TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky a mezioborových inženýrských studií

Studijní program: B 2612 – Elektrotechnika a informatika
Studijní obor: Elektronické informační a řídící systémy

Disassembler pro procesory x51

Disassembler for x51 processors

Bakalářská práce

Autor: **Aleš Havlas**
Vedoucí BP/DP práce: Ing. Tomáš Pluhař
Konzultant: Ing. Tomáš Martinec

V Liberci 19. 5. 2006

(*** zadání ***)

Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé BP a prohlašuji, že **s o u h l a s í m** s případným užitím mé diplomové práce (prodej, zapůjčení apod.).

Jsem si vědom(a) toho, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Bakalářskou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Datum 19. května 2006

Podpis

Poděkování

Dovolte mi touto cestou poděkovat dvěma osobám, které notnou měrou přispěly k úspěšnému vzniku mé bakalářské práce.

První z nich je slečna Jitka Rádlová, která mi byla velkou duševní podporou a zároveň prováděla jazykové korektury této zprávy.

Druhý pak je kolega Radovan Paška, jež mi nezanedbatelně pomohl s pochopením některých problematik fungování procesorů řady x51, vytvářením DLL knihoven v jazyce C++ a zároveň mi pomohl vytvořit grafické rozhraní pro demonstraci funkcí vytvořené DLL knihovny.

Aleš Havlas

Abstrakt

Práce se zabývá vytvořením disassembleru pro procesory řady x51, tedy aplikace, která umí přeložit strojový kód odpovídajícího procesoru zpět do na zdrojový v jazyce assembleru. Důraz je kladen na fakt, aby program nabídl na výstupu uživateli zdrojový kód v takové formě, aby jej bylo možno ihned zase zkompilovat. Mimo to má být součástí jiného softwaru, tudíž musí mít formu nějakého standardního modulu.

Zpráva práce obsahuje základní teorii o procesorech řady x51, jejich zdrojových i strojových kódech a dynamických knihovnách. Velmi obsáhle je popsán princip řešení celé aplikace od počátečního čtení zdrojových souborů, přes poznávání instrukcí a jejich operandů, až po pojmenovávání návěští a zápis výsledných souborů. Toto všechno samozřejmě včetně důvodů, proč bylo učiněno právě tak eventuelně, proč nebylo dosaženo požadovaných či doporučených cílů.

Nedílnou součástí celé práce je samozřejmě výsledný program, včetně zdrojových kódů a jednoduché ovládací aplikace, která umí demonstrovat jeho schopnosti.

Klíčová slova: disassembler, procesor řady x51, strojový kód, zdrojový kód, assembler

Abstract

This thesis deal with creating the disassembler for x51 series processors. It means the application, that is able to translate the machine code of corresponding processor back to source code in language of assembler. It is very important to offer on output source code in form, that can be immediately translate back to machine code. Except this should the application be the part of another software, so it must have interface for communication.

This text contains basic theory about x51 series processors, their source and machine codes and dynamic link libraries. The progress of creating is described very particulary from reading input files, over recognition of instructions and their operands, up to naming the signs and writing output files. This all of course with reasons, why was the progress just like this and/or alternatively, why the specified objectives was not reached.

Inseparable part of thesis is of course the application, with its source files included and simple graphic interface, that is able to demonstrate application’s abilities.

Keywords: disassembler, x51 series processor, machine code, source code, assembler

Obsah

Slovník zkratek a termínů

- **DLL (Dynamic Link Library)** – Dynamicky linkovaná knihovna. Zvláštní typ souboru obsahující zkompilovanou funkci, kterou nabízí k použití jiným aplikacím.
- **IntelHex** – Speciální formát pro ukládání strojového kódu procesorů v šestnáctkové soustavě vyvinutý firmou Intel.
- **návěští** – Místa v paměti, která jsou cílem relativních i absolutních odskoků.
- **CRLF (Windows, OS/2, DOS Line end)** – Znak pro ukončení řádku.
- **ORG** – Pseudoinstrukce assembleru určující, od kterého místa paměti má být uložen strojový kód.
- **.asm** – Přípona textových souborů se zdrojovým kódem v assembleru.
- **SFR (Special Function Register)** – Registr speciálních funkcí.
- **char** – Datový typ s velikostí jednoho bajtu (jeden znak).
- **unsigned char** – Datový typ pouze pro kladná čísla s velikostí jednoho bajtu.
- **EOF (End Of File)** – Znak konce souboru.
- **switch-case** – Podmínka pro větvení programu. (Přepni, pokud hodnota je…)
- **if** – Logická podmínka v programu (pokud).

Úvod

Účel práce

Zadání práce vzešlo z potřeb Katedry softwarové inženýrství naší fakulty, která (nejen) pro vyučovaný předmět *Počítače a mikropočítače* potřebuje kompletní softwarové vybavení pro programování, komunikaci a práci s procesory řady x51. Vzhledem ke skutečnosti, že již existující software tohoto typu (a to jak volně šiřitelný, tak komerční), je většinou účelově připraven na míru konkrétní činnosti, nepodporuje všechny potřebné standardy, či nedisponuje všemi požadovanými schopnostmi, bylo rozhodnuto, že v rámci bakalářských a diplomových prací (eventuelně ročníkových projektů) bude naprogramována skutečně univerzální aplikace (lépe řečeno balík aplikací), která uspokojí požadavky kladené na ní našimi vyučujícími i studenty.

Pokud výše nastíněný software má skutečně pokrýt všechny na něj kladené nároky, bude muset být velmi rozsáhlý a bude muset umět provádět mnoho různých činností – od překladu zdrojového kódu přes komunikaci po sériové lince až po rozklad strojového kódu zpět na zdrojový. To ovšem znamená fakt, že je nereálné, aby takový projekt zvládl zpracovat jeden člověk, neboť krom obrovského množství programování by jej čekalo i nastudování teorie zasahujících do mnoha oblastí elektroniky. Z tohoto důvodu bylo dále rozhodnuto, že tento program bude rozdělen na několik logických celků, které budou zpracovány jednotlivými studenty, a teprve tyto celky se posléze propojí v komplexní funkční aplikaci.

Požadavky na výsledek

Cílem mé práce bylo naprogramovat právě jednu z částí komplexního softwaru pro práci s procesory řady x51. Touto částí se stal *disassembler* – tedy aplikace pro převod strojového kódu zpět na zdrojový, programátorovi srozumitelný kód.

Schopnosti aplikace

Samozřejmým cílem bylo vytvořit tento program takovým způsobem, aby uměl přečíst soubor obsahující data strojového kódu, uměl z nich bezchybně vygenerovat kód zdrojový a dokázal upozornit na případné chyby a jiné nastalé problémy. Krom toho bylo zpočátku vytyčeno i několik cílů, kterých, bude-li to technicky možné, by bylo vhodné také dosáhnout. Šlo hlavně o implementaci schopnosti pojmenovat návěští podmíněných i nepodmíněných skoků slovy a nejen adresami v paměti, dále umět

nějakým (pokud možno) jednoduchým způsobem ověřit logiku vygenerovaného zdrojového kódu a eliminovat tak případná nestandardní parazitní data přidávané některými překladači a v neposlední řadě umět načíst alespoň dva základní formáty, ve kterých je strojový kód do souborů ukládán. Mimo to během tvorby průběžně vzniklo i zaniklo několik menších cílů, o nichž bude řeč v části věnované vlastnímu řešení zadané práce.

Forma aplikace

S ohledem na fakt, že aplikace nemá fungovat jako samostatný celek, ale jako součást softwarového balíku podobných programů, bylo samozřejmě nutné jí navrhnout nějaké rozhraní, pro propojení těchto modulů. Tímto rozhraním se nakonec stala velmi standardní a rozšířená *Dynamicky linkovaná knihovna* (DLL, dynamic link library). Tato knihovna je zvláštní typ souboru obsahující zkompilovanou funkci, kterou nabízí k použití jiným aplikacím.

DLL knihovnu však nelze spustit samostatně. Je třeba ji volat z nějakého spustitelného programu, což pro účely prezentování výsledků této práce není právě ideální. Z tohoto důvodu bylo dalším drobným, leč nutným cílem vytvoření velmi jednoduchého programu, který využije vytvořenou DLL knihovnu a demonstruje tak výsledky snažení.

Metodický přístup

Nastudování podkladů

Před započetím samotné práce na disassembleru bylo samozřejmě nutné doplnit si některé teoretické znalosti. Jednalo se zejména o nastudování principu a vlastností procesorů řady x51 a dále zopakování si programovacího jazyka (assembleru) pro tyto procesory. Samozřejmě během učení se assembleru bylo nezbytně nutné si některé vzorové programy zkusit napsat a pro lepší představu o jejich fungování i v překladači odkrokovat. Poslední teoretickou přípravou pak bylo vyzkoušení si některých již existujících disassemblerů, porovnání jejich výstupů a vytvoření si představy o tom, co by bylo vhodné zařadit a čeho se naopak vyvarovat.

Vlastní programování

Prvním krokem před započetím vlastní práce bylo zvolení si vyššího programovacího jazyka, protože nebyl zadáním práce jednoznačně určen. Uchýlil jsem

se k programovacímu jazyku *C* resp. (*C++*), neboť s tímto mám již poměrně značné zkušenosti (a tedy i znalosti o něm) a osobně mi nejvíce vyhovuje.

Programování disassembleru jsem pak prováděl po jednotlivých logických úsecích, které jsem si dopředu určil a které šly naprogramovat a otestovat samostatně. Nejprve samozřejmě přišlo čtení jednoduchého binárního souboru a poté vlastní rozpoznávání instrukcí procesoru. Následně se na to začaly nabalovat další funkční celky, jako kupříkladu rozpoznávání speciálních funkčních registrů, čtení jiných formátů souborů, rozpoznávání návěští u skoků… Během tohoto bylo samozřejmě nutné průběžně ošetřovat různé chybové stavy, které mohou nastat, a zároveň aplikaci průběžně testovat, zda funguje tak, jak je předpokládáno.

Výše uvedené programování a hlavně testování by nebylo dost dobře možné (nebo by bylo přinejmenším nepoměrně složitější), kdyby se celá aplikace vyvíjela rovnou jako DLL knihovna. Pro zjednodušení jsem tedy disassembler psal jako konzolovou aplikaci a až když byl hotový, převedl jsem jej na DLL knihovnu a k ní ještě vytvořil jednoduchý obslužný program.

1. Cíle práce

1.1 Vstupy aplikace

1.1.1 Podporované typy souborů

Strojový kód procesorů může být ukládán do celé řady typů souborů. Pro procesory řady x51 se však používají téměř výhradně pouze dva typy z těchto souborů. Jde jednak o prostý binární soubor, který obsahuje jen uložené přeložené posloupnosti instrukcí a jejich operandů v podobně „surových“ binárních dat.

Ještě o něco častěji než se souborem binárním se můžeme potkat s formátem *IntelHex*, který, jak jeho název napovídá, vyvinul původní tvůrce procesorů x51 – firma Intel, a jež je o mnoho sofistikovanější. Krom zakódovaných instrukcí a jejich operandů totiž obsahuje i údaj o adrese, na níž má být tento kód uložen, kontrolní součet pro ověření správnosti dat a v neposlední řadě má pevný formát, který lze zobrazit i v konvečních prohlížečích souborů.

Lze předpokládat, že při reálném použití aplikace bude vstupem právě jeden z těchto dvou typů souborů. Z tohoto důvodu bude nutné implementovat podporu obou dvou.

1.1.2 Další vstupy

S ohledem na skutečnost, že aplikace má pouze přeložit strojový kód zpět do zdrojového, se dá říci, že jakékoli další vstupy krom souboru zdrojového kódu jsou zcela zbytečné a jen komplikují cestu k výsledku. Předpoklad je tedy další „režijní“ vstupy pokud možno vůbec nezavádět, i když už dopředu existují náznaky, že to nebude dost dobře možné.

1.2 Výstupy aplikace

1.2.1 Zdrojový kód

Hlavním výstupem aplikace bude samozřejmě zdrojový kód assembleru přeložený z kódu strojového. Ten bude zapsán dle pravidel běžně používaných standardů do obyčejného textového souboru, ovšem odlišeného příponou *.asm*, která značí, že se jedná právě o zdrojový kód v assembleru. Tento kód by měl být do souboru zapsán a naformátován tak, aby ho bylo možné okamžitě zpětně přeložit do strojového kódu.

