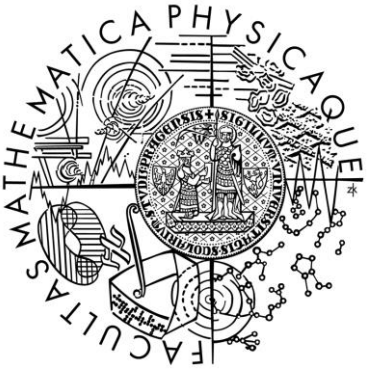
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Tomáš Faltín

Interaktivní disassembler pro procesory architektury Intel 64

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Jakub Yaghob, Ph.D.

Studijní program: Informatika

Studijní obor: Programování

Praha 2014

Rád bych tímto poděkoval vedoucímu této práce RNDr. Jakubu Yaghobovi, Ph.D. za pomoc a čas, který mi při psaní práce věnoval. Dále bych chtěl poděkovat kolegům z práce, kteří my poskytli čas na napsání práce a v neposlední řádě samozřejmě mojí rodině a kamarádům za podporu a trpělivost, kterou se mnou mají.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V …..... dne..... podpis

Název práce: Interaktivní disassembler pro procesory architektury Intel 64

Autor: Tomáš Faltín

Katedra / Ústav: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Jakub Yaghob, Ph.D., Katedra softwarového inženýrství

Abstrakt: Práce se zabývá implementací disassembleru-debuggeru pro procesory architektury Intel64. Disassembler si instrukce i jejich formát načítá z předpřipravených XML souborů, čímž je zajištěna rozšiřitelnost i pro budoucí instrukce a architektury. Disassembler dokáže po vložení libovolného programu v jednom z podporovaných formátů převést vykonávané instrukce vloženého programu do jazyka symbolických adres. Pomocí debuggeru je následně možné tento program spustit a kontrolovat tok vykonávaných instrukcí. Instrukce je možné procházet v pořadí, jakém jsou právě vykonávány anebo umístit na určitou instrukci programu breakpoint, na kterém se vykonávání program zastaví. Debugger je schopný zobrazit jednotlivá vlákna běžícího programu a také najít a rozpoznat základní vyšší programové struktury jako jsou podmíněný příkaz a cyklus.

Klíčová slova: disassembler, Intel 64

Title: Interactive disassembler for Intel 64 processors

Author: Tomáš Faltín

Department: Department of Software Engineering

Supervisor: RNDr. Jakub Yaghob, Ph.D., Department of Software Engineering

Abstract: The aim of this thesis is to create disassembler-debugger for Intel64 processors. Disassembler loads instructions and instruction's format from XML files which implies future extensibility for new instructions and architecture. After inserting of a program disassembler converts its instructions to assembly language. Debugger is able to run and control the program's instruction flow by stepping over single instruction or by setting some breakpoints that stops the program. Debugger can also show program's threads and find some basic programming structures like if statements and loops.

Keywords: disassembler, Intel 64

Obsah

Úvod ... 1
Cíle práce ... 1
Struktura práce ... 1
1 Background ... 3
1.1 Architektura ... 3
1.1.1 Architektura x86 a IA-32 ... 3
1.1.2 Architektura Intel 64 ... 4
1.2 Jazyk symbolických adres (assembly language) ... 4
1.3 Kódování instrukcí ... 5
1.3.1 Prefixy instrukcí ... 6
1.3.2 REX prefix ... 7
1.3.3 VEX prefix ... 7
1.3.4 Opcode ... 8
1.3.5 ModR/M ... 8
1.3.6 SIB ... 8
1.3.7 Displacement a Immediate ... 9
1.4 Debugger ... 9
1.4.1 Výjimky a přerušení ... 9
1.4.2 INT3 ... 9
1.4.3 EFLAGS ... 9
1.5 PE formát ... 10
2 Analýza problému ... 11
2.1 Podpora instrukcí ... 11
2.2 Načítání instrukcí ... 11
2.3 Program pro načítání ... 12
2.4 Zpracování instrukcí ... 13
2.4.1 Zpracování instrukce ... 14
2.4.2 Zpracování operandů ... 14
2.5 Hledání instrukcí ... 15
2.5.1 Velikost nejmenší jednotky ... 15
2.5.2 Zvolená struktura ... 15
2.5.3 Zotavení z chyb ... 16
2.6 Výpis instrukcí ... 16
2.7 Rozpoznávání programových struktur ... 17
2.8 Debugging ... 18
2.9 Uživatelské rozhraní ... 19
2.10 Ostatní disassemblery ... 20
2.10.1 IDA ... 20
2.10.2 OllyDbg ... 20
2.10.3 Ndisasm ... 20
3 Implementace ... 21
3.1 Architektura aplikace ... 21
3.2 UI ... 22
3.3 ConsoleUI ... 22
3.4 Dictionary ... 23
3.4.1 Třída Trie ... 23
3.4.2 Abstraktní třída Code ... 25

3.4.3 Třída Modul ... 25
3.4.4 Třída Instruction ... 26
3.4.5 Abstraktní třída AbstractOperand ... 26
3.4.6 Třída Dictionary ... 27
3.4.7 Třída Loader ... 28
3.5 Soubor instructions.xml ... 29
3.6 Soubor instruction_format.xml ... 30
3.6.1 Definice formátu instrukce ... 31
3.6.2 Definice pravidel pro operandy ... 33
3.6.3 Definice pravidel pro opcode ... 35
3.7 Debugger ... 36
3.8 CodeAnalyser ... 38
3.9 Načítané soubory ... 39
5 Závěr ... 40
5.1 Zhodnocení práce ... 40
5.2 Možnosti rozšíření a vylepšení ... 40
Seznam použité literatury ... 42
Seznam použitých zkratek ... 43
A Uživatelský manuál ... 44
A.1. Obsah CD ... 44
A.2. Požadavky programu ... 44
A.3. Obsluha programu ... 44
B Formáty souborů ... 48
B.1. Schéma souboru *instruction.xml* ... 48
B.2. Schéma souboru *instruction_format.xml* ... 49

Úvod

Při vývoji aplikací dochází často k tvorbě chyb. Pokud máme k dispozici zdrojový kód programu, ve kterém je chyba, nemívají s tím ladící nástroje problém. Pokud ovšem nemáme z nějakého důvodu ke zdrojovým kódům přístup, například protože se jedná o nějaký program nebo knihovnu, pro kterou už výrobce ukončil podporu, je dost často potřeba použít specializovaných programů – disassemblerů. Pro nalezení chyby je ideální mít disassembler provázaný s debuggerem, který nám zobrazí jednotlivé vykonávané instrukce a jednotlivá vlákna ve vícevláknové aplikaci. Protože jazyk symbolických adres je nízkoúrovňový, není s ním práce pro běžného programátora příjemná. Proto je dobré, pokud disassembler dokáže zobrazit vyšší programové struktury, které jsou již ve všech vyšších programovacích jazycích podobné.

Cíle práce

Cílem této práce je vytvoření disassembleru-debuggeru pro procesory architektury Intel64. Při vytváření se klade velký důraz na jeho pozdější rozšiřitelnost, ať už přidávání nových instrukcí nebo celých architektur s novým formátem instrukcí. Disassembler slouží k převodu strojového kódu programu do jazyka symbolických adres pro cílový procesor. Pro snadnější orientaci v kódu umí disassembler najít vyšší programové struktury, jako jsou podmíněné příkazy a cykly. Pro potřeby dynamické analýzy kódu je součástí programu i debugger. Debugger dovoluje překládaný program spustit a procházet ho společně s vykonáváním jednotlivých instrukcí, případně umístit na libovolné místo v kódu breakpoint, na kterém se výpočet přeloženého programu zastaví.

Struktura práce

V první kapitole se nachází popisy věcí, které je sice možno někde dohledat, ale pro tvorbu naší aplikace byly klíčové a proto jsme je zde chtěli zdůraznit. Uvádíme zde popis a stručný vývoj architektury. Dále je zde uvedeno základní kódování instrukcí a operandů do jazyka symbolických adres. Na konci kapitoly je část věnovaná principu debuggování a dekódování spustitelných souborů.

V druhé kapitole rozebíráme podrobně náš řešený problém s jednotlivými možnostmi, které jsme měli. Věnujeme se zde problému s načítáním, zpracováním a

následným uložením a vyhledáváním instrukcí. Jako další rozebíráme postup, jakým jsme rozpoznávali programové struktury v přeloženém kódu. Rozebíráme zde také problémy při implementaci debuggeru, jako je umísťování breakpointů. Věnujeme se zde uživatelskému rozhraní a tomu, proč jsme nakonec zvolili vytvoření konzolové aplikace. Na konci kapitoly je krátký popis a porovnání jiných disassemblerů, které stojí za zmínku.

Ve třetí kapitole se rozebírá implementace naší aplikace. Na začátku je zobrazena architektura celé naší aplikace a dále pak v jednotlivých podkapitolách popisujeme funkce jednotlivých částí aplikace.

Na závěr je uvedeno zhodnocení celé naší práce společně s dalšími možnostmi pokračování ve vývoji.

V příloze je uveden uživatelský manuál s postupem instalace programu a popis jeho ovládání doplněný o obrázky. V další části jsou umístěny XML schémata pro definování instrukcí a jejich pravidel.

1 Background

1.1 Architektura

Architekturou počítače označujeme jeho koncepční strukturu a logickou organizaci. Jedním ze základních prvků každého dnešního počítače je procesor. Jedná se většinou o složitý integrovaný obvod, který vykonává zadané strojové instrukce. Systémová architektura procesoru je tvořena datovými typy, instrukcemi, registry, adresováním, paměťovou architekturou, přerušením a zpracováním výjimek, vstupem a výstupem.

1.1.1 Architektura x86 a IA-32

Označení x86 značí třídu procesorů, které jsou zpětně kompatibilní s instrukční sadou procesoru Intel 8086. Jedná se o plně 16bitový procesor. Po něm následovali další procesory jako 80186, 80286 atd. Proto se pro tyto procesory vžilo označení x86. Další procesor s názvem 80386 už ale obsahoval i 32bitový režim. Samozřejmě byl stále zpětně kompatibilní s předchozími 16bitovými procesory. Architektura, která vycházela z tohoto procesoru, se označuje IA-32. Někdy je také tato architektura nesprávně označována právě jako x86.

Tento procesor byl tedy prvním s architekturou IA-32. Představil 32bitové registry, jejichž dolních 16bitů bylo používáno jako dřívější 16bitové registry. Pro podporu zpětné kompatibility s dřívějšími procesory architektury x86 je obsažen reálný režim. Jedná se o režim, ve kterém je možno přímo přistupovat k hardwaru, jelikož neobsahuje žádnou formu ochrany paměti. Není zde podporován například ani multitasking. Dalším režimem, který byl představený již s procesorem 80286, je chráněný režim. Ten již obsahuje jistou formu ochrany paměti – objevuje se zde podpora pro virtuální paměť, čehož může být dosaženo segmentováním. Pro podporu zpětné kompatibility byl vytvořen virtuální 8086 režim, který dovoluje běžet programy pro reálný režim, ale již za kontroly nějakého operačního systému.

Architektura IA-32 je dnes minimální doporučená pro běh moderních operačních systému. Všechny výše zmíněné režimy jsou dnes používány, byť již v menší míře než dříve. Reálný režim a chráněný režim jsou využívány při startu počítače, například k běhu BIOSu a startu operačního systému. Další režimy se využívají pro běh 16bitových nebo 32bitových aplikací.

1.1.2 Architektura Intel 64

Intel s firmou HP se na začátku vývoje snažili prosadit 64bitové procesory s architekturou IA-64, která byla odlišná a nekompatibilní s dosavadní používanou architekturou IA-32. Toho využívala firma AMD, která představila 64bitové procesory architektury AMD64 (značeno také jako x64, x86-64 nebo x86_64), které byly zpětně kompatibilní s architekturou IA-32. Pro velký úspěch těchto procesorů byl Intel nucen vytvořit klon této architektury, kterou pojmenovali IA-32e. Později byl název změněn na EM64T a následně znova změněn na Intel64. Dnešní osobní počítače jsou již tvořeny převážně 64bitovými procesory.

Jako s každou novou architekturou, přišla řada novinek a vylepšení. Například přibyly nové a větší registry, zvětšil se virtuální a fyzický adresový prostor a přibylo nové adresování relativně vůči IP[1]. Samozřejmě přibyl i nový režim procesoru – IA-32e. V tomto režimu je možné spouštět 64bitové programy. Pro spuštění starších, 16 a 32bitových aplikací pod 64bitový operačním systémem, slouží režim kompatibility. Starší programy, určené buď pro reálný, nebo virtuální 8086 režim, již v tomto režimu spustit nejdou.

1.2 Jazyk symbolických adres (assembly language)

Jazyk symbolických adres (JSA[2]) je nízkoúrovňový programovací jazyk, který je tvořen symbolickou reprezentací jednotlivých strojových instrukcí a konstant, potřebných pro vytvoření strojového kódu pro daný procesor. Jako u jiných jazyků se pro překlad jazyka symbolických adres do strojového kódu používá překladač, v tomto konkrétním případě se mu říká assembler.

Každá instrukce v JSA vypadá následovně:

jméno_instrukce operand1, operand2, operand3, operand4.

Operandy jsou u instrukcí volitelné, takže jsou instrukce bez operandů i instrukce se čtyřmi.

[1] Instruction pointer – registr s ukazatelem na další instrukci

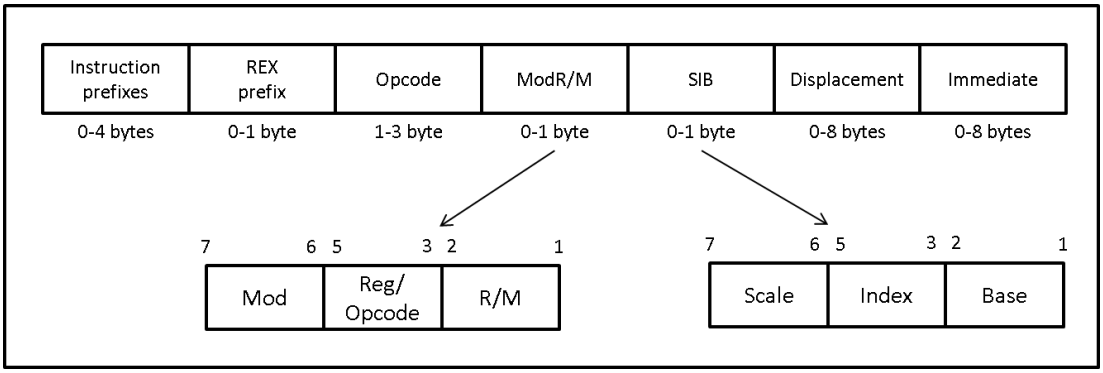
[2] Jazyk symbolických adres

1.3 Kódování instrukcí

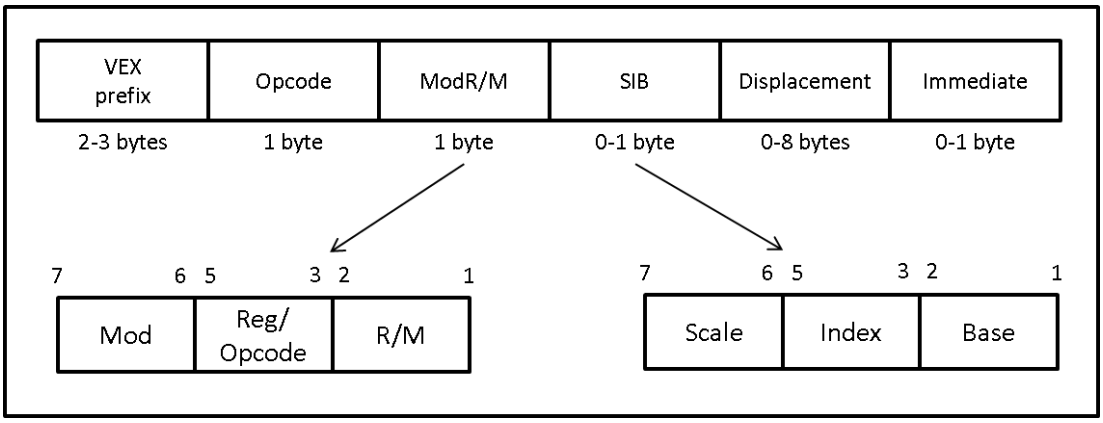
Každý dnešní procesor obsahuje několik set instrukcí. Ty lze dále rozdělit do několika tříd. Každá třída instrukcí může obsahovat vlastní kódování jednotlivých instrukcí. Následující informace jsou do podrobností vypsány v dokumentu [**1**]. Zde jsou uvedeny jen základní obecné informace.

