VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÝCH SYSTÉMŮ

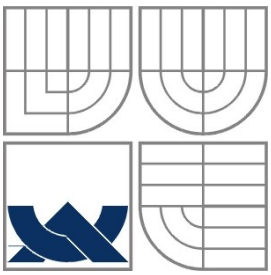
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

INTERAKTÍVNY DISASSEMBLER

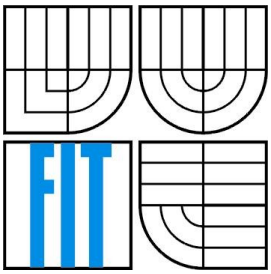
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE Milan Mrva
AUTHOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

INTERAKTIVNÍ ZPĚTNÝ ASSEMBLER

INTERACTIVE DISASSEMBLER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Milan Mrva
AUTHOR

VEDOUCÍ PRÁCE — Ing. Jakub Křoustek
SUPERVISOR

BRNO 2011

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2010/2011

Zadání bakalářské práce

Řešitel: **Mrva Milan**
Obor: Informační technologie
Téma: **Interaktivní zpětný assembler**
Interactive Disassembler
Kategorie: Překladače

Pokyny:

1. Seznamte se s metodami zpětného překladu kódů z binární formy do jazyka symbolických instrukcí.
2. Prostudujte architektury a instrukční sady běžně používaných mikroprocesorů.
3. Dle pokynů vedoucího navrhněte modifikace, které urychlí některé známé metody zpětného překladu.
4. Metody navržené v předchozím bodě použijte při implementaci interaktivního disassembleru pro vybrané instrukční sady.
5. Aplikace by měla poskytovat široké spektrum funkcí. Např. přímá editace programu, zvýrazňování syntaxe apod.
6. Zhodnoťte dosažené výsledky a diskutujte další možný vývoj projektu.

Literatura:

- Křoustek, J.: Analýza a transformace kódů, bakalářská práce, FIT VUT v Brně, Brno, 2007.
- Schwarz, B., Debray, S. K., Andrews, G. R.: Disassembly of Executable Code Revisited, University of Arizona, Tucson, 2002.
- Další dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Bez požadavků

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

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

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

Vedoucí: **Křoustek Jakub, Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2010
Datum odevzdání: 18. května 2011

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

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

V práci jsou popsané postupy a nástroje zpětného inženýrství v rámci softwaru. Uvedené jsou techniky ochrany před rozkladem či zkoumáním obsahu spustitelného souboru. Představené jsou příklady programů zabývajících se zpětným překladem. Dále se práce zaobírá architekturou procesoru s důrazem na mikroprocesory Intel a Motorola. Jsou ukázané rozdílné formáty spustitelných souborů. Byl implementovaný generický modulovatelný zpětný assembler.V tomto textu je představená jak jeho struktura, tak zásuvné moduly, které prezentují tři různé techniky disassemblovaní. Jeden z modulů používa vícevláknový parser, který je vlastním návrhem autora. Tyto přístupy jsou v závěru porovnané a je nastíněn další vývoj.

Abstract

This thesis describes procedures and tools of reverse engineering in terms of software development. There are introduces different techniques of protection against decomposition of executables. The work also mentions some programs used for decomposition analysis. Furthermore it contains information about architecture of processing units, with emphasis on microprocessors Intel and Motorola. Variety of executable formats is shown. Generic retargetable disassembler was implemented. There is a description of its structure and plugins. These plugins represents three algorithms used for disassembling a program. One of them applies a multi-process parsing engine, which is an own design by author of the thesis. At the end, these techniques are compared and further development is outlined.

Klíčová slova

Zpětný assembler, dekompilátor, debugger, reverzní inženirství, architektura mikroprocesoru, inštrukční sada, formaty spustitelných souborů

Keywords

Disassembler, decompiler, debugger, reverse engineering, microprocessor architecture, instruction set, executable file formats

Citace

Milan Mrva: Interaktívny Disassembler, bakalářská práce, Brno, FIT VUT v Brně, 2011

Interaktívny Disassembler

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jakuba Křoustka.
Další informace mi poskytli Ing. Richard Růžička, Ph.D a Ing. Josef Strnadel, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Milan Mrva
30. 4. 2011

Poděkování

Rád by som poďakoval hlavne môjmu vedúcemu Ing. Jakubovi Křoustkovi za konzultácie, rady a poskytnuté informácie ohladom celej problematiky spätného prekladu. Ing. Richardovi Růžičkovi Ph.D za predstavenie formátu s19 a Ing. Josef Strnadel, Ph.D. za poskytnuté testovacie súbory.

Obsah

1 Úvod

Pri vývoji určitého produktu, alebo aj rozvoji nejakej myšlienky zvyčajne začíname od najjednoduchšieho a postupne sa presúvame k zložitejším a komplexnejším problémom. Jedná sa o tzv. metódu zdola-nahor. Štruktúru problému si sami vytvárame.

Môže ale nastať prípad, že táto štruktúra, vec, či systém je už vytvorený a my ho chceme rozanalyzovať a spoznať. V biológií pri skúmaní procesov fungovanie ľudského tela, v chémií pri analýze zloženia atómu, či v genetike pri objavovaní DNA. Vtedy sa jedná o opačnú metódu zhora-nadol. Touto metódou sa zaoberá aj reverzné inžinierstvo. Snaží sa na základe už fungujúceho prípadu, zrekoštrouvať princípy a procesy na, ktorých je tento prípad postavený.

Spočiatku malo svoje vyžitie hlavne prírodovedných vedách, ale postupne čoraz viac preniká aj do technických disciplín. Inžinieri na celom svete sa snažia skopírovať technické riešenia od súperov, či spoznať do hĺbky vlastné systému, ktoré majú k dispozícií. Veľmi rozšírené je to vo vojenskej sfére. Ale aj v trhovej ekonomike u konkurujúcich podnikov.

Na poli informačných technológií tomu nie je inak. Môže sa jednať o rozloženie súborových formátov, komunikačných protokolov, či spustiteľných programov. Základným prostriedkom rozkladania programov je práve spätný preklad binárneho súboru do základného ľudsky čitateľného programovacieho jazyka – assembleru, tzv. disassembling.

Cieľom tejto práce je objasniť problematiku reverzného inžinierstva na poli informačných technológií a interaktívny disassembler navrhnúť a implementovať. Viac o reverznom inžinierstve v informatike, disassemblovaní i interaktivite u disassembleru je napísané v druhej kapitole.

Tretia kapitola obsahuje informácie o architektúre mikroprocesorov a mikrokontrolérov a rôznych formátoch spustiteľných súborov na PC a zabudovaných zariadeniach.

V štvrtej kapitole bude rozobraný návrh a implementácia samotného disassembleru. Jedná sa o obecný disassembler modulovateľný pomocou pluginov. Budú ukázané a zhodnotené rozličné postupy pri spätnom preklade na architektúre Motorola HC08.

Práca je zosumarizovaná v záverečnej časti. Dôraz je tu kladený na multiprocesový parsovací mechanizmus a sú načrtnuté ďalšie možné smery vývoja.

2 Reverzné inžinierstvo v informatike

Reverzné inžinierstvo (RE) je definované ako proces analýzy predmetného systému s cieľom identifikovať komponenty systému, ich vzájomné väzby alebo vytvoriť reprezentáciu systému v inej forme poprípade vo vyššej úrovni abstrakcie [1]. V rámci informačných technológií môžeme RE rozdeliť na hardwarové a softwarové. U hardwaru sa jedná o zistenie fyzického zapojenia integrovaných obvodov, mikročipov alebo celého prístroja (mp3 prehrávač, herná konzola...). Softwarové RE sa dá ďalej rozdeliť podľa zdrojových a cieľových dát na rozklad komunikačného protokolu, súboru, či programu. Pri rozklade programu môžeme podľa cieľových dát rozoznávať disassembling, dekompiláciu, či rozklad na UML model programu.

- Rozklad komunikačného protokolu – sledovaním sieťovej komunikácie na rozdielnych úrovniach môžeme zistiť tak obsah ako aj forma komunikácie. Na fyzickej úrovni to pomáha napr. pri skúmaní protokolu ovládača nejakého nedokumentovaného hardwarového zariadenia. Na aplikačnej vrstve internetového protokolu sa dá (pomocou programov ako tcpdump, či Wireshark) sledovať komunikácia sieťových programov. Napríklad protokol ICQ bol takto rozšifrovaný a vďaka tomu vznikli alternatívne klienta ako Miranda, qip a podobne.
- Rozklad súborových formátov – Získavanie informácie o štruktúre súboru alebo jeho obsah. Otváranie formátu alternatívnych programov - napr. Office/OpenOffice, grafické programy apod. Alebo získanie obsahu súboru, ktorý je viditeľný napr. hexadecimálnom formáte alebo s acsii kódu, príp. meta dáta.
- Disassembling – vykonáva opak toho čo prekladač assembleru. Spustiteľný súbor prekladá spätne do jazyka symbolických inštrukcií. Podrobnejšie sa budeme disassemblerom zaoberať v nasledujúcich kapitolách.
- Dekompilácia – je opakom kompilácie. Prekladá binárny súbor do vyššieho programovacieho jazyka (C, C++, Java...). Prvým stupňom dekompilácie je práve disassemblovanie súboru. O dekompilácií sa viac píše v kapitole 2.2.
- UML model – Pre pochopenie štruktúry komplexného programu je dôležité vedieť z akých komponentov program pozostáva. Na základe skokov v zdrojovom kóde sa dá zostaviť diagram toku programu. Diagram tried sa dá vytvoriť na základe záznamov v tzv. vtable.

Motivácia k spätnému inžinierstvu je rôzna. Od obyčajnej zvedavosti, cez „crackovanie“ plateného softwaru a nahrádzanie stratenej dokumentácie až po sofistikovanú priemyselnú špionáž.

- Hľadanie chýb v programoch - pri výstavbe debugeru je nutné v prvom rade rozložiť preložený program späť do programovacieho jazyka.
- Vzdelávanie – Jedna s metód učenia sa pre začínajúceho programátora je skúmanie cudzieho kódu. V prípade, že nie je kód k dispozícií, i rozloženého programu.
- Kompatibilita programov a súborov – Pri vývoji softvéru, tvorca obyčajne chce aby jeho program bol schopný pracovať i s alternatívnymi, či konkurentnými formátmi. Ich štruktúru alebo komunikačný protokol môže zistiť buď s dokumentácie alebo za pomoci RE. Napríklad program Samba vznikol vďaka RE.
- Kontrola, či niekto neskopíroval váš kód
- Usvedčenie s podvodu – Pokiaľ niekto tvrdí, že jeho program má vykonávať istú funkcionalitu, rozložením tohto programu si to možno overiť. Známy je prípad spoločnosti Syncronys, ktorá v roku 1995 predala cca 600tisíc kópií svojho produktu SoftRAM95. Ten mal zväčšiť kapacitu pamäte RAM a zvýšiť výkon pre počítače s novým operačným systémom Windows 95. Programátori Mark Russinovitch a Andrew Schulman zistili pomocou RE, že aplikácia len mení nastavenie Windows (ukazateľ pamäte a pod.) a rozširuje swap súbor. Inak používa kódy z Microsoft „Windows Development Kit“. [2]
- Antivírové spoločnosti používajú pokročilé reverzné metódy na odhalenie funkcionality a slabých miest počítačových vírusov.
- Počítačové pirátstvo, krádež software, prelamovanie ochranných známok, atď.

2.1 Disassembling a disassembler

Disassembler (DA) je program určený na spätný preklad spustiteľného súboru do jazyka symbolických inštrukcií alebo internej formy pseudoassembleru, špecifickej pre daný dissembler. Poznáme dve hlavné techniky disassemblingu [3].

- Po prvé – Lineárnu (Linear Sweep) – táto technika sa používa pri formátoch s jednotnou dĺžkou inštrukcie. Disassembler ide postupne od začiatku inštrukčnej

sekvencie a jednotlivé bity o dĺžke inštrukcie prekladá priamo do assembleru. Táto metóda je síce veľmi rýchla, ale vôbec sa nestará o to aké informácie spracováva. A keďže prekladače väčšinou zarovnávajú dáta a vkladajú tabuľky skokov do kódu, linear sweep je pomerne nepresná.

- Rekurzívna metóda je sofistikovanejšia. Kontroluje inštrukcie a pokiaľ narazí na inštrukciu skoku prejde na návestie skoku. Návestie označí ako už navštívené aby ho nabudúce neprekladalo. Potom pokračuje zas lineárne. Pokiaľ narazí na return vráti sa späť na posledný skok. Takto prejde celý súbor. Pri disassemblovaní zložitejších architektúr binárnych súborov je táto technika využiteľnejšia. Pre jednoduché inštrukčné sady stačí linear sweep.
- V praxi sa zvyčajne používa kombinácia obidvoch metód – tzv. hybridný algoritmus. Ten prechádza súbor rekurzívne a keď vyčerpá všetky možnosti pokračuje lineárnym prechodom.

V závislosti od účelu disassembleru je i jeho zložitosť. Pokiaľ chceme len vidieť zdrojový kód obyčajného programu stačí nám jednoduchý disassembler. Najkomplikovanejšie DA používajú crackery snažiaci sa odhaliť pôvodný kód nejakého chráneného programu. To preto lebo tieto musia byť schopné odhaliť rôzne techniky anti-disassemblingu.

2.1.1 Interaktivita u disassembleru

Základným obmedzením klasického DA je, že si nemá odkiaľ vytiahnuť názvy funkcií, premenných, procedúr a komentáre. Preto je schopný maximálne vypísať čistý zdrojový kód. Assemblerovský kód bez hlbších znalostí konkrétneho programu sa užívateľovi pomerne zle dekóduje.

Interaktívny disassembler oproti tomu umožňuje vkladanie komentárov, zadávanie mien funkcií či návestí a tak značne sprehľadňuje čítanie kódu. Samozrejme tieto údaje musí zadať užívateľ, ktorí danému kódu rozumie.

