnosť je možné zmeniť použitím nepovinného parametru „-overwrite“. Posledný parameter je „-a“, za ktorým nasleduje požadovaná akcia. Povolené sú tieto hodnoty:

asm – Aplikácia analyzuje hlavičku súboru a v prípade korektného súboru analyzuje časť s programovým kódom. Získané inštrukcie zapíše v textovej podobe do výstupného súboru.

all – Aplikácia analyzuje vstupný súbor a prekladá binárny kód do výstupného jazyka. Výstupný jazyk je zapísaný do súboru. Formát výstupu zodpovedá navrhnutému jazyku v kapitole 5.

proc <adresa> – Aplikácia analyzuje vstupný súbor a preloží podprogram začínajúci na zadanej adrese do výstupného jazyka. Výstup sa zapíše do zadaného súboru. Adresa je hexadecimálne číslo, ktorá určuje virtuálnu adresu prvej inštrukcie. Výsledný text podprogramu môže byť rozdielny od textu podprogramu v predchádzajúcom prípade. Je to spôsobené tým, že zo zadaného podprogramu nie je možné určiť všetky informácie ako z celého programu.

Za povinnými parametrami môžu nasledovať nepovinné parametre. Dostupné sú tieto tri:

-overwrite – Ak výstupný súbor existuje, prepíše ho bez varovania.

-exp <1..32> – Určuje maximálny počet jednoduchých výrazov, z ktorých môže byť vytvorený zložitejší. Preddefinovaná hodnota je 8. Hodnota 1 znamená, že sa nebudú vytvárať zložitejšie výrazy. Maximálna povolená hodnota je 32.

-level <1..8> – Hodnota určuje počet použitých analyzátorov. Analyzátory AnalyzerCall a AnalyzerSimpleCall sa vykonávajú vždy.

Príklady spustenia aplikácie:

```
disasm.exe -i vstup.exe -o vystup.txt -a asm
disasm.exe -i vstup.exe -o vystup.txt -a all
disasm.exe -i vstup.exe -o vystup.txt -a all -level 5
disasm.exe -i vstup.exe -o vystup.txt -a proc 0x411400
```

6.4 Zhodnotenie a návrh ďalšieho rozšírenia

Funkčnosť aplikácie bola overená na viacerých príkladoch. Tieto príklady sa nachádzajú na priloženom CD. Základ z príkladov tvoria jednoduché programy napísané v jazyku C a C++. Tieto príklady boli preložené 4 prekladačmi: MS Visual Studio 2003, 2005, 2008 a Gcc vo verzii 3.4.4. Prekladač Gcc bol použitý z Cygwin vo verzii 1.5.25. Každý z príkladov sa zameriava na otestovanie určitej časti.

Príklady na CD sú rozdelené podľa použitého prekladača a nastavených parametrov prekladu. U každého príkladu je priložený analyzovaný program, textový zápis inštrukcií a výstup prekladu. Tiež sa tam nachádza celkový výsledok analýzy. Pre každý podprogram je zobrazený celkový počet inštrukcií a počet analyzovaných inštrukcií v absolútnom čísle a v percentách. Na záver je zobrazený celkový počet podprogramov a výsledok analýzy. Pre neoptimalizované výstupy z testovaných prekladačov aplikácia dokáže korektne preložiť viac ako 90 % relevantného vstupu tvorený inštrukciami z inštrukčnej sady IA-32.

Aplikácia poskytuje funkčnosť požadovanú zadaním. Oproti zadaniu je pridaná analýza niektorých inštrukcií z inštrukčnej sady x87. Nasleduje zoznam najdôležitejších vlastností, ktoré aplikácia podporuje:

- podpora pre spustiteľné súbory pre operačný systém MS Windows 32-bitová verzia,
- podpora pre dynamické knižnice (DLL),
- analýza PE hlavičky,
- načítanie všetkých importovaných modulov a funkcií,
- preklad binárneho kódu do inštrukcií z inštrukčných sád IA-32, x87, MMX, SSE, SSE2, SS3, SS4,
- zápis inštrukcií do textového súboru,
- preklad z jazyka symbolických inštrukcií do navrhnutého jazyka,
- podpora naviac používaných inštrukcií z inštrukčnej sady IA-32,
- podpora niektorých inštrukcií z inštrukčnej sady x87,
- detekcia podprogramov,
- detekcia lokálnych a globálnych premenných v podprogramoch,
- určovanie rozsahu platnosti premenných,
- detekcia zložitých matematických výrazov,
- spájanie viacerých výrazov do jedného,
- možnosť nastavenia maximálneho počtu výrazov, ktoré majú byť spojené do jedného,
- detekcia polí,
- detekcia príkazov if, if – else, if – else – if,
- detekcia cyklov while, do – while a for,
- detekcia neštruktúrovaných cyklov s podmienkou uprostred,
- detekcia zložitejších podmienok pre podmienené príkazy a cykly,
- detekcia volaných podprogramov z iných modulov,
- detekcia niektorých predávaných argumentov podprogramom.

Aplikáciu by bolo možné rozšíriť o veľké množstvo nových činností. Azda najdôležitejším by bolo doplnenie prekladu ďalších inštrukcií najmä z inštrukčnej sady x87 a MMX.

Inštrukčná sada procesorov Intel obsahuje veľké množstvo inštrukcií. V súčasnosti nie je možné preložiť všetky inštrukcie do navrhnutého jazyka. Navrhnutý jazyk bude potrebné doplniť o ďalšiu funkčnosť. Príkladom môže byť práca s portami. V jazyku symbolických inštrukcií sa používajú inštrukcie IN a OUT. Inštrukcia IN získa jeden bajt z portu, ktorého adresa je zadaná ako parameter. Inštrukcia OUT pošle na port zadaný bajt.

V určitých prípadoch je možné zistiť predávané argumenty podprogramom. Zistenie, čo najväčšieho počtu argumentov zvyšuje kvalitu prekladu. Pre importované podprogramy z iných modulov neexistuje zoznam predávaných argumentov. Preto nie je možné zobraziť predávané argumenty. Jedným z riešením by bola analýza importovaného modulu a zistenie argumentov exportovaných podprogramov. Pre zvýšenie rýchlosti by bolo možné najbežnejšie moduly dopredu analyzovať a uložiť požadované výsledky do súboru. Tieto výsledky by sa podľa potreby načítali pri preklade súboru.

Ďalším krokom by bolo možné detekovať určité rysy z objektových programov. Pre detekciu tried je potrebné detekovať štruktúry. Potom by bolo možné detekovať triedy a ich metódy.

V súčasnosti sa čoraz viac používajú 64-bitové verzie operačného systému Windows. Spúšťacie súbory pre tieto systémy majú upravenú hlavičku PE. Oproti formátu pre 32-bitové verzie sú upravené niektoré položky. Najväčší rozdiel je vo veľkosti adries. Veľkosť adries pre 64-bitové systémy je 8 bajtov. Pre tieto systémy sa používa inštrukčná sada x86-64, ktorá je veľmi podobná sade IA-32.

7 Záver

Cieľom diplomovej práce je navrhnúť a implementovať aplikáciu na prevod binárneho kódu do vyššieho programovacieho jazyka. Súčasťou práce je tiež návrh programovacieho jazyka.

Táto práca sa zameriava na binárne súbory pre operačný systém MS Windows. Na začiatku práce sa čitateľ podrobne oboznámil so štruktúrou hlavičky binárneho súboru a so spôsobom dekódovania programového kódu do jazyka symbolických inštrukcií.

V kapitole číslo 5 je navrhnutý programovací jazyk. Pri tvorbe jazyka som sa inšpiroval existujúcimi jazykmi, najmä jazykom C a C++. Návrh jazyka je jednoduchý a pre potreby tejto práce postačuje. V prílohe číslo 3 je zobrazená časť programu zapísaná v tomto jazyku. V prílohe 1 a 2 je uvedený príslušný program zapísaný v jazyku C a v jazyku symbolických inštrukcií.