Při překladu instrukcí z JSA do strojového kódu dojde k nahrazení instrukcí byty. Maximálně může být použito 15 bytů k zakódování jedné instrukce. Každý z jednotlivých bytů má určitý název a roli.

Instrukce je zakódována do jednoho ze dvou formátů ukázaných na obrázcích 1 a 2. Formátování s VEX prefixem je použito například u AVX instrukcí. Každé z polí má jméno a je u něho vždy napsáno, kolik může zabírat bytů. Některá pole se opět skládají z dalších polí, a proto je na obou obrázcích zobrazen výřez s jednotlivými čísly bitů.



Obrázek 1: Formát kódování instrukcí



Obrázek 2: Formát kódování instrukcí s VEX prefixem

1.3.1 Prefixy instrukcí

Existují čtyři skupiny prefixů. Z každé skupiny se může v instrukci vyskytovat maximálně jeden prefix. Prefixy jsou většinou volitelné, ale existují i instrukce, pro které je určitý prefix povinný a používá se jako identifikátor.

Skupiny prefixů:

1. Lock a Repeat
    - LOCK prefix je zakódovaný pomocí bytu 0xF0. Pokud je prefix uveden, tak následující instrukce je provedena atomicky. Tento prefix může být použit pouze s určitými instrukcemi
    - REPNE/REPNZ prefix je kódován pomocí bytu 0xF2. Používá se jen s operacemi pracující s řetězci nebo vstupem a výstupem
    - REP/REPE/REPZ prefix je zakódován pomocí bytu 0xF3. Používá se taktéž jen s operacemi, které pracují s řetězci nebo s vstupem a výstupem
2. Segmentové prefixy a nápověda pro skoky – nápověda je použita pouze s instrukcemi skoku
    - Použití segmentového registru CS je kódováno jako 0x2E
    - Použití segmentového registru SS je kódováno jako 0x36
    - Použití segmentového registru DS je kódováno jako 0x3E
    - Použití segmentového registru ES je kódováno jako 0x26
    - Použití segmentového registru FS je kódováno jako 0x64
    - Použití segmentového registru GS je kódováno jako 0x65
    - Prefix 0x2E je použit, pokud skok není proveden
    - Prefix 0x3E je použit, pokud skok je proveden
3. Velikosti operandů.
    - Prefix 0x66 slouží k přepínáním na neimplicitní hodnotu velikostí operandů
4. Adresování
    - Prefix 0x67 slouží k přepínání na neimplicitní velikost použitou k adresování

1.3.2 REX prefix

REX prefix je prefix používaný v 64bitovém režimu. Pomocí tohoto prefixu definujeme použití 64bitových registrů anebo rozšířených stávajících registrů. Níže uvedená tabulka 1 ukazuje jednotlivé bity prefixu a jejich jméno a funkci.

Jméno	Bity	Definice
	7-4	0100
W	3	1 = 64bitové operandy
R	2	rozšíření pole reg
X	1	rozšíření pole index
B	0	rozšíření pole r/m, base nebo reg

Tabulka 1: Bity prefixu REX

1.3.3 VEX prefix

Prefix VEX byl představen se záměrem nahradit u nových instrukcí všechny běžně používané prefixy a uvozovací byty u pole opcode. Také zde nalezneme bity pro uložení čtvrtého operandu.

Existují dvě velikosti prefixu VEX. Rozlišit je od sebe můžeme pomocí prvního bytu, který je rozdílný. Prefix o velikosti tři byty se používá jako náhrada za prefix REX a instrukce s 3bytovým polem pro opcode, v ostatních případech si vystačíme s prefixem o velikosti dva byty. Níže uvedené tabulky 2 a 3 ukazují jednotlivé bity prefixu a jejich jméno a funkci.

Jméno	Byte	Bity	Definice
	1	7-0	0xC5
R	2	7	negace hodnoty REX.R
vvvv	2	6-3	číslo registru v jedničkovém doplňku
L	2	2	délka vektoru: 0 = skalár nebo 128bitový vektor, 1 = 256bitový vektor
pp	2	1-0	plní funkci prefixu: 01 = 0x66, 10 = 0xF3, 11 = 0xF2

Tabulka 2: Bity 2bytového prefixu VEX

Jméno	Byte	Bity	Definice
	1	7-0	0xC4
R	2	7	negace hodnoty REX.R
X	2	6	negace hodnoty REX.X
B	2	5	negace hodnoty REX.B
m-mmmm	2	4-0	simuluje uvozovací byty pole opcode: 00001 = 0x0F, 00010 = 0x0F38, 00011 = 0x0F3A
W	3	7	funkce záleží na poli opcode
vvvv	3	6-3	číslo registru v jedničkovém doplňku
L	3	2	délka vektoru: 0 = skalár nebo 128bitový vektor, 1 = 256bitový vektor
pp	3	1-0	plní funkci prefixu: 01 = 0x66, 10 = 0xF3, 11 = 0xF2

Tabulka 3: Bity 3bytového prefixu VEX

1.3.4 Opcode

Pole opcode slouží k identifikaci instrukce. Délka pole je buď 1, 2 nebo 3 byty. Pokud je instrukce kódována za pomoci prefixu VEX, má pole délku pouze jeden byte. Případné další byty jsou zakódovány v prefixu (viz tabulka 3). Pro délku pole 2 a 3 byty je jako první byte použito vždy číslo 0x0F.

1.3.5 ModR/M

Tento byte je používán u instrukcí, které používají operandy uložené v paměti. Na obrázku 1 je vidět, že se skládá celkem ze tří polí:

- Mod – společně s polem R/M kóduje buď číslo registrů, nebo adresní mód
- Reg/opcode – pole buď kóduje dodatečné bity pro opcode, nebo je v něm zakódované číslo registru
- R/M – specifikuje registr nebo adresní mód

1.3.6 SIB

Určité kombinace pole ModR/M si vyžádají zakódování dodatečných informací v následujícím bytu jménem SIB. Používá se k zakódování složitějšího adresního módu. Na obrázku 1 vidíme, že se skládá z následujících tří polí:

- Scale – udává násobící faktor
- Index – kóduje číslo násobeného registru
- Base – kóduje číslo bázového registru

1.3.7 Displacement a Immediate

Pole displacement je používáno u některých adresních módů pro vložení konstanty. Podobně je využíváno pole immediate, které se nachází až za prvně zmiňovaným polem. Používá se u instrukcí, které pracují s číselnými konstantami.

1.4 Debugger

Pro dynamickou analýzu kódu se běžně používá debugger. Debugger slouží k procházení kódu po jednotlivých instrukcích. V našem případě přichází v úvahu pouze procházení strojových instrukcí. Pro tuto činnost existuje podpora přímo v procesoru – výjimky a přerušení.

1.4.1 Výjimky a přerušení

Procesor obsahuje dva mechanismy pro přerušení běhu programu. Jedná se o mechanismus výjimek a mechanismus přerušení. Existuje několik druhů výjimek i přerušení. Některé z nich má možnost vygenerovat sám program, některé jsou generovány za určitých situací automaticky. Pokud je výjimka nebo přerušení vygenerováno, dojde k uložení stavu procesoru a následnému skoku na adresu v paměti, kde je uložena procedura určená k vykonání. Po jejím vykonání dojde k obnovení stavu procesoru a vrácení zpět na místo, kde k přerušení nebo výjimce došlo.

1.4.2 INT3

Pro zavolání přerušení nebo výjimky v programu slouží instrukce *INT*. Za touto instrukcí se udává číslo, které označuje číslo v tabulce vektorů přerušení. Po přeložení do strojového kódu se jedná o instrukci dlouhou dva byty. Existuje ale speciální verze této instrukce – *INT3*, která se překládá pouze jako jeden byte – 0xCC. To, že se jedná o jednobytovou instrukci, je důležité. K nastavení breakpointu dochází nahrazením bytu následné instrukce. Jelikož velikost nejmenší instrukce je jeden byte, musí být i stejně velká instrukce *INT3*. Pokud by byla větší, nedokázali bychom zastavit na menších instrukcích.

1.4.3 EFLAGS

Pro ukládání stavu procesoru slouží speciální registr, kterému se říká *EFLAGS*. Jedná se o registr velikosti čtyři byty, kde každý bit představuje tzv. flag. Každý z nich

má nějaký pevně stanovený význam. V 64bitovém režimu byl představen registr *RFLAGS*, což je vlastně rozšíření stávajícího registru do 64bitů.

Jeden z bitů, který je zajímavý pro debuggování, je trap flag. Tento bit slouží k požádání procesoru o vykonání jedné instrukce a následném zastavení. Po vykonání jedné instrukce je tento bit opět nenastaven.

1.5 PE formát

Systém Windows používá pro ukládání spustitelný souborů a knihoven formát PE. Zkratka PE [3] nám připomíná fakt, že formát není nijak závislý na použité architektuře. Jednou z jeho výhod je to, že jeho struktura uložení na disku odpovídá struktuře souboru načteného v paměti. Tato struktura je popsána v souboru *WINNT.H*. Zde se také nachází veškeré struktury, konstanty, makra a typy potřebné pro práci s formátem PE. Podrobné informace jsou k nalezení v dokumentu [**2**].

[3] Portable executable

2 Analýza problému

2.1 Podpora instrukcí

Jedno z kritérií kvality disassembleru je množství podporovaných instrukcí. Čím větší množství instrukcí dokáže rozkódovat, tím lépe. Již od začátku návrhu jsme se soustředili na to, abychom mohli podporovat ideálně všechny instrukce. Další věcí, na kterou jsme kladli důraz, byla rozšiřitelnost. Chtěli jsme, aby bylo v budoucnu jednoduše možné přidávat další instrukce, registry, možná i další architekturu.

Díky požadavku na rozšiřitelnost jsme okamžitě zavrhli napsání instrukcí přímo do zdrojových kódů. Další možností je jejich zapsání někam vedle do vlastního souboru. Pro přidávání a následné jednoduché zpracování jsme instrukce zapsali ve formátu XML[4]. Formát XML je standard pro práci s daty, je jednoduše čitelný a dobře se automaticky zpracovává.

Instrukce jsme získali z manuálu [**1**], který poskytuje firma Intel. Ten jsme následně zpracovali a převedli do námi požadovaného XML formátu. Bohužel jsme nikde nenašli volně dostupné zpracované instrukce ve tvaru, který by se nám hodil, takže je bylo potřeba ručně zpracovat. Jedna z výtek, kterou bych směřoval k manuálu, je jeho nekonzistence napříč celým manuálem, která velmi ztěžuje práci. Obzvlášť se tyto nesrovnalosti projeví při automatickém zpracování pomocí programu. Jako příklad bych uvedl značení instrukcí, které nemají žádné operandy. Pokud operandy instrukce má, tak se u instrukce uvádí znaky, podle kterých se v tabulce najdou použité operandy. Pokud instrukce operandy nemá, tak se u některých instrukcí vůbec tyto znaky neuvádějí a u některých instrukcí se znaky sice v tabulce uvedou, ale jednotlivé operandy jsou označeny jako nedostupné.

2.2 Načítání instrukcí

Další otázkou bylo, kdy nahrát instrukce do programu. Zde se naskytují pouze dvě možnosti. Buď nahrát instrukce do programu v době kompilace, nebo až za běhu programu.

První možností máme na mysli převod XML souboru pomocí nějakého skriptu přímo do jazyku, který je kompatibilní s jazykem, ve kterém program píšeme. Tady přicházelo v úvahu C++ možná C. Tabulky jednotlivých symbolů a instrukcí by zde

[4] Extensible markup language – rozšiřitelný značkovací jazyk

mohly být vloženy do tříd/struktur jako konstantní statické atributy. Nevýhodou by byla složitá transformace dat přímo do programovacího jazyka tak, aby šel správně přeložit. Navíc by musel být program při každé změně instrukcí znovu překládán. Na druhou stranu se nepředpokládá, že uživatel bude často měnit instrukční sadu. Všechny výše zmíněné nevýhody by byly vykoupeny rychlejším během programu oproti druhému řešení.

Druhou možností bylo nahrát data a vytvořit tabulky až za běhu aplikace. Zde není problém se změnou instrukcí, jelikož je XML soubor znova načítán při každém startu aplikace. Následně je v programu zpracován a uložen do používaných struktur. Tento přístup působí celkově jednodušeji. Druhou stránkou mince je opakované načítání a zpracovávání při startu aplikace. Opakovanému zpracovávání by se dalo vyhnout vytvořením binárního souboru s obsahem pomocných datových struktur. Tento soubor by se načítal při každém spuštění. Následně bychom potřebovali zjistit případnou změnu uloženého XML souboru, což by šlo například uložením jeho otisku a kontrolou při každém startu, zda se otisk změnil.

My jsme v programu zvolili načítání a zpracovávání souboru až za běhu programu, jelikož zpomalení programu nebylo až tak velké.

2.3 Program pro načítání

Jelikož jsme se rozhodli, že budeme načítat ze souborů ve formátu XML, potřebujeme umět nějak zpracovat načítaný soubor. Jednou z možností je napsat si vlastní parser. Tuto možnost jsme zavrhli, protože existuje spousta dobrých XML parserů, které tu práci udělají za nás. Jedna z výhod, kterou nám takové řešení poskytuje, je rychlost, jelikož se jedná o specializované programy na práci s XML. Výhodou napsání vlastního programu na zpracování XML je to, že bychom nemuseli udělat obecné řešení, ale pouze specializované na naše soubory, tudíž bychom možná mohli ušetřit nějaký čas, případně paměť.

Existují základní dva druhy parserů – SAX [5]a DOM[6]. My jsme se rozhodli pro DOM. Nevýhodou je samozřejmě větší paměťová náročnost oproti SAXu, ale lépe se

[5] Simple API for XML – jedná se o přístup proudového zpracování XML souborů

[6] Document Object Model – při zpracování XML souboru se vytvoří objektový model, který se následně zpracovává

s ním pracuje. Navíc budeme načítat soubory pouze jednou při startu, kde nebudeme extrémní výkon potřebovat.

Na zpracování jsme vybrali knihovnu Xerces-C++ [**3**]. Jedná o multiplatformní knihovnu, která poskytuje jak rozhraní SAX tak i DOM. Jedná se o velmi obsáhlou knihovnu. Jelikož jsme z počátku nevěděli, jaké všechny práce s XML souborem budeme provádět, zvolili jsme raději knihovnu, která obsahuje obrovské množství funkcí. Jedna z věcí, která jsme například chtěli udělat, ale už jsme ji nestihli, byla kontrola vloženého instrukčního formátu se schématem. Další výhodou je, že se jedná o živý projekt, takže se na něm stále pracuje a můžeme se na jeho kvalitu spolehnout. Knihovna je vydávána jako open-source pod licencí Apache License, verze 2.0 [**4**].

2.4 Zpracování instrukcí

Z načtených dat potřebujeme vytvořit strukturu, ve které budeme následně instrukce vyhledávat. Základním stavebním prvkem každé instrukce je opcode. Ten dříve stačil k identifikaci, ale s postupným přidáváním instrukcí se začaly další informace ukládat do prefixů a i do bytu následující opcode. Každá instrukce může mít jinak složený identifikátor. Společně s instrukcí je potřeba vyhodnotit její operandy. Ty můžou být jak na pevno dány instrukcí, tak i určeny až za běhu z okolních bitů.

Máme tedy více možností, jak uložit instrukce. Buď pomocí přímočarého řešení, kdy vygenerujeme všechny možné kombinace bitů a těm přiřadíme odpovídající instrukci a její operandy. Jednalo by se o velice rychlé řešení, kde by stačilo jen vyhledat dané byty a podle nich by nám byla vrácena příslušná instrukce i s operandy. Problém ale nastává s paměťovou náročností takového řešení. Předpokládejme zhruba, že průměrná délka instrukce jsou tři byty. Dále předpokládejme, že každý záznam má uložený alespoň ukazatel, jehož délka je v 32bitových systémech 4 byty. Pak dostaneme spotřebu paměti:

$4B \cdot 2^{28} = 2^{30}B = 1GB$

To je na program podle našich měřítek hodně a to jsme příklad hodně zjednodušili.