Pomocou disassembleru môžeme i meniť samotný binárny súbor. Pri zmene inštrukcií a dát si treba dávať pozor na adresy. Každé návestie je už namapované na presnú adresu a pokiaľ by sme nahradili napríklad 4bitovú inštrukciu 8 bitovou, všetky dáta za ňou by sa o 4 bity posunuli, čiže by už nefungovali skoky a to by spôsobilo nefunkčnosť celého programu. Nahrádzať môžeme len rovnako dlhými dátami, prípadne kratšími. V tom prípade sa nadbytočné bity vynulujú resp. nahradia

inštrukciami NOP (no operation) a interpret (procesor) ich preskakuje. Pokiaľ by užívateľ chcel robiť komplexnejšie zmeny v programe, je lepšie zdrojové súbory (získané pomocou DA) znovu skompilovať.

2.2 Dekompilácia

Dekompilácia je proces spätného prekladu zo spustiteľného programu do vyššieho programovacieho jazyka. Je opakom kompilácie - prekladu. Skladá sa z dvoch krokov. Prvým je preklad s binárky do pseudoassembleru – disassembling. Druhým je preklad s assembleru do cieľového jazyka. Cieľový jazyk nemusí byť nutne zhodný s jazykom, v ktorom bol program napísaný.

Disassemblerom sa podrobnejšie zaoberá predchádzajúca kapitola. Tu si povieme niečo o druhej fáze behu dekompilera. Assemblerovský kód symbolických inštrukcií prejde sémantickou analýzou, kde sa na základe rôznych vzorcov inštrukcií idúcich za sebou vytvoria pseudoinštrukcie vyššieho jazyka. Je potrebné zanalyzovať nielen samotný inštrukčný tok programu ale aj premenné a dátové štruktúry.

Keďže rôzne prekladače generujú rozdielnu syntax assemblerovských inštrukcií a tak aj rozdielne binárne súbory. Nie je prakticky možné pôvodný zdrojový kód spätne dekompilovať. Použitím optimalizácií, ktoré na úrovne assembleru kód zoštíhľujú a tým pádom rozbíjajú obvyklé idiómy, sa šanca, že kód zrekonštruujeme ešte znižuje. Naopak pridanie debugovacích informácií pri preklade, umožňuje dostať sa aj k pôvodným názvom premenných a funkcií.

Original source code	**Disassembled binary code**	**Decompiled source code**
`void main() {`	10684: save %sp, -112, %sp	int main(int argc, char **argv, char **envp) {

`int a, x;`		int local17; // argc{37}
		int local18; // argc{73} // "old a"
		int local19; // local18{73} // a

`a = 0;` `do {`	10688: clr %o0 1068c: sethi %hi(0x10400),	argc = 0; // Compiler reuses argc
`a = a+1;`	%l0	for a
`x = a;`	10690: add %o0, 1, %i0	local19 = argc;
`printf("%d`	10694: or %l0, 872, %o0	do {
`", a);`	10698: call printf	local18 = local19;

```
                1069c: mov    %i0, %o1              printf("%d ", local18 + 1);

 } while (a < 106a0: cmp    %i0, 9                local17 = local18 + 1;
10);            106a4: ble    0x10690               local19 = local17;
                106a8: mov    %i0, %o0            } while (local18 + 1 <= 9);
 printf("a is 106ac: sethi    %hi(0x10400),       printf("a is %d, x is %d\n", local18 + 1,
%d, x is        %g1                             local18 + 1);
%d\n", a, x); 106b0: mov    %i0, %o1
                106b4: mov    %i0, %o2
                106b8: call   printf
                106bc: or     %g1, 880, %o0
 return 0;                                        return 0;
}               106c0: ret                      }
                106c4: restore  %g0, 0, %o0
```

1 - Ukážka dekompilovaného kódu (Boomerang Decompiler) [4]

2.3 Ochrana pred spätným inžinierstvom

Anti-disassembling je technika zahmlievania binárneho súboru aby, bol odolnejší proti útočníkom. Dôvody pre ochranu proti disassemblingu sú rôzne. Môže sa jednať o ochranu intelektuálneho vlastníctva pred pirátmi, ale aj o protekciu malwaru, či spywaru pred antivírovými programami.

Proti dynamickému reverznému inžinierstvu existuje veľa druhov ochrany. Základný princíp spočíva v tom, že skúmaný program zistí, že je pripojený ku kontrolnému procesu a na základe toho sa zachová inak. Prísť na to môže viacerými spôsobmi:

- Microsoft API - volania funkcií IsDebuggerPresent, CheckRemoteDebuggerPresent [5]
- Názvu okna – skúmaný program zistí, aké programy sú v systéme pootvárane a hľadá debuger, ktorý pozná. Tento systém má tú nevýhodu, že umožňuje ochranu len proti určitým debugerom, ktorých meno má program v databáze. [5]

- Kontrola časomiery – samotná réžia kontrolného programu zaberá čas počas behu skúmaného programu. Ten si môže všimnúť, že jeho beh je pomalší ako normálne. [5]
- Breakpointy – breakpoint je miesto v programe, kde sa odovzdáva riadenie debugeru pomocou vyvolania výnimky. Pokiaľ je program už pripojený k debugeru výnimka sa nevyvolá [6].

Pri ochrane proti statickému skúmaniu aplikácie sa používajú rôzne systémy na oklamanie disassembleru:

- Upravenie hlavičky oddielu – zdvojenie, nastavenie neobvyklej veľkosti, atď. [7]
- Samomodifikujúci kód – dáta sa generujú do oblasti, pre spustiteľný kód až za behu programu. Disassembler ich tak nemôže objaviť. [8]
- Skok doprostred inštrukcie – adresa skoku sa nastaví tak, aby nebola zarovnaná na dĺžku inštrukcie, ale skočila do jej časti. Bity pred skokom sa tak vôbec nevykonajú. S čím mnoho disassemblerov nepočíta, pretože prekladač by to nikdy neurobil, a vytvára tak nezmyselné posunuté výsledky. [9]
- Nikdy neuskutočnený skok – v kóde sa uvedie skok, ktorý sa za behu nikdy nevykoná na posunutú adresu. DA, ktorý dokáže skákať doprostred inštrukcie, preloží nasledujúce bity na základe posunutých adries a potom ich už ďalší krát neprechádza. [5]

2.4 Programy na spätnú transformáciu kódu

Reverzná transformácia kódu sa v informatike využíva skoro tak dlho ako je obor starý. Za ten čas už vzniklo množstvo úspešných i menej úspešných programov. Uvediem aspoň tie najznámejšie s nich. Viac je ich uvedených napr. v [10]. Disassemblery:

- IDA – v súčasnej dobe asi najznámejší a najpoužívanejší interaktívny disassembler a debuger. Jedná sa o komerčný projekt vytvorený Ilfakom Guilfanovom. Aktuálna verzia je 6.1, staršie sú k dispozícií zdarma na nekomerčné používanie. Podporuje radu procesorov, grafickú reprezentáciu kódu, je modulovateľný a použiteľný na MS-Windows, Linuxe i Mac OS X. [11]

- objdump – linuxová aplikácia na zobrazovanie rôznych informácií o binárnych súboroch. Použiteľná ako jednoduchý disassembler . Nezvláda prácu so zahmlenými (anti-disassemblingovanými) súbormi.
- Lida – Linux Interactive Disassembler – pokročilý linuxový DA. Pracuje aj so súbormi s poškodenými hlavičkami alebo aj bez nich. Zvláda základnú kryptoanalýzu. Voľne šíriteľný. [7]

Dekompilerov nie je zďaleka tak veľa ako disassemblerov. To hlavne preto, že ich konštrukcia je omnoho zložitejšia a každý dekompiler je zároveň aj DA. Väčšinou sa vyskytujú dekopilery pre high level objektové jazyky, ktoré zanechávajú množstvo debugovacích informácií.

- Boomerang – multiplatformný open sourcový dekompiler do jazyka C. Pôvodne veľmi zaujímavý a ambiciózny projekt – no od roku 2006 nie sú žiadne správy o jeho vývoji. [4]
- Mocha – pravdepodobne najstarší dekompiler dostupný pre jazyk Java (Java bola predstavená v Máji 1995, Mocha v Júni 1996). Voľne stiahnuteľný. Vyvinutý Holanďanom Hanpeterom van Vlietom. Mocha na svoju dobu vzbudila značnú kontroverziu a bola autorom dočasne stiahnutá z obehu. Neskôr sa tam vrátila spolu so zatemňovačom kódu zvaným Crema. Problém s Java Dekompilátormi je ten, že kým prekladače väčšiny jazykov zanechávajú minimum informácií o zdrojovom kóde, u Java sa dá zreprodukovať skoro v pôvodnej verzií. Čo prakticky spôsobuje, že ako náhle uvediete na trh binárku Javy, pre človeka s dekompilátorom ste otvorili váš zdrojový kód. [12][13] Ďalšie Javové dekompilátory sú napr. Jad, DJ, JODE, atď.
- .NET Reflector – prvý a najrozšírenejší dekompilátor pre platformu .NET. Vyvinutý Lutzom Roederom. Od roku 2008 pod vývojom Red Gate Software, ktorý v februári tohto roku (2011) vyhlásili pôvodne bezplatný produkt za platený. Ako reakcia na tento ťah vznikli bezplatné alternatívy – ILSpy, JustCompile... [14]

Debuggery slúžia na sledovanie programu za behu. Klasický debugger, slúžiaci na hľadanie chýb vo vlastnom kóde, funguje len pri programe preloženom s debuggovacími informáciami.

- GDB – GNU Debugger – štandardný nástroj pre hľadanie chýb v GNU software. Podporuje mnoho unixových operačných systémov a programovacích jazykov (C, Ada,

Pascal...). GDB poskytuje možnosť tzv. vzdialeného debugovania – používanú na prácu s embedded systémami. Avšak neposkytuje grafické užívateľské rozhranie, to obsahuje až jeho grafická nadstavba DDD, prípadne Insight. [15]

- Olly Debugger – najrozšírenejší debugger pre systémy Windows. Sleduje registre, procedúry, volania API, zdieľané knižnice. Dovoľuje užívateľsky definované názvy návestí, popisy funkcí a komentáre. V súčasnej dobe je vo verzií 2.01. [16]

Tam kde existujú programy na reverzné inžinierstvo, tam sa zákonite musí nájsť aj ich protipól tzv. zatemňovače kódu – obfuskátory. Tie implementujú rozličné techniky anti-disassemblingu.

- Crema – aplikácia pre zneprehľadnenie kódy v Java. Rovnako ako Mocha vyvinutá Hanpeterom van Vlietom. Van Vliet použil Cremu najprv na skrytie kódu Mochy, potom, kde Mocha vyvolala búrlivú diskusiu o reverznom inžinierstve, ju uvoľnil pre verejné používanie. Neplatená verzia zatemňovača ale vkladala neplatné identifikátory, ktoré neprejdú bezpečnostnými kontrolami v mnohých prehliadačoch. Javovské obfuskátory ako Crema, alebo Hashjava apod. neznemožňujú spätný preklad do zdrojového kódu, ale menia názvy premenných a funkcií (na nič nehovoriace reťazce čísel) aby bolo pre útočníka komplikovanejšie sa vyznať v získanom kóde. [17] [13]
- Salamander .NET Obfucator – zatemňovač pre C#, VB.NET, C++.NET, J#, MSIL, atď. [18]
- Voormedia HTML Scrambler [19] – ukážka toho, že zatemňovacie techniky sa používajú nielen v nízkoúrovňových jazykoch.

2.4.1 Projekt Lissom

Lissom [20] je projekt, ktorý v súčasnosti beží na Fakulte informačných technológií VUT v Brne. Jeho cieľom je návrh a implementácia jazyka ISAC, slúžiaceho na popis mikroprocesorovej architektúry. Jazyk by mal napomôcť k zvýšenie modelovacie schopnosti aplikačne špecifických mikroprocesorov.

V rámci projektu sa vyvíja aj univerzálny assembler a disassembler, kompilátor a dekompilátor jazyka C.

3 Procesory a súborové formáty

Procesor (Central Processing Unit – CPU) je základná súčasť počítačového systému, ktorá vykonáva inštrukcie počítačového programu. CPU spracúva postupne každú inštrukciu na aritmeticko – logickom základe. Pojem centrálna procesorová jednotka sa v informatike používa od 60tich rokov 20. storočia [21]. Dizajn procesorov sa od vtedy rapídne zmenil, ale fundamentálne princípy ostávajú rovnaké. Na základe typu inštrukčnej sady môžeme CPU rozdeliť na dve základné kategórie.

- RISC – reduced instruction set computer (redukovaná inštrukčná sada) – obsahuje hlavne jednoduché inštrukcie. Dĺžka vykonávanie jednej inštrukcie je vždy jeden cyklus. Dĺžka (počet bitov) inštrukcií je vždy rovnaká. Mikroinštrukcie sú hardwarovo implementované na procesor, čím je výrazne zvýšená rýchlosť ich vykonávania. V súčasnosti sa využíva napr. v mikroprocesoroch ARM, ktoré sú implementované v mobilných telefónoch, smartphonoch, či iPodoch.
- CISC – complex instruction set computer (kompletná inštrukčná sada) – inštrukcií je relatívne veľa a majú premennú dĺžku i počet vykonávacích cyklov. V tomto prípade procesor v podstate obsahuje miniatúrny počítač vrátane programov. Tento počítač potom vyhodnocuje jednotlivé inštrukcie. Výhodou je, že jedna inštrukcia je schopná zaistiť komplexné služby. Najvýznamnejším reprezentantom tejto triedy mikroprocesorov je Intel x86.

Podľa umiestnenia inštrukcií tj. programového kódu a dát sa procesory delia na:

- Von Neumannovská architektúra – Dáta i inštrukcie sú v rovnakej pamäti. Umožňuje modifikovať inštrukcie za behu programu. Nazvaná podľa matematika Johna von Neumanna, ktorý navrhol prvý počítač s touto architektúrou - Edvac.
- Harvardská architektúra – Program a dáta sú vo fyzicky oddelených pamätiach. Dvojo pamätí dovoľuje paralelný prístup k obom naraz, čo zvyšuje rýchlosť spracovania. Názov vychádza z počítača Harvard Mark I, postavenom na tejto architektúre.

