FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Emulátor instrukční sady výukového procesoru
Student:	Jiří Šebele
Vedoucí:	doc. Ing. Jan Schmidt, Ph.D.
Studijní program:	Informatika
Studijní obor:	Počítačové inženýrství
Katedra:	Katedra číslicového návrhu
Platnost zadání:	Do konce zimního semestru 2019/20

Pokyny pro vypracování

Vytvořte emulátor instrukční sady jednoduchého procesoru, který by mohli používat studenti seznamující se s programováním v asembleru. Instrukční sada má být nastavitelná tak, aby se student s jednotlivými rysy asembleru mohl seznamovat postupně. Instrukční sada by se měla podobat procesoru AVR, který pak studenti programují. Emulátor musí dovolit základní ladicí úkony, jako je krokování, sledování a modifikace obsahu pamětí.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Hana Kubátová, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 2. března 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Emulátor instrukční sady výukového procesoru

Jiří Šebele

Katedra číslicového návrhu

Vedoucí práce: doc. Ing. Jan Schmidt, Ph. D.

14. května 2018

Poděkování

Děkuji svému vedoucímu za připomínky a pomoc při psaní práce. Děkuji své rodině a přátelům za podporu. Dále bych rád poděkoval všem respondetům průzkumu a všem účastníkům uživatelského testování. V neposlední řadě bych rád poděkoval své partnerce Arlette a svému příteli Davidovi, kteří mi byli po celou dobu psaní práce inspirací i podporou.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 14. května 2018 .....

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na tuto práci

Šebele, Jiří. *Emulátor instrukční sady výukového procesoru*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018. Dostupný také z WWW: ⟨`https://github.com/jiri/thesis-paper`⟩.

Abstrakt

Tato práce se zaměřuje návrh a implementaci překladače, emulátoru a ladící aplikace pro instrukční sadu jednoduchého procesoru, který pomůže úplným začátečníkům zorientovat se v problematice programování v asembleru. Překladač pak umožňuje omezit instrukční sadu pro úlohy, kde chceme demonstrovat konkrétní rysy asembleru.

Klíčová slova Rust, C++, Qt, Assembly, AVR, Emulátor

Abstract

This work focuses on the design and implementation of an assembler, emulator and a debugging application for an instruction set of a simple processor, which will enable beginners to orient themselves in the field of programming in the assembly language. It allows constraining of the instruction set for exercises, in which we want to demonstrate specific attributes of assembly programming.

Keywords Rust, C++, Qt, Assembly, AVR, Emulator

Obsah

Seznam obrázků

Seznam tabulek

Seznam ukázek kódu

Úvod

S postupujícím technologickým vývojem je elektronika se kterou se běžně setkáváme čím dál složitější. Instrukční sada procesorů s architekturou x86 od firmy Intel nyní obsahuje tisíce instrukcí[2], což je oproti původnímu procesoru 8086[3] závratné číslo. Tento vývoj je ve světle čím dál větší potřeby pro výpočetní výkon nevyhnutelný, ale rostoucí komplexita hardwaru vytváří nepřátelské prostředí pro naprosté začáteníky.

V dobách, kdy nejsofistikovanější domácí počítače jako Amiga nebo Commodore 64 byly poháněny osmibitovými procesory s jednoduchou instrukční sadou, bylo začít s programováním pro danou platformu často pouze otázkou přečtení manuálu, nebo pomoci od učitele ve škole. Dnes je však spouštění aplikací pod kontrolou operačního systému, zavaděč systému je v některých případech zamčený a instrukční sada tak složitá, že obsáhnout ji je pro začátečníka takřka nemožný úkol.

Přesto však existuje zjevný zájem o nízkoúrovňové programování. Populární platforma Arduino se rapidně blíží milionu aktivních uživatelů a díky dobré programové podpoře je nahrávání programů do mikroprocesorů jednodušší, než kdy dřív. Přesto však existuje bariéra, kterou představuje nutnost vlastnit fyzické zařízení schopné daný kód spouštět.

Cíle práce

V naší práci se pokusíme navrhnout virtuální platformu, která se pokusí být ještě jednodušší, než nejjednodušší z existujících mikroprocesorů a bude možné s ní experimentovat na kterémkoliv moderním domácím počítači.

Struktura práce

Tato práce je rozdělena do sedmi kapitol. V první kapitole prozkoumáme, s jakými problémy se začátečníci setkávají při programovaní pro mikrokontrolery. Dále se potom seznámíme s existující programovou podporu pro vývoj na platformě AVR. Prozkoumáme možnosti emulace běhu mikrokontrolerů z rodiny AVR na platformě x86.

V druhé kapitole rozvedeme specifikaci aplikačního software, který v rámci práce budeme vytvářet. Stejně jako samotná práce je pak i návrh rozdělen na čtyři části:

- architektura mikrokontroleru a instrukční sada,
- assembler, tj. aplikace, kterou budeme překládat symbolický zápis na strojový kód,
- emulátor, realizovaný formou knihovny,
- ladící aplikace, kterou budeme emulovat běh na platformě x86.

V následujících čtyřech kapitolách se budeme každé části věnovat detailněji. Vzhledem k tomu, že jsou na sobě jednotlivé části víceméně nezávislé, budeme se věnovat návrhu a realizaci každé z nich pohromadě. Nejdříve zvážíme dílčí technologie, které bychom při realizaci mohli použít a každou z nich detailněji prozkoumáme. Poté si přiblížíme detaily samotné realizace a popíšeme, jak byly řešeny problémy, na které při práci jistě narazíme. Na konci zhodnotíme výsledné řešení a jak se použité technologie osvědčily.

Ve sedmé a poslední kapitole námi navržený aplikační software otestujeme s několika uživateli. Kapitola obsahuje popis metodiky testování, jeho průběh a zhodnocení výsledků. Součástí je také návrh úloh, které při testování použijeme.

Kapitola 1

Analýza problému

V této kapitole se pokusíme zjistit, s čím mají začátečníci největší potíže při práci s mikrokontrolery. Prozkoumáme názory studentů FIT ČVUT, kteří absolvovali předmět BI-SAP. Dále se pak podíváme na existující programovou podporu pro vývoj programů pro platformu AVR.