Uložená data je tedy potřeba rozdělit. Použili jsme rozdělení, které bylo nasnadě. Nejprve je potřeba uložit instrukci. Po nalezení instrukce najdeme případné operandy a máme kompletní instrukci v JSA. Otázkou zůstává, co použít jako identifikátor k vyhledávání instrukcí.

2.4.1 Zpracování instrukce

Náš program prošel několika evolucemi, kdy jsme postupně zkoušeli různé návrhy. V první verzi programu jsme instrukce ukládali jen podle bytů v poli opcode. Prvním problémem bylo, že jsme měli několik instrukcí se stejným identifikátorem. Problém byl vyřešen po načtení všech bytů, kdy se za pomoci načtených prefixů určilo, o jakou instrukci se jedná. Následující řešení nebylo špatné, ale bylo pomalé. S klasickými instrukcemi nebyl až takový problém, ten nastal až u instrukcí, které používají prefix VEX. Jelikož se snaží prefix VEX nahradit funkcionalitu jiných prefixů a pole opcode (viz kapitola 1), mělo mnoho instrukcí stejný identifikátor. Přesunuli jsme de facto problém vyhledaní instrukcí až na dobu, kdy vybíráme mezi více instrukcemi za pomoci prefixů. Protože jsme nepředpokládali, že bude na výběr z mnoha instrukcí, byly instrukce prohledávány lineárně, což vedlo právě u instrukcí s prefixem VEX ke zpomalení. Pokud bychom následné vybírání podle prefixů udělali nějak chytřeji než lineárním průchodem, určitě bychom dostali nějaké zrychlení. Rozhodli jsme se ale raději přepracovat návrh a jako identifikátor použít společně s polem opcode i prefixy.

V další verzi jsme tedy jako identifikátor použili jak opcode, tak i použité prefixy. Jelikož prefixy občas obsahují i nadbytečné informace, které nejsou k nalezení instrukce potřeba, rozhodli jsme se tyto informace vygenerovat všechny. Záhy jsme narazili na problém, který už byl rozebírán výše. Spotřeba paměti programu narostla na zhruba 800MB. Z toho důvodu jsme provedli další změnu a to, že se vyhledává jen podle položek pole opcode a položek z prefixů, které jsou skutečně potřeba. Díky tomu jsme docílili požadovaného zmenšení spotřeby paměti na zhruba 30MB.

2.4.2 Zpracování operandů

Při zpracovávání instrukcí jsme co nejvíce vycházeli z manuálu [**1**], kde je pro každou instrukci vyznačeno, o jaké typy operandů se jedná a kde se nachází jejich bitová podoba. My jsme tento návrh převzali. Každý operand je určen dvojicí typ a místo, kde se nachází. Lépe to nelze udělat, jelikož stejný typ může být u jedné instrukce uložen v bytu ModR/M a například u instrukcí používající VEX prefix právě v prefixu. Navíc, pokud chceme zachovat co největší univerzálnost, tak se adresování pomocí dvojice hodí.

V první verzi programu si každá instrukce pro každý operand pamatovala typ a byte, ve kterém se nachází. Po nalezení instrukce jsme se podívali na jednotlivé

operandy a zjistili, kolik ještě potřebují načíst bytů pro správné vyhodnocení operandů. Po načtení správného počtu bytů, při zápisu instrukce došlo ke spárování typů operandů podle dané dvojice. Důležité z hlediska pomalého běhu bylo, že přestože jsme už v průběhu načítání instrukcí věděli, které typy bude instrukce potřebovat, tak jsme ji to nechávali rozhodnout až za běhu při párování. To se ukázalo jako velká brzda v rychlosti běhu programu. Jednou z výhod bylo, že start aplikace byl rychlejší, jelikož jsme nemuseli tak moc zpracovávat operandy při načítání. Obecně jsme se ale snažili, abychom co nejvíce práce odvedli při načítání aplikace, abychom mohli co nejrychleji provádět překlad instrukcí za běhu. Je to logické vyústění toho, že aplikace startuje pouze jednou, kdežto překlad může probíhat již vícekrát.

Proto jsme ve druhé verzi programu zvolili přístup, kdy se již při startu aplikace páruje instrukce se správným operandem podle jeho zadané dvojice. To je činnost, kterou jsme v první verzi programu prováděli až za běhu, při nalezení instrukce. Samozřejmě se tím prodloužil čas načítání aplikace, ale zrychlili jsme běh, který je pro nás důležitější.

2.5 Hledání instrukcí

2.5.1 Velikost nejmenší jednotky

Instrukce jsme se rozhodli vyhledávat po jednom bytu. Zvolili jsme to jako rozumný kompromis. Bývá to nejmenší jednotka, která se načítá ze souborů a navíc velikost nejmenší instrukce je právě jeden byte. Při použití více bytů by byl princip víceméně stejný jako v jednobytové verzi, s tím rozdílem, že by vše bylo větší. Řešili by se často speciální případy, jelikož instrukce bývají členěny právě po bytech. Celkově to působí, že zrychlení, které bychom získali načítáním dvou a více bytů naráz, by bylo zaplaceno zpomalením v oblasti zpracování těchto bytů. Samozřejmě by se i nejspíš zvýšila paměťová náročnost, jelikož některé věci, které byly původně uloženy v jednom bytu, by se musely uložit do bytů dvou.

2.5.2 Zvolená struktura

K uložení a vyhledávání identifikátorů jednotlivých instrukcí jsme použili naši datovou strukturu. Jedná se o strom, kde každý uzel má několik následníků. Strom procházíme od kořene, dokud nenarazíme na uzel s instrukcí. Snažíme se ho procházet ideálně po bytech. Jak už bylo vysvětleno v kapitole 2.3, nedá se toho vždy docílit,

pokud chceme zároveň zachovat malou paměťovou náročnost. Proto je občas nutné vyhledávat i po skupinkách bitů. Jednu z optimalizací, kterou jsme provedli, bylo to, že alespoň v první hladině stromu, jsou použity celé byty. Jelikož mnoho instrukcí může být identifikováno pouze pomocí celých bytů, nechtěli jsme je tedy zpomalovat delším vyhledáváním. Navíc tato optimalizace v našem případě nezvětšila o tolik paměťovou náročnost programu, a proto jsme ji použili.

Jinou možností by bylo používat hashovací funkci. Když se ale nad tím zamyslíme, tak strom v našem případě představuje víceúrovňové hashování. V každém uzlu se totiž přímo vydáme do potomka, který má hodnotu právě příchozího bytu, případně nějakých jeho bitů. Pokud tedy uvažujeme načítání po jednom bytu, tak už by nám ani moc nepomohlo.

2.5.3 Zotavení z chyb

Jeden problém nastane v případě, že instrukce není nalezena. To se může stát při hledání nějaké instrukce, která není načtena ze souboru s instrukcemi. V tomto případě se chceme nějak zotavit z chyby a pokračovat správně dál.

Předpokládejme tedy, že jsme hledali nějakou posloupnost bytů a následně jsme již došli do stavu, kdy neexistuje žádný následník uzlu, ve kterém se nacházíme. Prohlásíme tedy počáteční byte za chybný, vrátíme se a pokusíme se začít hledání z dalšího bytu za chybným bytem. Takto pokračujeme, dokud ve stromě nenalezneme správnou instrukci. Bohužel po nalezení chyby už nemůžeme nikdy s jistotou říct, že překládaný kód je ten původní. Může totiž nastat situace, že díky chybným bytům se můžeme dostat ve stromu k jiné instrukci a dále, pokud všechny následující byty budou vést ve stromě k instrukcím, tak jsme dostali kód, který nejspíš nebyl ten původní. Jednou možností je, pokud máme k dispozici debugger, si počkat, jak si s chybnými byty poradí procesor a následně se synchronizovat s ním.

Náš přístup zotavení z chyb by šel zrychlit, pokud bychom do stromu zavedli zpětné hrany a vytvořili z něho konečný automat. Jelikož předpokládáme, že se chybné byty budou objevovat velice výjimečně, použijeme pouze strom nikoliv automat.

2.6 Výpis instrukcí

Po nalezení správné instrukce je potřeba ji někam uložit. Jelikož jedna instrukce odpovídá několika bytům, pro velké programy se tedy jedná o mnoho dat,

které už není dobré ukládat v celku do paměti. Proto převedené instrukce uložíme do souboru a následně budeme načítat jen potřebné části kódu.

Nyní je otázkou, kde začít a kde skončit s načítáním. Víme, že instrukce se vykonávají v pořadí, v jakém jsou uvedeny ve zdrojovém kódu. Pro změnu pořadí vykonávaných instrukcí slouží v JSA například podmíněné skoky, nepodmíněné skoky nebo příkaz CALL. Důležité jsou samozřejmě i cílové adresy, na které tyto instrukce skočí. Bloky tedy budou tvořit instrukce, které se nacházejí mezi po sobě jdoucími instrukcemi skoku nebo jejich cílovými adresami. Ke každé hraniční adrese bloku si uložíme její reálnou adresu v uloženém souboru. Pro následné vypsání bloku pak již stačí zjistit, kde se nachází v souboru, a vypsat ho.

2.7 Rozpoznávání programových struktur

Jelikož JSA není tak jednoduchý na analýzu jako vyšší programovací jazyky, zabudovali jsme i podporu nalezení podmíněných příkazů a cyklů a volání funkcí pomocí funkce CALL. U podmíněných příkazů zobrazíme podmínku, která větvení způsobila a její cílový skok. U cyklů zobrazíme podmínku, která je nutná pro další cyklení, a začátek cyklu. Jelikož náš program má být pouze disassembler, nechtěli jsme se nějak moc ubírat tímto směrem. Přeci jenom je to již práce pro tzv. decompilery[7]. Proto náš program bude obsahovat jen velmi základní analýzu těchto struktur.

Pro potřeby analýzy si vytvoříme graf toku řízení. Jedná se o orientovaný graf, kde jednotlivé uzly jsou bloky. Bloky použijeme stejné, jako v předchozí kapitole 2.6. Jako hrany mezi bloky použijeme právě instrukce skoku. Pokud instrukce slouží ke skoku z bloku A do bloku B, vytvoříme tímto směrem orientovanou hranu. V takto vytvořeném grafu budeme vyhledávat programové struktury. Ze vstupního souboru zjistíme startovní adresu a následně budeme procházet graf pomocí DFS[8]. Pokud nalezneme cyklus, víme, že se jedná o cyklus i v instrukcích. K úniku z cyklu můžeme projít jednotlivé vrcholy, které se nacházejí v cyklu a zjistit, zda neobsahují ještě jinou odchozí hranu. Pokud ano, tak by to mohla být ta úniková. Pro nalezení podmíněných příkazů si budeme při průchodu grafem zjišťovat, z kterých vrcholů vede více hran, a

[7] Jedná se o program, který se snaží převést strojový kód zpět na zdrojový kód, ve kterém byl naprogramován

[8] Depth-First Search – prohládávání do hloubky

ty vrcholy použijeme. Na konec bloků, které odpovídají těmto vrcholům, přidáme podmínku. Podle toho, o jaký druh skoku se jedná, poznáme, co se testovalo. Nyní už stačí nalézt pouze poslední instrukci, která změnila hodnotu registru *FLAGS* a použít její argumenty v podmínce. Většinou se jedná o instrukci *CMP* nebo *TEST*, ale může se jednat i o některé aritmetické operace. My v programu rozpoznáváme porovnání pouze pomocí zmíněných instrukcí. Pro vypisování volání funkcí si akorát pamatujeme cílové adresy volání instrukce CALL. Větve grafu, které odpovídají těmto adresám, neprohledáváme, ale uložíme pro pozdější zpracování. Po zpracování hlavní funkce budeme výše popsanou analýzu provádět postupně pro všechny uložené funkce.

2.8 Debugging

Všechny části disassembleru, které jsme zmínili do této kapitoly, byly nezávislé na OS[9]. Jelikož je podpora pro debuggování zabudována přímo v procesoru a moderní OS nás nenechají k němu přistupovat přímo, musíme využít služeb OS. My jsme se rozhodli psát program pro OS rodiny Windows.

Jednou ze základních funkcí, kterou použijeme je funkce *WaitForDebugEvent*. Funkce se stará o odchytávání všech důležitých událostí spojených s debuggováním. Podporu pro přidávání a odebírání breakpointů si ale budeme muset vytvořit sami. Pokud známe adresu, na kterou se má breakpoint uložit, načteme obsah paměti na daném místě a uložíme si starý obsah bytu do mapy, kde jako klíč použijeme adresu. Pak místo starého bytu vložíme byte 0xCC, což je instrukce *INT3*, která vyvolá výjimku. Výše zmíněná funkce umí odchytávat i výjimky. Pokud dojde na výjimku, zkontrolujeme, zda se jedná o výjimku, kterou vyvolává breakpoint. Pokud ano, tak v mapě nalezneme původní byte a vrátíme ho na své místo. Poté můžeme normálně pokračovat ve vykonávání programu. Pokud bychom chtěli krokovat vykonávání programu, je potřeba nastavit trap flag. Ten je uložen v registru *EFLAGS*. Poté co tento příznak nastavíme, tak se procesor po další vykonané instrukci zastaví a funkce *WaitForDebugEvent* znova odchytí výjimku.

Jeden problémů, může nastat při vkládání breakpointu. Pokud adresa, na kterou vkládáme breakpoint, je prvním bytem instrukce, tak je vše v pořádku. V opačném případě nikoliv. Jeden z lepších případů je ten, kdy jsme se spletli o instrukci, tedy že

[9] Operační Systém

máme první byte, ale jiné instrukce než jsme chtěli. To nám sice zastaví debugger na špatné instrukci, ale pořád se nic neděje. Horší případ nastane, pokud jsme breakpoint vložili někam špatně. Pak jsme úplně změnili instrukci, v lepším případě pouze data, s kterými pracuje. Následně nejspíše dojde k tomu, že debuggovaný program skončí s chybou. Případ se špatně vypočtenou adresou breakpointu nastane tehdy, když špatně přeložíme strojový kód do JSA. Uživatel bude chtít vložit breakpoint na adresu, kde začíná domnělá instrukce a přitom ji vloží úplně jinam.

My v našem programu velmi optimisticky předpokládáme, že jsme všechny instrukce správně přeložili a nebude tedy docházet k těmto chybám. Bohužel daný přístup půjde těžko vylepšit. Jednou z možností by bylo analyzovat kód a zjistit tok instrukcí. Pokud bychom se přiblížili k našemu breakpointu, nechali bychom procesor krokovat jednotlivé strojové instrukce a tím bychom si mohli zkontrolovat, správné přeložení instrukcí a i následné uložení breakpointu.

2.9 Uživatelské rozhraní

Od začátku jsme nebyli pevně rozhodnuti, jaké vytvoříme uživatelské rozhraní. Rozhodovali jsme se mezi konzolovou a okenní aplikací. V dnešní době jsou již samozřejmě standardem okenní aplikace, ale i dnes se najde spousta aplikací, které takové rozhraní nemají (viz programy v systému UNIX). Jelikož naším hlavním cílem bylo vytvořit co nejlepší disassembler, rozhodli jsme se pro tu jednodušší variantu – vytvoření konzolové aplikace.

Při vytváření návrhu jsme si dali za cíl, aby případné pozdější rozšíření nebo výměna uživatelského rozhraní byla co nejjednodušší. Proto jsme použili lehce upravený architektonický vzor MVC [10]. Upravili jsme ho tak, že jsme spojili dohromady části View a Controller. Pro tuto část jsme vytvořili abstraktní třídu, kterou musí uživatelské rozhraní implementovat. Spojili jsme tak ty dvě části, abychom podpořili provázanost mezi těmito dvěma částmi. Je tedy nyní nutnost implementovat společně obě dvě části.