V moderných PC sa väčšinou používa kombinovaný prístup. Počítač navonok (tj. aj s pohľadu programátora) pôsobí ako Von Neumannovský koncept, ale vo vnútri procesoru je použitá Harvardská architektúra (registre pre dáta sú oddelené od inštrukčných). U mikroprocesorov sa používajú obidva prístupy.

Ďalej sa architektúry procesorov delia na big-endianové a little-endianové.

- Big-endian – vyznačuje sa tým, že pri ukladaní viacbytových dát sa významnejšie byty uložia na nižšiu adresu. Bity sú v jednom byte zoraradené tak, že najvýznamnejší (najľavejší) je 7. bit a najviac v pravo je umiestnený najmenej významný 0. bit. Pri grafickom zobrazovaní pamäte big-endianového systému je najvyššie umiestnená nultá adresa a smerom nadol adresa narastá.
- Little-endian – funguje opačne ako big-endian. Významnejšie byty sa ukladajú na vyššiu adresu. Pamäť sa zobrazuje od najvyšších adries po najnižšie. Usporiadanie bitov v byte je rovnaké.

Little-endian sa používa u procesorov Intelu ako x86, či 8051. Big-endian využíva napr. Motorola alebo IBM. V minulosti sa používal i tzv. middle endian kde sa ukladanie bytov začínalo od stredu. Niektoré architektúry (ARM, PowerPC...) umožňujú prepínať medzi jednotlivými prístupmi.

S hľadiska výstavby disassembleru je dôležitá tak architektúra procesoru ako aj operačný systém, použitý súborový formát, atď. U procesorov osobných počítačov sa bližšie pozrieme na architektúru EXE formátu u Microsoft Windows a ELF formátu systému Linux a predstavíme si aj mikroprocesorové textové formáty SREC a HEX.

3.1 Architektúra mikrokontrolérov

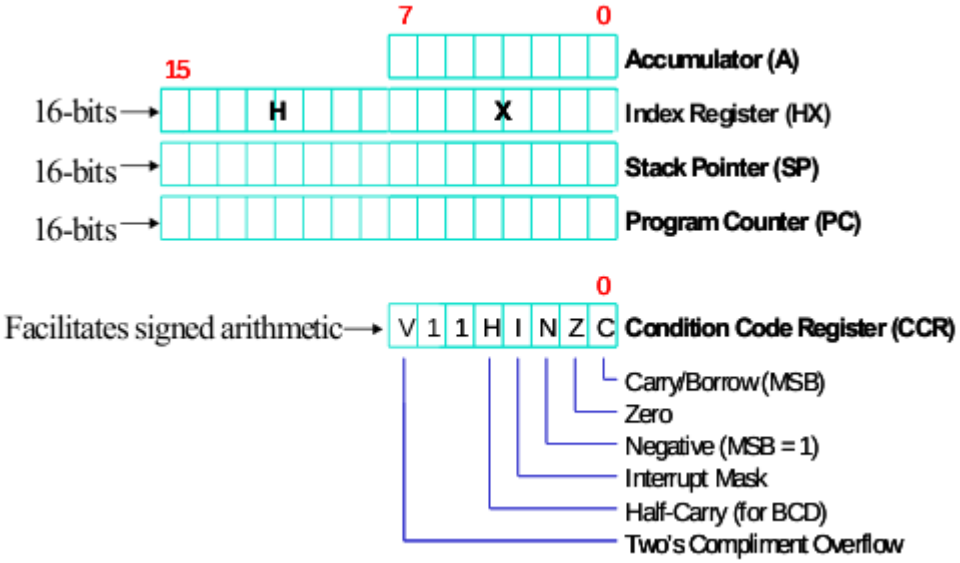
Mikroprocesor (uP) je základná procesorová jednotka (CPU) na čipe. Komponenty CPU obsahujú aritmeticko – logickú jednotku (ALU), inštrukčný dekodér, registre, radič, atď. Radič riadi chod celého mikroprocesoru, je tvorený registrom inštrukcií, obvodom pre dekódovanie inštrukcie a riadiacim obvodom. Súhrne sa to nazýva riadiaca časť.

Po pridaní vonkajšej operačnej pamäte a periférnych vstupne/výstupných jednotiek získavame počítač. Mikropočítač umiestnený na jednom čipe nazývame mikrokontrolér. Obvykle sa mikrokontroléry používajú v zabudovaných zariadeniach (embedded systems) ako napr. mp3-prehrávače, mikrovlnky, práčky, semafóry, atď.

3.1.1 Motorola 68HC08

Jednočipové mikrokontroléry Motorola (v súčasnosti Freescale) rady 68HC08 sú predstaviteľmi kategórie, jednoduchých 8-bitových mikrokontrolérov. Nadväzujú na radu HC05, s ktorou sú kompatibilné na úrovni zdrojového kódu (smerom hore). Štruktúra jadra vychádza z historického mikroprocesoru 6800 a staršieho jednočipového mikropočítača 6801. Jadrom kontroléru je mikroprocesor CPU08, oproti mikropočítaču 6801 bol zjednodušený o druhý strádač, no na druhej strane je CPU08 doplnené o inštrukcie pre prácu s bity v pamäti a v registroch periférií. Príbuznými mikrokontrolérmi sú 16-bitové 68HC16 a 68HC12, resp. 32-bitové procesory M680020 a M680040. [23][22]

Spomínaný mikroprocesor má architektúru von Neumannovho typu, tj. dáta i program sú umiestnené v rovnakom pamäťovom priestore. Inštrukčná sada je typu CISC. Procesor je vo formáte big-endian. Ako je vidieť na obrázku 2, CPU08 obsahuje minimum registrov. Jedná sa o univerzálny strádač (akumulátor), indexovací register (upravuje adresu operandu – obvykle pri vektorových operáciách), ukazateľ zásobníku, programový čítač (obsahuje návratové adresy pri skokoch a prerušeniach) a príznakový register. Vyznačuje sa rýchlim prístupom do pamäte, pomocou 16tich adresovacích módov. Hardwarovo podporuje násobenie i delenie. Zvyčajne (aj pri MC68HC908LJ12, použitom pri DA v tejto práci) sa na mikrokontroléry HC08 (HC908) využíva pamäť FLASH, umožňujúca rýchly zápis údajov. [22]



2 - Programovací model registrov u HC08

3.1.2 Intel MCS-51

Firma Intel v roku 1980 predstavila prvého predstaviteľa rady MCS-51, označovanej tiež ako 8051. Jedná sa o veľmi úspešnú a rozšírenú rodinu mikrokontrolérov, ktorú dodnes vyrába hromada výrobcov - okrem Intelu napríklad Atmel, Philips, AMD, Infeon.

Na rozdiel od HC08, sa jedná o little-endian systém, ktorý má modifikovanú Harvardskú architektúru, so sadou 111tich RISC inštrukcií. Väčšina s týchto inštrukcií sa vykonáva v jednom cykle. Okrem základných 16-bitových registrov, má aj univerzálne 8-bitové registre R0 - R7, sadu bitovo adresovateľných registrov a registre periférii. Všetky sú namapované do priestoru dátovej pamäte.

3.2 Formát binárnych súborov

Súborové formáty zvyčajne delíme na dva základné typy – textové a binárne. Pri textových sa binárne dáta automaticky interpretujú ako ascii znaky. Binárne sa ďalej delia na zvukové, bit mapy, spustiteľné... Existuje viacero metód ako operačný systém zistí o aký typ súboru sa jedná. Medzi najpoužívanejšie patria:

- Prípona súboru – spôsob odlišovania súborov, používané na systémoch Windows, MS-DOS, či Mac OS X. Určuje formát súboru podľa časti jeho názvu nasledujúcej po poslednej bodke. V pôvodnom FAT systéme bolo meno súboru limitované na 8 znakov a prípona na tri. Preto aj dnes je väčšina súborových prípon trojznakových.
- Hlavička súbory – väčšina binárnych formátov má svoju hlavičku umiestnenú na začiatku súboru (nemusí byť pravidlo) kde sa uchovávajú informácie o tomto súboru. V prvom rade typ súboru, pomocou tzv. magického čísla, čo je sekvencia bitov identifikujúca daná typ. Napr. u gif obrázkov je to (v ascii) GIF87a alebo GIF89a, u jpg JFIF...
- Explicitné metadáta – tie sa nachádzajú v súborovom systéme oddelene od súboru a jeho názvu. Môžu sa nachádzať v súbore, ktorý je priložený k danému súboru. To by mohlo viesť k tomu, že súbor s metadátami, potrebuje súbor s metadami[2] a k zacykleniu celého systému. Preto je to riešené, tak, že súbor s dátami a s metadámi sa spolu zabalia do jedného archívu (to zároveň odstraňuje aj problém straty metadát napr. pri posielaní cez sieť). Tento spôsob napríklad využíva OpenOffice pri svojich OpenDocumentoch. [10]

Spustiteľné súbory sú väčšinou binárneho charakteru. Existuje ale obrovská varianta rôznych formátov pre rôzne procesory a operačné systémy.

3.2.1 .EXE

Prípona spustiteľných súborov používaných na MS-DOS, Windows a OS/2. Vychádza s anglického EXEcutable (spustiteľný). Existuje viacero formátov tohto súboru. Rozlišujú sa podľa magického čísla (prvých dvoch bajtov súboru).

- MZ – starý 16 bitový MS-DOSový formát. Podľa iniciálok Marka Zbikowského - jedného z vývojárov operačného systému MS-DOS
- NE – new executable – stále 16 bitový formát, ktorý môže bežať na MS-DOSe 4.0 a vyššie ale aj na 32bitových windowsoch a OS/2.
- LE – linear executable – 16/32 bitov, Windows
- LX – linear executable – 32 bitov, OS/2
- PE – portable executable – súčasná verzia windowsových binárok (od NT 3.1). Vychádza s formátu COFF. Oproti nemu lepšiu podporu zdieľaných knižníc a rozšírené možnosti ukladania debugovacích informácií. Skladá sa s mnohých hlavičiek a sekcií, ktoré vravia dynamickému linkeru ako namapovať súbor do fyzickej pamäte. Typickým príkladmi sú sekcie *.text* a *.data*. Sekcia *.text* je read-only a obsahuje programovú časť súboru (spustiteľné dáta), kým sekcia *.data* obsahuje dátovú zložku a umožňuje i zápis.
- PE+ (64 bit) – verzia PE pre 64 bitové windowsy na platformách x86-64 a IA-64.

3.2.2 a.out

a.out je súborový formát používaný v starších verziách unixových operačných systémov. Názov vychádza z Assembler OUTput. Formát a.out obsahuje do 7 sekcií. [25]

- hlavičku exec – jediná povinná sekcia. Obsahuje údaje pre kernel o načítaní a spustení súboru.
- text segment
- data segment

- relokačnú tabuľku pre textový segment – upravuje ukazovatele kódovej časti pri kombinácií viacerých súborov
- relokačnú tabuľku pre dátové ukazovatele
- tabuľku symbolov – obsahuje adresy pomenovaných adries a funkcií
- tabuľku názvov – uchováva názvy symbolov

Jedná sa o základný a veľmi jednoduchý dizajn, poskytujúci nedostatočnú podporu pre zdieľané knižnice. Dlho sa udržal najmä v prostredí FreeBSD. No postupne bol nahradený formátom COFF resp. ELF.

3.2.3 COFF

Nástupný formát po a.out, vytvorený v laboratóriách AT&T. Sú z neho odvodené formáty ECOFF, XCOFF a v neposlednej rade Microsoftový PE. Pracuje už so zdieľanými knižnicami, ale nie s podporou pozične nezávislého kódu. Pokiaľ sa zdieľaný kód nepodarí umiestniť na preferovanú adresu v pamäti (je obsadená iným programom), loader musí prepočítať každú absolútnu adresu pridaním tzv. delta hodnoty. To značne spomaľuje naťahovanie aplikácie. Táto nevýhoda (oproti ELF) je ďalej prenesená aj na PE. Medzi jeho nevýhody patrí obmedzené množstvo sekcií, krátka maximálna dĺžka mien pre segmenty a nedostatočné možnosti umiestnenia debugovacích informácií pre moderné programovacie jazyky.

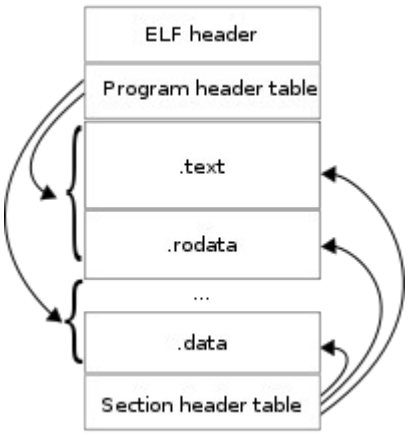
3.2.4 Formát ELF

ELF (Executable and Linkable Format) sa využíva hlavne na linux-like a unix-like operačných systémoch ale i niektorých embedded zariadeniach. V roku 1999 bol komisiou 86open schválený ako štandardný binárny formát pre unixové systémy na platforme x86. Na rozdiel od COFFu a PE formátu podporuje plne pozične nezávislý kód (position independent code) a obsahuje globálnu tabuľku skokov (global offset table - GOT).

GOT je v elf súbore umiestnený v samostatne sekcie zvanej *.got*. Táto tabuľka je privátna pre každý proces a ukazuje na miesto v fyzickej pamäti, kde sú uložené adresy zdieľaných funkcií a premenných. [24]

Ako ukazuje nasledujúci obrázok, súbor sa podobne ako PE skladá z hlavičiek, ktoré sú na začiatku, vo vnútry i na konci súboru. Ďalej môže obsahovať jednu alebo viac sekcii *.text* a *.data*.