Výsledná aplikácia ukazuje prácu s binárnymi súbormi pre operačný systém MS Windows. Aplikácia dokáže analyzovať hlavičku súboru a zistiť požadované informácie. Na základe zistených informácií potom nasleduje analýza programového kódu a detekcia jednotlivých inštrukcií. Ďalším krokom je prevod jazyka symbolických inštrukcií do navrhnutého jazyka. Pre neoptimalizované výstupy z testovaných prekladačov aplikácia dokáže korektne preložiť viac ako 90 % relevantného vstupu tvorený inštrukciami z inštrukčnej sady IA-32. Rozšírenie oproti zadaniu je analýza základných inštrukcií zo sady x87.

Program dokáže korektne detekovať výrazy spolu s ich úpravou. Napríklad zjednodušovanie výrazov, spájanie viacerých výrazov do jedného, odstránenie výrazov, ktoré nemajú sémantický vplyv na program. Ďalej vyhľadáva riadiace konštrukcie (if, if – else, if – else if) a cykly (while, do – while, for). Dokáže rozpoznať samostatné podprogramy. Pre podprogramy sa pokúša detekovať predávané parametre, lokálne a globálne premenné.

Program by bolo možné rozšíriť o detekciu ostatných inštrukcií, najmä z inštrukčnej sady x87 a MMX. Taktiež by bolo vhodné vylepšiť detekciu zložitejších dátových štruktúr. Po detekovaní štruktúry by bolo možné detekovať triedy a ich metódy. Ďalším vylepšením aplikácie by mohla byť analýza binárnych súborov pre 64-bitové operačné systémy.

Túto prácu je možné využiť pre podrobné oboznámenie sa so spôsobom prekladu vyšších programovacích jazykov do jazyka symbolických inštrukcií. Po rozšírení funkčnosti by bolo možné prácu využiť v počítačovej bezpečnosti. Z analyzovaných programov je možné získať určité charakteristické znaky, na základe ktorých je možné stanoviť podobný kód.

Literatúra

[1] Rudolf Marek: *Učíme se programovat v jazyce Assembler pro PC.* Computer Press, Brno, 2003, 228 s. ISBN 80-722-6843-0

[2] Češka Milan, Ježek Karel, Melichar Bořivoj, Richta Karel: *Konstrukce překladačů*. Vydavatelství ČVUT, Praha, 1999, 636 s. ISBN 80-01-02028-2

[3] *Intel 64 and IA-32 Architectures Software Developer`s Manual, Volume 1: Basic Architecture.* [online], URL http://www.intel.com/design/processor/manuals/253665.pdf

[4] *Intel 64 and IA-32 Architectures Software Developer`s Manual, Volume 2A: Instruction Set Reference, A-M.* [online], URL http://www.intel.com/design/processor/manuals/253666.pdf

[5] Intel 64 and IA-32 Architectures Software Developer`s Manual, Volume 2B: Instruction Set Reference, N-Z. [online], URL http://www.intel.com/design/processor/manuals/253667.pdf

[6] Matt Pietrek: *An In-Depth Look into the Win32 Portable Executable File Format.*[online] 2002. URL http://msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx

[7] Matt Pietrek: *An In-Depth Look into the Win32 Portable Executable File Format, Part 2.* [online] 2002. UR http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/default.aspx

[8] Matt Pietrek: *Peering Inside the PE: A Tour of the Win32 Portable Executable File Format.* [online] 1994. URL http://msdn2.microsoft.com/en-us/library/ms809762.aspx

[9] *Microsoft Portable Executable and Common Object File Format Specification.* [online] 2008. URL http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

[10] *Debug Help Library.* [online] 2007. URL http://msdn2.microsoft.com/en-us/library/ms679309%28VS.85%29.aspx

[11] Dimitri van Heesch: *Doxygen*. [online] 2008. URL http://www.stack.nl/~dimitri/doxygen

Zoznam príloh

Príloha 1. Príklad programu v jazyku C

```
=============================================================================
Zápis časti programu v jazyku C
=============================================================================

// deklarácia globálnej premennej
int g_globalna = 0;

// funkcia test
int test(int a, int b, int c) {

	int value1 = 0;
	int value2 = 1;
	int value3 = 2;
	int i;

	if (a == 0) {
		value1++;
		g_globalna+=2;
	}

	if (a > 0 && b <= 5) {
		value1++;
	} else {
		++value2;
	}

	for (i = 0; i < 10; i++) {
		value3 += i * b;
	}

	while (c > 0) {

		printf("%d\n", c);
		if (c == 8) {
			break;
		}
		c = c - 2;
	}

	value1 = (a + b - c) * (8 + b * c);
	i = 8;
	do {
		value1 += a;
		if (value1 > 8) {
			continue;
		}
		i--;
	} while (i > 0) ;
	return value1;
}
```