Disassemblery však obvykle slouží pro ladící účely, pro něž je pouhý výpis instrukcí assembleru a jejich operandů nedostačující. Pro plnohodnotné využití bývá potřeba také vidět adresy instrukcí a/nebo operandů, případně i číselné vyjádření instrukce či operandu. Z tohoto důvodu se ukazuje být nutné, aby disassembler generoval na svém výstupu soubory dva – druhý bude právě soubor pro ladící účely s výpisem adres a jejich obsahu.

1.2.2 Stavová hlášení

Každý dobře odladěný program podává průběžné zprávy o průběhu zpracování úkolu a samozřejmě také o případných nastalých chybách. Pro zadanou aplikaci, která bude fungovat pouze jako funkční celek jiného softwaru, platí toto tvrzení dvojnásob. Mateřský program totiž nebude mít nástroje jak sám zjistit, co se aktuálně děje, či kde při běhu nastala chyba. A nenabídnout uživateli požadovaný výsledek a zároveň ani nepodat zprávu/vysvětlení o tom, co se vlastně stalo, by bylo značně nešťastné řešení, odsuzující aplikaci do kategorie nepoužitelného softwaru. Proto nutným výstupem programu musí být také nějaká forma stavových hlášení informujících o průběhu, úspěšném ukončení, ale hlavně o případných chybách, jejich důvodech a eventuelně možných řešení.

1.3 Vlastní funkce disassembleru

1.3.1 Zpracování vstupních souborů

Strojový kód je logicky členěn na jednotlivé bajty – instrukce procesoru (samozřejmě s jejich případnými operandy). Ty mají velikost vždy jeden, dva nebo maximálně tři bajty, přičemž každý jednotlivý bajt má svůj logický smysl a význam. Proto bude bezpodmínečně nutné, aby aplikace uměla číst vstupní soubory po jednotlivých bajtech.

Aby měl celý proces převodu ze strojového kódu na zdrojový nějaký smysl a program byl odolnější proti „pádům“, bylo by nanejvýše vhodné, aby aplikace všemi dostupnými prostředky ověřila, zda vstupní soubor skutečně obsahuje strojový kód procesoru řady x51.

1.3.2 Poznávání instrukcí a jejich operandů

Samozřejmě nejdůležitějším cílem při tvorbě disassembleru je právě poznávání jednotlivých instrukcí a jejich operandů. To bude realizováno podle oficiální instrukční

sady firmy Atmel, současného majoritního výrobce procesorů řady x51. Samozřejmým souvisejícím požadavkem je zároveň také fakt, že toto poznávání musí být provedeno takovým způsobem, aby bylo bezchybné a jeho výstup zároveň byl zpracovatelný do výsledného souboru.

1.3.3 Návěští

Speciálně vytyčeným cílem je implementace poznávání tzv. návěští, tedy těch míst v paměti, která jsou cílem relativních i absolutních odskoků. Většina v rámci přípravy testovaných disassemblerů je totiž uvádí pouze v jejich číselné hodnotě. Tedy nepřiřazuje jim jména tak, jak to bylo uvedeno v původním zdrojovém souboru. To však má celou řadu nevýhod. Hlavní jsou ale dvě z nich – v prvé řadě je takový zdrojový kód pro programátora značně nepřehledný. Za druhé tímto způsobem napsaný kód není možné ihned zase zkompilovat. Jedním z hlavních cílů tedy je pojmenovat návěští slovy.

1.3.4 Nalezení parazitních dat

Některé překladače vkládají z různých (a mnohdy poměrně utajených) důvodů a účelů do strojového kódu data, které nemají s vlastním původním programem vůbec nic společného. Bohužel neexistuje v tomto žádný standard, takže tato data opravdu není možné v disassembleru nějak zpracovat. Oproti tomu je samozřejmě ale naprosto nežádoucí, aby se vyhodnocovala jako instrukce a jejich operandy, neboť tímto by se mohl výstup programu znehodnotit. Součástí aplikace by tedy měla být nějaká funkce, které ověří logiků výstupního kódu a případně označí ta data, která nadávají žádný logický smysl.

1.4 Výsledná forma aplikace

1.4.1 Forma samotné aplikace

Vzhledem k tomu, že vyvíjená aplikace má posloužit jako jeden funkční celek (modul) daleko rozsáhlejšího programu, musí mít nějaké (nejlépe standardizované) rozhraní pro začlenění do jiného softwaru. Jako toto rozhraní zvolíme univerzální a velmi rozšířenou DLL knihovnu. Výsledkem celého snažení by tedy měl být program ve formě DLL knihovny nabízející funkci disassembleru libovolné aplikaci, která splní požadavky na vstupy a bude schopna zpracovat nadefinované výstupy této knihovny.

1.4.2 Rozhraní pro prezentaci aplikace

Výsledná aplikace, tak jak byla naznačena v předchozím článku, bude ovšem funkční jedině za toho předpokladu, že ji nějaká jiná (nadřazená) aplikace spustí s nadefinovanými příslušnými vstupy. Pro potřeby prezentace výsledků této práce jde však o formu značně nevhodnou, neboť tato mateřská aplikace ještě není naprogramována a tudíž není technicky možné předvést výsledek celého snažení v činnosti. Proto jako další cíl přibývá ještě vytvořit jednoduchý program, který bude umět výslednou DLL knihovnu využít a demonstrovat tak její funkčnost. Pro původní účel práce se jedná o krok de facto naprosto zbytečný a tento software nebude následně vůbec užitečný. Ovšem na druhou stranu by neměl být větší problém jej velmi rychle vytvořit.

2. Teorie

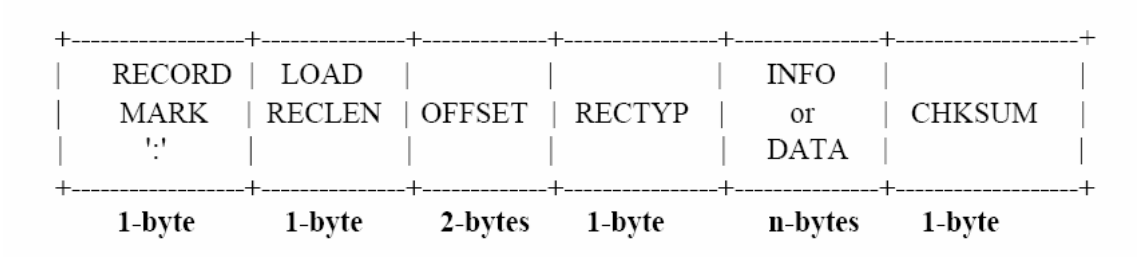
2.1 Typy souborů

2.1.1 Binární soubor

Binární soubory obsahují pouze „surová“ data a to má dvě základní výhody. V [1] bylo řečeno, že *„…výhodou binárních souborů je, že pro uchování stejného množství informace potřebují mnohem méně prostoru. Druhou výhodou binárních souborů je, že se s nimi pracuje mnohem rychleji než z textovými soubory. Důvody jsou dva – jednak jsou binární soubory kratší a jednak při zápisu čísla do textového souboru je nutné provést jeho konverzi z vnitřní reprezentace čísla v počítači na textovou podobu, což je časově náročné.*“

2.1.2 Soubor formátu IntelHex

Formát souboru IntelHex popsaný v [2], je vlastně textový soubor se zvláštní strukturou (viz obr. 2-1). Každý řádek je uvozen dvojtečkou. Za ní následuje jeden bajt vyjadřující velikost datové části (*n*), dále jsou dva bajty adresy v paměti, jeden bajt vyjadřující typ řádku (běžný řádek nebo konec souboru), dále následuje *n* bajtů datové



Obr. 2-1 - Struktura řádku v souboru IntelHex

části a předposlední bajt je kontrolní – součet všech předchozích (mimo uvozující dvojtečky) a jeho samotného musí být dělitelný 256. Posledním bajtem (ten na obr. 2-1 není již znázorněn, ale jeho přítomnost je logická) je standardní *CRLF (Windows, OS/2, DOS Line end*) – tedy ukončení řádku.

Formát IntelHex je snadno zobrazitelný i ve standardních textových prohlížečích, lze ověřit jeho správnost a nabízí i možnost zakódovat adresu instrukce (pseudoinstrukce *ORG*), což u binárního souboru lze jen se značnými obtížemi. Zřejmě také právě proto je široce podporován a využíván.

2.1.3 Zdrojový kód v assembleru

V [3] bylo řečeno, že „*…assembler je nejnižší programovací jazyk. Je to soubor instrukcí toho kterého procesoru, které se postupně zapisují do zdrojového souboru. Tyto instrukce mnemotechnicky vyjadřují jednotlivé povely pro procesor, jsou to zkratky anglických pojmů (např. instrukce DJNZ - decrement and jump is zero - dekrementuj a skoč, pokud je nula).*“ Pro zmíněný postupný zápis se vžila konvence taková, že co řádek zdrojového kódu, to jedna instrukce assembleru včetně jejích operandů. Přičemž vlastní název instrukce a operandy oddělujeme mezerou, více operandů mezi sebou pak čárkou. Tato zvyklost není závazná, pomáhá však k přehlednosti a je běžně používána, takže je nanejvýš vhodné se jí přidržet.

Assembler jako nižší programovací jazyk neobsahuje prakticky žádné zvláštnosti či speciality zápisu zdrojového kódu (cykly, porovnávání, matematické a logické operace atp.). De facto jedinými dvěma výjimkami z tohoto tvrzení jsou návěští a komentáře.

Návěští, jak již bylo zmíněno, označují místa, které jsou cílem podmíněných i nepodmíněných odskoků. Podle [3] je „*…návěští Vámi volitelný text, v podstatě cokoliv, kromě tzv. rezervovaných jmen.*“ A dále „*…návěští je od instrukce oddělené dvojtečkou*.“

Komentář pak slouží programátorovi pro zapisování informací o programu přímo do zdrojového kódu. Tyto poznámky jsou překladačem ignorovány, takže do strojového kódu se ani nedostanou. Mohlo by se tedy zdát, že pro účel disassembleru jsou k ničemu. To je však omyl, neboť představují ideální způsob, jak uživateli sdělit nějakou informaci, která může být důležitá, avšak není bezprostředně chybou a nemusí kvůli ní být tedy zastaveno zpracování strojového kódu. Způsob zápisu komentářů pak opět vysvětluje [3]: „*Cokoli ve zdrojovém kódu začíná středníkem, je bráno do konce řádku jako komentář.*“

Pro úplnost teoretických znalostí o zápisu zdrojového kódu assembleru bude vhodné ještě zopakovat v předchozím oddíle již zmíněnou informaci. Zdrojový kód má formát prostého textového souboru, ovšem kvůli odlišení bývá zvykem mu přidávat příponu *.asm*.

2.2 Disassembler

K čemu lze využít disassembler, jak funguje a jaké jsou jeho základní nevýhody vysvětluje velmi stručně a výstižně [4]: *„Napsat dobře fungující program nemusí být žádný velký problém. Ten však může nastat v okamžiku, kdy se od takovéhoto programu ztratí jeho zdrojový tvar, autor je neznámo kde a zmíněný program je zapotřebí nějakým velmi jednoduchým způsobem pozměnit, upravit, rozšířit, doplnit či jinak modifikovat – je tedy potřeba "vidět" do takového programu, který je k dispozici jen ve spustitelném (tj. binárním) tvaru. Zde by přišel velmi vhod takový prostředek, který by dokázal převést jednou přeložený program z jeho binárního tvaru zpět do tvaru zdrojového.*

Pokud byl příslušný program napsán v některém vyšším programovacím jazyku, pak takováto možnost nikdy nebude k dispozici - již jen z toho prostého důvodu, že vztah mezi zdrojovými a přeloženými programy není jednoznačný v tom smyslu, že překladem mnoha různých zdrojových programů může vzniknout jeden a tentýž přeložený program. Proces překladu tedy bohužel není obecně vratný. V principu však možné je - a to pro každý program v binárním (spustitelném) tvaru - převést jej do jazyka symbolických instrukcí neboli do assembleru. Tedy nahradit číselné tvary jednotlivých strojových instrukcí jejich symbolickým vyjádřením, které je pro člověka samozřejmě mnohem srozumitelnější a již samo o sobě výrazně přispívá k možnosti "vidět do" příslušného programu.

Tím ale možnosti převodu z binárního tvaru do asembleru ještě nemusí končit: další možností je i nahrazení číselných adres (použitých v roli operandů skokových instrukcí a instrukcí volání podprogramů) vhodnými symbolickými návěštími. Právě naznačený převod se v angličtině označuje jako disassembly (jako protiklad k procesu sestavení neboli "assembly") a program, který jej provádí, je tzv. disassembler. V češtině se často hovoří nepříliš správně o zpětném překladu (přičemž věcně správnější by asi bylo označení ve smyslu "překlad opačným směrem", protože například program napsaný původně v Pascalu se tímto způsobem nepřekládá zpět).