Sekcia *.text* obsahuje procesorové inštrukcie. *.data* obsahuje rezervované premenné. Pokiaľ v programe priradíme premennej hodnotu, tu sa prejaví zmena. [26]



3 – štruktúra ELF súboru

3.2.5 Formát SREC

Motorola S-record formát [27], tiež zvaný S19 alebo SREC bol vyvinutý v 70.rokoch ako ASCII formát pre spustiteľný kód pre procesor 6800. Názov vychádza z S-record 1 až 9 (možný typ záznamu v jednotlivom riadku). V súčasnosti sa používa u širokej rodiny mikroprocesorov, medzi inými i u HC08. Tento súbor sa pomocou PROM programátoru vkóduje priamo na mikrokontrolér. SREC pozostáva so sérii ascii záznamov. Záznam má nasledovnú štruktúru:

1. *Písmeno S* – start record
2. *Typ záznamu* – jedna číslica 0-9. Definuje typ dát.

Záznam	Popis	Počet bytov adresy
S0	Hlavička	2
S1	Dáta	2
S2	Dáta	3
S3	Dáta	4
S5	Počet záznamov	2
S7	Koniec bloku	4
S8	Koniec bloku	3
S9	Koniec bloku	2

3. *Veľkosť záznamu* – dve hexadecimálne číslice označujúce počet bajtov zvyšku záznamu. Jeden bajt pozostáva z dvoch hexadecimálnych číslic.

4. *Adresa* – umiestnenie prvého dátového bajtu na danom riadku súboru .s19 v pamäti mikrokontroléru.
5. *Data* – samotné operačné inštrukcie a operandy.
6. Checksum – kontrolný súčet. Posledný bajt komplementu ku súčtu veľkosti, adresy a dát. Slúži na kontrolu integrity dát pri prenose

Prvý záznam (S0) zvyčajne obsahuje komentár ako meno programu alebo verziu. Posledný záznam (S7, S8, S9) môže obsahovať počiatočnú adresu (pokiaľ je odlišná od 0x0).

```
S00F000068656C6C6F202020202000003C
S11F00007C0802A6900100049421FFF07C6C1B787C8C23783C6000003863000026
S11F001C4BFFFFE5398000007D83637880010014382100107C0803A64E800020E9
S111003848656C6C6F20776F726C642E0A0042
S5030003F9
S9030000FC
```

Start code
Typ záznamu
Počet bytov v riadku
Adresa
Dáta
Kontrolný súčet

4 – príklad súboru S19 [28]

3.2.6 Intel HEX

HEX [29] je súborový formát veľmi podobný SRECu. Rovnako ako on je textový a používa sa pre embedded systémy. Každý záznam začína značkou 0x03AH tj. v ASCII kóde znak ':'.

```
:10010000214601360121470136007EFE09D219040
:10011000146017EB7C20001FF5F16002148011988
:10012000194E79234623965778239EDA3F01B2CAA7
:100130003F0156702B5E712B722B732146013421C7
:00000001FF
```

Start code
Počet bytov v riadku
Adresa
Typ záznamu

Dáta
Kontrolný súčet

5 – príklad HEX súboru

4 Návrh a implementácia interaktívneho disassembleru

Jedným s cieľov tejto práce je návrh a implementácia disassembleru. Bol vytvorený interaktívny DA nesúci názov Baida (BAchelor Interactive DisAssembler). Program je implementovaný v jazyku C/C++, presnejšie v C++ knižnici Qt. Je modulovateľný prostredníctvom pluginov, ktoré implementujú rôzne parsovacie mechanizmy a architektúry spustiteľných súboru, a XML – súborov, ktoré implementujú rôzne inštrukčné sady.

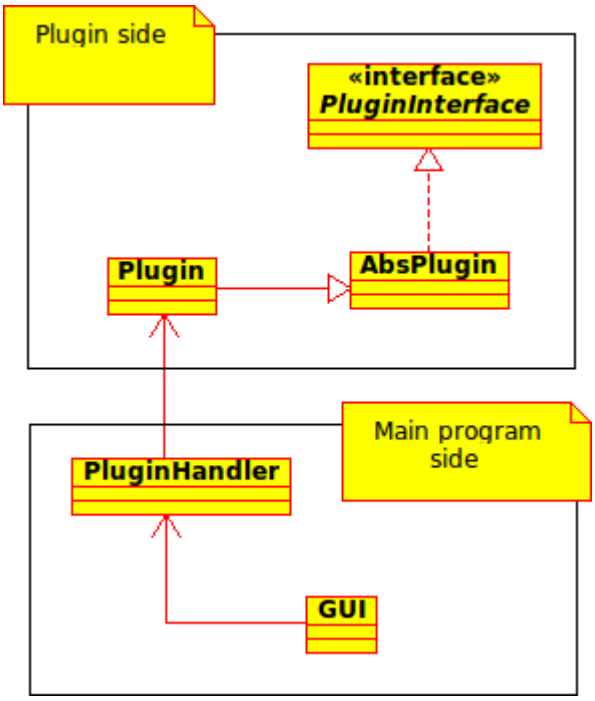
4.1 Požiadavky na program

DA by mal plniť nasledovné požiadavky.

- Mal by poskytovať komfortné užívateľské rozhranie
- Mal by fungovať na viacerých platformách – aspoň Linux a Windows
- Mal by umožňovať použitie rôznych parsovacích algoritmov, podporovať rozličné architektúry, procesory, formáty spustiteľných súborov a inštrukčné sady.
- Mal by obsahovať prvky interaktivity u DA. Konkrétne umožňovať editáciu rozloženého programu, pridávanie komentárov kódu, pomenovávať a premenovávať návestia skoku, zvýrazňovať vybrané inštrukcie v binárnom súbore atď.
- Mal by zahŕňať nejakú modifikáciu v prístupe k spätnému prekladu. V tomto prípade bol implementovaný viacvláknový parser, umožňujúci simultánny preklad vo viacerých procesoch.

4.2 Návrh a štruktúra

Program je z hľadiska štruktúry rozdelený na dve základné časti. Main Program – Gui, ktorý sa stará o zobrazenie užívateľského rozhrania, grafický výstup, otváranie dialógových okien a zapojenie jednotlivých zásuvných modulov. A zásuvný modul – Plugin, kde sa odohráva celý výpočet, spätný preklad do assembleru. Do hlavného programu sa odosielajú len štruktúrované dáta.

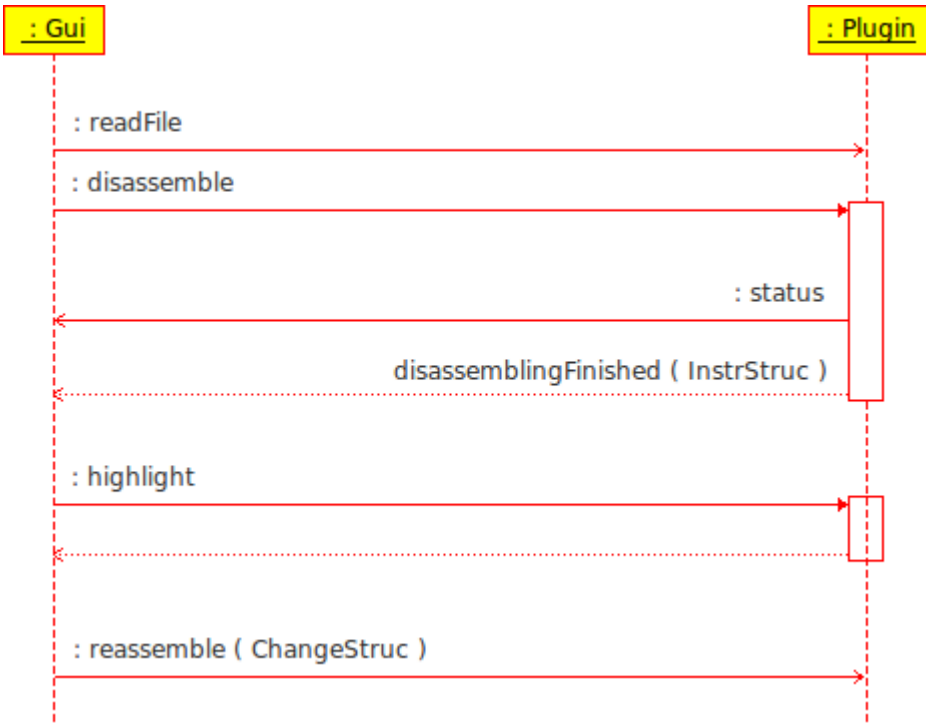


6 – Základná štruktúra aplikácie

Načítanie a uloženie spustiteľných súborov prebiehajú v časti zásuvného modulu. Je to s toho dôvodu, že v závislosti na použitom module sa číta buď binárny súbor (EXE, ELF...) alebo textový súbor (SREC, HEX...). A tie si už daný modul spracuje podľa špecifík vybranej architektúry. To umožňuje široko spektrálnu modularitu programu. Textový výstup assemblerovského kódu, zaobstaráva priamo hlavná časť – potrebné informácie má už z pluginu vytiahnuté.

4.2.1 Komunikačné rozhranie zásuvných modulov

Tieto dve časti spolu komunikujú pomocou protokolu ktorý pozostáva s virtuálnej triedy *AbsPlugin (ABStract Plugin)*, ktorá implementuje rozhranie *PluginInterface*. Každý zásuvný modul musí implementovať túto virtuálnu triedu. Dôvod prečo je použitá virtuálna trieda a nielen čisté rozhranie je ten, že komunikačný protokol Gui – Plugin využíva signály a sloty knižnice Qt. A tie sa pri komunikácií so zásuvným modulom dajú implementovať len s použitím abstraktnej triedy.



7 - Príklad komunikácie medzi modulmi

Gui a Plugin medzi sebou posielajú dva základné druhy dát. *InstrStruc* je štruktúra, ktorá sa predáva smerom s disassemblovacieho pluginu do GUI po dokončení spätného prekladu. Obsahuje riadky assemblerovského kódu. Každý riadok má adresu, názov inštrukcie, operandy, typ inštrukcie (pre farebné zvýraznenie syntaxe), prípadne meno návestia, skokovú adresu a komentár (pokiaľ by binárka obsahovala debugovacie informácie vrátane komentárov). V súčasnej verzií nie je implantované zvýraznenie syntaxe, ani zobrazenie skokovej adresy.

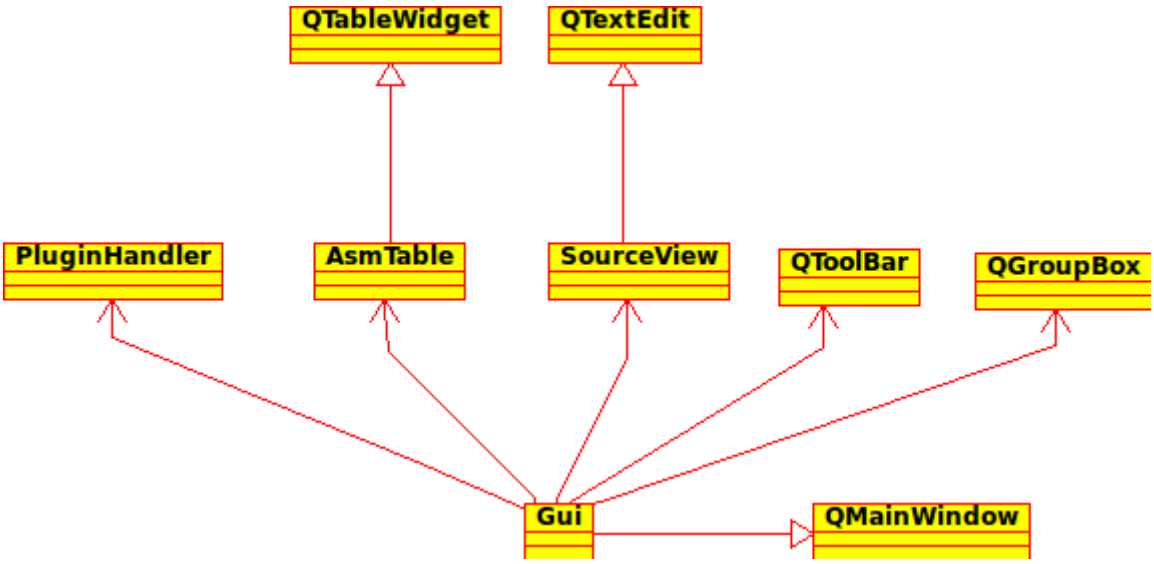
V prípade editácie súboru sa z hlavnej časti posiela vektor *ChangeStruct*, obsahujúci adresu a inštrukciu s operandom upravených riadkov.

Ďalej sa smerom z hlavného programu posielajú názvy súborov a koordináty označených buniek, slúžiace na zvýraznenie označených buniek v binárnom kóde. Opačným smerom sa odosiela obsah binárneho súboru v html formáte na zobrazenie a status, či prípadné chybové hlásenia prekladu.

4.3 Gui

Hlavná časť programu zastrešuje grafické užívateľské rozhranie a triedu pre integráciu pluginov *PluginHandler*. Základom je trieda *Gui*, ktorá dedí *QMainWindow* a obsahuje všetky ostatné widgety. Tá sa zobrazí po spustení programu.

PluginHandler uskladňuje všetky načítané zásuvné moduly. Preposiela príkazy aktuálne zvolenému pluginu a príjma od neho dáta. Pluginy interpretuje ako *AbsPlugin*.



8 - Diagram tried pre hlavný modul

Ostatné časti modulu sa zameriavajú na zobrazenie gui. Základným ovládacím prvkom je *ToolBar*. Obsahuje ikony *Disassemble* (preloží načítaný súbor do assembleru), *Load Binary* (načíta spustiteľný súbor), *Save Binary* (uloží reassemblovaný editovaný spustiteľný súbor) a *Save Text* (uloží assemblerovský kód programu ako textový .asm súbor).

9 - Ovládacia lišta

O výber zásuvného modulu sa stará *GroupBox* . Obsahuje informácie o vybranom module a načítanom súbore.