Príloha 2. Ukážka výstupu inštrukcií

```
;=====================================================================
; Adresa   Binárny kód inštrukcie  Inštrukcia
;=====================================================================

00411410:  55                     push        ebp
00411411:  8b ec                  mov         ebp, esp
00411413:  81 ec f0 00 00 00      sub         esp, 0xf0
00411419:  53                     push        ebx
0041141A:  56                     push        esi
0041141B:  57                     push        edi
0041141C:  8d bd 10 ff ff ff      lea         edi, [ebp-0xf0]
00411422:  b9 3c 00 00 00         mov         ecx, 0x3c
00411427:  b8 cc cc cc cc         mov         eax, 0xcccccccc
0041142C:  f3 ab                  rep stosd   dword ptr [edi]
0041142E:  c7 45 f8 00 00 00 00   mov         dword ptr [ebp-0x8], 0x0
00411435:  c7 45 ec 01 00 00 00   mov         dword ptr [ebp-0x14], 0x1
0041143C:  c7 45 e0 02 00 00 00   mov         dword ptr [ebp-0x20], 0x2
00411443:  83 7d 08 00            cmp         dword ptr [ebp + 0x8], 0x0
00411447:  75 16                  jne         address: 41145f
00411449:  8b 45 f8               mov         eax, dword ptr [ebp-0x8]
0041144C:  83 c0 01               add         eax, 0x1
0041144F:  89 45 f8               mov         dword ptr [ebp-0x8], eax
00411452:  a1 60 71 41 00         mov         eax, dword ptr [0x417160]
00411457:  83 c0 02               add         eax, 0x2
0041145A:  a3 60 71 41 00         mov         dword ptr [0x417160], eax
0041145F:  83 7d 08 00            cmp         dword ptr [ebp + 0x8], 0x0
00411463:  7e 11                  jle         address: 411476
00411465:  83 7d 0c 05            cmp         dword ptr [ebp + 0xc], 0x5
00411469:  7f 0b                  jg          address: 411476
0041146B:  8b 45 f8               mov         eax, dword ptr [ebp-0x8]
0041146E:  83 c0 01               add         eax, 0x1
00411471:  89 45 f8               mov         dword ptr [ebp-0x8], eax
00411474:  eb 09                  jmp         address: 41147f
00411476:  8b 45 ec               mov         eax, dword ptr [ebp-0x14]
00411479:  83 c0 01               add         eax, 0x1
0041147C:  89 45 ec               mov         dword ptr [ebp-0x14], eax
0041147F:  c7 45 d4 00 00 00 00   mov         dword ptr [ebp-0x2c], 0x0
00411486:  eb 09                  jmp         address: 411491
00411488:  8b 45 d4               mov         eax, dword ptr [ebp-0x2c]
0041148B:  83 c0 01               add         eax, 0x1
0041148E:  89 45 d4               mov         dword ptr [ebp-0x2c], eax
00411491:  83 7d d4 0a            cmp         dword ptr [ebp-0x2c], 0xa
00411495:  7d 0f                  jge         address: 4114a6
00411497:  8b 45 d4               mov         eax, dword ptr [ebp-0x2c]
0041149A:  0f af 45 0c            imul        eax, dword ptr [ebp + 0xc]
0041149E:  03 45 e0               add         eax, dword ptr [ebp-0x20]
004114A1:  89 45 e0               mov         dword ptr [ebp-0x20], eax
004114A4:  eb e2                  jmp         address: 411488