Se zpětným překladem jsou ovšem i některé principiální problémy. Ten největší vyplývá již ze samotné koncepce dnešních počítačů, stanovené ke konci druhé světové války americkým matematikem von Neumannem: ten totiž zavedl zásadu, že program a data jsou ve své podstatě jedno a totéž a že se mají uchovávat stejným způsobem na stejném místě (tj. v operační paměti); o tom, zda jde skutečně o program či data,

rozhoduje pouze způsob jejich interpretace procesorem. Nyní, při zpětném překladu, je ovšem správná interpretace ponechána na disassembleru. Ten se ale opravdu nemá "čeho chytit", resp. nemá podle čeho poznat, zda to, co v rámci binárního tvaru programu najde, jsou instrukce, nebo data.“

2.3 Procesory řady x51

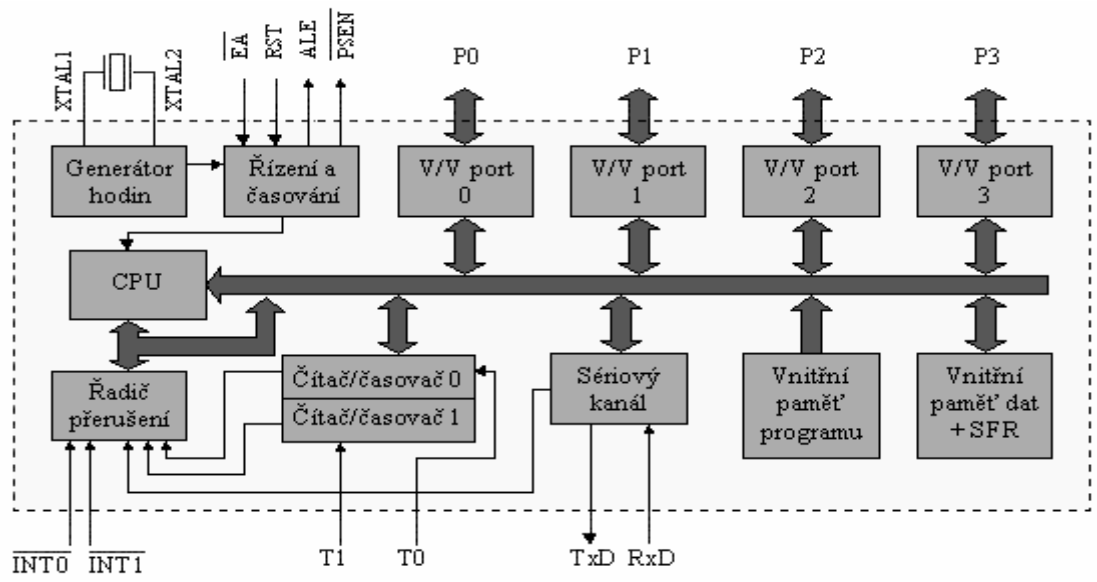
2.3.1 Struktura procesoru

Procesor Intel 8051, který je vzorem řady procesoru x51 měl, jak uvádí [8], *„…měl v roce 2000 za sebou již 20 let své existence. Přestože jde tedy o procesor velmi starý, je u návrhářů elektronických zařízení stále oblíben, stejně tak je i velmi často používán ve výuce na technických středních i vysokých školách. Mikroprocesor doznal do dnešní doby velmi mnoho variant s vylepšeními a doplňujícími periferiemi oproti jeho původní verzi.*“

Vnitřní architekturu procesoru řady x51 (schéma na obr. 2-2) popisuje ve stručnosti tentýž zdroj [8]: „*Mikroprocesor tvoří centrální procesorová jednotka (CPU), jejíž podstatnou částí je aritmeticko-logická jednotka. Ta umožňuje pracovat s jednotlivými bity paměti, vykonávat instrukce programu atd. Centrální procesorová jednotka je vnitřní 8-bitovou společnou sběrnicí propojena s pamětí programu a pamětí dat. Vnitřní paměť programu o velikosti 4kB může být typu ROM (8051), EPROM (8751) nebo mikroprocesor nemusí mít žádnou vnitřní paměť programu (8031). Vnitřní paměť dat je typu RAM o velikosti 128 bytů. Ke společné sběrnici jsou dále připojeny 4 vstupně/výstupní porty P0 až P3, které umožňují styk mikroprocesoru s vnějšími periferiemi.*

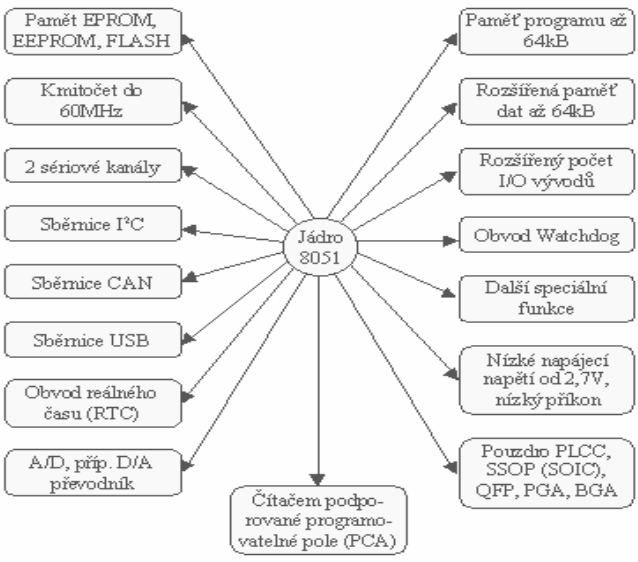
Chceme-li s mikroprocesorem používat větší pamět, než kterou nám poskytuje sám mikroprocesor, nebo chceme používat mikroprocesor 8031 bez vnitřní paměti programu, můžeme k mikroprocesoru připojit samostatnou vnější paměť programu a/nebo vnější paměť dat. K tomuto účelu jsou z mikroprocesoru vyvedeny řídící signály PSEN (paměť programu) a WR, RD (paměť dat). Pro snazší styk s periferiemi je mikroprocesor vybaven řadičem přerušení, který zpracovává 5 zdrojů přerušení - 2 externí (vývod INT0, INT1), od každého ze dvou čítačů/časovačů a od sériového kanálu. Jednotlivá přerušení mají definovanou prioritu na každé ze dvou volitelných úrovní priority. Mikroprocesor obsahuje dva 16-bitové čítače/časovače s volitelným režimem provozu. Pro snazší sériovou komunikaci s nadřízeným počítačem nebo

spolupracujícími mikroprocesory je mikroprocesor vybaven duplexním sériovým kanálem.“



Obr. 2-2 – Blokové schéma procesoru řady x51

Vzhledem k tomu, že (jak již bylo uvedeno) procesor 8051 vznikl před velmi dlouhou dobou, je logické, že existuje mnoho jeho různých rozšíření a variant. Právě proto také mluvíme o procesorech řady x51 a ne pouze o procesoru 8051. (Jméno firmy Intel, ač ho původní název obsahoval, se již vynechává, neboť procesory řady x51 již vyrábí jiné společnosti.) Stručný přehled těchto možných rozšíření lze vidět na obr. 2-3.



Obr. 2-3 – Rozšíření jádra procesoru 8051

2.3.2 Instrukční sada

Vzhledem k tomu, že procesory řady x51 jsou osmibitové, může teoreticky jejich instrukční sada obsahovat maximálně 256 (tj. 2^8) instrukcí. Reálně ovšem bývá do některých instrukcí rovnou zakódován i operand nebo jeho část (např. číslo registru či část adresy), takže ve výsledku je celkový počet možností, které mohou při poznávání instrukce nastat, výrazně nižší. (Soupis všech instrukcí je vložen na CD-ROM mezi přílohy této zprávy.)

Každá instrukce pracuje s jedním až třemi operandy. (S výjimkou některých instrukcí speciálních, které nemají operand žádný.) Tyto operandy se podle [8] dají rozdělit do devíti skupin. Jejich seznam a popis významů naleznete v tab. 2-1.

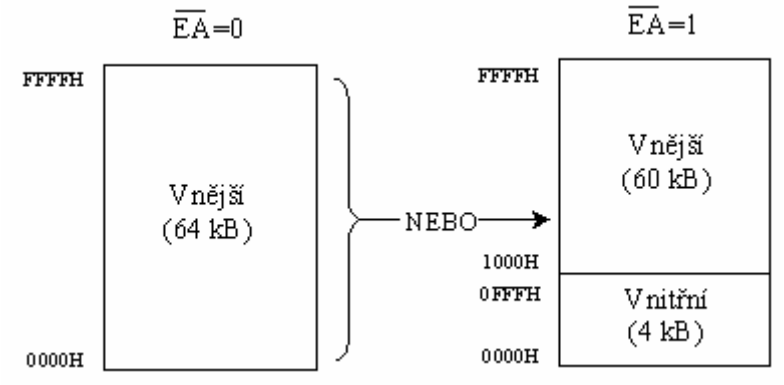
R_n	Registr R0 až R7 z právě vybrané registrové banky.
direct	Adresa 8-bitových dat v interní paměti nebo ve spec. funkčních registrech.
@R_i	Adresa 8-bitových dat v interní paměti, která se nachází v registru R0/R1.
#data	8-bitová konstanta
#data16	16-bitová konstanta.
addr16	16-bitová cílová adresa v paměti programu.
addr11	11-bitová cílová adresa v paměti programu.
rel	Odskok o -128 až +127 bajtů od prvního bajtu následující instrukce.
bit	Přímo adresovaný bit ve vnitřní paměti nebo spec. funkčních registrech.

Tab. 2-1 – Typy operandů procesoru řady x51

2.3.3 Organizace paměti

Procesory řady x51 mají dva oddělené paměťové prostory. Jednak prostor pro paměť programu a jednak prostor pro paměť dat. Podle [6] je *„…u základního mikroprocesoru 8051 vnitřní paměť programu velikost 4kB a vnější paměť má max.velikost 64kB. Mikroprocesor je vybaven vstupem EA, který řeší překrývání*

vnitřního a vnějšího paměťového prostoru. Je-li vstup EA = 0, potom paměť programu je tvořena celou vnější pamětí. Je-li vstup EA = 1, potom se instrukce v paměťovém prostoru 0000H až 0FFFH čtou z vnitřní paměti programu a mimo tento prostor (tedy z 1000H až FFFFH) ze zbývajících 60kB vnější paměti programu.“ (Viz obr. 2-4.)

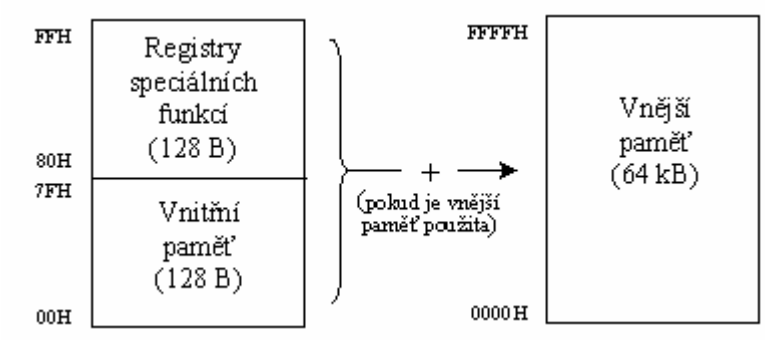


Obr. 2-4 – Struktura paměťového prostoru programu

Oproti tomu, bylo řečeno [6], že „*…vnitřní paměť dat má u základního typu 8051 velikost 128 bytů a tvoří ji (bráno od adresy 00h) čtyři banky, z nichž každá má po osmi registrech (vždy R0 až R7). Registry z neaktivní banky nelze adresovat symbolickým označením (R0, R1 … R7), ale pouze přímou adresou (absolutně). Za nimi následuje tzv. bitová oblast. Pro ni je vyhrazeno 16 bytů na adresách 20h až 2Fh. Bity z této oblasti jsou přímo adresovatelné. V oblasti od 30h do 7Fh se nachází zbývající vnitřní paměť. Tu lze adresovat pouze po bytech (přímé i nepřímé adresování). Nad touto oblastí, tedy na adresách 80h až FFh, se nacházejí registry speciálních funkcí (tzv. SFR).*