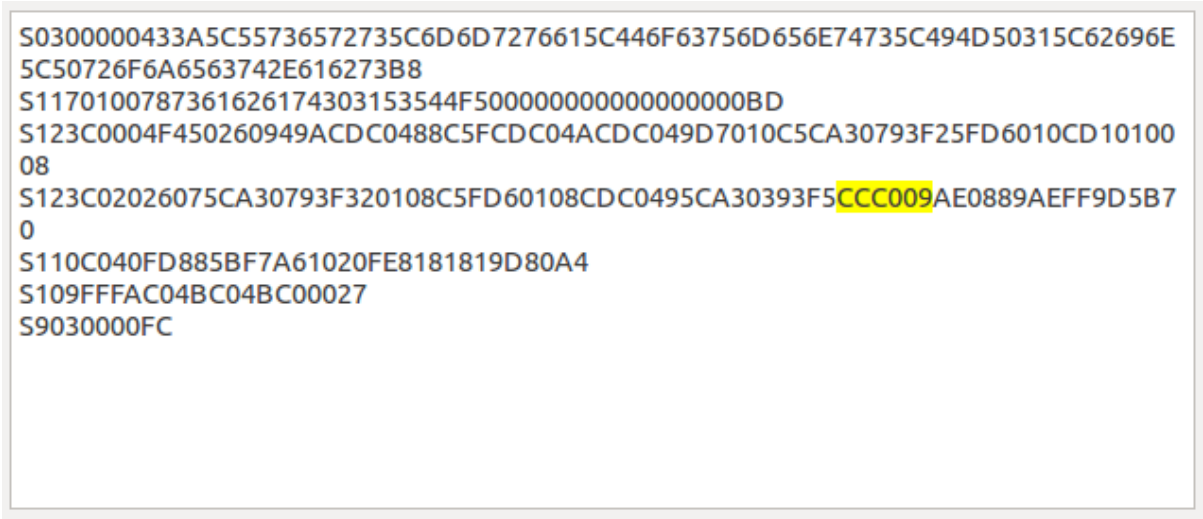
10 - GroupBox

Na zobrazenie výsledného asm kódu Baida používa *AsmTable*. Ten je potomkom *QTableWidget.* Umožňuje zobraziť adresu, návestie, meno inštrukcie, operandy a komentár. Všetko okrem adresy dovoľuje i editovať.

	Address	Label	Instruction	Parameters	Comment
1	0xC000	_Startup	CLRA		
2	0xC001		LDHX	0x0260	
3	0xC004		TXS		
4	0xC005		CLI		
5	0xC006		JSR	[0xC048]	
6	0xC009	FROM0xC036_	CLRH		
7	0xC00A		CLRX		
8	0xC00B	FROM0xC017_	JSR	[0xC04A]	Volanie funkcie do_something()
9	0xC00E		JSR	[0xC049]	
10	0xC011		STA	off[0x010C]	
11	0xC014		INCX		
12	0xC015		CPX	0x07	
13	0xC017		BLE	rel[0xF2]	

11 - Zobrazovač zdrojového kódu – AsmTable

Nasledujúcim zobrazovacím nástrojom je *SourceView*, ktorý slúži na zobrazovanie obsahu zdrojové binárneho súboru. Dáta, ktoré sa tu zobrazujú posielajú jednotlivé pluginy v html formáte. To dovoľuje zásuvným modulom implementovať vlastný formát zobrazovania binárky. Protokol zásuvných modulov umožňuje zvýrazňovať položky vybrané v *AsmTable*. *SourceView* je implementovaný ako *QTextEdit*.



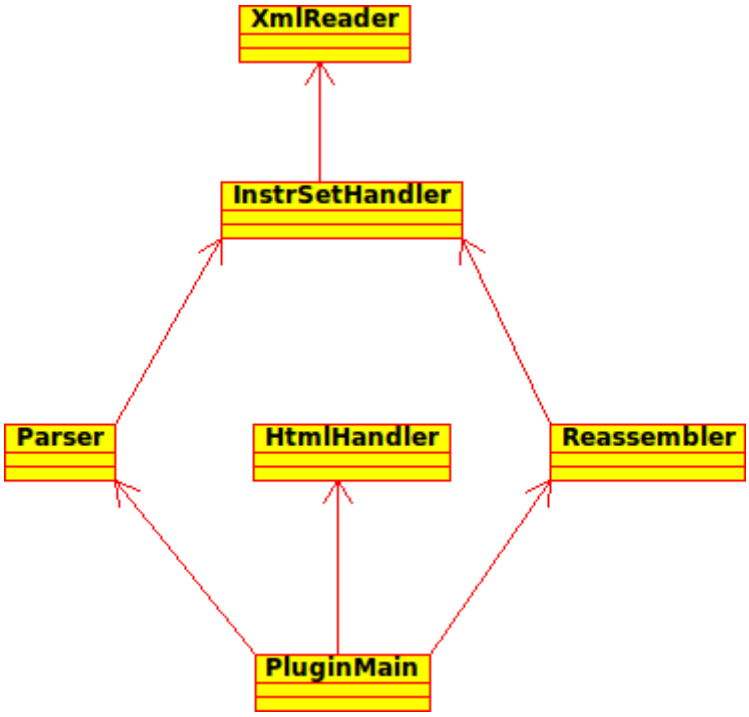
12 - Zobrazovač obsahu binárneho súboru

Užívateľské rozhranie dopĺňa statusbar, na ktorom sa ukazujú informačné a chybové hlásenia. A menubar, poskytujúci alternatívnu možnosť ovládania programu.

4.4 Zásuvné moduly

Zásuvné moduly slúžia na vykonávanie spätného prekladu do asm kódu, reassemblovanie upraveného kódu do binárky a zobrazovanie binárneho kódu. Modul môže byť naprogramovaný pre preklad hocijakej architektúry alebo typu súboru .

Všetky pluginy, ktoré boli v rámci tejto bakalárskej práce implementované, spája spoločná základná štruktúra. Určité triedy nie sú v niektorých pluginoch použité, iné môžu byť rozšírené. Viac si o tom povieme pri jednotlivých parsroch.



13 - Základná štruktúra pluginu

Trieda *PluginMain* sa stará o komunikáciu s hlavným modulom a slúži ako základ pre ostatné triedy. Musí implementovať abstraktnú supertriedu *AbsPlugin*.

V prípade priatia požiadavku na rozklad súboru zavolá *Parser*, ktorý obsahuje disassemblovací algoritmus. *Parser* si vytiahne dáta o inštrukčnej sade z *InstrSetHandleru*, ktorý ich uskladňuje. Získal ich cez *XmlReader* z XML súboru. XML súbor obsahuje nasledovné podrobnosti o jednotlivých inštrukciách.

- Názov (mnemonic) – napr. ADD
- opkód (opcode) – bitový formát inštrukcie. napr. AB
- typ operandu – informácie o formáte a dĺžke nasledujúceho operandu
- typ inštrukcie – v tomto prípade sa rozlišujú 3 základné typy inštrukcií: branch, return a default. Táto informácia umožňuje parsru sa orientovať vo vetvení binárky.
- krátky popis inštrukcie – napr. „Add with Carry“

Podobne trieda *Reassembler* sa stará o rekonštrukciu binárneho súboru po úpravách. Tiež používa dáta s triedy *InstrSetHandler*.

HtmlHandler zabezpečuje zobrazovanie obsahu binárneho súboru. Na gui úrovni sa tieto dáta ukazujú v *SourceView*.

4.4.1 Použitá inštrukčná sada

Na demonštráciu funkčnosti programu a pluginov bola použitá inštrukčná sada MC68HC908LJ12. Jedná sa o inštrukčnú sadu mikrokontroléru HC08 (viď kapitolu 3.1.1) typu CISC. Binárne dáta ukladá v SREC formáte (kapitola 3.2.5), dáta v rovnakom priestore ako inštrukcie.

Obsahuje 291 inštrukcií zvyčajne operujúcich v dvoch až piatich cykloch. Inštrukcie sú poväčšine jednobytové, niektoré (začínajúce prefixom 9E) dvojbytové. Môžu mať nasledovné typy operandov:

- bez operandu
- 1jednobitový
- 1dvojbitový
- 2 jednobitové

Ako bolo spomenuté vyššie inštrukčná sada podporuje 16 adresovacích módov. Väčšina z nich sa používa pri prístupe k dátam. Pri vetvení programu sa využívajú len dva.

- Priama adresa – operandom je 2 bytová adresa na ktorú program skáče. Používa sa pri vzdialených skokoch ako napr. volanie funkcií.
- Relatívna adresa – používa sa pri krátkych skokoch. 1 bytový operand sa pričíta k aktuálnej adrese tým sa získa adresa skoku. Pre skoky smerom k nižším adresám sa používa dvojkový doplnok. To umožňuje skoky v rozsahu +127 (0x7F) až -128 (0x80).

4.4.2 Lineárny parser

Lineárny plugin implementuje algoritmus „linear sweep“. Parser prejde celý binárny súbor a rad za radom vypíše všetky dáta, interpretované ako inštrukcie s operandmi. Nevýhodou je, že tento mechanizmus identifikuje aj dáta ako inštrukcie. Výhodou je vysoká rýchlosť a fakt, že vždy preloží celý súbor .

V rámci toho pluginu je implementovaná aj funkcia zvýrazňovania bytov vybranej inštrukcie či operandu (viz. vyššie).

4.4.3 Rekurzívny parser

Tento parser demonštruje rekurzívny algoritmus. Najprv si vytvorí dátové štruktúry reprezentujúce jednotlivé byty v SREC súbore. Potom začnúc od adresy označenej ako počiatočná aplikuje rekurzívnu metódu prechod. Návratové adresy si ukladá do FIFO zásobníku. Pomenováva návestia skoku.

Špeciálnou vlastnosťou tohto pluginu je možno reassemblovania vytvoreného kódu. Užívateľ môže editovať získané inštrukcie a dáta. Funkčnosť je obmedzená na prepisovanie dát len rovnako dlhými dátami a dátami rovnakého typu tj. inštrukcie za inštrukcie a operandy miesto operandov.

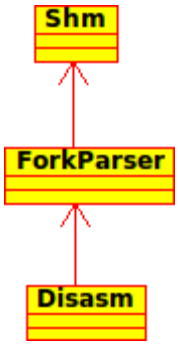
4.4.4 Viacprocesový kombinovaný parser

Posledný zásuvný modul implementuje hybridný algoritmus. Významný je ale hlavne tým, že využíva na svoj beh viacero vlákien.

Na začiatku sa proces rozdelí na dve vlákna. Prvé vlákno vytvorí zdieľanú pamäť a čaká na skončenie druhého. To lineárne disassembluje program. Ako náhle narazí na skokovú inštrukciu, opäť sa rozdelí. Materský proces pokračuje v lineárnom rozklade súboru, kým dcérske skočí na cieľové návestie a obdobne pokračuje. Procesy končia, keď narazia na návratovú inštrukciu alebo už disassemblovaný kód. Pokiaľ už preložený kód je iný ako aktuálne prekladaný, proces, ktorý naň narazil ako posledný sám seba označí za nesprávny a odroluje sebou preložené byty až po posledné vetvenie programu. Rodičovské procesy vždy čakajú na ukončenie dcérskych.

Keď sa ukončia všetky disassembovacie procesy, vlákno, ktoré vytvorilo pamäť zoradí preložený obsah tejto pamäte a pošle ho hlavnému programu.

V prípade toho plugin je štruktúra parsru spomínaná v kapitole 4.4 rozdelená na tri časti.



14 - Rozdelenie parsovacích tried u multivláknového pluginu

Trieda *Disasm* spracuje vstupný s.19 súbor a vytvorí hashovaciu tabuľku do ktorej uloží adresy a dátové byty. Túto tabuľku predá *ForkParseru*, ktorý sa stará o rozdelenie riadenia na jednotlivé procesy a samotný reverzný preklad. Trieda *Shm* zapuzdruje implementáciu zdieľanej pamäte.

Nevýhodou tohto pluginu je, že kvôli použitiu linuxových knižníc na vetvenie pomocou funkcie *fork()*, nie je preložiteľný na systéme Windows.

4.5 Ovládanie a beh aplikácie

Užívateľ najprv vyberie konkrétny plugin. V prípade, že nie je k dispozícií, môže použiť definované rozhranie na inštaláciu zásuvných modulov. Jedná sa v podstate o systém, kde vybraný plugin sám kopíruje seba a súbory potrebné pre svoj beh (inštrukčné sady a podobne) do adresára na to určeného.

Potom vyberie binárny súbor na spätný preklad. Ten sa uloží do pamäte pluginu (čiže po zmene pluginu ho potreba opäť načítať). Dá príkaz na disassemblovanie súboru. Zásuvný modul naplní *AsmTable*. Ďalej môže užívateľ pracovať so získanými dátami a exportovať ich buď do textového súboru ako assembler, alebo (v prípade, že použil plugin, ktorý má implementovaný reassembler) do binárneho súboru.

4.6 Porovnanie modulov

Lineárny parser je podľa očakávaní najrýchlejší. To vďaka jednoduchej konštrukcií. Na danej architektúre je aj pomerne presný. Jeho nevýhodou je, že preloží dáta ako inštrukcie, čo môže spôsobiť posunutie celého adresovania. Tiež to v praxi spôsobuje vytváranie neexistujúceho kódu.

Rekurzívny parser prechádza všetky dáta dvakrát pri vytváraní hashovacích štruktúr a pri ich spracovaní. Čo spôsobuje, že je prinajlepšom dvakrát pomalší ako lineárny. Pri práci s malými súbormi to však nie je pre užívateľa postrehnuteľné. Nevytvára neexistujúci kód, ale na druhej strane pri väčších binárnych súboroch sa stáva, že nepreloží celý obsah.

Ukázalo sa, že hybridný algoritmus produkuje najpresnejší kód. Rýchlosťou sa dá porovnať s rekurzívnym. Implementovaný multivláknový plugin je ale o mnoho pomalší. Je to spôsobené réžiou pri vytváraní zdieľanej pamäte a prepínaní procesov. Otázne je ako by sa plugin zachoval pri prekladaní väčších programov na mnohojadrových procesoroch. Tam by sa mohla prejaviť výhoda práce s viacerými procesmi.