004114A6:  83 7d 10 00            cmp         dword ptr [ebp + 0x10],0x0
004114AA:  7e 2e                  jle         address: 4114da
004114AC:  8b f4                  mov         esi, esp
004114AE:  8b 45 10               mov         eax, dword ptr [ebp + 0x10]
004114B1:  50                     push        eax
004114B2:  68 3c 57 41 00         push        0x41573c
004114B7:  ff 15 bc 82 41 00      call        dword ptr ds:[0x4182bc]
004114BD:  83 c4 08               add         esp, 0x8
```

```
004114C0:  3b f4                    cmp         esi, esp
004114C2:  e8 74 fc ff ff           call        address: 41113b
004114C7:  83 7d 10 08              cmp         dword ptr [ebp + 0x10], 0x8
004114CB:  75 02                    jne         address: 4114cf
004114CD:  eb 0b                    jmp         address: 4114da
004114CF:  8b 45 10                 mov         eax, dword ptr [ebp + 0x10]
004114D2:  83 e8 02                 sub         eax, 0x2
004114D5:  89 45 10                 mov         dword ptr [ebp + 0x10], eax
004114D8:  eb cc                    jmp         address: 4114a6
004114DA:  8b 45 08                 mov         eax, dword ptr [ebp + 0x8]
004114DD:  03 45 0c                 add         eax, dword ptr [ebp + 0xc]
004114E0:  2b 45 10                 sub         eax, dword ptr [ebp + 0x10]
004114E3:  8b 4d 0c                 mov         ecx, dword ptr [ebp + 0xc]
004114E6:  0f af 4d 10              imul        ecx, dword ptr [ebp + 0x10]
004114EA:  83 c1 08                 add         ecx, 0x8
004114ED:  0f af c1                 imul        eax, ecx
004114F0:  89 45 f8                 mov         dword ptr [ebp-0x8], eax
004114F3:  c7 45 d4 08 00 00 00     mov         dword ptr [ebp-0x2c], 0x8
004114FA:  8b 45 f8                 mov         eax, dword ptr [ebp-0x8]
004114FD:  03 45 08                 add         eax, dword ptr [ebp + 0x8]
00411500:  89 45 f8                 mov         dword ptr [ebp-0x8], eax
00411503:  83 7d f8 08              cmp         dword ptr [ebp-0x8], 0x8
00411507:  7e 02                    jle         address: 41150b
00411509:  eb 09                    jmp         address: 411514
0041150B:  8b 45 d4                 mov         eax, dword ptr [ebp-0x2c]
0041150E:  83 e8 01                 sub         eax, 0x1
00411511:  89 45 d4                 mov         dword ptr [ebp-0x2c], eax
00411514:  83 7d d4 00              cmp         dword ptr [ebp-0x2c], 0x0
00411518:  7f e0                    jg          address: 4114fa
0041151A:  8b 45 f8                 mov         eax, dword ptr [ebp-0x8]
0041151D:  5f                       pop         edi
0041151E:  5e                       pop         esi
0041151F:  5b                       pop         ebx
00411520:  81 c4 f0 00 00 00        add         esp, 0xf0
00411526:  3b ec                    cmp         ebp, esp
00411528:  e8 0e fc ff ff           call        address: 41113b
0041152D:  8b e5                    mov         esp, ebp
0041152F:  5d                       pop         ebp
00411530:  c3                       ret
```