Vnější paměť dat s kapacitou až 64kB je přístupná přes 16-bitový pomocný ukazatel dat DPTR. Určitou nevýhodou je to, že DPTR může být pouze naplněn konkrétní adresou paměťového místa nebo inkrementován (DPTR může být zmenšován pouze odečítáním hodnoty od jeho dílčích částí DPH a DPL). Vnější paměť dat může být přístupná i s pomocí registrů R0 nebo R1 příslušné aktivní banky, které se využívají k 8-bitovému nepřímému adresování. Zapsáním 8-bitové hodnoty do výstupní brány P2, která představuje horní část adresy při adresování vnější paměti, vybereme jeden z 256 bloků paměti RAM o velikosti 256 bytů. Nepřímým adresováním pomocí registrů R0 a R1 potom lze zapsat nebo přečíst vybraný byte ve zvoleném bloku. Dolní část adresy vnější paměti se nastaví zapsáním 8-bitové hodnoty do výstupní brány P0 (horní+dolní

část adresy = 8+8 bitů, tedy celkem 16-bitová adresa, tou lze tedy adresovat právě oněch 64kB paměti).“ (Viz obr. 2-5)



Obr. 2-5 – Struktura paměťového prostoru dat

Zastavme se ještě na chvíli u SFR. O nich píše [6] toto: „*Oblast SFR je tvořena 21 registry, které jsou umístěny v paměťovém prostoru na adresách 80h až FFh. Nachází se tedy za zbývající vnitřní pamětí RAM nebo ve stejném paměťovém prostoru jako leží rozšířená paměť dat u novějších typů 8051. Z tohoto důvodu jsou SFR přístupné pouze pomocí přímého adresování bytů nebo bitů.*“ (Viz obr. 2-6)

F8H								FFH
F0H	B							F7H
E8H								EFH
E0H	ACC							E7H
D8H								DFH
D0H	PSW							D7H
C8H								CFH
C0H								C7H
B8H	IP							BFH
B0H	P3							B7H
A8H	IE							AFH
A0H	P2							A7H
98H	SCON	SBUF						9FH
90H	P1							97H
88H	TCON	TMOD	TL0	TL1	TH0	TH1		8FH
80H	P0	SP	DPL	DPH				PCON 87H

Bitově adresovatelné

Obr. 2-6 – Registry speciálních funkcí

2.4 DLL knihovny

Jak již bylo řečeno v úvodu, DLL knihovna je zvláštní typ souboru obsahující zkompilovanou funkci, kterou nabízí k použití jiným aplikacím. Pro přesnější představu o jejich hlavních rysech, výhodách i nevýhodách a smyslu použití bude opět nejlépe

citovat odborný zdroj [5]: *„Dynamicky linkované knihovny (DLL, dynamic link library) jsou další možností, jak psát modulární software. Knihovnou DLL máme na mysli zvláštní druh diskového souboru, označený příponou DLL, který obsahuje zkompilované funkce, zdroje, globální data. Základem je, že DLL poté nabízí k užití různé exportované funkce. Klientský program si pak v případě potřeby, připojí příslušnou knihovnu a tyto funkce importuje. Samotný operační systém Microsoft Windows je sestaven ze spolupracujících dynamických knihoven.*

Smysl užití dynamických knihoven spočívá v tom, že proces načítání a uvolňování v aktuálním okamžiku potřebných knihoven ve svém důsledku výrazně šetří operační paměť. Další klad knihoven DLL můžete najít ve vícenásobném využití funkcí knihoven u různých aplikací. Na druhé straně využíváte-li dynamického linkování knihoven při programování s MFC (i jinde), musíte vždy zajistit při přenosu programu přítomnost potřebných DLL knihoven pro běh programu. Odměnou vám bude menší velikost .exe souboru programu. Druhou možností je statické linkování programu, při kterém jsou všechny potřebné funkce přilinkovány k výslednému .exe souboru. Výsledkem bude podstatně větší .exe soubor, ale nemusíte se na druhou stranu zajímat o přítomnost dalších knihoven.“

3. Metodika práce

3.1 Studium teorie

3.1.1 Před započetím práce

Prvním nezbytným krokem před započetím samotné práce bylo samozřejmě zjistit, jak vlastně principielně funguje disassembler. Ten fakt, že tato aplikace načte strojový kód nějakého programu a po zpracování jej vypíše ve formě instrukcí assembleru, je bez pochyby jasný, nicméně to, co se děje uvnitř, není již zcela zřejmé a jednotliví tvůrci se o tom logicky příliš nezmiňují. V tomto ohledu mi asi nejvíce pomohla série anglicky psaných článků *Let's build a compiler!* od J.W.Crenshawa, Ph.D. a samozřejmě také nejedna porada s konzultantem mé práce, Ing. T.Martincem.

Další znalosti, které bylo nutno získat, se týkaly procesorů řady x51 – jak fungují, jak pracují s pamětí, co vše (ne)umí a v neposlední řadě také syntaxe assembleru, v němž je zapisován zdrojový kód programů pro ně. V tomto ohledu byly největším přínosem znalosti získané během studia na Technické univerzitě v Liberci. Konkrétně se jednalo o předmět *Počítače a mikropočítače*, který shodou okolností přednášel konzultant mé práce Ing. T.Martinec. Pro doplnění dalších faktů pak nejlépe posloužily manuály procesorům řady x51 od firmy Atmel v elektronické podobě a webové stránky D.Hankovce, které se zabývají procesory řady x51 a jež lze naleznout na adrese *www.dhservis.cz*. Doporučenou odbornou literaturu jsem nakonec nevyužil, neboť se mi zdála být nesrozumitelnou. Každou informaci jsem v ní musel vyhledávat příliš dlouho, a tak jsem ztrácel souvislosti. Zmiňované zdroje jsou naopak psány velmi stručně a účelně, což mi naprosto vyhovovalo.

Posledním krokem, který jsem před zahájením samotné práce na zadání učinil, bylo vyzkoušení si některých již existujících disassemblerů, porovnání jejich výstupů a rozhodnutí se, jak by zhruba měl vypadat výstup mé aplikace. Porovnával jsem disassemblery volně šiřitelné, které jsem nalezl pomocí fulltextového vyhledávače Google na internetu, a i několik disassemblerů komerčních, jež máme k dispozici v učebnách univerzity. S rozhodováním, co zařadit a co ne mi, samozřejmě velmi pomohl konzultant mé práce Ing. T.Martinec, neboť on má s prací s podobnými

programy celou řadu praktických zkušeností, a tak velmi dobře ví, co je pro reálné použití nejvhodnější.

3.1.2 Během práce

Před započetím práce samozřejmě nebylo možné nastudovat všechny potřebné podklady, neboť jsem dopředu nevěděl, na co při tvorbě můžu narazit a kde se skrývají různá problematická zákoutí. Na druhou stranu, díky nepodcenění teoretické přípravy jsem se do podobné situace dostával jen velmi zřídka, přičemž problém většinou spočíval v chybějící (nebo zapomenuté) znalosti o fungování procesorů řady x51 nebo o jejich instrukční sadě.

Valnou většinu výše zmíněných situací jsem bez problémů zvládl vyřešit načtením příslušných informacích v již zmiňovaných zdrojích – v manuálech k procesorům řady x51 od firmy Atmel v elektronické podobě a na webových stránkách D.Hankovce, které se zabývají procesory řady x51 a jež lze nalézt na adrese *www.dhservis.cz*. Zbývající nevelké procento chybějících teoretických znalostí jsem pak doplnil konfrontací s konzultantem mé práce Ing. T.Martincem nebo se svým kolegou z Technické univerzity v Liberci R.Paškou, jež souběžně se mnou zpracovával bakalářskou práci na podobné téma (*Assembler pro procesory řady x51*) a měl tedy také nastudováno mnoho teoretických podkladů.

3.2 Vlastní programování

3.2.1 Volba vyššího programovacího jazyka

Vzhledem k tomu, že zadáním mé práce byl určen fakt, že výstupní program má mít formu DLL knihovny, bylo dopředu jasné, že bude nutné zvolit nějaký vyšší programovací jazyk a jeho takové prostředí, jež bude schopno DLL knihovnu zkompilovat. Tento jazyk a prostředí však zadáním přímo definovány nebyly, záleželo tedy na mém rozhodnutí.

Po krátké úvaze jsem došel k závěru, že tu jsou dvě možnosti. Buď na Technické univerzitě v Liberci velmi oblíbené a vyučované *Delphi* a nebo jazyk *C* (resp. *C++*), který jsem se už před dlouhým časem naučil z vlastní iniciativy, a tudíž jsem také schopen v něm naprogramovat takto rozsáhlý projekt. Nakonec jsem se přiklonil k jazyku C, neboť mně osobně vyhovuje daleko více a podle mých osobních zkušeností je programování v něm daleko přehlednější a nepoměrně méně často se vyskytují různé

chyby vzniklé překlepy. (Respektive překladače jazyka C umí tyto chyby daleko přesněji lokalizovat. Narozdíl od Delphi, kde nezřídka např. při chybě na patnáctém řádku je zahlášena chyba až na řádku třicátém.) Mimo to jsem také zvyklý psát webové aplikace v jazyce PHP, které vycházejí právě z jazyka C, a mám tedy jeho syntaxi daleko lépe zažitou, což při vytváření takovéhoto většího projektu bylo velmi důležité a usnadnilo to celý proces.

Po volbě vyššího programovacího jazyka přišlo na řadu rozhodnutí, v jakém vývojovém prostředí pracovat. U zpočátku nabízejícího se prostředí *Borland C++* jsem po několik řádcích programů zjistil, že trpí podobnými nešvary jako výše zmiňované Delphi, takže jsem od něj velmi rychle upustil. Konečná volba pak padla na *Microsoft Visual C++*, jež mi doporučil Ing. P.Šolín, který se programováním aplikací pod operační systémy Windows intenzivně zabývá. Toto prostředí mi nakonec kvůli své strohosti a účelovosti naprosto vyhovovalo a díky tomu bylo vytvoření mé práce o dost snazší.

V závěrečných fázích práce však přeci jen došlo i na mnou nepříliš užívané vývojové prostředí *Borland Delphi*. Pro vzniklou DLL knihovnu bylo totiž nutné vytvořit ovládací program s grafickým rozhraním, což díky tomu, že z vlastní iniciativy programuji výhradně webové aplikace, v jazyce C++ neumím. Naopak vytváření takových rozhraní v Delphi znám z několika různých předmětů vyučovaných na Technické univerzitě v Liberci, tudíž by bylo více než zbytečné studovat principy dalšího jazyka.

3.2.2 Způsob programování

S ohledem na fakt, že vlastnímu programování a problémům při něm nastalých bude věnován celý oddíl této zprávy, zmíním se zde o metodice programování jen velmi stručně.

Poučen z článku J.W.Crenshawa, Ph.D., rozvhrnul jsem programování aplikace na mnoho malých funkčních celků. Ústředním principem tedy bylo naprogramovat vždy malou samostatně funkční část disassembleru, otestovat ji, zda pracuje správně, a teprve poté pokračovat v práci dále. Zároveň s tímto bylo samozřejmě vhodné dodržovat tzv. zlatou programátorskou zásadu: *„Použiješ-li něco více než dvakrát, udělej to jako funkci.“*

Nutno ovšem podotknout, že tyto myšlenky jsou sice krásné, bohužel jde ale o teorii, která se v praxi (obzvláště bez větších zkušeností) dodržuje jen velmi velmi těžko. A tak i já jsem se během programování disassembleru dostal do situace, v níž se zdrojový kód mé aplikace stal naprosto nepřehledným propletencem, kde většina funkční části byla nevhodně umístěna v jedné olbřímí funkci, a do nějž prakticky nešlo přidat další schopnosti, aniž by se tím narušil chod jiných částí programu. Po této zkušenosti jsem se rozhodl svůj výtvor přepracovat znovu od začátku. To už jsem ovšem věděl, jak jednotlivé části rozvrhnout a na co si dát pozor, tudíž na druhý pokus jsem již svou aplikaci vytvořil takovým způsobem, jak je popisováno v předchozím odstavci.

Velmi důležitou částí každé aplikace je rozpoznávání chybových stavů. Tedy zamezení „pádů“ programu a podání informace uživateli o tom, co se stalo. Vzhledem k tomu, že cílem mé práce bylo vytvořit pouze jednu funkci daleko rozsáhlejšího programu, nemusel jsem vymýšlet, jak chybové stavy řešit. Mnou vytvářená aplikace prostě a jednoduše proběhne buď úspěšně nebo se její běh přeruší z důvodu nastalé chyby. A v takovém případě pouze stačí, aby byl ohlášen typ chyby. Její řešení již pak bude ležet na bedrech „mateřského“ programu, který disassembler, jehož vytvářím, spouští.