Nyní odstupem času usuzujeme, že se jednalo o špatnou volbu a bylo by lepší použít rozděleného vzoru MVC. Provázanost mezi View a Controllerem není až tak důležitá. V některých případech nám dokonce ovládání pomocí konzole a náhled zobrazený například v okně přijde jako dobrý nápad.

[10] Model-View-Controller

2.10 Ostatní disassemblery

2.10.1 IDA

IDA [**5**] je nejlepší disassembler, který se dá momentálně sehnat. Samozřejmě, že je si toho výrobce vědom a nechá si za to dobře zaplatit. Na druhou stranu za to poskytuje výbornou technickou podporu. Volně k dispozici je starší verze 5.0, která i tak umí hodně. Základní vlastností je převod do JSA. Komerční verze dokáže načíst většinu myslitelných spustitelných formátu a binárních souborů pro všechna možná zařízení, nejen klasické počítače, verze zdarma umožňuje načítání pouze 32bitových aplikací ve formátu PE. To, v čem IDA skutečně na rozdíl od ostatních programů vyniká, je analýza kódu. Dokáže například zobrazit hodnoty řetězců, rozpozná systémová volání, umí zobrazit připojené knihovny. Samozřejmostí je možnost spuštění debuggeru nad přeloženým kódem. Rozšíření programu je možné pomocí pluginů, které se píší v jazyku Python.

2.10.2 OllyDbg

OllyDbg [**6**] je, jak už název napovídá, spíše debugger. Dovoluje načtené programy převést do JSA. Nepodporuje až tak širokou škálu formátů jako IDA. V čem se mu ale blíží, je jeho statická analýza. Dokáže zobrazit všechny možné informace, načtené knihovny, rozeznává řetězce a dokáže rozeznat i volání standardních API a funkcí z jazyka C. Jedna z užitečných funkcí je i editace přeloženého kódu jeho následném spuštění. Jednou z nevýhod je podpora pouze pro instrukční sadu x86. Ač je program poskytován jako shareware, tak je úplně zdarma bez žádných dalších omezení.

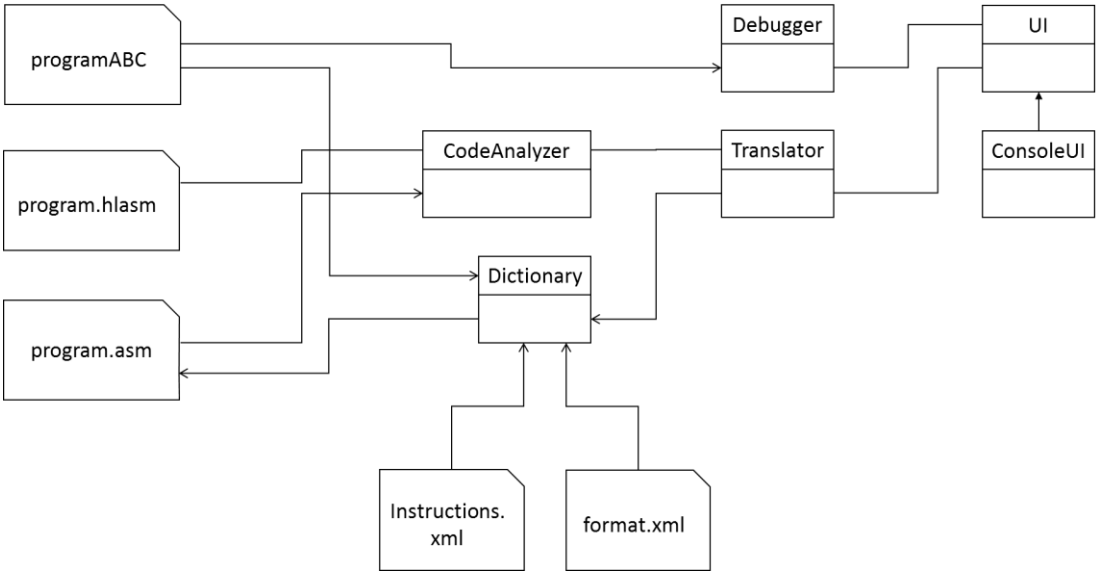
2.10.3 Ndisasm

Ndisasm [**7**] je jednoduchý disassembler dodávaný zdarma společně s assemblerem NASM. Jedná se o multiplatformní open source program, který není interaktivní a běží v konzoli. Jediné co umí, je vypsat JSA pro daný procesor a architekturu. Nemá k dispozici žádné další speciální nástroje, ať třeba debugger nebo analýzu kódu. Na rozdíl od výše zmíněných ale umí překládat 64bitový kód.

3 Implementace

3.1 Architektura aplikace

Na obrázku 3 je nakreslená architektura celé aplikace.



Obrázek 3: Architektura aplikace

Středobodem aplikace je třída *UI*. Jedná se o abstraktní třídu, od které je poděděna třída *ConsoleUI*. Třída komunikuje s dvěma třídami, s třídou *Translator* a s třídou *Debugger*. Třída *Translator* funguje jako fasáda a jsou v ní umístěny další dvě důležitější třídy, *Dictionary* a *CodeAnalyzer*.

Třída *Dictionary* načítá svoje informace z dvou konfiguračních souborů v XML. Až dostane jméno programu určeného pro překlad od třídy *UI*, přeloží ho a výsledek uloží do dalšího souboru s názvem *program.asm*.

Třída *CodaAnalyzer* se stará o práci s přeloženým kódem. Na požádání pošle přeložený kód nebo případně provede analýzu tohoto kódu a pošle její výsledek. Tento výsledek je uložen v souboru *program.hlasm*.

Třída *Debugger* se stará o práci s debuggerem. Pokud dostane příkaz od *UI*, spustí překládaný program a posílá signály o jednotlivých událostech. Z *UI* přicházejí další povely nutné k ovládání debuggeru.

3.2 UI

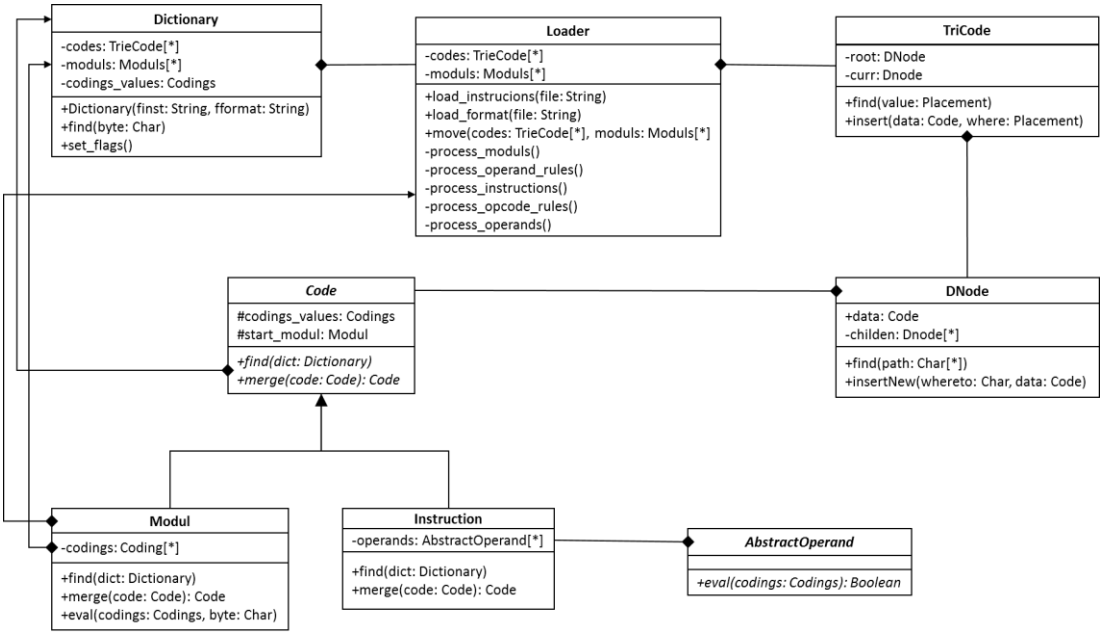
Abstraktní třída *UI* má za úkol abstrahovat uživatelské rozhraní. Od začátku byla navržena tak, aby bylo jednoduché přidávat její potomky a vyměnit tak celé prostředí. Pro zobrazovací část *UI* obsahuje virtuální funkce, které je potřeba předefinovat. Jelikož v zobrazovací části často dochází k vyvolání událostí asynchronně, je každá tato funkce používána jako callback, který se zavolá při dané události. Pomocí tohoto mechanismu je ostatně implementováno mnoho uživatelských rozhraní. Pro ovládací část jsou k dispozici objekty tříd, které je možno použít k ovládání.

3.3 ConsoleUI

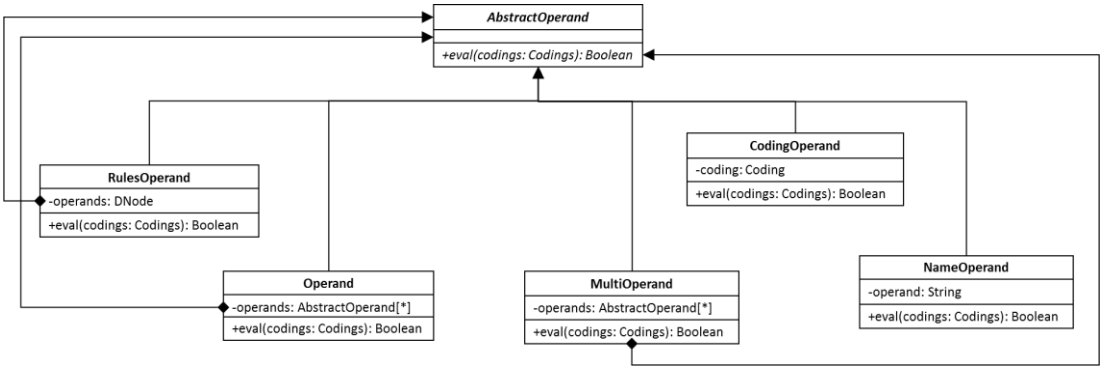
Abychom mohli vytvořit konzolové uživatelské rozhraní, vytvořili jsme potomka abstraktní třídy *UI* a pojmenovali ho *ConsoleUI*. Tato třída implementuje potřebné funkce pro zobrazování. Výsledky jsou jednoduše vypsány do konzole. Jelikož jsme chtěli vytvořit pohodlné ovládání pomocí klávesnice, bylo potřeba nějak odchytávat jednotlivé stisky kláves. Kvůli tomu jsme si vytvořili smyčky, jejichž úkolem nebylo nic jiného, než poslouchat stisky klávesnice a následně provést správnou akci. K odposlouchávání ve smyčce jsme použili standardní funkci *GetAsyncKeyState*. Dále bylo potřeba vždy alespoň na chvíli smyčku uspat, abychom dali šanci i ostatním vláknům něco provést. Z počátku jsme měli vytvořené ovládání pomocí F kláves. Jelikož jsou tyto klávesy v konzoli již rezervovány pro jiné akce, docházelo často ke kolizím. Proto jsme zvolili ovládání pomocí čísel.

3.4 Dictionary

Třída *Dictionary* se stará o překlad ze strojových instrukcí do JSA. Jelikož je možné třídu používat i samostatně, nejen společně s aplikací, rozhodli jsme se z ní vytvořit samostatnou statickou knihovnu. Na obrázcích 4 a 5 je zobrazena její architektura.



Obrázek 4: architektura třídy Dictionary

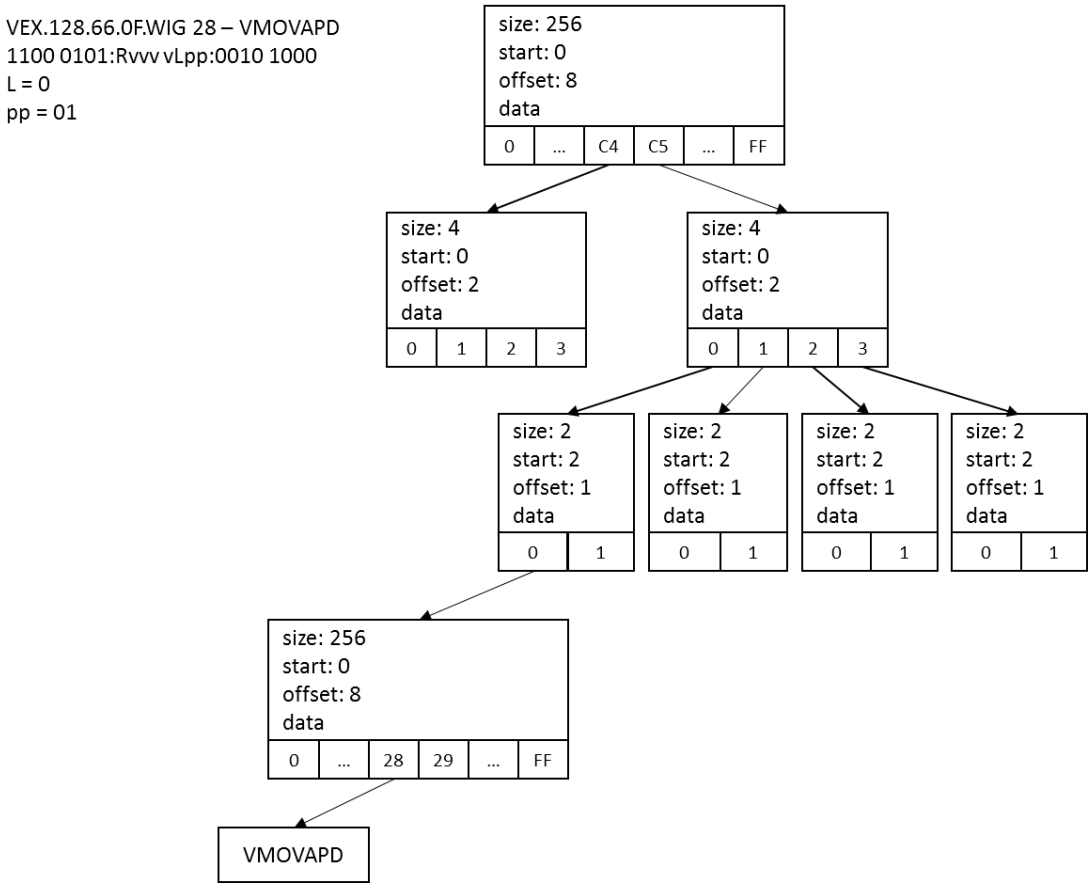


Obrázek 5: Architektura třídy AbstractOperand

3.4.1 Třída Trie

Třída *Trie* je implementace naší vyhledávací struktury určené k vyhledání instrukcí. Jedná se o šablonovanou třídu, jejíž specializace na abstraktní třídu *Code* se nazývá *TriCode*.

Třída je implementována jako vyhledávací strom. Strom má jeden kořen. Uzel ve stromě může mít následníky a jejich počet je mocninou čísla dvě. Proměnný počet následníků je nutný z důvodu ukládání i částí bytu. Proto si každý uzel, kromě počtu svých potomků, pamatuje pozici v bytu, na které má začít hledat a počet bitů. Bity z tohoto intervalu jsou použity jako index k vyhledávání potomků. Samozřejmě každý uzel obsahuje šablonový typ data, v našem případě ukazatel na třídu *Code*. Takto vytvořená struktura slouží k tomu, abychom byli schopni zakódovat celý identifikátor instrukce, který může obsahovat i požadavek na prefixy. Na obrázku 6 je ukázán příklad uložení instrukce ve stromu.



Obrázek 6: Příklad uložení instrukce

Je jednoduché nahlédnout, že vyhledání jednoho bytu má konstantní časovou složitost. V nejhorším případě totiž budeme muset projít všech osm bitů, což odpovídá osmi vrstvám stromu. Při vytváření nejprve najdeme místo, kam uzel patří, případně vytvoříme cestu k novému uzlu. Nalezení cesty pak trvá O(B) kroků, kde B je počet

bytů, ze kterých se skládá identifikátor. Do třídy *TrieCode* ukládáme potomky třídy *Code*, což jsou třídy *Instruction* a *Modul*.