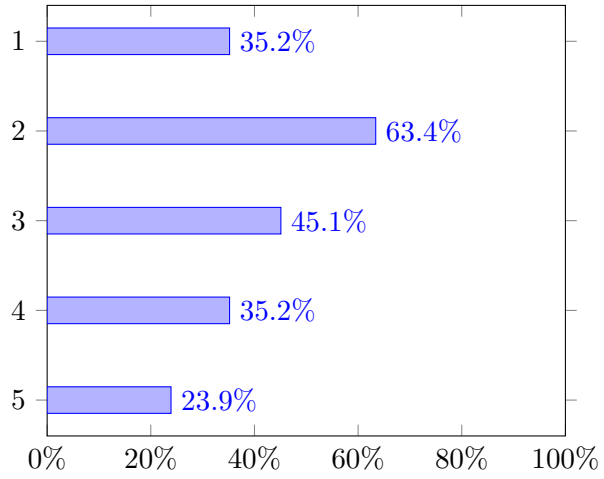
1.1 Dotazník

Na FIT ČVUT se studenti poprvé setkávají s vývojem pro mikrokontrolery nejčastěji v druhém semestru v předmětu BI-SAP. Jsou proto ideálním zdrojem informací o tom, jak začátečníci k programování mikrokontrolerů přistupují, s jakými se setkávají potížemi a co jsou pro ně nejobtížnější překážky k překonání. Prozkoumáme tudíž jejich spokojenost s průběhem výuky pomocí krátkého dotazníku.

Samotný dotazník je velice jednoduchý, ptá se pouze zda respondent souhlasí s následujícími tvrzeními:

1. Assembly mi dělalo potíže.
2. Nemohl jsem dělat úkoly doma / bez přípravku.
3. Vyvíjet pro AVR nešlo na macOS / Linuxu.
4. Programy nešlo dobře ladit.
5. AVR je příliš komplikovaná platforma.

Původním plánem bylo nechat dotazník zveřejněný, dokud nenasbíráme alespoň 100 respondentů. V poměrně krátkém časovém intervalu jednoho



Obrázek 11: Procenta kladných odpovědí na otázky mezi respondenty

dne jsme získali 110 odpovědí a jejich sběr jsme proto ukončili. Ručně jsme protřídili odpovědi, které nebyly validní a zůstalo nám 95 odpovědí. Výsledek dotazníku je poté uveden v grafu na obrázku 11.

Z odpovědí je nejenom patrné, že je problematické samotné programování v jazyce symbolických adres, ale že jednoznačně největším problémem je závislost na fyzickém přípravku, který musí studenti mít, aby mohli plnit úlohy. Dále se pak jako problém ukázala také nízká kvalita programové podpory pro vývoj pro platformu AVR.

1.2 Současný stav řešení

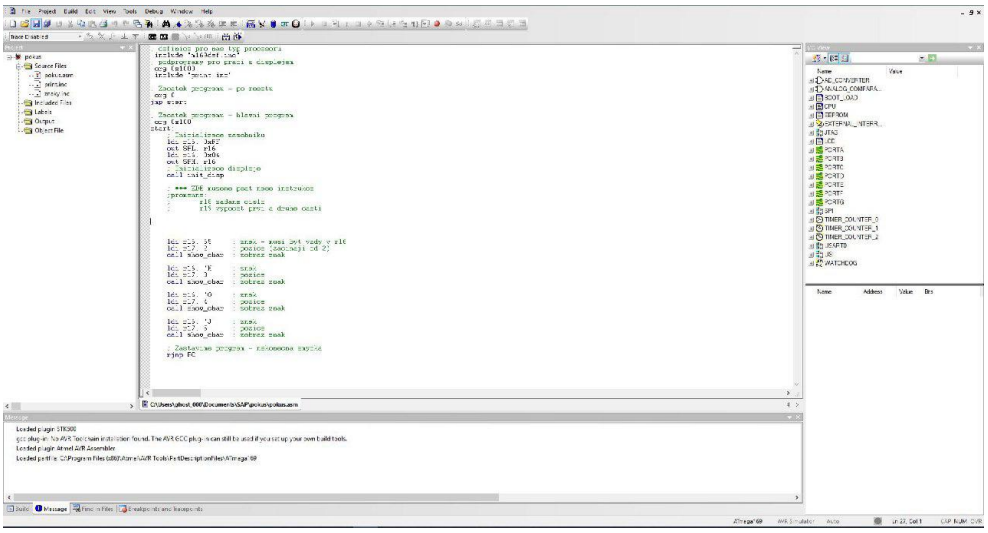
Aplikací, která se v současné době na FIT ČVUT používá při výuce předmětu BI-SAP je AVR studio (konkrétně verze 4)[4]. Jedná se o kompletní vývojovou sadu přímo od společnosti Atmel[1]. Aplikace je však kompatibilní pouze se systémem Microsoft Windows[5], je ji proto těžké nebo nemožné zprovoznit na alternativních operačních systémech. To je pro některé uživatele zásadní problém a aplikace proto není kompletním řešením.

Nejpřímočařejším řešením vývoje pro platformu AVR je manuální překlad a nahrávání programu do zařízení. Z našeho oblíbeného jazyka vyprodukujeme patřičným překladačem binární soubor, který poté slinkujeme a nahrajeme do zařízení pomocí programátoru a aplikace. Toto řešení je pro zkušené uživatele

[1] V roce 2016 došlo k odkoupení společnosti Atmel konkurenční společností Microchip. Od té doby došlo k přejmenování aplikace **AVR studio** na **Atmel studio**, avšak některé verze si stále zachovávají původní jméno.

často nejjednoduší, avšak obzvlášť pro začátečníky pracující v operačním systému, ve kterém není užití konzolových aplikací možné nebo jednoduché, je tato cesta takřka nemožná.