3.2.3 Vhodnost formy aplikace pro vývoj

Jak již bylo v této zprávě mnohokráte řečeno, výsledná požadovaná forma disassembleru byla DLL knihovna, jež půjde použít jakou součást daleko rozsáhlejší a komplexnější aplikace. Bohužel DLL knihovny skýtají pro programování řadu nevýhod. Hlavním problémem se stává fakt, že průběh funkcí v nich obsažených je totiž zvenčí jakoby skryt. To je ale pro vývoj a ladění jakéhokoli programu poměrně nemilý fakt, neboť předpoklad, že se vše povede na první pokus a nebude potřeba důkladně prozkoumat běh některých funkcí krok za krokem, můžeme bez nadsázky nazvat utopií nebo přinejmenším velkou naivitou.

Proto jsem se při programování disassembleru uchýlil k lehké (na první pohled) komplikaci. Celý program jsem totiž nejprve vyvíjel jako konzolovou aplikaci spouštěnou a ovládanou příkazovým řádkem. A až teprve ve chvíli, kdy byla kompletně hotová, odladěná a bezchybně funkční, jsem ji převedl do požadované podoby DLL knihovny. Tento pro nezasvěceného zbytečný krok se nakonec ukázal být velmi

moudrým tahem, neboť při ovládání programu přes příkazový řádek lze velmi snadno doplnit mnoho různých vývojových funkcí, které poslouží pro hledání chyb a posléze budou zase smazány. Pokud bychom chtěli něco takového praktikovat s DLL knihovnou, museli bychom pokaždé (nebo minimálně jednou) vymýšlet a programovat nějaké rozhraní, pomocí nějž bude knihovna tyto ladící údaje poskytovat.

4. Vlastní řešení

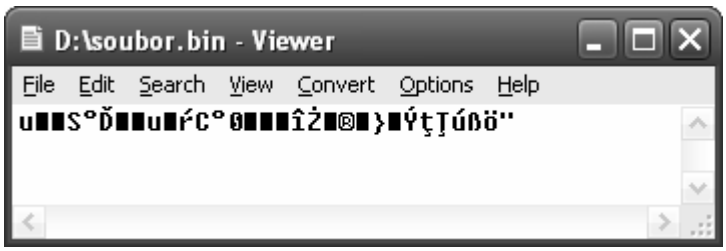
4.1 Vstupy aplikace

Jako vstup do aplikace byly zadány dva různé typy souborů. Vzhledem k tomu, že jejich struktura je dosti odlišná, nešlo použít stejné postupy pro jejich zpracování. Nicméně funkce obsahující tyto postupy dělají v zásadě to samé a jejich výstup je taktéž stejný, takže nemělo cenu je programovat odděleně. Místo toho jsem zvolil přístup rozvětvení funkce na dvě části. Kterou větví bude vstupní soubor zpracován záleží na tom, jakého je typu. Z tohoto důvodu bylo nutné přeci jen zavést ještě jeden vstup. Krom souboru (resp. jeho jména) ještě jeho typ, který bude určen obyčejným znakem. A to buď znakem '*B*' v případě binárního souboru, nebo znakem '*H*' v případě souboru ve formátu IntelHex. Tuto proměnnou bylo nutno definovat jako globální, aby mohla být použita právě i pro větvení podprogramů (funkcí), které aplikace ke svému běhu využívá.

Vzhledem k tomu, že strojový kód je logicky ukládán po jednotlivých bajtech, bylo nutné jej také po jednotlivých bajtech číst. To se nakonec ukázalo být daleko složitější, než se zpočátku zdálo, neboť vyšší programovací jazyky už zjevně moc nepočítají s tím, že by někdo mohl chtít pracovat s takovými malými daty jako je jeden bajt. Naštěstí datový typ *char* toto umožňuje, i když je původně určen pro práci se znaky a ne s čísly. Toto tvrzení však není úplně přesné, protože do samotného datového typu *char* bychom požadovaný jeden bajt po přečtení ze vstupního souboru sice uložili, ale jeho interpretace by nebyla správná, neboť rozsah tohoto typu je -127 až + 128 a my potřebujeme rozsah 0 až 255, poněvadž tak jsou vyjádřeny jednotlivé instrukce procesorů řady x51 (viz [8]). Museli jsme tedy použít datový typ *unsigned char*, který umožňuje uložit pouze kladná čísla.

4.1.1 Čtení binárního souboru

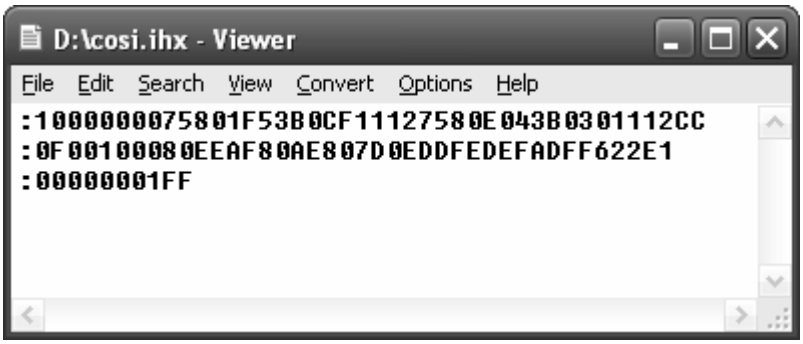
V teoretické části jsem uvedl, že binární soubor obsahuje jen „surová" (nijak nezakódovaná) data, bez jakýchkoliv jiných dat, které by kupříkladu vytvářela systém souboru (obr. 4-1). Z tohoto důvodu se omezilo jeho čtení na využití jedné standardní funkce. Každé zavolání této funkce způsobí přečtení následujícího bajtu strojového kódu do požadované proměnné. Tento bajt můžeme rovnou bez jakýchkoliv dalších úprav začít vyhodnocovat.



Obr. 4-1 – Náhled obsahu binárního souboru

4.1.2 Čtení souboru typu IntelHex

Naprogramovat čtení souborů ve formátu IntelHex bylo proti souborům binárním nepoměrně těžší. Tyto totiž mají určitou strukturu a tudíž krom dat strojového kódu obsahují ještě data další, která vytváří formát souboru (obr. 4-2). Výstupem funkce, která čte zdrojový soubor, však musela být pouze data strojového kódu. Ostatní údaje bylo nutné zahazovat nebo zpracovat jinak.



Obr. 4-2 – Náhled obsahu souboru IntelHex

Zda je následující čtený bajt strojový kód nebo pouze režijní údaj, lze snadno zjistit z pozice bajtu na řádku, který právě čteme. Nadefinoval jsem si tedy globální proměnnou, která uchovává údaj, kde se právě nacházíme. (Globální proto, aby toto číslo bylo zachováno i po úspěšném skončení běhu funkce – vrácení bajtu strojového kódu. Jeden řádek totiž může obsahovat i více bajtů strojového kódu, které není potřeba načíst všechny najednou.) Porovnáním tohoto čísla s další globální proměnnou – velikostí datové části řádku – a konstantními údaji, které vycházejí ze specifikací formátu IntelHex, lze snadno zjistit, zda se právě nacházíme na platném bajtu strojového kódu a můžeme běh funkce bezpečně ukončit, přičemž vrátíme právě tento bajt, nebo na některém z režijních údajů.

V případě, že se nacházíme právě na režijním údaji, musíme určitě číst soubor dále. Co ale s těmito daty? To je samozřejmě závislé na tom, který pomocný bajt zrovna načítáme. V případě uvozující dvojtečky, konce řádku a kontrolního součtu neprovedeme nic a přepíšeme data dalším bajtem. (Ověření kontrolního součtu bude prováděno ještě před během samotného disassembleru. Důvody vysvětlím v následujícím pododdíle.) Pokud bude načtena délka datové části řádku, uložíme ji do příslušné globální proměnné. Načtenou adresu dat uložíme do (dosud nezmíněné) globální proměnné pro uchování adresy (vysvětlíme také v následujícím pododdíle). A nakonec, bude-li typ načteného řádku „běžný řádek“, neprovedeme opět nic. Ovšem bude-li načten typ „konec souboru“, informujeme o této skutečnosti hlavní funkci programu.

V neposlední řadě bylo nutné dát pozor na fakt, že formát IntelHex je de facto textový soubor a znaky v něm obsažené jsou tedy zakódovány. Pokud tedy kupříkladu čteme hexadecimální jednobajtové číslo FF (255 desítkově), musíme přečíst ne jeden bajt, jak by se mohlo zdát, ale bajty dva. Tedy dvě čísla 70, která jsou kódem znaku ‘F‘. (Pouze uvozující dvojtečka je skutečně jednobajtová.) Toto překódování jsem provedl pomocí jednoduché podmínky a místo přečteného kódu zařadil již příslušné číslo. Pokud je čten první ze dvou znaků daného bajtu, musí se jeho hodnota samozřejmě vynásobit šestnácti.