Príloha 3. Ukážka výstupu prekladu

```
; ================================================================
; Zápis časti programu v navrhnutom jazyku
; ================================================================
;
; Podprogram sa začína reťazcom „Procedure: “, za ktorým nasleduje
; jednoznačný názov podprogramu.
; Názov argumentu začína reťazcom „arg_“.
; Názov lokálnej premennej začína reťazcom „loc_var_“.
; Názov globálnej premennej začína „global_“, kde nasleduje typ
; premennej a jeho jednoznačný identifikátor.
; Príkazy nemusia byť ukončené žiadnym oddeľovačom.
; Každý príkaz musí byť na samostatnom riadku.


; ================== P R O C E D U R E ================================

Procedure: subroutine_411410

; Ref call:
;     subroutine_411145 + 0x0

Arguments:

      reg_ecx = &int32
      reg_eax = &int32
      arg_2f = dword
      arg_32 = dword
      arg_34 = dword


Variables:

      loc_var_18 = int32
      loc_var_28 = dword
      loc_var_29 = dword
      loc_var_2a = dword
      loc_var_2b = dword
      loc_var_2c = dword
      loc_var_2d = dword
      loc_var_2e = dword
      loc_var_33 = dword
      loc_var_35 = dword
      loc_var_36 = dword

Begin:

    loc_var_28 = reg_ebp
    reg_ebp = reg_esp - 0x4
    loc_var_29 = reg_ebx
    loc_var_2a = reg_esi
    loc_var_2b = reg_edi
    reg_esp = reg_esp - 0x4 - 0xf0 - 0x4 - 0x4 - 0x4
    reg_edi = &(loc_var_29)
    reg_ecx = 0x3c
```

```
    for (loc_var_18 = 0x0, loc_var_18 < reg_ecx, ++loc_var_18) do

        reg_edi[loc_var_18 * 0x4] = -0x33333334

    endfor

    loc_var_2c = 0x0
    loc_var_2d = 0x1
    loc_var_2e = 0x2

    if (arg_2f == 0x0)

        loc_var_2c = loc_var_2c + 0x1
        global_dword_417160 = global_dword_417160 + 0x2
    endif


    if (arg_2f > 0x0 && arg_32 <= 0x5)

        loc_var_2c = loc_var_2c + 0x1

    else
        loc_var_2d = loc_var_2d + 0x1

    endif


label_41147f:

    for (loc_var_33 = 0x0,loc_var_33 < 0xa,loc_var_33 = loc_var_33+0x1) do

        loc_var_2e = loc_var_33 * arg_32 + loc_var_2e

    endfor


label_4114a6:

    while (arg_34 > 0x0)

        reg_esi = reg_esp
        loc_var_35 = arg_34
        loc_var_36 = 0x41573c
        reg_esp = reg_esp - 0x4 - 0x4

        call MSVCR80D.dll_printf()

        reg_esi == (reg_esp + 0x8)

        call subroutine_41113b(reg_ecx)


        if (arg_34 == 0x8)

            break

        endif

        arg_34 = arg_34 - 0x2

    endwhile
```

```
label_4114da:
    reg_ecx = arg_32 * arg_34 + 0x8
    loc_var_2c = (arg_2f + arg_32 - arg_34) * (arg_32 * arg_34 + 0x8)
    loc_var_33 = 0x8

label_4114fa:

    repeat

        loc_var_2c = loc_var_2c + arg_2f

        if (loc_var_2c > 0x8)


            continue

        endif

        loc_var_33 = loc_var_33 - 0x1

label_411514:

    until (loc_var_33 <= 0x0)

    reg_eax = loc_var_2c
    reg_edi = loc_var_2b
    reg_esi = loc_var_2a
    reg_ebx = loc_var_29
    reg_ebp == (reg_esp + 0x4 + 0x4 + 0x4 + 0xf0)

    call subroutine_41113b(reg_ecx)

    reg_esp = reg_ebp + 0x4
    reg_ebp = loc_var_28

    return


End
```

Príloha 4

CD

Obsah priloženého CD

Jednotlivé časti sú uložené v nasledujúcich podadresároch:

`text`

Tu je umiestnená elektronická podoba tejto diplomovej práce vo formáte OpenDocument (OpenOffice 2.x) a PDF (Portable Document Format 1.4).

`src`

Tu sú umiestnené zdrojové kódy a všetky súbory potrebné pre kompiláciu aplikácie.

`doc`

Adresár obsahuje programovú dokumentáciu vo formáte HTML. Dokumentácia bola vygenerovaná zo zdrojových súborov pomocou programu Doxygen.

`bin`

Tento adresár obsahuje preloženú aplikáciu spolu s návodom na spustenie.

`testy`

Adresár obsahuje testovacie príklady. Celkovo je tu viac ako 30 testovacích príkladov. Príklady sú rozdelené podľa použitého prekladača a nastavených parametrov prekladu. U každého príkladu je priložený analyzovaný program, textový zápis inštrukcií a výstup prekladu. Tiež sa tam nachádza zhodnotenie prekladu. Pre každý podprogram je zobrazený počet inštrukcií a počet analyzovaných inštrukcií v absolútnom čísle a v percentách. Na záver je zobrazený celkový počet podprogramov a výsledok analýzy.