Pro platformu AVR existuje několik dalších aplikací, které usnadňují vývoj a nahrávání programu. Ukážeme si z nich dvě – simavr[6] a Arduino IDE[7].



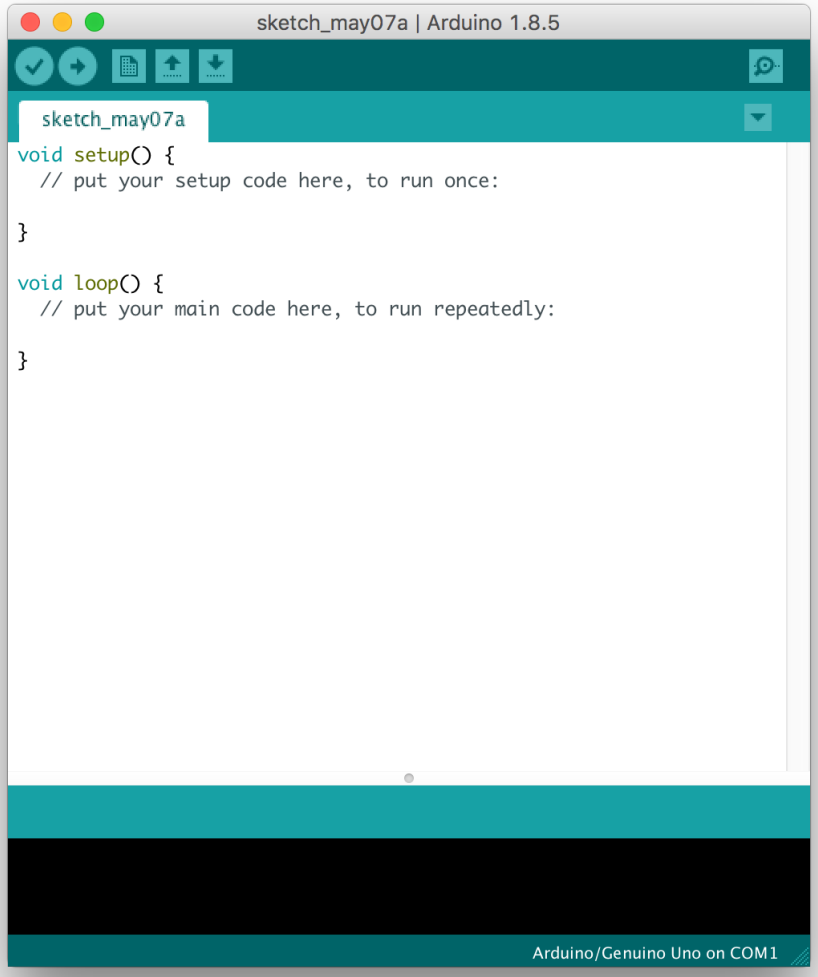
Obrázek 12: AVR studio 4

Zajímavým projektem je simavr[6]. Jedná se o konzolovou aplikaci, která umožňuje emulovat běh různých mikrokontrolerů z rodiny AVR. Poskytuje výbornou integraci s ladícím programem gdb, zpětnou logickou analýzu pomocí souborů ve formátu VCD (Value change dump) a snadnou rozšiřitelnost pomocí externích vizualizačních nástrojů. Díky tomu se jedná o výborný nástroj pro lazení složitějších programů. Kvůli tomu je však aplikace nepřívětivá k uživatelům, kteří nejsou již zkušenými programátory, nebo nemají detailní přehled o platformě AVR. Zároveň není aplikací nijak zprostředkován překlad zdrojového kódu do strojového. Uživatel je tudíž nucen řešit tento problém externím překladačem, jako například AVRA[8], nebo Avr-GCC[9].

Jednou z aplikací cílené na naprosté začátečníky je Arduino IDE[7]. Jedná se o aplikaci usnadňující vývoj pro prototypovací platformu Arduino. Aplikace vyčnívá mezi konkurencí svým simplistickým vzhledem a jednoduchostí používání. V aplikaci se programuje v jazyce C, obohaceném o knihovnu funkcí unsadňující ovládání mikrokontroleru. Překlad a nahrání programu do mikrokontroleru proběhne stiskem tlačítka „Upload“. Aplikace také usnadňuje komunikaci s externím zařízením po sériové lince pomocí zabudované konzole. Nikterak však uživateli neusnazuje lazení běžícího programu a uživatel je tak

ponechán napospas chybám, které jako začátečník nevyhnutelně udělá.

Mezi její nevýhody však patří, že se pokouší odstínit své uživatele od příliš mnoha aspektů programování pro mikrokontrolery. Nahrávání i překlad programu probíhá automaticky. Aplikace za uživatele doplní i hlavičkové soubory pro danou platformu. Knihovna funkcí poskytnuta uživateli, ačkoliv usnadňuje vývoj, nereprezentuje dobře operace probíhající „pod kapotou“. Jedná se proto o znalosti, které jsou nepřenositelné na platformy s Arduinem nekompatibilní.



Obrázek 13: Arduino IDE

1.3 Shrnutí

Z uvedených aplikací žádná není ideálním řešením pro naprosté začátečníky, pokusíme se proto navrhnout vlastní. Pokusíme se naši platformu navrhnout co uživatelksy nejpřívětivější, aby s ní mohl pracovat i naprostý začátečník. Zároveň se však pokusíme aby byla dostatečně realistická na to, aby následný přechod k platformě AVR byl co nejjednodušší.

Kapitola 2

Specifikace a cíle práce

Po dokončení analýzy současných řešení jsme připraveni definovat které problémy budeme v práci řešit a které problémy spadají mimo rámec práce.

2.1 Využití

Reálných platforem existuje na trhu hned několik a většina z nich s sebou nese i programovou podporu orientovanou na danou platformu. Každá z platforem je však omezena svým fyzickým návrhem a programová podpora je orientována primárně na profesionální vývoj, nikoliv na výuku a začátečníky. Programová podpora často není multiplatformní, postrádá simulátor nebo není simulátor orientovaný na uživatele, ale strojové testování. Velká část z nich potom spoléhá na fyzické vývojové sady, které může být pro začátečníky často obtížné vybrat nebo zakoupit.