3.4.2 Abstraktní třída Code

Třída *Code* je abstraktní třída, která slouží k tomu, abychom do stromu mohli ukládat jak moduly, tak instrukce. Moduly říkáme jednotlivým polím, ze kterých se skládá instrukce. Moduly se následně ještě dělí na menší části a těm říkáme kódování (v programu je použitý anglický název *coding*). Moduly jsou uloženy ale pouze ty, které se nacházejí před polem opcode. Jedná se totiž o prefixy.

Každý z prefixů začíná nějakou pevně danou posloupností bitů. Ty použijeme jako identifikátor k uložení do stromu. Nikdy by se nemělo stát, že budou mít modul a instrukce stejný identifikátor. V tom případě zahlásí program při načítání, o kterou instrukci a modul šlo a nechá ve stromu to, co bylo vloženo jako první. Kvůli tomu, obsahuje třída metodu *merge*. Není samozřejmě problém s tím, když identifikátor instrukce obsahuje nějaký modul. To je dokonce častý jev u instrukcí s prefixem VEX.

Třída *Code* obsahuje virtuální metodu *find*, která je zavolána v případě, že je nalezen uzel stromu s nějakými daty. Další příchozí byty nejsou používány k vyhledávání ve stromě, ale jsou přesměrovány rovnou funkci *find* posledně nalezenému objektu typu *Code*. Každý z potomků má implementované chování, jak nakládá s příchozími byty.

Atribut *start_modul* slouží k uložení odkazu na první modul, který potřebuje instrukce ke svému vyhodnocení. Dalším atributem je *codings_values*. Jedná se o množinu s názvy a hodnotami kódování.

3.4.3 Třída Modul

Třída *Modul* slouží k uložení jednotlivých polí strojových instrukcí. Každý modul obsahuje několik kódování, ty jsou uložena v poli. Kódování se skládá z názvu, délky a případné hodnoty.

Ve funkci *find* je implementována logika pro vyhodnocení hodnot jednotlivých kódování. S každým příchozím bytem se vyhodnotí příslušná kódování a uloží se do množiny s momentálními hodnotami kódování. Funkce *eval* je volána při vyhodnocování operandů. Funkce pomocí nového bytu vyhodnotí modul a uloží jeho kódování.

3.4.4 Třída Instruction

Třída *Instuction* je použita k uložení instrukcí. Proto obsahuje atribut *operands*, který ukládá ukazatele na operandy. Funkce *find* slouží k vyhodnocování instrukce. K vyhodnocování se používá následující algoritmus:

1. Zavolej funkci *eval* na modulu, který je potřeba vyhodnotit a vyhodnoť ho
2. Pokud není modul vyhodnocený, skonči
3. Pro každý operand zavolej funkci *eval*, která vyhodnotí operand a vrátí další potřebný modul
4. Pokud potřebujeme k vyhodnocování ještě další moduly, tak skonči
5. Vypiš instrukci a její operandy, vyčisti všechny pomocné struktury a skonči

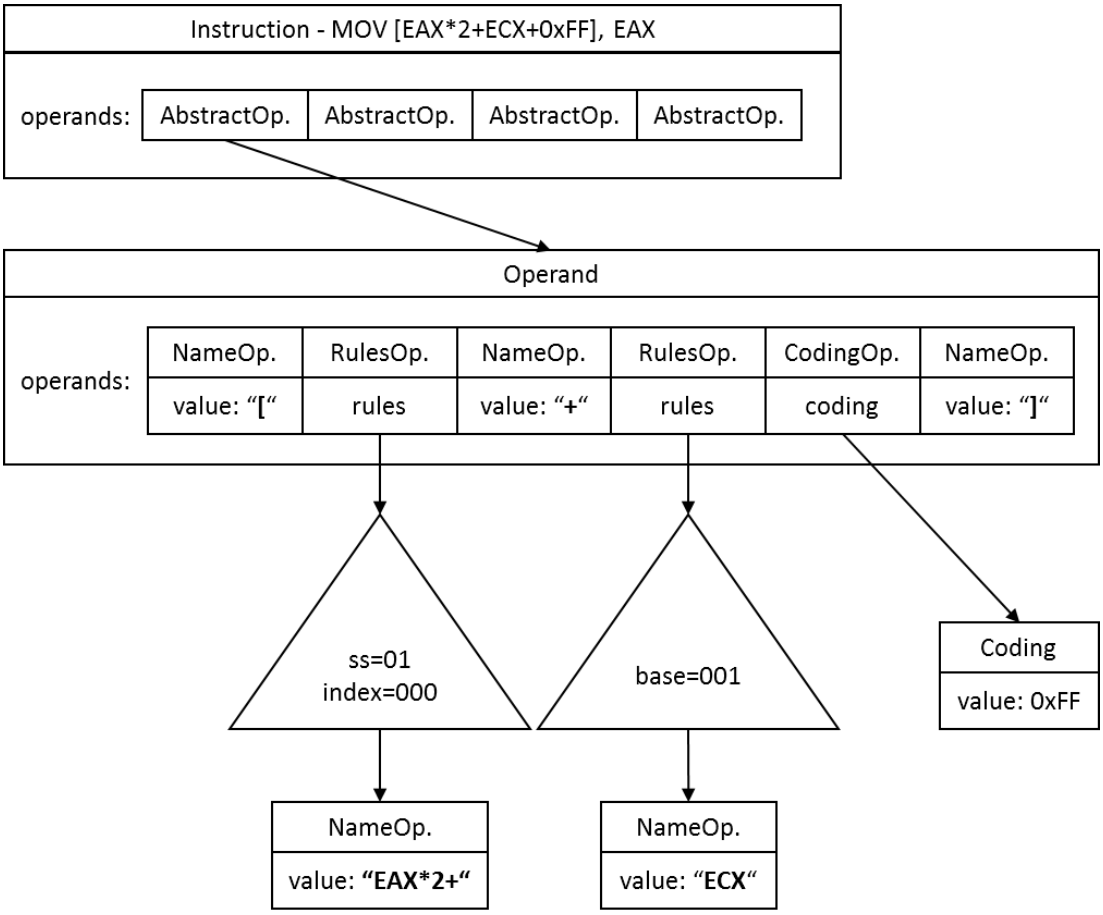
3.4.5 Abstraktní třída AbstractOperand

Abstraktní třída *AbstractOperand* slouží k uložení operandu. Jelikož operand může být velmi složitý a my jsme chtěli návrh operandu co nejvíce zobecnit, rozhodli jsme se pro návrh operandu použít návrhový vzor kompozit (anglicky composite). Každý z potomků obsahuje virtuální metodu eval, která se po zavolání pokusí vyhodnotit operand. Pokud neuspěje, tak vrátí nazpět jméno modulu a kódování, které potřebuje ke svému úspěšnému vyhodnocení. Existuje několik druhů operandů:

- *Operand* – skládá se z několika za sebou jdoucích operandů. Po vyhodnocení se vypíší přesně v tom pořadí, v jakém jsou uloženy. Používá se k vytvoření kombinací více operandů
- *RulesOperand* – obsahuje šablonovou třídu *Trie*, ve které jsou uloženy operandy. Pro vyhodnocení je potřeba projít celým stromem až k listu, kde se nachází cílový operand. Používá se, pokud se více operandů liší v hodnotě stejného kódování
- *MultiOperand* – obsahuje lineárně uložené operandy, z kterých je potřeba vybrat jeden podle daného kódování. Používá se v případech, kdy mají operandy stejné kódování i hodnoty a liší pouze v použití prefixů
- *CodingOperand* – slouží k okopírování hodnoty, která je přímo uložena jako hodnota daného kódování. Ta se přeloží jako a vrátí se její hodnota.

- *NameOperand* – slouží jako operand, který přímo obsahuje řetězec.

Na obrázku 7 je ukázán příklad složení prvního operandu.



Obrázek 7: Příklad skládání operandu

3.4.6 Třída Dictionary

Třída *Dictinary* je hlavní třída, která se stará o překlad bytů do JSA. Obsahuje tedy pomocné struktury *codes*, což jsou třídy typu *TrieCode*, které obsahují stromy s instrukcemi pro každý mód procesoru. Další strukturou je pole modulů, kde jsou uloženy moduly, které se nacházejí za polem opcode.

Třída obsahuje tři důležité metody. Metoda *set_flags*, slouží k nastavení módu procesoru. Pomocí parametru se může přepínat mezi jednotlivými módy. Druhá metoda je konstruktor, kde se zavolá inicializace, o kterou se stará třída *Loader*. Po ukončení načítání, za předpokladu, že všechno proběhlo správně, se přesunou do třídy *Dictionary* všechny potřebné struktury. Metoda, která se stará o předklad bytů se jmenuje *find*. Pokud existuje uložený objekt typu Code, tak mu předá řízení. Pokud,

žádný takový objekt nemá, tak se posune ve stromě na dalšího potomka. Pokud žádný neexistuje, znamená to, že jsme žádnou instrukci nenašli a musíme se zkusit vrátit.

3.4.7 Třída Loader

Třída *Loader* slouží k načtení souborů s instrukcemi a jejich formátem z XML souborů do námi požadovaných struktur. Pokud dojde ke správnému načtení, přesuneme používané struktury do třídy *Dictionary*. K tomu slouží metoda *move*. O načítání z XML souborů se starají dvě metody *load_instructions* a *load_format*. Ty postupně zavolají funkce na zpracování instrukcí, operandů a pravidel.

Jako první zpracováváme moduly. Do každého modulu vložíme příslušné kódování. Pokud se jedná o moduly před modulem opcode (prefixy), tak je vkládáme rovnou do stromu s instrukcemi, jinak moduly uložíme do pole. Dále zpracováváme pravidla pro operandy. Ty si ukládáme do třídy *OperandFactory*, která se stará o vytváření operandů. Vychází z návrhového vzor továrna (anglicky factory). Operandy se zde ukládají do mapy, kde klíčem jsou trojice *<mód, jméno pravidla, jméno operandu>* a hodnoty jsou operandy. Při vkládání operandů se stejným klíčem dojde k jejich sloučení, pokud je to možné. Poslední věci určené k načtení z *instructions_format.xml* jsou pravidla pro opcode. Zde si načteme do připravených tabulek zvlášť pravidla pro prefixy a pravidla pro postfixy.

Z *instructions.xml* si nejprve načteme operandy opět do struktury *OperandFactory*. Pomocí jmen totiž určíme jméno pravidla pro operand. Následně procházíme a zpracováváme instrukci, tak jak jsou napsány ve vstupním souboru. Jediné, co ještě potřebujeme u instrukcí zpracovat, jsou operandy a opcode. Z pravidel pro prefixy a postfixy získáme ještě dodatečné požadavky na jednotlivé bity. V postfixových pravidlech se může objevit speciální pravidlo s operátorem +, jehož hodnotu zpracujeme a přičteme k poslednímu bytu pole opcode. Takto si vygenerujeme všechny možnosti polí opcode. Pro jednu instrukci v XML souboru to může být i několik instrukcí. Je potřeba samozřejmě zpracovávat zvlášť instrukce pro jednotlivé módy procesoru, jelikož operandy se liší v závislosti na módu procesoru. Poslední nutnou úpravou je vytvořit a správně připojit k instrukci její operandy. Pak už zbývá jen pro všechny vygenerované opcode vytvořit instrukci a vložit ji do *CodeTrie*.

3.5 Soubor instructions.xml

Soubor *instructions.xml* slouží k popisu instrukcí a operandů, které instrukce používají. Jeho formát je velmi podobný tomu, ve kterém jsou instrukce a operandy napsané v manuálu [**1**]. Obrázek 8 ukazuje příklad definování instrukce v souboru. Pro definování instrukcí je potřeba dodržet strukturu zobrazenou na obrázku. Přesné schéma tohoto souboru je napsáno v příloze B.1.

```
<?xml version="1.0" encoding="UTF-8" ?>
<instructions>
  <operands>
    <operand name="NP" op1="NA" op2="NA" op3="NA" op4="NA" />
    <operand name="I" op1="imm" op2="NA" op3="NA" op4="NA" />
    <operand name="RM" op1="ModRM:reg" op2="ModRM:r/m" op3="NA" op4="NA"  />
    <operand name="RVM" op1="ModRM:reg" op2="VEX.vvvv" op3="ModRM:r/m" op4="NA"  />
  </operands>
  <instruction name="AAD" >
    <formats>
      <format form="AAD" opcode="D5 0A" op_enc="NP" bit64="0" bit32="1" bit16="1" desc="ASCII
adjust AX before division." />
      <format form="AAD imm8" opcode="D5 /ib" op_enc="I" bit64="0" bit32="1" bit16="1"
desc="Adjust AX before division to number base imm8." />
    </formats>
  </instruction>
  <instruction name="ADDSD/VADDSD" >
    <formats>
      <format form="ADDSD xmm, xmm/m64" opcode="F2 0F 58 /r" op_enc="RM" bit64="1" bit32="1"
bit16="0" desc="SSE2 Add the low double-precision floating-point value fromxmm/m64to xmm." />
      <format form="VADDSD xmm, xmm, xmm/m64" opcode=".VEX.NDS.LIG.F2.0F.WIG 58 /r"
op_enc="RVM" bit64="1" bit32="1" bit16="0" desc="AVX Add the low double-precision floating-
point value from xmm/mem to xmm and store the result in xmm." />
    </formats>
  </instruction>
</instructions>
```

Obrázek 8: Příklad definování isntrukcí

Mezi elementy *operands* je potřeba nadefinovat používané operandy. Element *operand* musí mít všechny zobrazené atributy. Atribut jméno slouží ke spojení operandu s instrukcí. U instrukce v atributu *op_enc* se napíše jméno použitého operandu. Pokud operand s uvedeným jménem není nalezen, instrukce je ignorována.

Dalšími atributy jsou jednotlivé operandy. Počet operandů je nastaven v aplikaci konstantou na čtyři. Pokud nechceme, aby operand byl užíván, použijeme řetězec *NA*. Pokud chceme zakódovat do opcode i operand, použijeme řetězec *opcode*. Pomocí názvu uvedeném v operandu se odkazujeme na jméno pravidla, zapsaném v souboru *format.xml*.

Formát instrukce je zapsán jako atributy v elementu *format*. Není potřeba pro každou instrukci psát nový element *instruction* ani jeho atribut *name*. My jsme ho používali v dřívějších dobách a nyní jsme je tam pro přehlednost nechali. Všechny

atributy kromě *desc*, který by měl obsahovat popis instrukce, jsou povinné. V atributu *form* se udává formát instrukce. Úplně stejně se bude vypisovat v JSA. Operandy ve formátu slouží jako jeden z klíčů k vyhledání operandu a budou při vypisování nahrazeny. Instrukce i operandy by měli být odděleny mezerami, abychom mohli správně rozdělit řetězec a najít jeho části.

V atributu *opcode* se nachází identifikátor instrukce, podle kterého bude nalezena. Identifikátor se může skládat ze tří částí. První část tvoří prefixy. Začínají tečkou a i jako oddělovač jim slouží tečka. Tyto prefixy by měli odpovídat jménům pravidel pro opcode, uložených v druhém XML souboru. Za prefixy a mezerou, jako oddělovačem, se nachází jediná povinná část atributu. Musí se zde nacházet posloupnost bytů zapsaných v šestnáctkové soustavě. Jednotlivé byty musí být odděleny mezerami. Za posloupností bytů se může nacházet další část, které říkáme postfixy. Jedná se opět o názvy pravidel pro opcode, s tím rozdílem, že jsou aplikována až po posloupnosti bytů. Zapisují se s uvozovacím lomítkem a oddělují se od sebe klasicky pomocí mezer. Tyto pravidla smí adresovat pouze moduly, které se nacházejí za polem opcode. Pravidla pro postfixy v jedné instrukci musí adresovat pouze stejný modul. Pokud některý z prefixů respektive postfixů obsahuje jméno neexistujícího pravidla, je daný prefix respektive postfix ignorován.

Následují atributy *op_enc*, které udává jméno kódování operandu a tři atributy s názvem *bit16*, *bit32*, *bit64*, které říkají, v jakých módech je instrukce validní. Nula znamená, že instrukce v daném módu neexistuje, jednička znamená opačnou funkčnost.