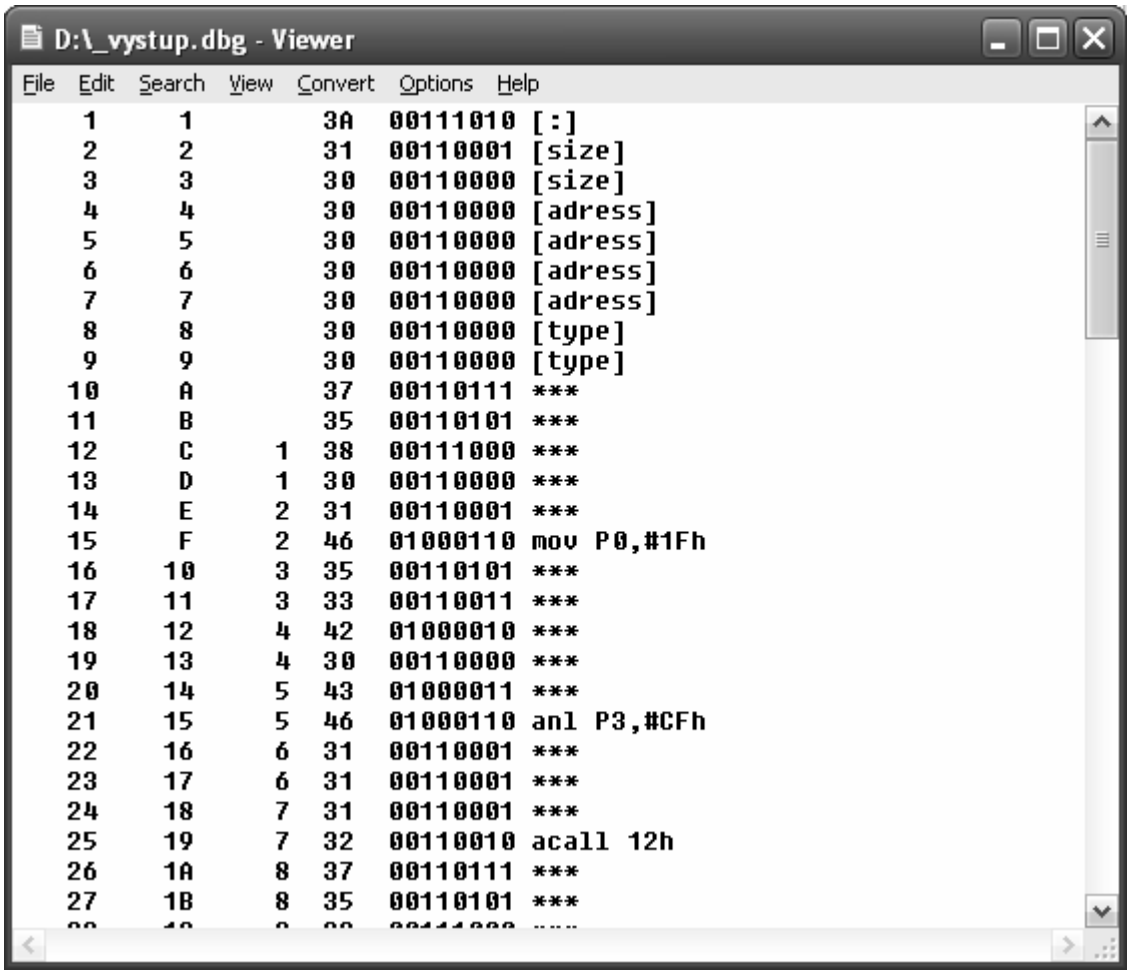
4.2 Výstupy aplikace

4.2.1 Výstup do souborů

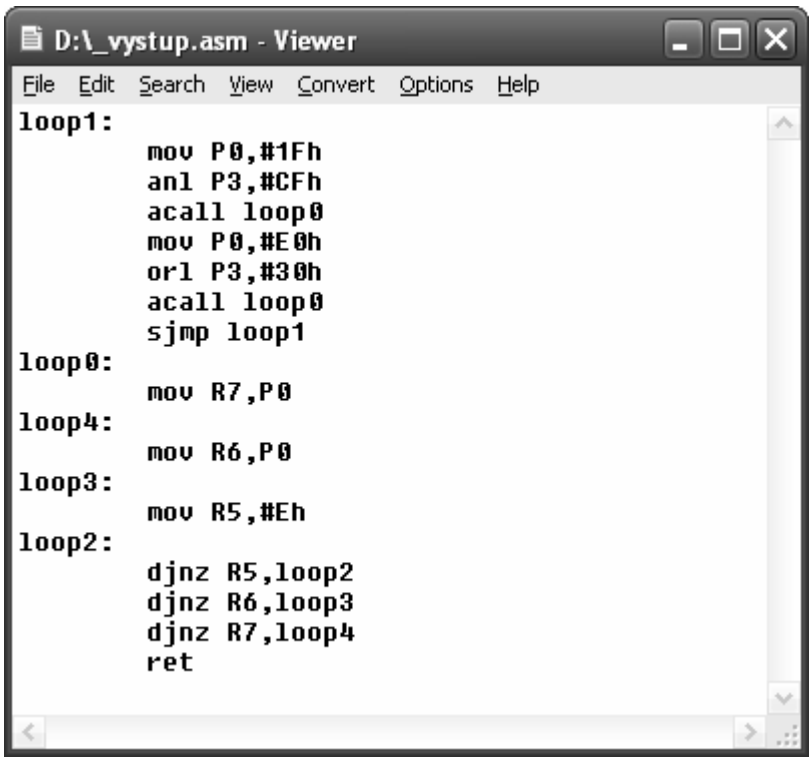
Již v cílech této práce bylo naznačeno, že výstupní soubory by měly být dva. Jednak „ladící“ soubor s výpisem jednotlivých bajtů, jejich adres a příslušných instrukcí a jednak soubor s „čistým“ zdrojovým kódem assembleru – tj. takovým kódem, který půjde rovnou opět zkompilovat. Oba dva jsou běžné textové soubory (pouze mají zvláštní přípony), které lze vytvořit a zapisovat do nich pomocí standardních funkcí.

V případě souboru pro ladění nenastali žádné komplikace. Jak postupně jednotlivá data strojového kódu čteme, tak je do tohoto souboru s příslušnými údaji zapisujeme (obr. 4-3). Ovšem u souboru s „čistým“ zdrojovým kódem bylo již zpočátku jasné, že takto jednoduše to provést nelze. Problematickým místem se staly instrukce skoků, které mohou směřovat (a nezřídka samozřejmě směřují) do míst před aktuální pozicí (obr. 4-4). O tom ale samozřejmě při postupném vyhodnocování nemůžeme ve

chvíli, kdy jsme na cílovém místě, vědět a nelze sem tudíž příslušné návěští zapsat. A zpětné doplňování údajů do textového souboru je velmi obtížné. Proto jsem rozhodl, že celý proces postupného čtení vstupního souboru a poznávání instrukcí bude proveden dvakrát. Poprvé bude vytvořen ladící soubor, v němž budou cíle skoků uvedeny číselně a návěští zde nebudou vůbec, neboť soubor (jak již bylo řečeno) obsahuje adresy jednotlivých bajtů. Zároveň s ním ale vznikne i pole adres, které jsou cílem všech podmíněných i nepodmíněných odskoků. Při druhém průběhu pak na každé platné adrese bude testováno, zda sem nesměřuje některý z těchto uvedených odskoků. V případě, že ano, vypíše se jeho název do příslušného místa. (Ten samý název bude pak samozřejmě vypsán i k příslušné instrukci místo číselné adresy.)



Obr. 4-3 – Náhled obsahu výstupního („ladícího“) souboru



Obr. 4-4 – Náhled obsahu výstupního („čistého“) souboru

Výše naznačené řešení má na první pohled zásadní nevýhodu. Zpracování vstupního souboru bude trvat dvakrát tak dlouho a tedy teoreticky dvakrát tak více zatíží počítač, což není známkou dobře odladěného programu. To není ale úplně tak pravda. Je třeba vzít v úvahu například fakt, že nebude muset být alokována operační paměť pro další proměnné, které by bylo nutné v případě vytváření obou výstupů najednou zcela jistě nadefinovat. Dále musíme počítat se skutečností, že jiné řešení výpisu návěští by vyžadovalo složité programové konstrukce, které by ve výsledku měly možná ještě pomalejší běh, než tento způsob. Připočteme-li k tomu ještě fakt, že vstupní soubory mohou mít velikost maximálně 64kB a jejich zpracování bude tedy na současných počítačích záležitostí velice rychlou, lze prohlásit, že naznačené řešení se na použitelnosti aplikace v reálném provozu nikterak nepodepíše.

4.2.2 Chybová a stavová hlášení

Druhým a posledním výstupem aplikace jsou krom souborů ještě chybová a stavová hlášení. Jejich počet není velký a byla realizována prostřednictvím zobrazení

textové zprávy, která uživateli vysvětlí, kde se stala chyba nebo jinak informuje o průběhu zpracování vstupního souboru.

4.3 Poznávání instrukcí

Připomeňme si nyní stručně kostru aplikace, kterou jsem již vytvořili definováním vstupů a výstupů. Hlavní funkce programu proběhne dvakrát a při každém běhu vytvoří jeden výstupní soubor. Jeho vytváření bude probíhat postupně tak, že se zavolá podprogram, který načte ze vstupního souboru bajt strojového kódu, hlavní funkce jej vyhodnotí, zavolá ji znovu pro další bajt atd.

4.3.1 Konec poznávání

Zapřemýšlíme-li nad výše uvedenou kostrou, zjistíme, že se de facto jedná o „základní verzi“ disassembleru – sice bez pokročilých schopností, ale funkční. Pouze s jedním detailem. Nevíme, kdy průběh poznávání instrukcí ukončit. Řešení je ovšem nasnadě. Konec běhu hlavní části programu nastává ve chvíli, kdy dojdeme na konec vstupního souboru.

Z tohoto důvodu jsem před spuštění samotného vyhodnocování umístil jednoduchý podprogram, který postupným čtením souboru po bajtech až do znaku *EOF* (End Of File, konec souboru) zjistí jeho velikost. Čtené údaje se samozřejmě nijak nezpracovávají, pouze se přepisují následujícími.

Nezpracovávání čtených údajů při zjišťování velikosti souboru se zprvu zdálo být logické, ale posléze jsem přeci jen z dodržování této zásady lehce slevil. Vzhledem k tomu, že v tomto místě běhu programu se ještě nepoznávají instrukce – tedy neprovádí se hlavní činnost aplikace – je zde nanejvýš vhodné ověřit správnost dat. (Samozřejmě jen u formátu IntelHex. U binárního souboru to provést nelze.) Pokud toto ověření totiž neproběhne v pořádku, ušetříme procesorový čas, který by se jinak věnoval vyhodnocování instrukcí až do místa, kde ověření selže. A navíc na výstupu aplikace se neobjeví nic (krom chybového hlášení), což je zcela jistě lepší varianta, než kdyby uživatel dostal nějaký nedokončený soubor.

Vraťme se ale k hlavní myšlence tohoto článku. Ve chvíli, kdy jsem měl k dispozici hodnotu velikosti souboru, bylo ukončení poznávání snadnou záležitostí. Celé to obstarává cyklus, který načítá postupně bajty od prvního až po velikost souboru a vyhodnocuje je. Každý průběh tímto cyklem tedy znamená rozpoznání jedné

instrukce. V případě vícebajtových instrukcí, kdy se načtou při průběhu cyklu další bajty ze zdrojového souboru, pak stačí pouze s každým načtením inkrementovat řídící proměnnou.

Uvedený způsob má ještě jednu výhodu. Aktuální hodnota řídící proměnné cyklu je zároveň i adresou příslušného bajtu, kterou vypisujeme do „ladícího“ souboru. Tedy alespoň v případě binárního souboru. V případě souboru IntelHex bylo nutné zavést ještě jednu proměnnou, která se inkrementuje pouze při načtení platného bajtu strojového kódu.

4.3.2 Vlastní poznávání

Samotné poznávání instrukcí nebylo záležitostí ani tak programátorského umu a znalostí, jako spíše trpělivosti. I když trochu invence bylo přeci jen potřeba. Teoreticky může nastat 256 možných situací, takže nezkušeného programátora by tato situace vedla k podmínce *switch-case* (přepni do části, pokud hodnota je…). To ovšem není úplně šikovné, neboť mnoho instrukcí má v sobě již zakódovanou část operandu, tudíž na její poznání zbývá pouze sedm až pět bitů z celkových osmi, takže tato instrukce zabírá daleko více pozic z 256 celkových než jen jednu.

Situaci jsem tedy vyřešil obyčejnými podmínkami *if* (pokud), které u každé instrukce testovaly pouze příslušné bity a tedy pro nalezení zmíněných instrukcí zabírajících více možností stačila jedna podmínka (obr. 4-5). Navíc toto vyhodnocování probíhalo jen dokud se nenalezla vyhovující podmínka. Posléze jsem nastavil pomocnou booleovskou proměnnou na nulu, další možnosti již testovány nebyly. (Pokud je instrukce již poznaná, žádná jiná to být nemůže.)

S tímto řešením ovšem přišla drobná komplikace. Moje aplikace musí umět pracovat s binárními čísly, což však překladač jazyka C neumožňoval. Musel jsem tedy doplnit funkce pro převod čísla na jeho binární vyjádření pomocí řetězce jedniček a nul (pro poznávání instrukcí) a naopak (pro získání částí operandů z kód instrukce). Nutno podotknout, že to nebylo právě jednoduché, neboť práce s řetězci není v jazyku C právě nejjednodušší. Na druhou stranu to ale velmi přispělo k „čistotě“ kódu programu a jeho optimalizaci, což jistě stálo za to.

```
if(zpracovano) {                                    // @@@ MOV A,Rn @@@

    instrukce[0] = '1';                             // MOV A,Rn => 1 1 1 0 1 r r r
    instrukce[1] = '1';
    instrukce[2] = '1';
    instrukce[3] = '0';
    instrukce[4] = '1';
    instrukce[5] = 'X';
    instrukce[6] = 'X';
    instrukce[7] = 'X';

    OK = 1;

    for(k = 0; k < 5; k++) {

        if(binarne[k] == instrukce[k]) ;
        else OK = 0;
    }

    if(OK) {

        pracovni[3] = '@';
        pracovni[2] = binarne [7];
        pracovni[1] = binarne [6];
        pracovni[0] = binarne [5];

        UnBinar(pracovni);
        if(dvakrat) fprintf(fw, "\t");
        fprintf(fw, "mov A,R%ld\n", dekadicky);

        zpracovano = 0;
    }
```

Obr. 4-5 – Podmínka poznávající instrukci, jež má v sobě zakódován operand

4.3.3 Výpis instrukce

Po poznání instrukce už stačilo pouze ji vypsat. Samozřejmě ve většině případů bylo nutné ještě doplnit operandy. A to jak zakódované přímo v instrukci, tak uložené v dalších bajtech. To ale s již existujícími funkcemi pro čtení souboru bylo otázkou pouze zavolání této funkce a vypsání její návratové hodnoty ve správném tvaru. Ve výjimečných případech, kdy operand byl zakódován jak v přímo v instrukci, tak v následujícím bajtu, byla před jeho vypsáním ještě nutná konverze na binární řetězce, složení těchto řetězců do jednoho a jeho zpětná konverze na číslo.

Výpis instrukce bylo také nutné odlišit podle toho, zda se jednalo o první kolo vyhodnocování a tedy vytváření „ladícího“ souboru nebo o průběh druhý a tedy vytváření souboru „čistého“ kódu v assembleru.