Cílem práce bude proto navrhnout sadu nástrojů, která není omezena reálnými problémy fyzických platforem a jejíž charakteristiky jsou optimalizovány pro použití začátečníky a výuku, nikoliv profesionální užití v reálném světě. Cílem platformy je poté naučit začátečníky naprosté základy, s nimiž se jim poté bude pracovat o něco lépe. Platforma se také bude podobat reálné platformě AVR, která je v současné době velmi rozšířená v amatérské sféře, hlavně díky projektu Arduino.

2.1.1 Začátečníci

Naprostí začátečníci jsou hlavní cílovou skupinou projektu. Instrukční sada je navržena co nejmenší, aby ji začátečníci mohli obsáhnout co nejrychleji a mohli znát všechny nástroje, které mají k dispozici při řešení problémů. Pro začátečníka bude důležitá multiplatformnost ladící aplikace a překladače, spolu s faktem, že k vývoji není potřeba fyzická vývojová sada. Začátečníci

by při používání naší platformy měli získat zevrubnou představu, co programování pro mikrokontroler obnáší, v čem se liší od systémového programování a jak se přistupuje k řešení problémů v nízkoúrovňovém jazyce.

2.1.2 Pokročilí

Ačkoliv pokročilí uživatelé nejsou cílovou skupinou našeho projektu, existuje díky modulární architektuře projektu možnost použít námi navržený překladač a emulátor ve vlastních projektech, chceme-li přidat programovatelnost pomocí nízkoúrovňového jazyka. Mezi takové případy užití může spadat například návrh alternativních aplikací pro výuku programování nebo počítačových her. V neposlední řadě pak není složité adaptovat překladač ani emulátor pro alternativní instrukční sadu.

2.2 Funkcionalita

Výsledkem práce by měla být sada aplikačního softwaru, knihoven a dokumentů popisující kompletní proces vývoje a parametry spouštění kódu na virtuálním procesoru.

2.2.1 Analýza architektury AVR

Nejdříve prozkoumáme architekturu procesorů z rodiny AVR. Inspirujeme se mikrokontrolerem ATtiny12, který je jeden z nejjednodušších AVR mikrokontrolerů. Z dokumentace ATtiny12 od společnosti Atmel[10] zjistíme následující:

- mikroprocesor disponuje 32 víceúčelovými, 8-bitovými registry,
- mikroprocesor disponuje 1KB programové paměti a 64 byty paměti EEPROM,
- na procesoru se nachází 6 tzv. GPIO pinů,
- procesor operuje v řádu jednotek Mhz.

Při analýze nemusíme zabíhat do přílišných detailů – snažíme se ji pouze aproximovat, nikoliv simulovat. Výše uvedené charakteristiky jsou pro nás tudíž směrodatné.

Dále se podíváme na instrukční sadu procesorů AVR. Jedná se o instrukční sadu typu RISC, disponuje tedy méně, rychlejšími instrukcemi. Instrukce ve své strojové podobě mají délku jednoho nebo více 16-bitových slov. Některé instrukce jsou však redundantní a dají se nahradit kombinací ostatních instrukcí.

2.2.2 Mikroprocesor a strojový kód

První částí práce bude popis podoby virtuálního mikroprocesoru, který naše platforma bude používat. Takový procesor by měl být co nejjednoduší, aby bylo pro naprostého začátečníka obsáhnout všechny informace potřebné k jeho použití v co nejmenjším čase. Měl by být podobný rodině mikroprocesorů AVR od firmy Atmel, se kterými se potom začátečník bude setkávat například na předmětech BI-SAP a BI-ARD, nebo při samostatných projektech využívající širokou řadu projektů z rodiny Arduino[2]. Jeho instrukční sada by měla být dostatečně kompaktní na její snadné zapamatování a navržená tak, aby bylo jednoduché kódovat a dekódovat instrukce bez použití manuálu nebo externího programu. Finálně by potom procesor měl umožňovat snadnou adaptaci pro další využití v jiných projektech, aby nebylo nutné vyvíjet nová řešení a dále tím fragmentovat již tak široké spektrum zařízení.

Mikroprocesor bude založen na modifikované harvardské architektuře, která je kombinací harvardské architektury a Von Neumannovy architektury. Bude tedy jednu paměť na program, ze které půjde pouze číst a jednu paměť na data, ze které půjde jak číst, tak do ní zapisovat. Tohle je architektura, kterou používá rodina mikroprocesorů AVR[10].

Samotný mikroprocesor spouští pouze strojový kód a není zatížen překladem jazykem na vyšší úrovni. Strojový kód je bitovou reprezentací zakódovaných instrukcí. Instrukce pro náš procesor se budou skládat z jednoho a více bytů. První byte enkóduje vždy pouze typ instrukce, o který se jedná. Následující byty enkódují parametry dané instrukce. V samotném strojovém kódu se nijak neodrážejí návěstí definované v jazyce symbolických adres.

Mikroprocesor by měl také podporovat několik vstupních a výstupních operací, obsluhovaných pomocí oddělené paměti a přerušení:

- grafický výstup podobou displeje namapovaného do paměti,
- komunikaci po seriálové lince,
- omezená sada tlačítek.

2.2.3 Emulátor

Jelikož námi navržený procesor bude pouze virtuální, nebude existovat fyzický integrovaný obvod, který by byl schopný spouštět náš strojový kód. Proto, abychom mohli strojový kód spouštět, vytvoříme emulátor, který bude emulovat běh procesoru na jiné, hostující platformě.

[2]Více na https://www.arduino.cc/

Emulátor bude koncipován formou knihovny, kterou bude možné použít při vývoji dalšího softwaru. Tím se zajistí konzistentní chování napříč více programy, ve kterých bude knihovna použita. Zároveň tak bude velmi snadné naimplementovat novou aplikaci, která je schopna spouštět strojový kód naší virtuální platformy.