3.6 Soubor instruction_format.xml

Soubor *instruction_format.xml* slouží k nadefinování struktury a pravidel instrukcí. Jeho první část tvoří definice formátu strojové instrukce. V druhé části jsou definice pravidel pro operandy a třetí část se skládá z definic pravidel pro opcode. Na obrázku 9 je zobrazena požadovaná XML struktura. Přesné schéma souboru je napsáno v příloze B.2.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<instruction_format>
  <format>
  </format>
  <operand_rules>
  </operand_rules>
  <opcode_rules>
  </opcode_rules>
</instruction_format>
```

Obrázek 9: Struktura souboru *instruction_format.xml*

3.6.1 Definice formátu instrukce

Definice formátu instrukce se uzavírá mezi elementy s názvem *format*. Tato část slouží k definici polí ve strojové instrukci. Každé pole můžeme nějak pojmenovat. Taktéž můžeme přiřadit jméno i jednotlivým skupinám bitů. Je důležité definovat formát obecně s ohledem na všechny instrukce. V našem souboru jednotlivým polím ve strojové instrukci odpovídají moduly a jednotlivým skupinám bitů odpovídají kódování. Kódování slouží k tomu, abychom se na jejich hodnoty mohli odkazovat v používaných pravidlech.

Jediným povinným modulem, který se určitě musí v definici objevit je modul se jménem *opcode*. Ostatní moduly jsou již volitelné. Modulům, které se budou nacházet před modulem opcode říkáme prefixy, modulům za modulem opcode říkáme postfixy. Na obrázku 10 je ukázán příklad definice modulů.

```
<modul name="vex">
  <coding>
    <byte code="C5" size="1"/>
    <bits name="vex" size="0" />
    <bits name="R" size="1" opcode="0"/>
    <bits name="vvvv" size="4" opcode="0"/>
    <bits name="L" size="1" opcode="1"/>
    <bits name="pp" size="2" default="00" opcode="1"/>
  </coding>
  <coding>
    <byte code="C4" size="1"/>
    <bits name="vex" size="0" />
    <bits name="R" size="1" opcode="0"/>
    <bits name="X" size="1" opcode="0"/>
    <bits name="B" size="1" opcode="0"/>
    <bits name="mmmmm" size="5" opcode="1"/>
    <bits name="W" size="1" opcode="1"/>
    <bits name="vvvv" size="4" opcode="0"/>
    <bits name="L" size="1" opcode="1"/>
    <bits name="pp" size="2" default="00" opcode="1"/>
  </coding>
</modul>
<modul name="opcode" />
<modul name="modrm">
  <coding>
    <bits name="mod" size="2" />
    <bits name="reg" size="3" />
    <bits name="rm" size="3" />
  </coding>
</modul>
```

Obrázek 10: Příklad definice modulu

Každý z modulů má povinný atribut *name*. Pomocí toho jména můžeme jednoznačně určit, kam kódování patří. Důležitou vlastností modulu je to, že jeho velikost musí být zarovnaná na velikost v bytech. Speciální funkci modulu *opcode* už jsme zmiňovali. Všechny ostatní moduly mají povinnost obsahovat kódování. Kódování se zapisuje mezi elementy *coding*. Jak je vidět na obrázku 10, může modul obsahovat více typů kódování. To, že existuje uvnitř modulu více skupin kódování, znamená, že může být v instrukci použito výhradně jedna z nich. To, že se uvnitř více skupin nachází kódování se stejnými jmény, nevadí, jelikož bude vždy použito právě jedno kódování a tedy je jasné, o kterou skupinu bitů jde. Jediným požadavkem je, aby kódování se stejným názvem měly stejné velikosti. Stejných názvů se dá využít k tomu, abychom obě kódování mohli spárovat se stejnými pravidly. Tak, jak jsou kódování zapsána v modulu, tak také budou použita. O správné načítání a úpravu vzhledem k pořadí bytů a bitů se postará program. Velikost jednotlivých kódování se pozná podle názvu elementu – buď *byte* pro byty nebo *bits* pro bity a následného atributu *size*, který udává velikost v příslušné jednotce. Z počátku skupiny je nutné uvést pro kódování jejich uvozující posloupnost bitů, případně byte. Tato hodnota se udává do atributu *code*. Součet velikosti těchto uvozovacích bitů/bytů nesmí být větší než jeden byte. Kódování následující první uvozující bity musí mít definováno místo

atributu *code* atribut *name*, kam se napíše jeho jméno. Všechny do teď zmíněné atributy, kromě atributu *code,* fungovaly ve všech modulech. Následující atributy jsou určeny výhradně pro prefixy.

Jedním již zmíněným atributem je *code*. Další je atribut *opcode*. Ten může nabývat dvou hodnot – 1 a 0. Tento atribut slouží k upřesnění významu kódování. Pokud je kódování použito v některém z pravidel pro prefixy v *opcode*, je potřeba ho opatřit hodnotou 1, v opačném případě se použije hodnota druhá. Ta slouží k označení kódování, která jsou typicky používána pro uložení operandů. Pokud dojde k tomu, že instrukce nepoužije pravidlo pro kódování, přestože by mělo, tak mohou nastat dvě možnosti. Normální postup je, že se vygenerují všechny možnosti daného kódování a instrukce se ve stromě uloží na více míst. Pokud bychom tomuto chování chtěli zabránit, můžeme použít atribut *default*, jehož hodnota se použije automaticky a k žádnému generování již nedojde. Jednou specialitou je to, že pokud uvozující kódování je menší než byte, tak se zbylé bity až do velikosti bytu vygenerují. Jedná se o optimalizaci, abychom na první úrovni *TrieCode* měli pouze celé byty. V příkladu si můžeme povšimnout kódování s velikostí 0. Takové kódování se může použít například kvůli testování přítomnosti modulu.

3.6.2 Definice pravidel pro operandy

Pravidla pro operandy jsou uzavřena mezi elementy s názvem *operand_rules*. Každé z pravidel je následně popsáno mezi elementy *operand_rule*. Na obrázku 11 je znázorněn příklad definice pravidel pro operandy.

```
<operand_rules>
  <operand_rule name="$base$" mode="16bit 32bit">
    <c op1="sib.base" op2="111" />

    <rule s="EDI" />
  </operand_rule>
  <operand_rule name="$sib$" mode="16bit 32bit">
    <c op1="sib.ss" op2="00" />
    <c op1="sib.index" op2="000" />

    <rule s="@EAX+$base$" />
  </operand_rule>
  <operand_rule name="ModRM:r/m" mode="16bit">
    <c op1="modrm.mod" op2="01" />
    <c op1="modrm.rm" op2="100" />

    <rule t="m16:16" s="@WORD FAR [$sib$+'Sdisplacement.disp8']" cop1="ipg4.addrsize_prefix"
cop3="ipg3.opsize_prefix" />
    <rule t="16:16" s="@WORD FAR [$sib$+'Sdisplacement.disp8']" cop1="ipg4.addrsize_prefix"
cop3="vex.pp" cop4="01" />
    <rule t="m16:16" s="@DWORD FAR [$sib$+'Sdisplacement.disp8']"
cop1="ipg4.addrsize_prefix"/>
    <rule t="m16:16" s="@WORD FAR [SI+'Sdisplacement.disp8']" cop1="ipg3.opsize_prefix" />
    <rule t="m16:16" s="@WORD FAR [SI+'Sdisplacement.disp8']" cop1="vex.pp" cop2="01" />
    <rule t="m16:16" s="@DWORD FAR [SI+'Sdisplacement.disp8']" />
  </operand_rule>
</operand_rules>
```

Obrázek 11: Příklad definice pravidel pro operandy

Každé pravidlo musí mít definovaný název. Ten se zapisuje do atributu *name*. Pokud název začíná a končí znakem *$*, znamená to, že se jedná o pravidlo pro proměnnou s daným názvem. Důležité pravidlo pro definování proměnných je to, že její definice se musí vyskytovat před jejím použitím. Atribut s názvem *mode* je nepovinný. Pokud není uveden, pravidla platí pro všechny módy procesoru. Pokud je hodnota uvedena, můžeme nadefinovat, pro které módy dané pravidlo má platit. Jednotlivé módy se od sebe oddělují mezerami. Hodnoty jsou *16bit*, *32bit* a *64bit*. První část, z které se skládá pravidlo, je podmínka a značí se elementem s názvem *c*. První atribut *op1* je povinný. Měl by obsahovat jméno kódování, ke kterému se vztahuje. Jméno se udává ve formátu *modul.coding*, tedy nejprve jméno modulu následované jménem kódování. Obě jména jsou oddělena tečkou. Pokud není přítomen druhý atribut *op2*, je kódování testováno pouze na přítomnost, nikoliv na konkrétní hodnotu. Pokud přítomen je, je pravidlo použito pouze, pokud je reálná hodnota kódování rovna požadované hodnotě. Může být uveden libovolný počet pravidel. Jediným požadavkem je to, aby pravidla se stejným jménem a módem měla stejná kódování v pravidlech. Z hodnot pravidel se totiž vytvoří strom, ve kterém se bude podle hodnot vyhledávat. Pokud tedy pravidla nebudou obsahovat stejné podmínky, nedojde ke správnému uložení pravidla.

Podmínky následuje výčet pravidel. Vkládají se jako hodnoty atributů elementu *rule*. Jediný, který musí být vždy přítomen, je atribut se jménem *s*. Jako hodnota operandu se použije přímo jeho hodnota a operand se uloží jako objekt typu *NameOperand*. Pokud ovšem začíná řetězec znakem @, budou se všechny speciální znaky obsažené v řetězci interpretovat speciálním způsobem. Speciální znaky jsou *$* a *'*. Vše, co je obsaženo mezi dvěma znaky *$*, se bere jako jméno proměnné, kterým je následující výraz nahrazen. Pomocí tohoto speciální znaku je tedy možno skládat více operandů dohromady. Ty jsou následně spojeny do jednoho objektu *Operand*. Pokud je nalezen výraz oddělený s obou stran znaky *'*, je hodnota interpretována jako jméno kódování a operand se nahradí reálnou hodnotou kódování. Jedná se o objekt typu *CodingOperand*. Tento operand se používá například k vypsání konstant jako operandů. To, jestli je hodnota interpretována jako číslo se znaménkem, se dává vědět pomocí písmen: S – se znaménkem, U – bez znaménka. Párové speciální symboly by se nikdy neměli křížit. Po nalezení speciální symbolu se totiž hledá stejný další symbol a vše mezi je použito jako název.

Dalším atributem je *t*. Hodnota atributu určuje jméno operandu ve formátu instrukce. Pro párovaní operandu s instrukcí je potřeba znát trojici *<mód, jméno operandu, jméno ve formátu instrukce (target)>*. Pomocí této trojice jsme schopni jednoznačně spojit správný operand se správnou instrukcí. Jelikož u proměnné je cíl zřejmý – proměnná, není potřeba tento atribut psát. Poslední jsou atributy *cop1*, *cop2*, *cop3* a *cop4*. Ty slouží jako náhrada podmínek přímo u pravidel. Jelikož každé z pravidel může záviset na samostatné podmínce, ale je zbytečné tvořit celý strom operandů, dá se to zapsat takto. Daná pravidla jsou pak ukládána jako objekty *MultiOperand*. Atributy s lichými čísly musí obsahovat jména kódování, ty se sudými čísly jejich hodnoty. Samozřejmě není potřeba hodnotu zadávat, pokud nám jde pouze o přítomnost kódování.

3.6.3 Definice pravidel pro opcode

Pravidla pro opcode jsou vypsána uvnitř elementu *opcode_rules*. Slouží k tomu, abychom byli schopni dodefinovat potřeby prefixů a postfixů. Syntaxe i sémantika je stejná jako u operandů. Přibyli pouze nějaké možnosti speciálně pro opcode. Na obrázku 12 je vidět příklad definice pravidel.