5 Záver

V tejto bakalárskej práci bolo predstavená reverzné inžinierstvo v rámci informačných technológií. Boli uvedené rôzne druhy aplikácií a prístupov. Ďalej bola uvedená architektúra procesoru a rozličné súborové formáty spustiteľných súborov, ktoré používajú operačné systémy a mikroprocesory. Špeciálne sa venovalo mikrokontrolérom HC08, formátu SREC a príslušnej inštrukčnej sade.

Získané informácie a vedomosti boli zužitkované pri tvorbe univerzálneho disassemblovacieho programu a zásuvných modulárnych parserov. Na základe výsledkov jednotlivých modulov je možno pozorovať funkčnosť algoritmov spätného prekladu.

Inovatívny prístup bol použitý pri návrhu viacvláknového prekladača, ktorý používa samostatné vlákna pre každé návestie. Implementovaný plugin síce nepreukázal lepší výkon , ale predostrel nový nápad, ktorý sa môže neskôr pri vývoji podobného nástroja v praxi použiť.

Ďalší vývoj sa by sa mohol uberať buď smerom vývoja vyrobeného programu, implementáciou viacerých modulov pre rôzne typy mikroprocesorov a rozličné inštrukčné sady. Optimalizáciou multiprocesového parsru, prípadne širším rozvojom tejto myšlienky.

Literatúra

[1] J. Sochor.: *Údržba softwaru*. Zpravodaj ÚVT MU. ISSN 1212-0901, 1996, roč. VI, č. 3, s. 15-20

[2] Kaner, C.: *Article 2B and Reverse Engineering*. Uniform Commerical Code Bulletin. November 1998. Dostupné na URL: <http://www.badsoftware.com/reversea.htm>

[3] Schwarz, B., Debray, S. K., Andrews, G. R.: *Disassembly of Executable Code Revisited*. University of Arizona, Tucson, 2002.

[4] The Boomerang Decompiler Project: *Boomerang*. 2002 – 2006 [cit. 8. Mája 2011]. Dostupné na URL: <http://boomerang.sourceforge.net>

[5] Pierce, C.: *MindshaRE: Anti-Reversing Techniques* . August 2008 [cit. 30. Apríla 2011] Dostupné na URL: <http://dvlabs.tippingpoint.com/blog/2008/08/07/mindshare-anti-reversing-techniques>

[6] Falliere, N.: *Windows Anti-Debug Reference.* September 2007 [cit. 30. Apríla 2011] Dostupné na URL: <http://www.symantec.com/connect/articles/windows-anti-debug-reference>

[7] Schallner, M.: *Linux Interactive DisAssembler - Project Homepage*. 2004 [cit. 30. Apríla 2011]. Dostupné na URL: <http://lida.sourceforge.net/>

[8] Zemánek, J.: *Aktivní SMC*. August 2001 [cit. 30. Apríla 2011]. Dostupné na URL: <http://www.builder.cz/art/asembler/aktSMC.html>

[9] Zemánek, J.: *Anti-disassembling - základy*. Júl 2001 [cit. 30. Apríla 2011]. Dostupné na URL: <http://www.builder.cz/art/asembler/anti_dis.html>

[10] Křoustek, J.: *Analýza a Transformace kódů*, bakalářská práce, Brno, FIT VUT v Brně, 2007.

[11] *The IDA Pro* – Project Homepage. Dostupné na URL: <http://www.hex-rays.com/idapro/>

[12] Smith, E.: *Mocha, the Java Decompiler*. Máj 2007 [cit. 11. Mája 2011]. Dostupné na URL: <http://www.brouhaha.com/~eric/software/mocha/>

[13] Dr. Dobb's Journal: *Decompile Once, Run Anywhere (Web Techniques, Sep 1997)*. Január 2002 [cit. 11. Mája 2011]. Dostupné na URL: <http://drdobbs.com/184414282?pgno=9>

[14] *.NET Reflector*. Wikipedia. Máj 2011 [cit. 11. Mája 2011]. Dostupné na URL: <http://en.wikipedia.org/wiki/.NET_Reflector>

[15] *.GNU Debugger*. Wikipedia. November 2010 [cit. 11. Mája 2011]. Dostupné na URL: <http://cs.wikipedia.org/wiki/GNU_Debugger>

[16] *Olly Debugger* – Project Homepage. Dostupné na URL: <http://www.ollydbg.de/>

[17] Sriram, KB.: *Hashjava*. Január 1997 [cit. 11. Mája 2011]. Dostupné na URL: <http://www.ru.j-npcs.org/usoft/WWW/www_blackdown.org/kbs/hashjava.html>

[18] *Salamander .Net Obfucator* – Project Homepage. Dostupné na URL: <http://www.remotesoft.com/salamander/obfuscator.html>

[19] Voormedia HTML Scrambler. 2011. Dostupné na URL: <http://www.voormedia.com/en/tools/html-obfuscate-scrambler.php>

[20] *Lissom* – Project Homepage. 2006 – 2011. Dostupné na URL: <http://www.fit.vutbr.cz/research/groups/lissom/>

[21] *Central Prosessing Unit.* Wikipedia. Máj 2011 [cit. 8. Mája 2011]. Dostupné na URL: <http://en.wikipedia.org/wiki/Central_processing_unit>

[22] Schwarz, J., Růžička, R., Strnadel, J.: *Studijní opora pro předmět IMP*. VUT Brno, FIT, 2006.

[23] Váňa, V.: *Začínáme pracovat s mikrokontroléry Motorola HC08 Nitron*. BEN, Praha, 2003.

[24] Wienand, I.:*Computer Science from the Bottom Up - Chapter 9. Dynamic Linking.* 2004-2009 [cit. 8. Mája 2011]. Dostupné na URL: <http://bottomupcs.sourceforge.net/csbu/x3824.htm>

[25] *A.OUT (5)*. FreeBSD Man Pages. Jún 1993 [cit. 8. Mája 2011]. Dostupné na URL: <http://www.gsp.com/cgi-bin/man.cgi?section=5&topic=a.out>

[26] *Executable and Linkable Format*. Wikipedia. Apríl 2011 [cit. 10. Mája 2011]. Dostupné na URL: <http://en.wikipedia.org/wiki/Executable_and_Linkable_Format>

[27] Dickens, T.: *S19 File Format*. Apríl 2004 [cit. 12. Mája 2011]. Dostupné na URL: <http://home.earthlink.net/~tdickens/68hc11/docs/s19_file_format.html>

[28] *SREC (File Format)*. Wikipedia. Február 2011 [cit. 12. Mája 2011]. Dostupné na URL: <http://en.wikipedia.org/wiki/SREC_(file_format)>

[29] *Intel HEX*. Wikipedia. Apríl 2011 [cit. 12. Mája 2011]. Dostupné na URL: <http://en.wikipedia.org/wiki/Intel_HEX>

Zoznam príloh

Príloha 1. Obsah CD

contents.txt	Obsah CD
make.sh	Skript pre preklad programu
manual.pdf	Návod k inštalácií a používaniu programu
./doc/	Elektronická verzia tejto práce
./bin/	Preložená spustiteľná verzia programu (Linux 64bit)
./src/	Zdrojové súbory
./examples/	Testovacie .s19 súbory

Príloha 2. Inštrukčná sada Motorola MC68HC908LJ12

Table 6-1. Instruction Set Summary (Sheet 1 of 8)

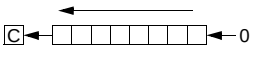
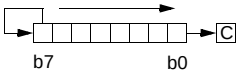
Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
ADC #*opr* ADC *opr* ADC *opr* ADC *opr*,X ADC *opr*,X ADC ,X ADC *opr*,SP ADC *opr*,SP	Add with Carry	A ← (A) + (M) + (C)	↕	↕	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP1 SP2	A9 B9 C9 D9 E9 F9 9EE9 9ED9	ii dd hh ll ee ff ff ff ee ff	2 3 4 4 3 2 4 5
ADD #*opr* ADD *opr* ADD *opr* ADD *opr*,X ADD *opr*,X ADD ,X ADD *opr*,SP ADD *opr*,SP	Add without Carry	A ← (A) + (M)	↕	↕	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP1 SP2	AB BB CB DB EB FB 9EEB 9EDB	ii dd hh ll ee ff ff ff ee ff	2 3 4 4 3 2 4 5
AIS #*opr*	Add Immediate Value (Signed) to SP	SP ← (SP) + (16 « M)	–	–	–	–	–	–	IMM	A7	ii	2
AIX #*opr*	Add Immediate Value (Signed) to H:X	H:X ← (H:X) + (16 « M)	–	–	–	–	–	–	IMM	AF	ii	2
AND #*opr* AND *opr* AND *opr* AND *opr*,X AND *opr*,X AND ,X AND *opr*,SP AND *opr*,SP	Logical AND	A ← (A) & (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP1 SP2	A4 B4 C4 D4 E4 F4 9EE4 9ED4	ii dd hh ll ee ff ff ff ee ff	2 3 4 4 3 2 4 5
ASL *opr* ASLA ASLX ASL *opr*,X ASL ,X ASL *opr*,SP	Arithmetic Shift Left (Same as LSL)		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	38 48 58 68 78 9E68	dd ff ff	4 1 1 4 3 5
ASR *opr* ASRA ASRX ASR *opr*,X ASR *opr*,X ASR *opr*,SP	Arithmetic Shift Right		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	37 47 57 67 77 9E67	dd ff ff	4 1 1 4 3 5
BCC *rel*	Branch if Carry Bit Clear	PC ← (PC) + 2 + rel ? (C) = 0	–	–	–	–	–	–	REL	24	rr	3
BCLR *n*, *opr*	Clear Bit n in M	Mn ← 0	–	–	–	–	–	–	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	11 13 15 17 19 1B 1D 1F	dd dd dd dd dd dd dd dd	4 4 4 4 4 4 4 4

Table 6-1. Instruction Set Summary (Sheet 2 of 8)

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
BCS *rel*	Branch if Carry Bit Set (Same as BLO)	PC ← (PC) + 2 + *rel* ? (C) = 1	–	–	–	–	–	–	REL	25	rr	3
BEQ *rel*	Branch if Equal	PC ← (PC) + 2 + *rel* ? (Z) = 1	–	–	–	–	–	–	REL	27	rr	3
BGE *opr*	Branch if Greater Than or Equal To (Signed Operands)	PC ← (PC) + 2 + *rel* ? (N ⊕ V) = 0	–	–	–	–	–	–	REL	90	rr	3
BGT *opr*	Branch if Greater Than (Signed Operands)	PC ← (PC) + 2 + *rel* ? (Z) | (N ⊕ V) = 0	–	–	–	–	–	–	REL	92	rr	3
BHCC *rel*	Branch if Half Carry Bit Clear	PC ← (PC) + 2 + *rel* ? (H) = 0	–	–	–	–	–	–	REL	28	rr	3
BHCS *rel*	Branch if Half Carry Bit Set	PC ← (PC) + 2 + *rel* ? (H) = 1	–	–	–	–	–	–	REL	29	rr	3
BHI *rel*	Branch if Higher	PC ← (PC) + 2 + *rel* ? (C) | (Z) = 0	–	–	–	–	–	–	REL	22	rr	3
BHS *rel*	Branch if Higher or Same (Same as BCC)	PC ← (PC) + 2 + *rel* ? (C) = 0	–	–	–	–	–	–	REL	24	rr	3
BIH *rel*	Branch if $\overline{IRQ}$ Pin High	PC ← (PC) + 2 + *rel* ? $\overline{IRQ}$ = 1	–	–	–	–	–	–	REL	2F	rr	3
BIL *rel*	Branch if $\overline{IRQ}$ Pin Low	PC ← (PC) + 2 + *rel* ? $\overline{IRQ}$ = 0	–	–	–	–	–	–	REL	2E	rr	3
BIT #*opr* BIT *opr* BIT *opr* BIT *opr*,X BIT *opr*,X BIT ,X BIT *opr*,SP BIT *opr*,SP	Bit Test	(A) & (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP1 SP2	A5 B5 C5 D5 E5 F5 9EE5 9ED5	ii dd hh ll ee ff ff ff ee ff	2 3 4 4 3 2 4 5
BLE *opr*	Branch if Less Than or Equal To (Signed Operands)	PC ← (PC) + 2 + *rel* ? (Z) | (N ⊕ V) = 1	–	–	–	–	–	–	REL	93	rr	3
BLO *rel*	Branch if Lower (Same as BCS)	PC ← (PC) + 2 + *rel* ? (C) = 1	–	–	–	–	–	–	REL	25	rr	3
BLS *rel*	Branch if Lower or Same	PC ← (PC) + 2 + *rel* ? (C) | (Z) = 1	–	–	–	–	–	–	REL	23	rr	3
BLT *opr*	Branch if Less Than (Signed Operands)	PC ← (PC) + 2 + *rel* ? (N ⊕ V) =1	–	–	–	–	–	–	REL	91	rr	3
BMC *rel*	Branch if Interrupt Mask Clear	PC ← (PC) + 2 + *rel* ? (I) = 0	–	–	–	–	–	–	REL	2C	rr	3
BMI *rel*	Branch if Minus	PC ← (PC) + 2 + *rel* ? (N) = 1	–	–	–	–	–	–	REL	2B	rr	3
BMS *rel*	Branch if Interrupt Mask Set	PC ← (PC) + 2 + *rel* ? (I) = 1	–	–	–	–	–	–	REL	2D	rr	3
BNE *rel*	Branch if Not Equal	PC ← (PC) + 2 + *rel* ? (Z) = 0	–	–	–	–	–	–	REL	26	rr	3
BPL *rel*	Branch if Plus	PC ← (PC) + 2 + *rel* ? (N) = 0	–	–	–	–	–	–	REL	2A	rr	3
BRA *rel*	Branch Always	PC ← (PC) + 2 + *rel*	–	–	–	–	–	–	REL	20	rr	3

Table 6-1. Instruction Set Summary (Sheet 3 of 8)