Z toho důvodu však musí emulátor také zprostředkovat snadnou implementaci vstupních a výstupních rozhraní uživatelem knihovny. Požadavky na formu vstupu a výstupu se mohou výrazně lišit dle užití. Například terminálová aplikace bude nejspíše mít seriálovou komunikaci zprostředkovanou pomocí standardního vstupu a výstupu. Grafická aplikace naopak bude nejspíše poskytovat vlastní seriálovou konzoli. Emulátor by také měl poskytovat možnost snadno rozšířit rozhraní procesoru o vlastní funkcionalitu.

Součástí projektu bude také rozsáhlá testovací sada , která pomůže zajistit stabilitu fungování programu a kontrolu chyb vznikajících změnami v kódu. Bude testovat chování každé instrukce samostatně a poté interakce instrukcí mezi sebou.

2.2.4 Jazyk

Jelikož samotný procesor rozumí pouze strojovému kódu, bude další částí práce definice jazyka symbolických adres, který nám umožní zapisovat instrukce pro náš procesor ve formátu čitelném lidmi. Tento jazyk bude definovat mnemoniky k jednotlivým instrukcím a pseudoinstrukce které nejsou reprezentovány ve strojovém kódu ale ovliňují překlad samotný. Dále pak umožní lepší organizaci programu pomocí návěstí, které umožní pojmenovat konkrétní adresu v programu. Jazyk bude definován pomocí gramatiky, podle které se bude číst ze zdrojového souboru.

Mezi pseudoinstrukce podporované jazykem by měly patřit následující:

- ukládání řetězců a uživatelských dat do programové paměti,
- nastavování pozice v binárním souboru,
- vložení obsahu z jiného souboru.

2.2.5 Překladač

Překlad z jazyka symbolických adres do strojového kódu čitelného pro emulovaný procesor bude zprostředkován konzolovou aplikací.

Výstupem aplikace bude samotný binární soubor se strojovým kódem. Volitelně potom soubor obsahující mapu mezi návěstími a pozicemi v kódu, která pomůže zpřehlednit dekompilovanou verzi kódu v ladících nástrojích.

Dále překladač bude umožňovat povolit pouze některé instrukce. To může být nápomocné u jednoduchých úloh, které v začátečnících často podněcují použití příliš komplikovaného řešení, nebo u úloh, kde chceme demonstrovat nějakou vlastnost programování pro mikroprocesory omezením instrukční sady, která by jinak pomohla problém vyřešit příliš jednodušše.

2.2.6 Ladicí program

Abychom usnadnili uživatelům naší platformy lazení programů a odstraňování chyb, bude součástí práce i grafická aplikace, která umožní krokovat chod programu, sledovat za běhu obsah registrů a paměti.

2.3 Závěr

Projekt bude sestávat celkem ze čtyř částí, z nichž jedna bude spočívat pouze v návrhu samotné platformy a zbývající tři části budou tvořit programovou podporu pro vývoj programů na námi definované platformě. Části programové podpory budou navrženy do jisté míry nezávisle, aby se v budoucnu daly snáze upravovat a integrovat s jinými projekty.

Kapitola 3

Mikroprocesor

První částí práce je návrh samotného mikroprocesoru. Ten je inspirován platformou AVR, kterou jsme analyzovali v sekci 2.2.1. Většina rozhodnutí učiněných při návrhu platformy je tedy inspirována přímo jí.

3.1 Instrukční sada

Instrukční sada je modelována jako podmnožina instrukční sady AVR, avšak upravená pro jednoduchost a větší edukativní hodnotu. Byla odstraněna většina instrukcí, které nejsou kritické pro řešení problémů, nebo jejichž funkcionalita se dá nahradit vícero jinými instrukcemi. Tedy například většina variant instrukcí s okamžitou hodnotou místo druhého parametru, jako `addi`, `andi` nebo `subi`. Zároveň každý operační znak má pouze jedinou mnemoniku a naopak – nedochází ke změně operačního znaku podle parametrů, se kterými je daná mnemonika zkombinována. Tím se předejde zbytečným chybám způsobeným překlepy nebo přehlédnutím se při čtení instrukčního manuálu.

Dalším z hlavních rozdílů je odstranění instrukcí provádějících skoky podle výsledku srovnávacích operací. Instrukce `cmp` je často vnitřně implementována jako aritmetické odčítání bez zápisu zpět, je proto užitečné chápat, jak porovnání dvou čísel ovlivní příznaky v procesoru. V naší instrukční sadě jsou proto podmíněné skoky možné jen podle příznaků. Díky tomu jsou uživatelé nuceni pochopit jak funguje porovnávací instrukce interně a zároveň je díky tomu instrukční sada jednodušší.

Naše platforma též neklade důraz na znaménka čísel uložených v paměti nebo v registru. Význam bitů v registrech je ponechán plně na uživateli platformy. Chybí proto příznak pro znaménko a další vlastnosti procesorů pracujících se zápornými čísly v dvojkovém doplňku.

Skupina	Operační znak	Mnemonika
Pomocné	0x00	nop
	0x02	sleep
	0x03	break
	0x04	sei
	0x05	sec
	0x06	sez
	0x07	cli
	0x08	clc
	0x09	clz
Aritmetika	0x10	add
	0x11	adc
	0x12	sub
	0x13	sbc
	0x14	inc
	0x15	dec
	0x16	and
	0x17	or
	0x18	xor
	0x19	cp
	0x1A	cpi
Větvení	0x20	jmp
	0x21	call
	0x22	ret
	0x23	reti
	0x24	brc
	0x25	brnc
	0x26	brz
	0x27	brnz
Manipulace s daty	0x30	mov
	0x31	ldi
	0x32	ld
	0x33	st
	0x34	push
	0x35	pop
	0x36	lpm
	0x3A	in
	0x3B	out

Tabulka 31: Přehled instrukční sady

Instrukční sada je detilněji popsána v příloze B.

3.2 Registry

Mikroprocesor disponuje šestnácti víceúčelovými osmibitovými registry $R0$ – $R15$. Dále potom speciálními registry PC a SP, označujícími současnou adresu spouštěné instrukce a vrcholem zásobníku respektive.