4.4 Další funkce

4.4.1 Poznávání SFR

Některé instrukce assembleru umožňují přímý přístup do paměti. Tento se zdaleka nejčastěji využívá pro práci s speciálními funkčními registry, kde jsou

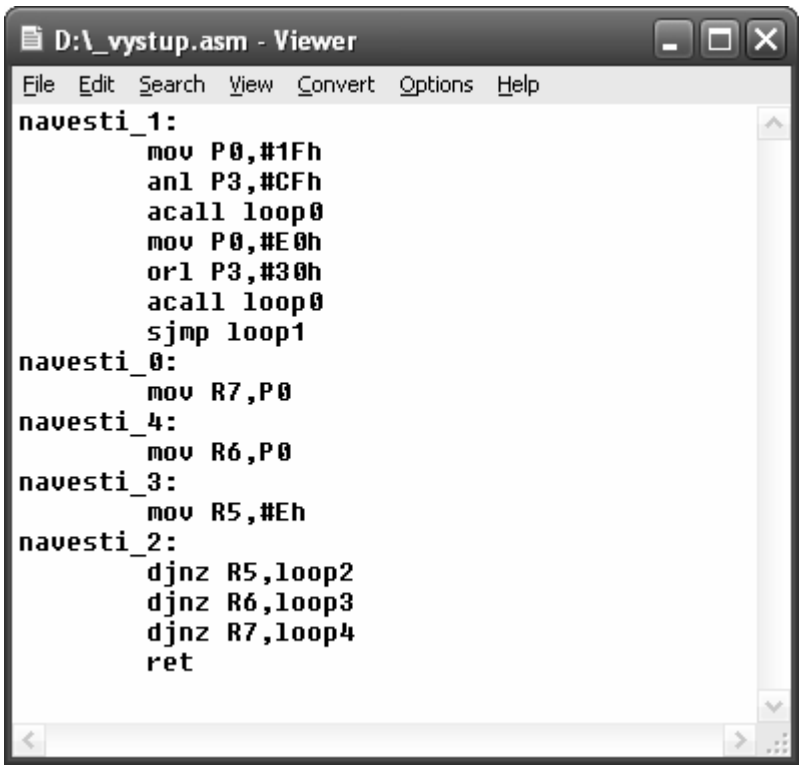
shromážděny důležité informace o stavu mikroprocesoru a jeho periferních obvodů, ale zároveň i informace ovlivňující jeho další činnost. Tyto registry mají sice svojí pevnou číselnou adresu, avšak většina z nich má zároveň i svůj slovní název (obr. 2-6). Pro přehlednost a smysluplnost přeloženého kódu bylo samozřejmě bezpodmínečně nutné použití těchto registrů v kódu poznat a doplnit jejich slovní názvy namísto číselné adresy.

Při řešení tohoto úkolu nezbylo nic jiného, než se uchýlit ke klasickému začátečnickému, ale programátorsky značně nevhodnému a nešikovnému principu, který by se dal stručně popsat slovy „*jedna proměnná testovaná desítkami podmínek na svůj obsah*“. Aby byl kód aplikace přeci jen alespoň nějakým způsobem optimalizován, vytvořil jsem zdvojení vyhodnocovacích podmínek. Jak to celé funguje? Nejprve je rozpoznán jeden z jedenácti bajtů, které obsahují pojmenované SFR a teprve v rámci něj se testuje adresa konkrétního registr. V případě, že je přímo adresován bajt mimo SFR, se tak samozřejmě na vyhodnocování konkrétních registrů uvnitř SFR vůbec nedostane a ušetříme tak průchod několika desítkami podmínek.

4.4.2 Návěští

Způsob pojmenovávání návěští jsem naznačil už v článku 4.2.1, nicméně pokusím se pro lepší představu o tom, jak to celé funguje, celý postup shrnout do souvislé myšlenky.

Narazí-li při prvním průběhu vyhodnocování aplikace na podmíněný či nepodmíněný odskok, prozkoumá pole s cílovými adresami odskoků a pokud tam ještě taková cílová adresa není, zařadí ji na první volné místo tohoto pole. K instrukci do „ladícího“ souboru pak vypíše tuto adresu pouze číselně. Při druhém průběhu vyhodnocování prozkoumá aplikace na každém bajtu, kde takový odskok může být (tedy první bajt každé instrukce), pole s cílovými adresami odskoků. Pokud v něm příslušnou adresu nalezne, zařadí na toto místo návěští ve tvaru *navesti_#*, přičemž místo znaku # vloží proměnnou s pořadovým číslem příslušné adresy v tomto poli (obr. 4-6). Adekvátním způsobem se zachová také v tom případě, kdy narazí na instrukci podmíněného či nepodmíněného odskoku. Tím je zajištěno, že návěští a odskoky budou stejně pojmenovány. A to dokonce i v případě, že na jedno návěští odkazuje více instrukcí zároveň, což je poměrně běžný stav.



Obr. 4-6 – Pojmenovávání návěští ve výsledném zdrojovém kódu

Výše zmíněný styl pojmenovávání návěští mi nebyl při konzultacích ohledně této práce doporučen. Ideálnějším způsobem by bylo volit jména kupříkladu tak, aby bylo na první pohled vidět, kterého typu instrukce je toto návěští cílem. (Kupříkladu tedy *navesti_djnz_#* pro odskok z instrukce DJNZ.) Tento princip však kromě toho, že by byl značně komplikovaný na naprogramování a tudíž i náročnější na výpočet pro počítač, skýtá jeden daleko zásadnější problém. Návěští, která by byla cílem odskoku z instrukcí více typů, by měla více jmen. Tudíž bychom se dostali do situace, že za cenu větší práce programátora i aplikace získáme ještě menší přehlednost výsledného zdrojového kódu assembleru. Proto jsem se rozhodl tento způsob pojmenovávání neimplementovat.

4.4.3 Ověření logiky kódu

Výstup z disassembleru by měl být skutečně platný zdrojový kód. Bohužel, jak již bylo řečeno, některé překladače přidávají do strojových kódu různá nestandardizovaná doplňková data, která disassembler samozřejmě vyhodnotí jako

instrukce. Takto vytvořený kód však postrádá logiku a bylo by vhodné jej přinejmenším nějak označit.

Původní myšlenkou bylo, že do těchto „parazitních“ dat se nemůže program (program, který vykonává procesor řady x51) během svého běhu dostat. Tudíž pokud bychom udělali nějaký skript, který dokáže postupně projít zdrojový kód tak, že instrukce, přes které projde, označí jako platné, měl by být tento problém vyřešen. Skript si nemusí vůbec všímat významu instrukce. Jedině v případě, že se jedná o skok by si tento skok uchoval a posléze (u skoku podmíněného, u skoku nepodmíněného okamžitě) prošel i větev, do níž skok odkazuje.

Bohužel toto řešení se ukázalo být v praxi neúčinné, neboť nezřídka se při procházení kódu dostal výše nastíněný skript i do oblasti dat, která již neměla s původním zdrojovým kódem vůbec nic společného. Tento efekt si lze vysvětlit jedině tak, že přístup různých překladačů je natolik rozdílný, že podobně jednoduchý podprogram nemůže „parazitní“ data odhalit. Pokud bychom tedy chtěli ověřit logiku dat, museli bychom naprogramovat celý simulátor činnosti procesoru řady x51, který se pokusí provést výsledný zdrojový kód tak, jak by činil právě procesor řady x51. S velkou pravděpodobností by tak odhalil ty části kódu, jež postrádají smysl. Simulátor pro procesory řady x51 však již nebyl předmětem této práce.

Z výše uvedeného důvodu jsem se nakonec rozhodl již vytvořený podprogram pro ověřování logiky kódu do výsledné aplikace nezařadit, neboť nemá cenu prezentovat, že aplikace umí to či ono, když ve skutečnosti funguje tento kód jen ve slabé třetině testovaných případů. Přeci jen cílem této práce bylo vytvoření fungující aplikace, která pomůže při výuce na Technické univerzitě v Liberci, nikoliv polovičatého softwaru, který se i přes jeho zjevné nedostatky budeme snažit vnutit nějakému zákazníkovi.

Nutno ovšem podotknout, že mnoho případných nelogických částí kódu je označováno již při vyhodnocování instrukcí. Kupříkladu u podmíněných i nepodmíněných skoků testujeme, zda cílová adresa vůbec existuje, a pokud ne, dopíše se do zdrojového kódu komentář s příslušným faktem. Taktéž návěští na místech, kde nemají co dělat (jinde než na prvním bajtu instrukce) se do zdrojového kódu vůbec nevypisují. Lze tedy prohlásit, že i tento bod zadání byl alespoň částečně v místech, kde to bylo technicky možné a účinné, splněn.

4.5 Finální úpravy

4.5.1 Převod aplikace na DLL knihovnu

Po naprogramování celé aplikace již zbylo pouze převést ji do požadované formy – tedy na DLL knihovnu. Zpočátku se zdálo, že tento proces bude záležitostí na „minutu“, což se bohužel tak úplně nepotvrdilo.

V čem převedení na DLL knihovnu vlastně spočívá? Ve vývojovém prostředí jsem založil nový projekt, ale nikoliv jako konzolovou aplikaci, avšak právě jako DLL knihovnu, čímž jsem automaticky dostal k základní soubory potřebné k jejímu vytvoření. Do hlavního zdrojového kódu jsem překopíroval zdrojový kód a okomentoval výstupy typické pro konzolovou aplikaci (smazání příkazové řádky, výpisy textu do ní atp.). Hlavní funkci aplikace jsem pak musel přejmenovat a nadefinovat jako vstupní bod DLL knihovny. Jejím názvem (prozaicky jsem zvolil „*disassembler*“) se pak bude volat celý disassembler z „mateřské“ aplikace.

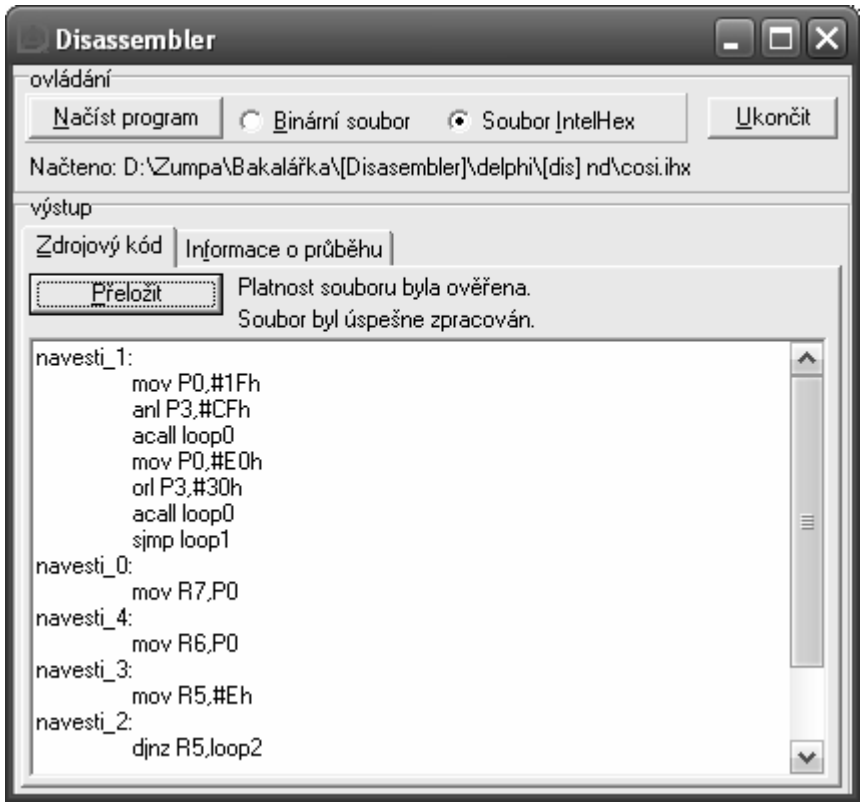
Nejdůležitějším a nejproblematičtějším bodem však bylo nadefinování vstupů a výstupů z DLL knihovny. Vzhledem k tomu, že se kvůli rozdílnému přístupu k paměti v „mateřské“ aplikaci a DLL knihovně ukázalo být značně problematické předávání čehokoliv jiného než prostých čísel mezi nimi, byly nutné nějaké zásahy do původního principu a řídících proměnných. Navíc bylo třeba vzít v úvahu, že celá aplikace se nyní chová jako funkce, tudíž může mít pouze jednu jedinou návratovou hodnotu. Rozhodl jsem se tedy, že vstupní i výstupní soubory budou mít pevné pojmenování, které bude zakomponováno přímo do DLL knihovny, aby se ušetřilo problematické předávání řetězců. Bude pak starostí „mateřské“ aplikace, aby DLL knihovně předložila správně pojmenovaný vstupní soubor a dokázala načíst a případně pod jiným názvem uložit standardně pojmenované výstupní soubory. Vstupem do knihovny pak bude jedno číslo určující, zda se jedná o soubor binární nebo formátu IntelHex. (Změna proti původní identifikaci pomocí znaků ‘*B*’ a ‘*H*’.) A konečně výstupem bude opět jedno číslo, které bude reprezentovat určitý stavový / chybový kód. (Přesný popis vstupů a výstupů vytvořené DLL knihovny naleznete v příloze této zprávy na CD-ROM.)