```
<opcode_rules>
  <opcode_rule name="VEX">
    <c op1="vex.vex" />
  </opcode_rule>
  <opcode_rule name="128">
    <c op1="vex.L" op2="0" />
  </opcode_rule>
<opcode_rule name="/0">
    <c op1="modrm.reg" op2="000" />
</opcode_rule>
<opcode_rule name="/+i">
    <c op1="+7" />
    <rule t="ST(i)" s="ST7" />
  </opcode_rule>
</opcode_rules>
```

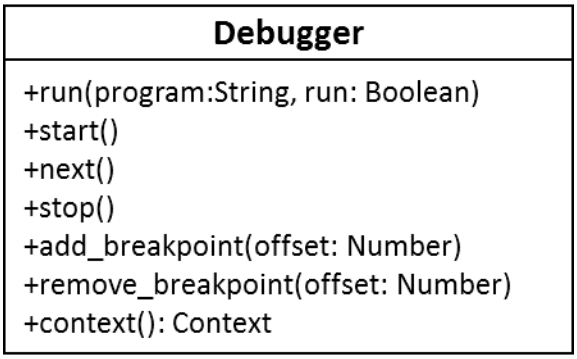
Obrázek 12: Příklad definice pravidel pro opcode

Každé jednotlivé pravidlo je uloženo uvnitř elementu *opcode_rule*. Tento element obsahuje jméno, podle kterého je pravidlo správně použito. Pokud jméno neobsahuje lomítko, jedná se pravidlo pro prefix. Nesmí se zde psát tečka, jako se píše v definici u instrukcí. Pokud se jedná o pravidlo pro postfix, musí být jeho jméno uvozeno lomítkem. Vnitřní část se vyhodnocuje úplně stejně jako u pravidel pro operandy. Pokud je přítomna pouze podmínka, musí být pro nalezení dané instrukce/modulu splněna. Nová věc, která přibyla oproti operandům, je speciální znak + v atributu podmínky. Ten bývá použit společně s elementy *rule*, ze kterých se vytvoří operand. Následně se zjistí číslo uložené za + a později, při vytváření instrukcí s daným pravidlem, se vezme toto číslo, přičte k poslednímu bytu v poli opcode a předáme operand ukládané instrukci, která ho vloží na správné místo podle dvojice *<mód, jméno operandu v instrukci>*.

3.7 Debugger

Debugger je samostatná třída, která se stará o proces debuggování. Pro podporu debuggování je potřeba podpory procesoru, z čehož vyplývá, že služby pro debuggování nám poskytuje OS. Jelikož bychom někdy do budoucna mohli chtít zpřístupnit disassembler i s podporou debuggování pro jiné platformy, bylo potřeba všechny specifické věci uzavřít uvnitř třídy, přesně tak, jak velí zásady OOP[11]. Při rozšiřování by pak již nebyl problém vytvořit nadřazenou abstraktní třídu, která by nám udávala rozhraní debuggeru. Na obrázku 13 je zobrazeno rozhraní třídy *Debugger*.

[11] Objektově Orientované Programování



Obrázek 13: třída *Debugger*

Třída dává k dispozici několik metod k jejímu ovládání. Pomocí metody *run* se spouští proces debuggování, kdy pomocí druhého parametru se nastaví breakpoint na první zpracovávaný řádek. Další metody se starají o ovládání debuggeru. Metoda *context* nám vrací naší speciální strukturu *Context*, která obsahuje všechny zajímavé údaje o debuggeru. Jsou zde jména vláken, registry procesoru, prováděná instrukce a uložené breakpointy. Události, které probíhají v debuggovaném programu, jsou zpracovávány asynchronně. Proto je potřeba v konstruktoru třídy *Debugger* zadat funkce, které jsou zavolány s vyvoláním příslušné události. *Debugger* je vytvářen společně s abstraktní třídou UI, kdy jsou do konstruktoru třídy *Debugger* vloženy virtuální metody této abstraktní třídy. Proto je nutné je implementovat v každém jejím potomkovi. Pokud nechceme zpracovávat všechny události, které debugger zpracovává, vytvořili jsme prázdné funkce, které mohou být místo toho zavolány.

Debugger je implementován za pomoci standardních funkcí z knihovny *kernel32.dll*. Nejprve vytvoříme proces pomocí funkce *CreateProcess*, kdy nastavíme příznak *dwCreatingFlag* na *DEBUG_PROCESS*. Tím docílíme toho, že budeme moci začít debuggovat náš program. Následně ve smyčce voláme funkci *WaitForDebugEvent*. Tato funkce čeká na událost, a pokud ji zachytí, tak vrátí předvyplněnou strukturu typu *DEBUG_EVENT*. Tato strukturu obsahuje union, který pro každý typ události obsahuje informace. My potřebné informace uložíme do naší struktury *Context* a zavoláme callback pro příslušnou událost. Po zpracování události zpracujeme povely, které jsme dostali. Pokud máme skončit, zavoláme funkci *TerminateProcess*. Pokud chceme pokračovat, zavoláme funkci *ContinueDebugEvent*. Dokud nezavoláme tuto funkci, jsou všechna vlákna zastavena a nic nevykonávají.

Pro vkládání breakpointů potřebujeme umět číst a zapisovat do paměti procesu. K tomu použijeme funkce *ReadProcessMemory* a *WriteProcessMemory*. Díky tomu

můžeme načíst paměť z adresy další prováděné instrukce. Ta je uložena v registru IP. Tento byte si uložíme a vložíme na jeho místo breakpoint. Při zpracovávání breakpointů pak starý byte vrátíme na své místo. Jelikož procesor nahrává věci, které bude zpracovávat, do cache, je důležité po každém zápisu do paměti vyčistit cache a to provedeme zavoláním funkce *FlushInstructionCache*.

3.8 CodeAnalyser

Třída *CodeAnalyser* se stará o analýzu přeloženého kódu. Vstupem této třídy je souboru *program.asm*, který vytvoří třída *Dictionary*. Nad tímto souborem probíhá analýza. Sekvenčně procházíme vstupní soubor a hledáme zajímavé instrukce. Zajímavými instrukcemi se myslí instrukce skoku a volání. Jelikož náš program obsahuje pouze základní analýzu programu, nebylo potřeba zatím vytvářet nějaký obecný návrh. Zajímavé instrukce jsou tedy pevně uloženy ve zdrojových kódech. Do budoucna by samozřejmě bylo ideální načítat používané instrukce z instrukčního XML souboru. Po nalezení instrukce ji zpracujeme – zjistíme adresu skoku a volání a uložíme si její adresu společně s místem, kde se nachází ve vstupním souboru. Toto párování je použito pro vypisování ze souboru. Z adres skoku a volání vytvoříme graf toku řízení. Tuto analýzu provádíme vždy, jelikož si v ní indexujeme adresy bloků v souboru.

Po zpracování vstupního souboru dochází na analýzu kódu. Jelikož tato analýza není potřebná, rozhodli jsme se ji spustit paralelně v druhém vlákně. Po jejím dokončení se zobrazí možnost jejího zobrazení v aplikaci. K analýze slouží námi vyrobený graf toku řízení. Ten procházíme standardně pomocí DFS. Při procházení si označujeme vrcholy grafu. Existují tři druhy vrcholů, nenavštívené, navštívené otevřené a navštívené uzavřené. Otevřené znamená, že přes ně pomocí DFS procházíme, uzavřené jsou ty, které jsme již nalezli a odešli z nich pryč. Při průchodu grafem tedy vždy zkontrolujeme vrchol, do kterého jsme přišli. Pokud je uzavřený, nemá ho znova cenu prohledávat, protože jsme to již udělali. Pokud je otevřený, znamená to, že skrz něho pomocí DFS již procházíme, tedy jsme našli cyklus, což odpovídá i cyklu v kódu. Dále pokračujeme s DFS do nenavštíveného vrcholu, případně se vrátíme a zkusíme vedlejší cestu. Pokud z nějakého uzlu v grafu vedou dvě hrany, víme, že jsme narazili na větvení, pokud se tedy nejedná o cyklus. Při průchodu grafem vypisujeme nalezené instrukce v blocích do výstupního souboru

program.hlasm. Při vypisování si pamatujeme indexy na jednotlivé adresy v souboru, abychom mohli jednoduše vypisovat i z tohoto souboru.

3.9 Načítané soubory

Všechny spustitelné soubory v systému Windows dnes používají pro uložení formát PE. V našem programu dovolujeme překládat pouze spustitelné soubory tohoto typu. Knihovny a jiné typy náš program nepodporuje.

Po načtení souboru je potřeba načíst hlavičku daného souboru. Z té vyčteme adresu na další hlavičku. Takto postupujeme až ke třetí hlavičce – volitelné hlavičce. Tato hlavička obsahuje důležité věci, jako jsou typ souboru, adresu na začátek kódu, délku kódu a adresu vstupního bodu. Tato hlavička je naštěstí pro námi podporované typy souborů povinná.

5 Závěr

Cílem této práce bylo vytvořit disassembler-debugger pro procesory Intel64. Program je implementován tak, že pokrývá nejen kýženou architekturu, ale je možně ho i rozšiřovat o nové instrukce nebo případně na jiné architektury. Dosáhli jsme toho díky načítání instrukcí i jejich formátu z připravených souborů, což probíhá v době načítání programu. Pro lepší orientaci je k dispozici základní analýza kódu, která zobrazí v přeloženém kódu smyčky a větvení. Program je také možné spustit a krokovat jeho chod aplikací, při současném zobrazení přeloženého zdrojového kódu v JSA. K dispozici jsou základní povely pro ovládání debuggeru.

5.1 Zhodnocení práce

Povedlo se nám vytvořit disassembler dostatečně obecný na to, aby ho nebyl problém dále jednoduše rozšířit o další instrukce a architektury. Tyto informace jsou definovány v přiložených souborech ze který si je program načítá. Není tedy potřeba kvůli změně instrukční sady měnit program, ale stačí pouze popsat cílovou architekturu v XML souborech podle daných pravidel. Program také obsahuje, na rozdíl od jiných podobných programů, prostředky pro jednodušší práci s JSA, ať debugger nebo základní analýzu. Jednou z nevýhod programu je jeho pomalost a z důvodu implementace debuggeru i uzavřenost na platformu Windows.

5.2 Možnosti rozšíření a vylepšení

Samozřejmě náš program není dokonalý. Jednou z věcí, která je vždy nutná u aplikací, je opravování chyb. I přesto, že jsme náš program testovali, jsme neodhalili všechny chyby, které se v aplikaci nacházejí. Dalších možností, jak by bylo možné aplikaci rozšiřovat a vylepšovat je několik:

- Zrychlení běhu programu. Program nebyl zaměřen primárně na rychlost, ale důležitější byla rozšiřitelnost. Pro reálné použití na větších program by bylo ale potřeba program zrychlit. Z provedeného měření vyplývá, že nejdéle v programu trvá zpracování operandů u jednotlivých instrukcí
- Rozšířit podporu na jiné platformy procesorů. Mělo by stačit předpřipravit další XML soubory pro instrukce a operandy. A upravit stávající do přehlednější podoby za použití funkčností programu, jako jsou například proměnné.

- Vytvořit okenní prostředí pro běh aplikace, jelikož konzole nedovoluje tak komfortní ovládání jako ovládání v okně pomocí myši.
- Rozšiřovat podporu načítaných formátů i na jiné typy souborů a přidat možnost připojení k již běžícím programům a zobrazení jejich načtených DLL souborů.
- Vytvořit multiplatformní program. Zde by bylo potřeba vytvořit pro každou platformu vlastní debugger, případně platformu a běh procesoru virtualizovat.
- Rozšiřovat statickou analýzu programu. Zde je možnost u některých programů využít přiložené soubory typu *pdb*. Celkově by bylo ideální se statickou analýzou kódu přiblížit k decompilerům. Tedy rozpoznávat co nejvíce programových struktur, nacházet v paměti řetězce nebo rozpoznávat systémová volání a standardní funkce některých jazyků.

Seznam použité literatury

1. INTEL CORPORATION. *Intel® 64 and IA-32 Architectures*. 325462.
2. CORPORATION, M. *Microsoft Portable Executable and Common Object File Format Specification*. 2013.
3. Xerces-C++ XML Parser. *Xerces-C++ XML Parser* [online]. [cit. 2014-07-11]. Dostupné z: http://xerces.apache.org/xerces-c/
4. The Apache Software Foundation. *Apache License, Version 2.0* [online]. [cit. 2014-07-11]. Dostupné z: http://www.apache.org/licenses/LICENSE-2.0
5. *IDA* [online]. [cit. 2014-06-30]. Dostupné z: https://www.hex-rays.com/products/ida/index.shtml
6. *OllyDbg* [online]. [cit. 2014-06-30]. Dostupné z: http://www.ollydbg.de/
7. *NASM home page* [online]. [cit. 2014-06-30]. Dostupné z: http://www.nasm.us
8. GAMMA ERICH, H. R. J. R. V. J. *Disign Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley Professional, 1994. ISBN 978-0201633610.
9. CodeProject. *Writing a basic Windows debugger* [online]. [cit. 2014-07-13]. Dostupné z: http://www.codeproject.com/Articles/43682/Writing-a-basic-Windows-debugger
10. CodeProject. *Writing Windows Debugger - Part 2* [online]. [cit. 2014-07-14]. Dostupné z: http://www.codeproject.com/Articles/132742/Writing-Windows-Debugger-Part
11. *64 bits* [online]. 2010 [cit. 2014-07-14]. Dostupné z: https://software.intel.com/en-us/articles/all-about-64-bits
12. MSDN. *Peering Inside the PE: A Tour of the Win32 Portable Exucatable File Format* [online]. 1994 [cit. 2014-07-14]. Dostupné z: http://msdn.microsoft.com/en-us/library/ms809762.aspx
13. EILAM, E. *Reversing: Secrets of Reverse Engineering*. 2005. ISBN 978-0-7645-7481-8.
14. ALFRED V. AHO, R. S. J. D. U. *Compilers: Principles, Techniques, and Tools*. 1st. 1986. ISBN 978-0201100884.
15. HYDE, R. *The Art of Assembly Language*. 2010. ISBN 978-1-59327-207-4.

Seznam použitých zkratek

V této sekci se nachází seznam všech zkratek, které jsou v práci použity

- **XML** – eXtensible Markup Language – rozšiřitelný značkovací jazyk
- **JSA** – Jazyk Symbolických Adres
- **CPU** – Central Processing unit – procesor
- **BIOS** – Basic Input Output Systém – jedná se o systém, který se stará o základní funkce počítače
- **OS** – Operační Systém
- **IP** – Instruction Pointer – registr, který obsahuje ukazatel na instrukci
- **OOP** – Objektově Orientované Programování
- **MVC** – Model-View-Controller – architektonický návrhový vzor
- **DLL** – Dynamically Linked Library – dynamicky linkovaná knihovna
- **DFS –** Depth-First Search – prohledávání do hloubky
- **SAX –** Simple API for XML – API pro seriové zpracovávání XML
- **DOM –** Document Object Model – objektová reprezentace XML

A Uživatelský manuál

V této příloze se nachází uživatelský manuál, který popisuje práci s aplikací od její instalace až po její spuštění a ovládání.

A.1. Obsah CD

Na přiloženém CD se nachází tyto složky:

- **src** – Obsahuje zdrojové kódy a projekt pro Visual Studio 2013
- **bin** – Obsahuje náš vytvořený spustitelný program a potřebné knihovny
- **bin/conf –** Obsahuje konfigurační soubory, jejich schémata a soubor, do kterého se zapisují chybová hlášení
- **doc** – Obsahuje bakalářskou práci
- **tests** – Obsahuje testovací soubory

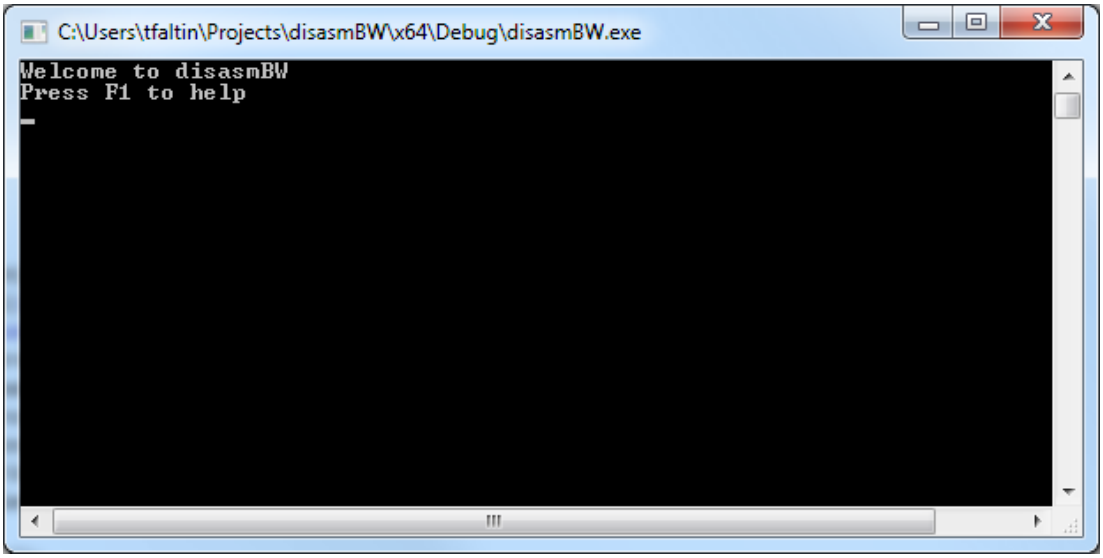
A.2. Požadavky programu

Program vyžaduje OS Windows 7 a vyšší. Na jiných OS program nebyl testován, takže nemůžeme zaručit jeho funkčnost.

A.3. Obsluha programu

Pro instalaci program na cílový počítač stačí zkopírovat složku *bin* z CD do cílového počítače. V této složce se nachází program *disasmBW.exe*, který slouží ke spuštění programu. Pro vlastní konfiguraci disassembleru stačí jít do složky *conf* a nakonfigurovat podle požadavků konfigurační soubory. Soubory musí být validní vzhledem k uloženým schématům. Pokud při načítání nebo při běhu aplikace nastane problém, je napsán v souboru *log.txt*.

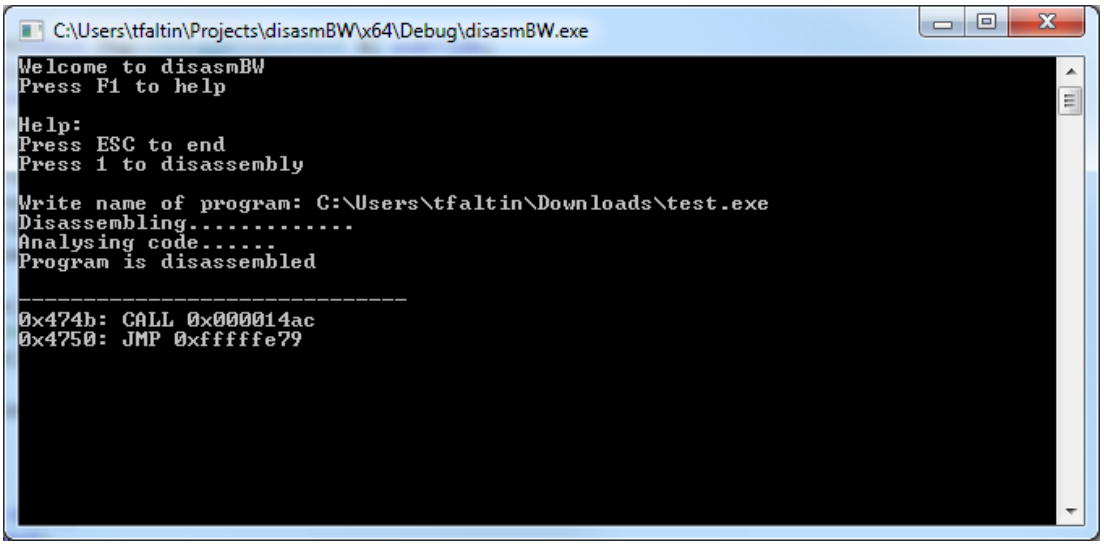
Po spuštění aplikace by se mělo zobrazit konzolové okno, jako je na obrázku 14. Rychlost načítání konfiguračních souborů závisí na jejich velikosti a rychlosti počítače.



Obrázek 14: Startovní obrazovka programu

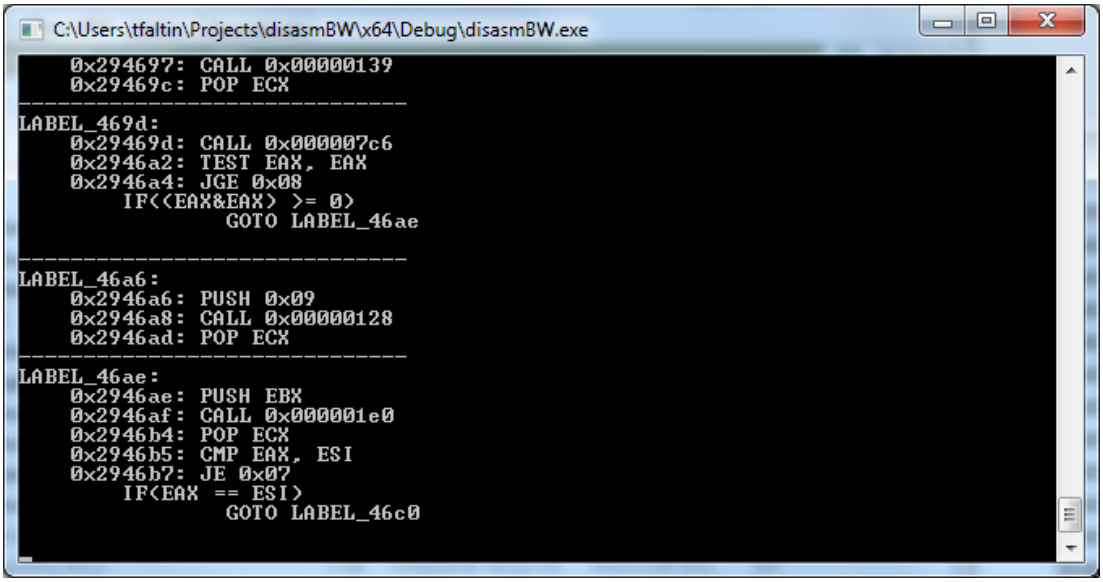
Pro více informací zmačkněte klávesu F1. Tato klávesa slouží k zobrazení nabídky v celém programu. Pro ukončení slouží klávesa ESC. Ta funguje také v celém programu a slouží vždy k vrácení o úroveň zpět. Pro spuštění disassembleru stačí

zmačknout klávesu 1. Následně jste vyzváni k zadání názvu programu. Pokud není možné program disassemblovat, program na to upozorní a vrátí se na startovní obrazovku. Během disassemblování vypisuje program na obrazovku tečky. Po skončení překladu se objeví obrazovka, jako na obrázku 15. Po provedení analýzy je spuštěna v paralelním vlákně další analýza, která nalezne vyšší programové struktury. Po jejím ukončení se v nabídce objeví nová možnost pro zobrazení těchto struktur.



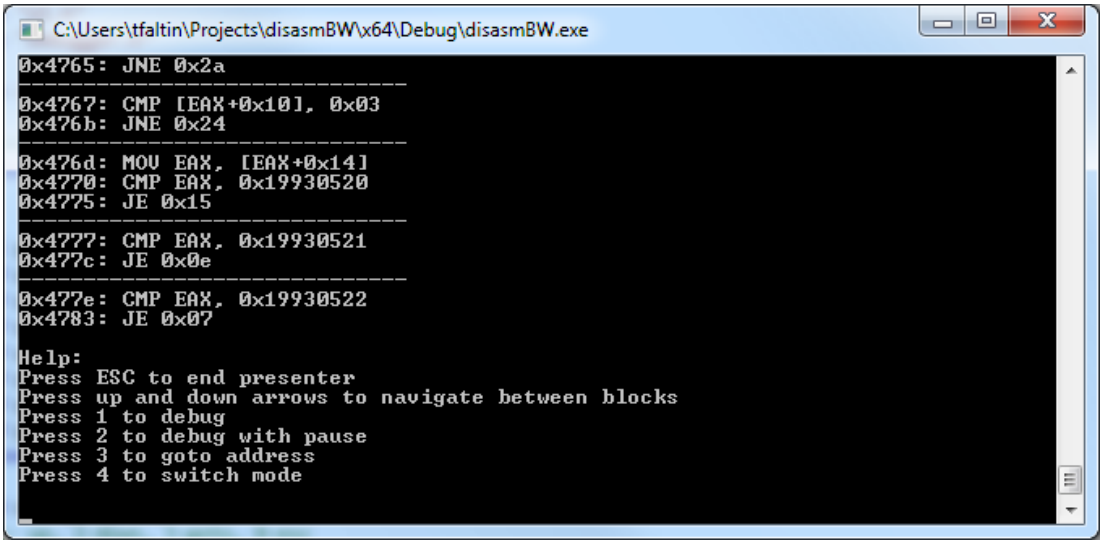
Obrázek 15: Obrazovka po přeložení programu

Program po úspěšném načtení zobrazí blok přeloženého kódu. Automaticky je nalezen a vypsán první blok, který obsahuje vstupní bod v programu. Přeložený kód je zobrazován po blocích. Mezi jednotlivými bloky se můžeme pohybovat šipkami nahoru a dolu. Pokud se narazí na konec nebo začátek přeloženého programu, žádné další bloky nejsou vypisovány. Pro skok na určitou adresu je potřeba zmačknout klávesu 3 a vepsat adresu. Pokud není adresa nalezena, nikam se neskočí. Po stisknutí klávesy 4 dojde k přepnutí do zobrazování vyšších programových struktur. Pokud tato možnost ještě není k dispozici, možnost se v menu neobjeví. Na obrázku 16 je ukázána obrazovka společně programu s nalezenými vyššími programovými strukturami.



Obrázek 16: Obrazovka s vyššími programovými strukturami

Pro spuštění debuggování jsou určený klávesy 1 a 2. Druhá z nich spustí debuggování a zastaví se na začátku vykonávaného programu. Na obrázku 17 jsou ukázány bloky programu a menu programu, v tomto zobrazovacím módu.

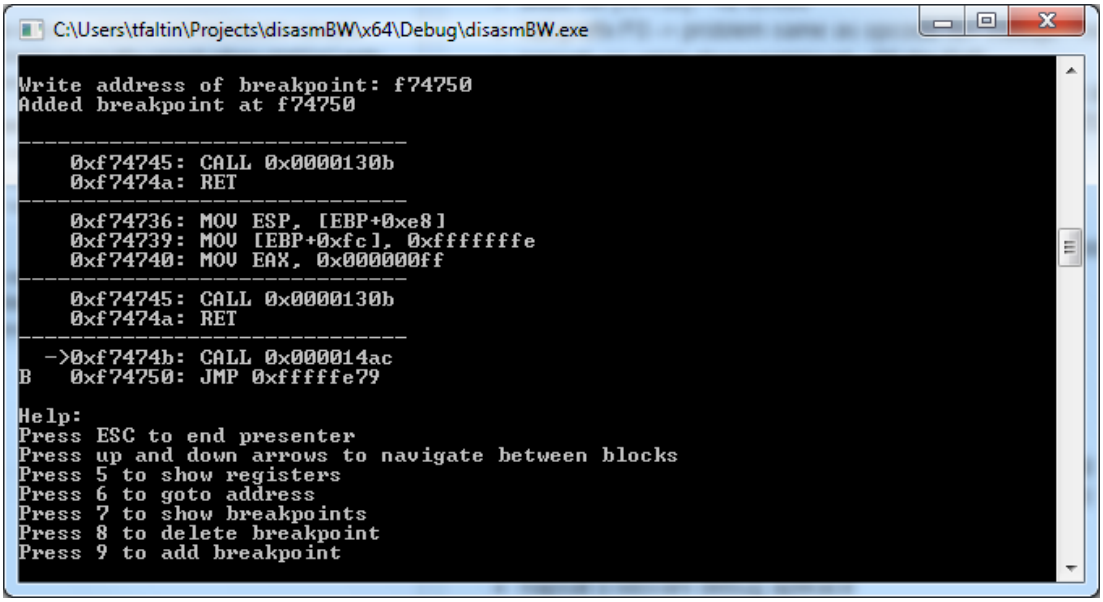


Obrázek 17: Obrazovka s bloky

Program při debuggování zobrazuje všechny zajímavé události, které se přihodily. Po nalezení breakpointu se běh programu zastaví. Běh je také možné zastavit stisknutím tlačítka 6. Pro následné pokračování slouží klávesa 5. Pokud bychom chtěli skočit pouze na další instrukci, stačí stisknout klávesu 0. Pro zobrazení právě běžících vláken je určeno tlačítko 8. Pro zobrazení informací o vlákně je připraveno tlačítko 9.

Při zobrazení informací o vlákně jste v případě více vláken požádáni o zadání identifikátoru vlákna. Po správném zadání program zobrazí zdrojový kód běžícího vlákna. Pokud ale vlákno běží v části paměti, kterou nemáme přeloženou, například v nějaké knihovně, zobrazí se pouze adresa a nepřeložená instrukce. Pokud máme přeložený zdrojový kód k dispozici, zobrazí se blok, ve kterém se nachází budoucí prováděná instrukce a ukazatel na ni. Pokud je na nějakou instrukci umístěn

breakpoint, zobrazí se před instrukcí velké písmeno B. Na obrázku 18 je ukázáno, jak vypadá vypisování bloků v debuggeru a menu, které je k dispozici.



Obrázek 18: Obrazovka v debuggeru

V módu zobrazování zdrojového kódu je možné provádět další akce a zobrazovat dodatečné informace. Pomocí šipek můžeme opět procházet kód a pomocí klávesy 6 skočit na adresu. Tlačítko 5 slouží k zobrazení hodnot všech základních registrů. K práci s breakpointy slouží tlačítka 7 – ukáže všechny breakpointy, 8 – smaže breakpoint, 9 – přidá nový breakpoint.

Pro vrácení o úroveň zpět slouží klávesa ESC. Postupným mačkáním této klávesy dojde i k vypnutí programu.

B Formáty souborů

Tato příloha obsahuje XML schémata ke konfiguračním souborům. Schémata je taktéž možno nalézt na CD ve složce *bin/conf*.

B.1. Schéma souboru *instruction.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="instructions" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="instructions">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="operands" minOccurs="1" maxOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="operand" minOccurs="0" maxOccurs="unbounded" >
                <xs:complexType>
                    <xs:attribute name="name" type="NonEmptyString" />
                    <xs:attribute name="op1" type="OperandType" />
                    <xs:attribute name="op2" type="OperandType" />
                    <xs:attribute name="op3" type="OperandType" />
                    <xs:attribute name="op4" type="OperandType" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="instruction" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="formats">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="format" minOccurs="1"
maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="form" type="FormType" />
                        <xs:attribute name="opcode" type="OpcodeType" />
                        <xs:attribute name="op_enc" type="xs:string" />
                        <xs:attribute name="bit64" type="NumberBoolType" />
                        <xs:attribute name="bit32" type="NumberBoolType" />
                        <xs:attribute name="bit16" type="NumberBoolType" />
                        <xs:attribute name="desc" type="xs:string"
use="optional"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" type="NonEmptyString" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="NonEmptyString">
    <xs:restriction base="xs:string">
      <xs:minLength value="1" />
    </xs:restriction>
  </xs:simpleType>
```

```
  <xs:simpleType name="NoOperandType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="NA" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="OperandType">
    <xs:union memberTypes="NonEmptyString NoOperandType" />
  </xs:simpleType>
  <xs:simpleType name="FormType">
    <xs:restriction base="xs:string">
      <xs:pattern value="([A-Z|[0-9])+(( [^,\s]*)(, [^,\s]*){0,3})?" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="OpcodeType">
    <xs:restriction base="xs:string">
      <xs:pattern value="((((\.([A-Z]|[a-z]|[0-9])*)* )?)(([0-9]|[a-f]|[A-
F]){2}( ([0-9]|[a-f]|[A-F]){2})*)(( /\+([a-z]|[0-9]|[A-Z])*)?( /([a-z]|[0-
9]|[A-Z])*)*)" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="NumberBoolType">
    <xs:restriction base="xs:short">
      <xs:enumeration value="0" />
      <xs:enumeration value="1" />
    </xs:restriction>
  </xs:simpleType>
      </xs:schema>
```