V kontrastu s platformou AVR jsme snížili počet registrů na šestnáct, abychom mohli kódovat index registru jako čtyřbitovou hodnotu a zvýšili tak čitelnost strojového kódu.

Dále máme dvojici šestnáctibitových registrů X a Y, které hodnotou odpovídají dvojicím registru $R12 : R13$ a $R14 : R15$. Registr X označuje cílovou adresu v paměti pro zapisovací instrukci `st`, registr Y potom adresu pro čtecí instrukce `lpm` a `ld`.

Zásobník reprezentovaný registrem SP začíná ve výchozím stavu na adrese `0xFFFF`. Roste směrem k nižším adresám. Ukazatel zásobníku ukazuje vždy na prázdný prostor nad vrcholem zásobníku, nikoliv na vrchol.

3.3 Pamět

Podobně jako platforma AVR[10], i náš mikroprocesor disponuje hned třemi druhy paměti: programovou, operační a pamětí pro obsluhu vstupu a výstupu.

Tyto tři paměti jsou modelovány jako tři separátní paměťové bloky, a pokrývají své adresní prostory (16, 16 a 8 bitů respektive). V nižších částech paměti však nejsou namapované registry, tak jak je tomu u AVR[10] a paměť zodpovídající za obsluhu vstupu a výstupu má separátní prostor adres, nezávislý na programové a operační paměti.

Paměť pro obsluhu vstupu a výstupu není fyzickou pamětí, nýbrž adresním prostorem na jehož pozicích jsou namapované obsluhy jednotlivých rozhraní. To je odraženo i v samotném emulátoru, který ukládá pro každou pozici dvě obslužné funkce. Ty potom obsluhují chování instrukcí `in` a `out` pro patřičnou adresu. V nevyužitém stavu pak zápis do této paměti hodnotu zahazuje a čtení vždy vrací nulovou hodnotu.

3.4 Instrukční cyklus

Mikroprocesor začíná spouštět program od adresy `0x0000`, na které je umístěný reset vektor. Ještě před načtením adresy dojde k obsluze přerušení, pokud jsou zapnuta (tj. příznak přerušení je nastaven). Pokud nastalo přerušení, dojde

k zavolání příslušného vektoru přerušení a zakázání přerušení odnastavením jejich příznaku. Všechny instrukce probíhají v jednom cyklu a jsou tedy stejně složité.

3.5 Přerušení

Přerušení jsou při startu spouštění vypnuty a musí se ručně povolit instrukcí `sei`. Díky tomu je na možné začátku výuky psát kód přímo od počáteční adresy a nestrachovat se, že přepisujeme vektory přerušení. Každý vektor má 16 bytů a jsou pro ně vyhrazeny adresy `0x0000` – `0x0100`.

0x0000	Reset	Počáteční bod spouštění
0x0010	VBlank	Indikuje obnovovací frekvenci displeje
0x0020	Button	Indikuje, že bylo stisknuto tlačítko
0x0040	Serial	Indikuje, že na sériový port přišla nová data

Tabulka 32: Vektory přerušení

3.6 Rozhraní

Rozhraní jsou na naší platformě obsluhovány pomocí instrukcí `in` a `out`. Změna v jejich stavu je indikována přerušeními, které provedou před načtením instrukce skok na patřičný vektor, kde můžeme rozhraní obsloužit a vrátit se ke spouštění zbytku programu tak kde jsme přestali pomocí instrukce `reti`.

Tato část mikroprocesoru se od platformy AVR liší nejvíce. Mikroprocesor nedisponuje žádnými víceúčelovými porty (v anglické literatuře též známy jako „GPIO pins“). Rozhraní mikroprocesoru jsou dané fixně a rozšiřitelné pouze úpravou programu implementujícího chování pomocí knihovny s emulátorem.

Mezi podporované rozhraní patří sériový port, který umožňuje textovou komunikaci s vnějším světem. Dále pak 7 tlačítek, rozložené do pozicových šipek, centrálního potvrzovacího tlačítka a dvou uživatelských tlačítek, jejichž funkce není mají reprezentovat směrová tlačítka, tlačítko pro potvrzení a víceúčelová tlačítka A a B.

Konečně potom platforma disponuje malým displejem s osmibitovou paletou a rozměry 160 pixelů na šířku a 144 pixelů na výšku. Displej má obnovovací frekvenci 60Hz a jeho obsah odpovídá jednomu bytu na pixel, počínajíc adresou `0x8000` v operační paměti.

Adresy obslužné paměti pro jednotlivá rozhraní a pozice vektorů přerušení lze nalézt v příloze B.

Kapitola 4

Překladač

První samostatnou programovou částí práce je překladač z jazyka symbolických adres do strojového kódu našeho mikroprocesoru. Nemá přímou závislost na ostatních částech práce, pouze na návrhu samotné platformy. Je proto ideálním kandidátem na první část práce.

4.1 Možnosti řešení

Ideální formou pro překladač je konzolová aplikace, aby formou následoval většinu ostatních jazykových překladačů, jako je GCC nebo Clang. Aplikace bude primárně řešit následující problémy:

- čtení a zápis textových i binárních souborů,
- syntaktický rozbor textu podle gramatiky jazyka,
- práce s konzolovými argumenty.

Je proto vhodné vybrat programovací jazyk, v jehož ekosystému existuje pro každý dílčí problém knihovna, která daný problém řeší co nejlépe. Ušetříme si tak zbytečnou práci a zbytečně se nebudeme vystavovat šanci že naše řešení bude obsahovat chyby a neošetřené krajní případy. V námi implementovaném kódu se pak chybám pokusíme předejít automatizovaným testováním. Je proto vhodné, aby námi zvolený jazyk poskytoval funkcionalitu, která testování kódu udělá co nejjednoduším.

4.1.1 C++

První volbou jazyka pro implementaci překladače je jazyk C++ – jazyk s dlouhou historií[11], širokou nabídkou knihoven a rozsáhlou standardní knihovnou.