Při vytváření DLL knihovny jsem se dostal do situace, kterou bez nadsázky mohu i zpětně nazvat velmi zoufalou. Zdrojový kód konzolové aplikace, který bez problému fungoval, po převedení do DLL knihovny odmítal jakoukoliv spolupráci. Sáhodlouhým testováním jsem zjistil, že problém je v globálních proměnných, které na

první pohled jakoby nefungovaly. Ovšem žádná syntaktická chyba v programu nebyla. Po mnoha hodinách práce zkoumání jsem nakonec přišel na to, že globální proměnné se při použití v DLL knihovně musí nejen deklarovat, ale rovnou i inicializovat. Pokud se totiž inicializují až v některé funkci (připomínám, že u DLL knihovny je i hlavní část kód pouhou funkcí), ostatní funkce „vidí“ tuto proměnnou jako nezinicializovanou – prázdnou. Tento fakt byl asi největším problémem, s kterým jsem se během práce potkal.

4.5.2 Aplikace pro demonstraci funkce DLL knihovny

Pro účely demonstrace funkčnosti DLL knihovny jsem nakonec ještě vytvořil jednoduchou grafickou aplikaci (obr. 4-7). Tato utilita nechá uživatele vybrat libovolný vstupní soubor, zvolit jeho typ a po vyžádání zavolá funkci z DLL knihovny. Pevně dané jméno vstupního souboru je vyřešeno tak, že aplikace dočasně překopíruje obsah vstupního souboru do jiného, pojmenovaného právě tímto pevným názvem. Po ukončení běhu funkce DLL knihovny tento soubor zase smaže. Výstup už je pak velmi prostý. Aplikaci do dvou záložek načte oba výstupy z DLL knihovny – tedy jako „ladící“ soubor, tak soubor s „čistým“ zdrojovým kódem.



Obr. 4-7 – Grafické rozhraní pro využití DLL knihovny

Aby toto spojení grafického rozhraní a DLL knihovny bylo funkční, je potřeba aplikaci oznámit, kde má knihovnu hledat a jakou funkci z ní přečíst. To obstará definice, kde na prvním řádku je jméno funkce tak, jak bude dále využita v obslužném programu a typ její návratové hodnoty. Druhý řádek obsahuje definici volání funkce, což je způsob, jakým spolupracují obě součásti. Nakonec to nejdůležitější, informace o umístění knihovny a název funkce tak, jak jej DLL knihovna obsahuje.

Zmíněný typ volání funkce se ukázal být jako velmi důležitá součást celého programu. Díky kombinaci knihovny naprogramované v *C++* a rozhraní v *Delphi* docházelo bez této definice k problémům s využití knihovny. Důvodem byl odlišný typ volání, který standardně každé rozhraní volí jinak a díky čemuž obě součásti nebyly schopné navzájem komunikovat. Po definování typu volání již bylo fungování celého spojení hladké i přes skutečnost, že každá část byla naprogramována v odlišném prostředí a dokonce i v odlišném programovacím jazyku.

5. Výsledky

5.1 Vytvořená data

5.1.1 Schopnosti výsledné aplikace

Schopnosti výsledného disassembleru de facto přesně odpovídají požadavkům na něj zpočátku kladeným. Aplikace tedy umí společně s informací, který z nich je do vkládán ke zpracování, načíst dva typy souborů (binární a IntelHex), vyhodnotit je po jednotlivých bajtech strojového kódu a udělat z nich kód zdrojový. (Je-li to možné, tak otestuje i syntaktickou správnost vstupního souboru.) Samozřejmě správně pozná všechny možné instrukce a jejich operandy. Mimo to umí i pojmenovávat speciální funkční registry, inteligentně vypisovat a pojmenovávat návěští a v omezené míře i ignorovat nebo opoznámkovat chyby či neplatné části kódu. Dále dokáže během práce podávat některé relevantní informace o své činnosti a v případě nastalé chyby zahlásí, co je v nepořádku. Naopak v případě, že svou práci dokončí bez nastalých komplikací, uloží výsledky do dvou různých souborů – jeden s detailním výpisem, druhý už pouze se zdrojovým kódem v assembleru.

5.1.2 Rozhraní výsledné aplikace

Díky zvolenému způsobu vytváření aplikace jsem tedy nakonec vytvořil tři různé programy.

Prvním byla konzolová aplikace se funkcí disassembleru, která se tedy musí spouštět a ovládat přes příkazový řádek. Její výhodou je, že s uživatelem komunikuje zdaleka nejvíce, protože u ní nebyl problém s více vstupy a výstupy. Lze jí tedy zadat libovolný název souboru, jeho typ a zda se má (u formátu IntelHex) ověřovat kontrolní součet. Výstupem pak jsou informace o průběhu (např. velikost zpracovávaného souboru) a samozřejmě dva výstupní soubory.

Druhým programem je pak tato konzolová aplikace přetvořená do DLL knihovny. (Tedy DLL knihovna s funkcí disassembleru.) Díky této konverzi přišla o možnost větší komunikace s uživatelem – ta se omezuje pouze na jedno vstupní a jedno výstupní číslo. Vše ostatní je pevně dáno (jako název vstupního souboru) a aplikace očekává, že nějaký nadřazený program jí toto připraví. Stejně tak očekává, že „přeloží“ požadavky uživatele na vstupní číslo a výstupní číslo na zprávu uživateli. Na druhou

stranu ji díky tomuto rozhraní lze implementovat do libovolného programu, který dokáže tyto požadavky splnit.

Třetím a posledním programem je velice jednoduchá aplikace, která umí komunikovat, připravit vstupy a vyhodnotit výstupy výše zmíněné DLL knihovny. Sama o sobě nemá žádný význam, ale společně s touto DLL knihovnou ji lze v operačních systémech Windows použít jako plnohodnotný disassembler pro procesory řady x51 s grafickým rozhraním.

5.2 Získané znalosti

5.2.1 Znalosti o procesorech řady x51

Díky studiu teoretických podkladů, ale hlavně díky následnému překládání instrukcí mohu prohlásit, že během práce jsem se velmi dobře naučil porozumět významu jednotlivých instrukcí assembleru a jejich operandů. Samozřejmě jsem také získal notné znalosti o tom, jak postupuje procesor při vykonávání jednotlivých instrukcí. V tomto ohledu asi nejvíce znalostí přineslo pochopení principu, na kterém fungují absolutní a zvláště pak relativní odskoky. V neposlední řadě jsem také načerpal znalosti o organizaci a využití paměti u procesorů řady x51. Jediná část problematiky, o které jsem se nemusel příliš moc dozvědět, je využití čítačů a časovačů, neboť tyto jsou z pohledu disassembleru pouhými speciálními funkčními registry a pochopení jejich významu není prakticky vůbec nutné.

5.2.2 Programování

Díky této práci jsem také výrazně oživil a prohloubil své znalosti z oblasti programování. Obzvláště se jedná o práci s binárními i textovými soubory v jazyce C a také využití podprogramů (funkcí), předávání proměnných mezi nimi a využití globálních proměnných.

Zcela novou znalostí se pak stalo vytváření DLL knihovny, z kterého jsem měl zpočátku velké obavy. Jak jsem již zmiňoval, velký problém sice nastal, ale na druhou stranu jeho řešení nevyžadovalo studování rozsáhlých podkladů, ale pouze dosti trpělivosti a umění hledání chyb. Samotné vytvoření DLL knihovny se ukázalo jako poměrně prosté.

Hledání chyb a ladění programu zabralo vůbec nemalou část (možná i nadpoloviční) času stráveného nad programováním disassembleru. Do situace, že jsem

několik hodin strávil krokováním běhu aplikace, vypisováním obsahu proměnných a hledáním chyb, jsem se dostal během práce hned několikrát. V tomto ohledu jsem tedy také načerpal mnoho zkušeností. Nejdůležitější z nich je asi ta, že chyby, které na první pohled nemůžou nikdy nastat, nastávají obvykle v jiné části programu, než v které člověk problém hledá. A že občas je lepší příliš nepřemýšlet, ale důkladně si projít všechny kroky a zkusit, zda se vše chová tak, jak je očekáváno.

Závěr

Kvalita výsledků

Dle mého názoru jsou dosažené výsledky velmi uspokojivé a v některých ohledech dokonce lehce předčili leckterá má očekávání.

Schopnosti aplikace

Co se týče schopností aplikace, neshledávám jediné místo, jehož výstup by nebyl vyhovující. Obzvláště pak funkce na vyhledávání a pojmenovávání návěští nakonec funguje velmi precizně, přestože vymyslet její princip i naprogramovat ji bylo značně problematické. Jediná drobná výtka by snad mohla směřovat k ověřování logiky kódu, avšak to, jak již bylo řečeno, nebylo možné na úrovni disassembleru dostatečně efektivně řešit.

Forma aplikace

Taktéž forma aplikace splňuje relativně vysoké nároky, a to i přesto, že cílem práce bylo vytvořit funkční modul pro jiný software (DLL knihovnu) a na možnost samostatného využití byl kladen jediný nárok – aby bylo možné její schopnosti prezentovat. Jediné místo, kde by asi bylo možná úroveň ještě zvýšit, jsou vstupy a výstupy DLL knihovny. Pevně dané jména souborů a „komunikace“ za pomoci pouhých čísel není úplně optimální.

Porovnání s již existujícími disassemblery

Vytvořená aplikace určitě snese srovnání s již existujícími programy stejných schopností. V některých ohledech je dokonce předčí – mám tím na mysli zejména již několikrát zmíněné „inteligentní“ pojmenovávání návěští a dále uživatelsky pohodlné rozhraní a možnost začlenit funkci disassembleru do libovolného komplexnějšího programu díky jeho rozhraní DLL knihovny.

Dosažení vytyčených cílů

Vstupy aplikace

Vytvořená aplikace skutečně umí zpracovat dva nejrozšířenější formáty souborů zdrojového kódu – jak binární soubor, tak soubor formátu IntelHex.

Pouze cíl nezavádět další vstupy nebyl úplně splněn, avšak jedna proměnná se dvěma možnými stavy není až takovou komplikací a kvalitu produktu rozhodně žádným způsobem nesnižuje.

Výstupy aplikace

Taktéž na výstupu aplikace získáme oba požadované soubory – jak soubor s „čistým“ zdrojovým kódem v assembleru, tak soubor včetně výpisů hodnot jednotlivých bajtů, jejich adres atd., který poslouží pro účely ladění.

Aplikace také ve slušné míře informuje uživatele o průběhu své činnosti a má ošetřeny a umí uživateli popsat všechny chyby, které mohou během zpětného překladu nastat.

Vlastní funkce

Aplikace skutečně umí zpracovat data z obou formátů souborů, bezchybně vyhodnotit všechny instrukce assembleru procesorů řady x51 včetně jejich operandů, poznat speciální funkční registry a pojmenovat i vypsat slovně jednotlivá návěští. Jedině poznávání „parazitních“ dat přidávaných některými překladači není na příliš vysoké úrovni, ovšem ne vinou nízké snahy či kvality aplikace.

Forma aplikace

Požadavek na formu aplikace, tedy DLL knihovnu včetně jednoduchého grafického ovládacího rozhraní byl splněn beze zbytku. Krom toho byla v průběhu vytvořena i konzolová verze.

Citace

[1] HEROUT, Pavel. *Učebnice jazyka C*. Třetí upravené vydání. České Budějovice: KOPP, 1998. Kapitola 6, s. 82-83. ISBN 80-85828-21-9

[2] HANKOVEC, David. *Jak se naučit programovat*. 2002-2006. URL: http://www.dhservis.cz/dalsi_1/popis.htm

[3] INTEL, kolektiv autorů. *Hexadecimal Object File: Format Specification*. 1988. URL: http://www.interlog.com/~speff/usefulinfo/Hexfrmt.pdf

[4] PETERKA, Jiří. *Disassembly*. 1994.

URL: http://www.earchiv.cz/a94/a450c120.php3

[5] POKORNÝ, Martin. *Výuka programování G1: DLL knihovny*. 2006. URL: http://tamnekde.unas.cz/data/prg/prg_g1/prg_g1_15.php

[6] HANKOVEC, David. *Popis 8051*. 2002-2006. URL: http://www.dhservis.cz/popis.htm

[7] ATMEL, kolektiv autorů. *Memory Organization*. 1997. CD ROM: x51_memory_organization.pdf

[8] ATMEL, kolektiv autorů. *Microcontroller Instruction Set*. 1997 CD ROM: x51_instruction_set.pdf