B.2. Schéma souboru *instruction_format.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="instruction_format"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="instruction_format">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="format" minOccurs="1" maxOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="modul" minOccurs="1" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence maxOccurs="unbounded">
                    <xs:element name="coding" minOccurs="0">
                      <xs:complexType>
                        <xs:choice maxOccurs="unbounded">
                          <xs:element name="byte">
                            <xs:complexType>
                              <xs:attribute name="name" type="xs:string" />
                              <xs:attribute name="size" type="xs:short" />
                              <xs:attribute name="code" type="xs:hexBinary"
use="optional" />
                              <xs:attribute name="desc" type="xs:string"
use="optional" />
                            </xs:complexType>
                          </xs:element>
                          <xs:element name="bits">
                            <xs:complexType>
                              <xs:attribute name="code" type="BitStringType"
use="optional" />
```

```
                              <xs:attribute name="name" type="xs:string" />
                              <xs:attribute name="size" type="xs:short" />
                              <xs:attribute name="desc" type="xs:string"
use="optional"/>
                              <xs:attribute name="opcode"
type="NumberBoolType" use="optional"/>
                              <xs:attribute name="default"
type="BitStringType" use="optional"/>
                            </xs:complexType>
                          </xs:element>
                        </xs:choice>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name" type="xs:string" />
                  <xs:attribute name="mode" type="ModeType" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="operand_rules" minOccurs="1" maxOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="operand_rule" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="c" type="CodingType" minOccurs="0"
maxOccurs="unbounded" />
                    <xs:element name="rule" type="RuleType"
maxOccurs="unbounded"/>
                  </xs:sequence>
                  <xs:attribute name="name" type="OperandRuleNameType" />
                  <xs:attribute name="mode" type="ModeType" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="opcode_rules" minOccurs="1" maxOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="opcode_rule" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="c" maxOccurs="1" type="OpcodeCodingType"
/>
                    <xs:element name="rule" type="RuleType" minOccurs="0"
maxOccurs="unbounded"/>
                  </xs:sequence>
                  <xs:attribute name="name" type="xs:string" />
                  <xs:attribute name="mode" type="ModeType" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="OpcodeType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="opcode" />
```

```
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="BiteSizeType">
    <xs:restriction base="xs:short">
      <xs:minInclusive value="0" />
      <xs:maxInclusive value="7" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="NumberBoolType">
    <xs:restriction base="xs:short">
      <xs:enumeration value="0" />
      <xs:enumeration value="1" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="BitStringType">
    <xs:restriction base="xs:string">
      <xs:pattern value="(0|1){1,7}" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="ModeType">
    <xs:list itemType="ModeTypeItem" />
  </xs:simpleType>
  <xs:simpleType name="ModeTypeItem">
    <xs:restriction base="xs:string">
      <xs:enumeration value="64bit" />
      <xs:enumeration value="32bit" />
      <xs:enumeration value="16bit" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="OperandRuleNameType">
    <xs:restriction base="xs:string">
      <xs:pattern value="([^\$\s]+)|\$[^\s\$]+\$" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="CodingNameType">
    <xs:restriction base="xs:string">
      <xs:pattern value="[^\s.]+\.[^\s.]+" />
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="RuleType">
    <xs:attribute name="t" type="xs:string" />
    <xs:attribute name="s" type="SourceType" />
    <xs:attribute name="cop1" type="CodingNameType" use="optional"/>
    <xs:attribute name="cop2" type="BitStringType" use="optional"/>
    <xs:attribute name="cop3" type="CodingNameType" use="optional"/>
    <xs:attribute name="cop4" type="BitStringType" use="optional"/>
  </xs:complexType>
  <xs:simpleType name="SourceType">
    <xs:restriction base="xs:string">
      <xs:pattern
value="[^@]+|(@([^\$']*('[^\$']*')?[^\$']*(\$[^\$']*\$)?[^\$']*)+)" />
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="CodingType">
    <xs:attribute name="op1" type="CodingNameType" />
    <xs:attribute name="op2" type="BitStringType" />
  </xs:complexType>
  <xs:simpleType name="OpcodeCodingNameType">
    <xs:restriction base="xs:string">
      <xs:pattern value="(\+[0-7])|[^\s.]+\.[^\s.]+" />
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="OpcodeCodingType">
```

```
    <xs:attribute name="op1" type="OpcodeCodingNameType" />
    <xs:attribute name="op2" type="BitStringType" />
  </xs:complexType>
</xs:schema>
```