Mezi zásadní nevýhody C++ však patří absence balíčkovacího systému. Kvůli tomu se musíme buď spolehnout, že patřičné hlavičkové soubory a statické knihovny budou poskytnuty uživatelem (nejčastěji pomocí systémového balíčkovače), nebo je musíme přiložit k práci jako součást našeho kódu. Dále pak C++ postrádá standardní nástrojovou sadu pro kompilaci větších projektů, musíme proto používat externí nástroje.

Jedním takovým nástrojem je CMake[12], používaný pro automatizaci překladu programu. Nejenom, že nám umožní snadno měnit překladač a přidávat závislosti do programu, ale také nám pomůže zajistit konzistentní překlad systému na různých platformách. Projekt CMake vznikl v roce 2001[12], je proto dospělou a stabilní volbou, která málokoho překvapí.

Knihoven pro práci s konzolovými argumenty existuje v ekosystému C++ hned několik. Je tomu tak pravděpodobně hlavně díky široké základně konzolových aplikací napsaných v C++. Nejjednodušší volbou je knihovna Boost.Program_options obsažená v knihovně Boost[13]. Existuje velká šance, že budeme knihovnu Boost používat i v jiných částech programu. Například pokud chceme podporovat verzi C++ starší nežli C++17[14], můžeme pro přístup k souborům použít Boost.Filesystem, který odpovídá stejnému rozhraní, které se v C++17 nachází[13].

Pro syntaktický rozbor textu můžeme použít knihovnu PEGTL[15]. Knihovna PEGTL je založena na flexibilní metodologii a umožňuje psát kód provádějící syntaktický rozklad modulárním a snadno rozšiřitelným způsobem. Mezi její výhody patří její malý rozsah, moderní rozhraní využívající funkce jazyka C++ a přehlednost výsledného kódu. Knihovna PEGTL je implementována jako hlavičkový soubor bez dalších závislostí[15].

```
using namespace tao::pegtl;

struct comment : if_must<one<';'>, anything> { };
```

Ukázka 4.1: Definice jednoduchého pravidla přijímajícího komentáře

4.1.2 Python

Narozdíl od ostatních dvou voleb je Python jazyk interpretovaný, nikoliv kompilovaný. Velké množství knihoven, které pro Python existují, v kombinaci s expresivní syntaxí z něj dělají dobrého kandidáta na návrh konzolové aplikace. Poskytuje jak vlastní balíčkovací systém jménem **pip**[16], tak zabudovanou knihovnu pro psaní testů[17]. Jeho největší nevýhodou je potom fakt, že ke spouštění programů napsaných v Pythonu je potřeba interpreter, který koncový uživatel nemusí mít na svém počítači nainstalovaný.

Jednou z knihoven umožňující práci s konzolí je knihovna Click[1]. Využívá funkcionalitu Pythonu známou jako *dekorátory*, díky kterým je její využití otázkou jednoduché anotace funkcí jejimi parametry. Její použití je demonstrováno v ukázce 4.2.

```
import click

@click.command()
@click.option('--count', default=1,
              help='Number of greetings.')
@click.option('--name', prompt='Your name',
              help='The person to greet.')
def hello(count, name):
    for x in range(count):
        click.echo('Hello %s!' % name)

if __name__ == '__main__':
    hello()
```

Ukázka 4.2: Zdrojový kód aplikace používající Click, převzato z [1]

Pro syntaktickou analýzu programu je i v Pythonu možné použít knihovnu, která nám umožní definovat pravidla samotné analýzy jako gramatiku. Může to být například knihovna Parsimonius[18], jejíž použití neodbočuje výrazně od jiných knihoven podobného typu. Samotné čtení programu ze souboru je pak zajištěno funkcionalitou poskytnout přímo v jazyce.

4.1.3 Rust

Rust je víceúčelový programovací jazyk financovaný a vyvinutý organizací Mozilla Research[19]. Ačkoliv samotný jazyk vznikl v roce 2006[19], první stabilní verze Rustu byla vydána teprve v roce 2015[19]. Jedná se tedy ve srovnání s ostatními možnostmi o poměrně nový jazyk. Přesto byl však Rust v anketě populárního webu Stack Overflow zvolen nejmilovanějším jazykem v letech 2016[20], 2017[21] a 2018[22].

Jedná se o nízkoúrovňový jazyk, navržený jako alternativa jazyků C a C++. Rust je kompilovaný, staticky typovaný a funkcionální[19]. Mezi jeho hlavní výhody v porovnání s C a C++ patří výborný systém na správu závislostí Cargo, bohatší standardní knihovna a příslib větší paměťové bezpečnosti. Dále mezi výhody patří snadná spolupráce s jazyky C a C++, nebo výborná podpora pro paralelismus a vícevláknové programování. Ani jedna z uvedených vlastností se však na našem programu vzhledem k jeho jednoduchosti neprojeví. Zabezpečení přístupu do paměti a validita programu je kontrolována při kompilaci[19], přináší tudíž minimální zatížení při samotném

běhu. Silný typový systém napomáhá ke zlepšení optimalizace kódu. Rychlostně je proto Rust srovnatelný s C++[23].

Mezi jeho nevýhody pak patří slabší, ale přesto bohatý ekosystém knihoven, menší uživatelská základna[22] a časté změny, ke kterým v jazyce stále dochází.

Stejně jako pro C++ a Python i pro Rust existuje nejedna knihovna, která pomáhá psát kód pro syntaktickou analýzu. Pro naši potřebu se zdá být nejlepší knihovna rust-peg[24]. Oproti C++ má však výhodu v tom, že umí generovat kód přímo z jazykové gramatiky. Samotná gramatika potom není psaná přímo v kódu jazyka, který se k tomu často syntakticky nehodí. V případě knihovny rust-peg dochází ke generaci v tzv. „build skriptu“, o jehož spouštění se stará Cargo.