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
BRCLR *n*,*opr*,*rel*	Branch if Bit *n* in M Clear	PC ← (PC) + 3 + *rel* ? (Mn) = 0	–	–	–	–	–	↕	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	01 03 05 07 09 0B 0D 0F	dd rr dd rr dd rr dd rr dd rr dd rr dd rr dd rr	5 5 5 5 5 5 5 5
BRN *rel*	Branch Never	PC ← (PC) + 2	–	–	–	–	–	–	REL	21	rr	3
BRSET *n*,*opr*,*rel*	Branch if Bit *n* in M Set	PC ← (PC) + 3 + *rel* ? (Mn) = 1	–	–	–	–	–	↕	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	00 02 04 06 08 0A 0C 0E	dd rr dd rr dd rr dd rr dd rr dd rr dd rr dd rr	5 5 5 5 5 5 5 5
BSET *n*,*opr*	Set Bit *n* in M	Mn ← 1	–	–	–	–	–	–	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	10 12 14 16 18 1A 1C 1E	dd dd dd dd dd dd dd dd	4 4 4 4 4 4 4 4
BSR *rel*	Branch to Subroutine	PC ← (PC) + 2; push (PCL) SP ← (SP) – 1; push (PCH) SP ← (SP) – 1 PC ← (PC) + *rel*	–	–	–	–	–	–	REL	AD	rr	4
CBEQ *opr*,*rel* CBEQA #*opr*,*rel* CBEQX #*opr*,*rel* CBEQ *opr*,X+,*rel* CBEQ X+,*rel* CBEQ *opr*,SP,*rel*	Compare and Branch if Equal	PC ← (PC) + 3 + rel ? (A) – (M) = $00 PC ← (PC) + 3 + rel ? (A) – (M) = $00 PC ← (PC) + 3 + rel ? (X) – (M) = $00 PC ← (PC) + 3 + rel ? (A) – (M) = $00 PC ← (PC) + 2 + rel ? (A) – (M) = $00 PC ← (PC) + 4 + rel ? (A) – (M) = $00	–	–	–	–	–	–	DIR IMM IMM IX1+ IX+ SP1	31 41 51 61 71 9E61	dd rr ii rr ii rr ff rr rr ff rr	5 4 4 5 4 6
CLC	Clear Carry Bit	C ← 0	–	–	–	–	–	0	INH	98		1
CLI	Clear Interrupt Mask	I ← 0	–	–	0	–	–	–	INH	9A		2
CLR *opr* CLRA CLRX CLRH CLR *opr*,X CLR ,X CLR *opr*,SP	Clear	M ← $00 A ← $00 X ← $00 H ← $00 M ← $00 M ← $00 M ← $00	0	–	–	0	1	–	DIR INH INH INH IX1 IX SP1	3F 4F 5F 8C 6F 7F 9E6F	dd ff ff	3 1 1 1 3 2 4

Table 6-1. Instruction Set Summary (Sheet 4 of 8)

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
CMP #*opr* CMP *opr* CMP *opr* CMP *opr*,X CMP *opr*,X CMP ,X CMP *opr*,SP CMP *opr*,SP	Compare A with M	(A) – (M)	↕	–	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP1 SP2	A1 B1 C1 D1 E1 F1 9EE1 9ED1	ii dd hh ll ee ff ff ff ee ff	2 3 4 4 3 2 4 5
COM *opr* COMA COMX COM *opr*,X COM ,X COM *opr*,SP	Complement (One's Complement)	M ← ($\overline{M}$) = $FF – (M) A ← ($\overline{A}$) = $FF – (M) X ← ($\overline{X}$) = $FF – (M) M ← ($\overline{M}$) = $FF – (M) M ← ($\overline{M}$) = $FF – (M) M ← ($\overline{M}$) = $FF – (M)	0	–	–	↕	↕	1	DIR INH INH IX1 IX SP1	33 43 53 63 73 9E63	dd ff ff	4 1 1 4 3 5
CPHX #*opr* CPHX *opr*	Compare H:X with M	(H:X) – (M:M + 1)	↕	–	–	↕	↕	↕	IMM DIR	65 75	ii ii+1 dd	3 4
CPX #*opr* CPX *opr* CPX *opr* CPX ,X CPX *opr*,X CPX *opr*,X CPX *opr*,SP CPX *opr*,SP	Compare X with M	(X) – (M)	↕	–	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP1 SP2	A3 B3 C3 D3 E3 F3 9EE3 9ED3	ii dd hh ll ee ff ff ff ee ff	2 3 4 4 3 2 4 5
DAA	Decimal Adjust A	$(A)_{10}$	U	–	–	↕	↕	↕	INH	72		2
DBNZ *opr,rel* DBNZA *rel* DBNZX *rel* DBNZ *opr*,X,*rel* DBNZ X,*rel* DBNZ *opr*,SP,*rel*	Decrement and Branch if Not Zero	A ← (A) – 1 or M ← (M) – 1 or X ← (X) – 1 PC ← (PC) + 3 + *rel* ? (result) ≠ 0 PC ← (PC) + 2 + *rel* ? (result) ≠ 0 PC ← (PC) + 2 + *rel* ? (result) ≠ 0 PC ← (PC) + 3 + *rel* ? (result) ≠ 0 PC ← (PC) + 2 + *rel* ? (result) ≠ 0 PC ← (PC) + 4 + *rel* ? (result) ≠ 0	–	–	–	–	–	–	DIR INH INH IX1 IX SP1	3B 4B 5B 6B 7B 9E6B	dd rr rr rr ff rr rr ff rr	5 3 3 5 4 6
DEC *opr* DECA DECX DEC *opr*,X DEC ,X DEC *opr*,SP	Decrement	M ← (M) – 1 A ← (A) – 1 X ← (X) – 1 M ← (M) – 1 M ← (M) – 1 M ← (M) – 1	↕	–	–	↕	↕	–	DIR INH INH IX1 IX SP1	3A 4A 5A 6A 7A 9E6A	dd ff ff	4 1 1 4 3 5
DIV	Divide	A ← (H:A)/(X) H ← Remainder	–	–	–	–	↕	↕	INH	52		7
EOR #*opr* EOR *opr* EOR *opr* EOR *opr*,X EOR *opr*,X EOR ,X EOR *opr*,SP EOR *opr*,SP	Exclusive OR M with A	A ← (A ⊕ M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP1 SP2	A8 B8 C8 D8 E8 F8 9EE8 9ED8	ii dd hh ll ee ff ff ff ee ff	2 3 4 4 3 2 4 5

Table 6-1. Instruction Set Summary (Sheet 5 of 8)

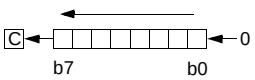
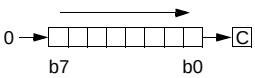
Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
INC *opr* INCA INCX INC *opr*,X INC ,X INC *opr*,SP	Increment	M ← (M) + 1 A ← (A) + 1 X ← (X) + 1 M ← (M) + 1 M ← (M) + 1 M ← (M) + 1	↕	–	–	↕	↕	–	DIR INH INH IX1 IX SP1	3C 4C 5C 6C 7C 9E6C	dd ff ff	4 1 1 4 3 5
JMP *opr* JMP *opr* JMP *opr*,X JMP *opr*,X JMP ,X	Jump	PC ← Jump Address	–	–	–	–	–	–	DIR EXT IX2 IX1 IX	BC CC DC EC FC	dd hh ll ee ff ff 	2 3 4 3 2
JSR *opr* JSR *opr* JSR *opr*,X JSR *opr*,X JSR ,X	Jump to Subroutine	PC ← (PC) + *n* (*n* = 1, 2, or 3) Push (PCL); SP ← (SP) – 1 Push (PCH); SP ← (SP) – 1 PC ← Unconditional Address	–	–	–	–	–	–	DIR EXT IX2 IX1 IX	BD CD DD ED FD	dd hh ll ee ff ff 	4 5 6 5 4
LDA #*opr* LDA *opr* LDA *opr* LDA *opr*,X LDA *opr*,X LDA ,X LDA *opr*,SP LDA *opr*,SP	Load A from M	A ← (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP1 SP2	A6 B6 C6 D6 E6 F6 9EE6 9ED6	ii dd hh ll ee ff ff ff ee ff	2 3 4 4 3 2 4 5
LDHX #*opr* LDHX *opr*	Load H:X from M	H:X ← (M:M + 1)	0	–	–	↕	↕	–	IMM DIR	45 55	ii jj dd	3 4
LDX #*opr* LDX *opr* LDX *opr* LDX *opr*,X LDX *opr*,X LDX ,X LDX *opr*,SP LDX *opr*,SP	Load X from M	X ← (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP1 SP2	AE BE CE DE EE FE 9EEE 9EDE	ii dd hh ll ee ff ff ff ee ff	2 3 4 4 3 2 4 5
LSL *opr* LSLA LSLX LSL *opr*,X (Same as ASL) LSL ,X LSL *opr*,SP	Logical Shift Left		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	38 48 58 68 78 9E68	dd ff ff	4 1 1 4 3 5
LSR *opr* LSRA LSR*X* LSR *opr*,X LSR ,X LSR *opr*,SP	Logical Shift Right		↕	–	–	0	↕	↕	DIR INH INH IX1 IX SP1	34 44 54 64 74 9E64	dd ff ff	4 1 1 4 3 5

Table 6-1. Instruction Set Summary (Sheet 6 of 8)

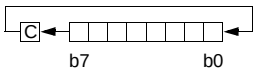
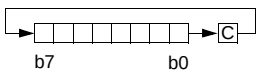
Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
MOV *opr,opr* MOV *opr,*X+ MOV *#opr,opr* MOV X+*,opr*	Move	$(M)_{Destination} \leftarrow (M)_{Source}$ H:X ← (H:X) + 1 (IX+D, DIX+)	0	–	–	↕	↕	–	DD DIX+ IMD IX+D	4E 5E 6E 7E	dd dd dd ii dd dd	5 4 4 4
MUL	Unsigned multiply	X:A ← (X) × (A)	–	0	–	–	–	0	INH	42		5
NEG *opr* NEGA NEGX NEG *opr*,X NEG ,X NEG *opr*,SP	Negate (Two's Complement)	M ← –(M) = $00 – (M) A ← –(A) = $00 – (A) X ← –(X) = $00 – (X) M ← –(M) = $00 – (M) M ← –(M) = $00 – (M)	↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	30 40 50 60 70 9E60	dd ff ff	4 1 1 4 3 5
NOP	No Operation	None	–	–	–	–	–	–	INH	9D		1
NSA	Nibble Swap A	A ← (A[3:0]:A[7:4])	–	–	–	–	–	–	INH	62		3
ORA *#opr* ORA *opr* ORA *opr* ORA *opr*,X ORA *opr*,X ORA ,X ORA *opr*,SP ORA *opr*,SP	Inclusive OR A and M	A ← (A) | (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP1 SP2	AA BA CA DA EA FA 9EEA 9EDA	ii dd hh ll ee ff ff ff ee ff	2 3 4 4 3 2 4 5
PSHA	Push A onto Stack	Push (A); SP ← (SP) – 1	–	–	–	–	–	–	INH	87		2
PSHH	Push H onto Stack	Push (H); SP ← (SP) – 1	–	–	–	–	–	–	INH	8B		2
PSHX	Push X onto Stack	Push (X); SP ← (SP) – 1	–	–	–	–	–	–	INH	89		2
PULA	Pull A from Stack	SP ← (SP + 1); Pull (A)	–	–	–	–	–	–	INH	86		2
PULH	Pull H from Stack	SP ← (SP + 1); Pull (H)	–	–	–	–	–	–	INH	8A		2
PULX	Pull X from Stack	SP ← (SP + 1); Pull (X)	–	–	–	–	–	–	INH	88		2
ROL *opr* ROLA ROLX ROL *opr*,X ROL ,X ROL *opr*,SP	Rotate Left through Carry		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	39 49 59 69 79 9E69	dd ff ff	4 1 1 4 3 5
ROR *opr* RORA RORX ROR *opr*,X ROR ,X ROR *opr*,SP	Rotate Right through Carry		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	36 46 56 66 76 9E66	dd ff ff	4 1 1 4 3 5
RSP	Reset Stack Pointer	SP ← $FF	–	–	–	–	–	–	INH	9C		1

Table 6-1. Instruction Set Summary (Sheet 7 of 8)

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
RTI	Return from Interrupt	SP ← (SP) + 1; Pull (CCR) SP ← (SP) + 1; Pull (A) SP ← (SP) + 1; Pull (X) SP ← (SP) + 1; Pull (PCH) SP ← (SP) + 1; Pull (PCL)	↕	↕	↕	↕	↕	↕	INH	80		7
RTS	Return from Subroutine	SP ← SP + 1; Pull (PCH) SP ← SP + 1; Pull (PCL)	–	–	–	–	–	–	INH	81		4
SBC #*opr*	Subtract with Carry	A ← (A) – (M) – (C)	↕	–	–	↕	↕	↕	IMM	A2	ii	2
SBC *opr*									DIR	B2	dd	3
SBC *opr*									EXT	C2	hh ll	4
SBC *opr*,X									IX2	D2	ee ff	4
SBC *opr*,X									IX1	E2	ff	3
SBC ,X									IX	F2		2
SBC *opr*,SP									SP1	9EE2	ff	4
SBC *opr*,SP									SP2	9ED2	ee ff	5
SEC	Set Carry Bit	C ← 1	–	–	–	–	–	1	INH	99		1
SEI	Set Interrupt Mask	I ← 1	–	–	1	–	–	–	INH	9B		2
STA *opr*	Store A in M	M ← (A)	0	–	–	↕	↕	–	DIR	B7	dd	3
STA *opr*									EXT	C7	hh ll	4
STA *opr*,X									IX2	D7	ee ff	4
STA *opr*,X									IX1	E7	ff	3
STA ,X									IX	F7		2
STA *opr*,SP									SP1	9EE7	ff	4
STA *opr*,SP									SP2	9ED7	ee ff	5
STHX *opr*	Store H:X in M	(M:M + 1) ← (H:X)	0	–	–	↕	↕	–	DIR	35	dd	4
STOP	Enable $\overline{\text{IRQ}}$ Pin; Stop Oscillator	I ← 0; Stop Oscillator	–	–	0	–	–	–	INH	8E		1
STX *opr*	Store X in M	M ← (X)	0	–	–	↕	↕	–	DIR	BF	dd	3
STX *opr*									EXT	CF	hh ll	4
STX *opr*,X									IX2	DF	ee ff	4
STX *opr*,X									IX1	EF	ff	3
STX ,X									IX	FF		2
STX *opr*,SP									SP1	9EEF	ff	4
STX *opr*,SP									SP2	9EDF	ee ff	5
SUB #*opr*	Subtract	A ← (A) – (M)	↕	–	–	↕	↕	↕	IMM	A0	ii	2
SUB *opr*									DIR	B0	dd	3
SUB *opr*									EXT	C0	hh ll	4
SUB *opr*,X									IX2	D0	ee ff	4
SUB *opr*,X									IX1	E0	ff	3
SUB ,X									IX	F0		2
SUB *opr*,SP									SP1	9EE0	ff	4
SUB *opr*,SP									SP2	9ED0	ee ff	5

Table 6-1. Instruction Set Summary (Sheet 8 of 8)

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
SWI	Software Interrupt	PC ← (PC) + 1; Push (PCL) SP ← (SP) – 1; Push (PCH) SP ← (SP) – 1; Push (X) SP ← (SP) – 1; Push (A) SP ← (SP) – 1; Push (CCR) SP ← (SP) – 1; I ← 1 PCH ← Interrupt Vector High Byte PCL ← Interrupt Vector Low Byte	–	–	1	–	–	–	INH	83		9
TAP	Transfer A to CCR	CCR ← (A)	↕	↕	↕	↕	↕	↕	INH	84		2
TAX	Transfer A to X	X ← (A)	–	–	–	–	–	–	INH	97		1
TPA	Transfer CCR to A	A ← (CCR)	–	–	–	–	–	–	INH	85		1
TST *opr* TSTA TSTX TST *opr*,X TST ,X TST *opr*,SP	Test for Negative or Zero	(A) – $00 or (X) – $00 or (M) – $00	0	–	–	↕	↕	–	DIR INH INH IX1 IX SP1	3D 4D 5D 6D 7D 9E6D	dd ff ff	3 1 1 3 2 4
TSX	Transfer SP to H:X	H:X ← (SP) + 1	–	–	–	–	–	–	INH	95		2
TXA	Transfer X to A	A ← (X)	–	–	–	–	–	–	INH	9F		1
TXS	Transfer H:X to SP	(SP) ← (H:X) – 1	–	–	–	–	–	–	INH	94		2

A	Accumulator
C	Carry/borrow bit
CCR	Condition code register
dd	Direct address of operand
dd rr	Direct address of operand and relative offset of branch instruction
DD	Direct to direct addressing mode
DIR	Direct addressing mode
DIX+	Direct to indexed with post increment addressing mode
ee ff	High and low bytes of offset in indexed, 16-bit offset addressing
EXT	Extended addressing mode
ff	Offset byte in indexed, 8-bit offset addressing
H	Half-carry bit
H	Index register high byte
hh ll	High and low bytes of operand address in extended addressing
I	Interrupt mask
ii	Immediate operand byte
IMD	Immediate source to direct destination addressing mode
IMM	Immediate addressing mode
INH	Inherent addressing mode
IX	Indexed, no offset addressing mode
IX+	Indexed, no offset, post increment addressing mode
IX+D	Indexed with post increment to direct addressing mode
IX1	Indexed, 8-bit offset addressing mode
IX1+	Indexed, 8-bit offset, post increment addressing mode
IX2	Indexed, 16-bit offset addressing mode
M	Memory location
N	Negative bit
n	Any bit
opr	Operand (one or two bytes)
PC	Program counter
PCH	Program counter high byte
PCL	Program counter low byte
REL	Relative addressing mode
rel	Relative program counter offset byte
rr	Relative program counter offset byte
SP1	Stack pointer, 8-bit offset addressing mode
SP2	Stack pointer 16-bit offset addressing mode
SP	Stack pointer
U	Undefined
V	Overflow bit
X	Index register low byte
Z	Zero bit
&	Logical AND
|	Logical OR
⊕	Logical EXCLUSIVE OR
( )	Contents of
–( )	Negation (two's complement)
#	Immediate value
«	Sign extend
←	Loaded with
?	If
:	Concatenated with
↕	Set or cleared
—	Not affected

Table 6-2. Opcode Map

	Bit Manipulation		Branch	Read-Modify-Write						Control		Register/Memory							
	DIR	DIR	REL	DIR	INH	INH	IX1	SP1	IX	INH	INH	IMM	DIR	EXT	IX2	SP2	IX1	SP1	IX
MSB / LSB	0	1	2	3	4	5	6	9E6	7	8	9	A	B	C	D	9ED	E	9EE	F
0	5 BRSET0 3 DIR	4 BSET0 2 DIR	3 BRA 2 REL	4 NEG 2 DIR	1 NEGA 1 INH	1 NEGX 1 INH	4 NEG 2 IX1	5 NEG 3 SP1	3 NEG 1 IX	7 RTI 1 INH	3 BGE 2 REL	2 SUB 2 IMM	3 SUB 2 DIR	4 SUB 3 EXT	4 SUB 3 IX2	5 SUB 4 SP2	3 SUB 2 IX1	4 SUB 3 SP1	2 SUB 1 IX
1	5 BRCLR0 3 DIR	4 BCLR0 2 DIR	3 BRN 2 REL	5 CBEQ 3 DIR	4 CBEQA 3 IMM	4 CBEQX 3 IMM	5 CBEQ 3 IX1+	6 CBEQ 4 SP1	4 CBEQ 2 IX+	4 RTS 1 INH	3 BLT 2 REL	2 CMP 2 IMM	3 CMP 2 DIR	4 CMP 3 EXT	4 CMP 3 IX2	5 CMP 4 SP2	3 CMP 2 IX1	4 CMP 3 SP1	2 CMP 1 IX
2	5 BRSET1 3 DIR	4 BSET1 2 DIR	3 BHI 2 REL		5 MUL 1 INH	7 DIV 1 INH	3 NSA 1 INH		2 DAA 1 INH		3 BGT 2 REL	2 SBC 2 IMM	3 SBC 2 DIR	4 SBC 3 EXT	4 SBC 3 IX2	5 SBC 4 SP2	3 SBC 2 IX1	4 SBC 3 SP1	2 SBC 1 IX
3	5 BRCLR1 3 DIR	4 BCLR1 2 DIR	3 BLS 2 REL	4 COM 2 DIR	1 COMA 1 INH	1 COMX 1 INH	4 COM 2 IX1	5 COM 3 SP1	3 COM 1 IX	9 SWI 1 INH	3 BLE 2 REL	2 CPX 2 IMM	3 CPX 2 DIR	4 CPX 3 EXT	4 CPX 3 IX2	5 CPX 4 SP2	3 CPX 2 IX1	4 CPX 3 SP1	2 CPX 1 IX
4	5 BRSET2 3 DIR	4 BSET2 2 DIR	3 BCC 2 REL	4 LSR 2 DIR	1 LSRA 1 INH	1 LSRX 1 INH	4 LSR 2 IX1	5 LSR 3 SP1	3 LSR 1 IX	2 TAP 1 INH	2 TXS 1 INH	2 AND 2 IMM	3 AND 2 DIR	4 AND 3 EXT	4 AND 3 IX2	5 AND 4 SP2	3 AND 2 IX1	4 AND 3 SP1	2 AND 1 IX
5	5 BRCLR2 3 DIR	4 BCLR2 2 DIR	3 BCS 2 REL	4 STHX 2 DIR	3 LDHX 3 IMM	4 LDHX 2 DIR	3 CPHX 3 IMM		4 CPHX 2 DIR	1 TPA 1 INH	2 TSX 1 INH	2 BIT 2 IMM	3 BIT 2 DIR	4 BIT 3 EXT	4 BIT 3 IX2	5 BIT 4 SP2	3 BIT 2 IX1	4 BIT 3 SP1	2 BIT 1 IX
6	5 BRSET3 3 DIR	4 BSET3 2 DIR	3 BNE 2 REL	4 ROR 2 DIR	1 RORA 1 INH	1 RORX 1 INH	4 ROR 2 IX1	5 ROR 3 SP1	3 ROR 1 IX	2 PULA 1 INH		2 LDA 2 IMM	3 LDA 2 DIR	4 LDA 3 EXT	4 LDA 3 IX2	5 LDA 4 SP2	3 LDA 2 IX1	4 LDA 3 SP1	2 LDA 1 IX
7	5 BRCLR3 3 DIR	4 BCLR3 2 DIR	3 BEQ 2 REL	4 ASR 2 DIR	1 ASRA 1 INH	1 ASRX 1 INH	4 ASR 2 IX1	5 ASR 3 SP1	3 ASR 1 IX	2 PSHA 1 INH	1 TAX 1 INH	2 AIS 2 IMM	3 STA 2 DIR	4 STA 3 EXT	4 STA 3 IX2	5 STA 4 SP2	3 STA 2 IX1	4 STA 3 SP1	2 STA 1 IX
8	5 BRSET4 3 DIR	4 BSET4 2 DIR	3 BHCC 2 REL	4 LSL 2 DIR	1 LSLA 1 INH	1 LSLX 1 INH	4 LSL 2 IX1	5 LSL 3 SP1	3 LSL 1 IX	2 PULX 1 INH	1 CLC 1 INH	2 EOR 2 IMM	3 EOR 2 DIR	4 EOR 3 EXT	4 EOR 3 IX2	5 EOR 4 SP2	3 EOR 2 IX1	4 EOR 3 SP1	2 EOR 1 IX
9	5 BRCLR4 3 DIR	4 BCLR4 2 DIR	3 BHCS 2 REL	4 ROL 2 DIR	1 ROLA 1 INH	1 ROLX 1 INH	4 ROL 2 IX1	5 ROL 3 SP1	3 ROL 1 IX	2 PSHX 1 INH	1 SEC 1 INH	2 ADC 2 IMM	3 ADC 2 DIR	4 ADC 3 EXT	4 ADC 3 IX2	5 ADC 4 SP2	3 ADC 2 IX1	4 ADC 3 SP1	2 ADC 1 IX
A	5 BRSET5 3 DIR	4 BSET5 2 DIR	3 BPL 2 REL	4 DEC 2 DIR	1 DECA 1 INH	1 DECX 1 INH	4 DEC 2 IX1	5 DEC 3 SP1	3 DEC 1 IX	2 PULH 1 INH	2 CLI 1 INH	2 ORA 2 IMM	3 ORA 2 DIR	4 ORA 3 EXT	4 ORA 3 IX2	5 ORA 4 SP2	3 ORA 2 IX1	4 ORA 3 SP1	2 ORA 1 IX
B	5 BRCLR5 3 DIR	4 BCLR5 2 DIR	3 BMI 2 REL	5 DBNZ 3 DIR	3 DBNZA 2 INH	3 DBNZX 2 INH	5 DBNZ 3 IX1	6 DBNZ 4 SP1	4 DBNZ 2 IX	2 PSHH 1 INH	2 SEI 1 INH	2 ADD 2 IMM	3 ADD 2 DIR	4 ADD 3 EXT	4 ADD 3 IX2	5 ADD 4 SP2	3 ADD 2 IX1	4 ADD 3 SP1	2 ADD 1 IX
C	5 BRSET6 3 DIR	4 BSET6 2 DIR	3 BMC 2 REL	4 INC 2 DIR	1 INCA 1 INH	1 INCX 1 INH	4 INC 2 IX1	5 INC 3 SP1	3 INC 1 IX	1 CLRH 1 INH	1 RSP 1 INH		2 JMP 2 DIR	3 JMP 3 EXT	4 JMP 3 IX2		3 JMP 2 IX1		2 JMP 1 IX
D	5 BRCLR6 3 DIR	4 BCLR6 2 DIR	3 BMS 2 REL	3 TST 2 DIR	1 TSTA 1 INH	1 TSTX 1 INH	3 TST 2 IX1	4 TST 3 SP1	2 TST 1 IX		1 NOP 1 INH	4 BSR 2 REL	4 JSR 2 DIR	5 JSR 3 EXT	6 JSR 3 IX2		5 JSR 2 IX1		4 JSR 1 IX
E	5 BRSET7 3 DIR	4 BSET7 2 DIR	3 BIL 2 REL		5 MOV 3 DD	4 MOV 2 DIX+	4 MOV 3 IMD		4 MOV 2 IX+D	1 STOP 1 INH	*	2 LDX 2 IMM	3 LDX 2 DIR	4 LDX 3 EXT	4 LDX 3 IX2	5 LDX 4 SP2	3 LDX 2 IX1	4 LDX 3 SP1	2 LDX 1 IX
F	5 BRCLR7 3 DIR	4 BCLR7 2 DIR	3 BIH 2 REL	3 CLR 2 DIR	1 CLRA 1 INH	1 CLRX 1 INH	3 CLR 2 IX1	4 CLR 3 SP1	2 CLR 1 IX	1 WAIT 1 INH	1 TXA 1 INH	2 AIX 2 IMM	3 STX 2 DIR	4 STX 3 EXT	4 STX 3 IX2	5 STX 4 SP2	3 STX 2 IX1	4 STX 3 SP1	2 STX 1 IX

INH Inherent
IMM Immediate
DIR Direct
EXT Extended
DD Direct-Direct
IX+D Indexed-Direct
REL Relative
IX Indexed, No Offset
IX1 Indexed, 8-Bit Offset
IX2 Indexed, 16-Bit Offset
IMD Immediate-Direct
DIX+ Direct-Indexed
SP1 Stack Pointer, 8-Bit Offset
SP2 Stack Pointer, 16-Bit Offset
IX+ Indexed, No Offset with Post Increment
IX1+ Indexed, 1-Byte Offset with Post Increment

*Pre-byte for stack pointer indexed instructions

MSB / LSB	0
0	5 BRSET0 3 DIR

Low Byte of Opcode in Hexadecimal

High Byte of Opcode in Hexadecimal

Cycles
Opcode Mnemonic
Number of Bytes / Addressing Mode