Pro práci s konzolovými argumenty je potom vhodné užít knihovnu Clap[25]. Knihovna Clap nám nejenom umožní přehledně zjistit, které argumenty s jakými hodnotami byly předány naší aplikaci, ale obstará za nás například generování informací o užití programu. Díky typovému systému pak není možné opomenout případy, že argument není definován, nebo nemá hodnotu jakou bychom očekávali.

```
extern crate clap;
use clap::App;

fn main() {
  App::new("assembler")
      .version("0.0.1")
      .about("Simple educational assembler")
      .author("Jiří Š.")
      .get_matches();
  }
}
```

Ukázka 4.3: Zdrojový kód aplikace používající Clap

4.2 Zvolená technologie

Pro samotnou implementaci programu byl nakonec zvolen jazyk Rust a knihovny rust-peg a Clap zmíněné v sekci 4.1.3. Rust byl zvolen primárně kvůli komfortu a bezpečí, které poskytuje, dobré programové podpoře a kvalitní knihovně pro práci s konzolovými argumenty. Nedílnou součástí rozhodnutí pak byly autorovy existující zkušenosti s jazykem.

```
$ assembler --help
assembler 1.0.0
Jiří Š. <sebelji1@fit.cvut.cz>
Simple educational assembler

USAGE:
    assembler [FLAGS]

FLAGS:
    -h, --help Prints this message
    -V, --version Prints version information
```

Ukázka 4.4: Automaticky vygenerovaná dokumetace k ukázce 4.3

4.3 Realizace

Základním stavebním blokem programu je struktura `Compiler` popsaná v ukázce 4.5. Ukládá současný stav překladu, výslednou binární podobu souboru a struktury potřebné k správnému zpracování návěstí.

```
pub struct Compiler {
  cursor: u16,
  output: [u8; 0x10000],
  label_map: HashMap<Label, u16>,
  needs_label: Vec<(u16, Label, Nibble)>,
  last_major_label: Label,
  enabled_instructions: Option<HashMap<Opcode, String>>,
  file_stack: FileStack,
}
```

Ukázka 4.5: Definice struktury `Compiler`

Prvním problémem, který bylo při realizaci potřeba vyřešit, bylo nahrazování návěstí za jejich skutečné adresy. Vzhledem k faktu, že u některých návěstí v době jejich použití v kódu ještě neznáme jejich pozici, musíme provádět překlad ve dvou průchodech.

V prvním průchodu přeložíme veškeré instrukce a u instrukcí, které vyžadují adresu definovanou návěstím zapíšeme pouze nulovou adresu a uložíme si pozici v programu spolu s adresou, kterou na ni budeme doplňovat. Zároveň si v prvním průchodu zapamatujeme pozici každého návěstí, které potkáme.

V druhém průchodu potom pouze projdeme seznam míst, na které je potřeba doplnit adresu a nahradíme dříve zapsanou nulovou adresu za skutečnou hodnotu. Pokud se narazíme na místo, které potřebuje adresu návěstí,

které v programu nebylo definováno, ukončíme překlad chybou.

```
fn resolve_labels(&mut self) -> Result<(), String> {
  for (position, label, nib) in self.needs_label.iter() {
    let addr = self.label_map.get(label)
        .ok_or(format!("Undefined label '{}'!", label))?;

    match nib {
      Nibble::Both => {
        self.output[*position as usize + 0]
            = ((addr & 0xff00) >> 8) as u8;
        self.output[*position as usize + 1]
            = ((addr & 0x00ff) >> 0) as u8;
      },
      Nibble::High => {
        self.output[*position as usize]
            = ((addr & 0xff00) >> 8) as u8;
      },
      Nibble::Low => {
        self.output[*position as usize]
            = ((addr & 0x00ff) >> 0) as u8;
      },
    }
  }

  Ok(())
}
```

Ukázka 4.6: Funkce doplňující adresy návěstí v druhém průchodu

Implementace lokálních návěstí byla vcelku přímočará. Při překladu si pouze pamatujeme poslední návěstí, které začínalo velkým písmenem. Pokud potom narazíme na definici nebo užití návěstí začínajícího tečkou, přidáme na jeho začátek poslední „velké“ návěstí. Zbytek problému už za nás pak vyřeší existující kód na doplňování adres. Lokální návěstí definované dříve, než jakékoliv „velké“ návěstí se umístí do mapy bez zvláštní úpravy.

Kompilační direktiva `include` byla o něco těžší oříšek na rozlousknutí. První návrh aplikace s ní nepočítal, četl proto celý program najednou pomocí gramatického pravidla `program`. Bylo tedy třeba přeorganizovat aplikaci, aby dělila zdrojový kód na řádky ručně a syntaktickou analýzu prováděla řádku po řádce.

Problém byl vyřešen strukturou `FileStack`, která simuluje zásobník ukládající informace o překládaných souborech. Poté stačí ukládat na zásobník

čtené soubory, na který jsou přidány buď začátkem překladu, nebo direktivou `include` a odebrány když je čtení souboru u konce. Ve chvíli kdy je zásobník se soubory prázdný je překlad dokončen. Před přidáním souboru na zásobník také zkontrolujeme, jestli se již v zásobníku nenachází, abychom předešli cyklické závislosti mezi soubory.

```
struct FileStack {
  filenames: Vec<String>,
  lines: Vec<Vec<(usize, String)>>,
}

impl FileStack {
 fn pop(&mut self) -> Option<(String, (usize, String))> {
    if self.filenames.is_empty() {
      None
    }
    else if let Some(line) = self.lines.last_mut()
        .and_then(|x| x.pop()) {
      let filename = self.filenames.last_mut()
          .expect("Inconsistent state in FileStack");
      Some((filename.clone(), line))
    }
    else {
      self.filenames.pop();
      self.lines.pop();
      self.pop()
    }
  }
}
```

Ukázka 4.7: Definice struktury `FileStack` a funkce `pop`.

Posledním problémem bylo omezování instrukční sady v souladu se zadáním. Tento problém byl vyřešen pomocí jednoduchého předání kolekce povolených instrukcí kompilátoru, který potom při zpracování nepovolené instrukce ukončí program chybovou hláškou. Předání seznamu povolených instrukcí probíhá přidáním parametru ukazujícího na soubor ve formátu JSON, obsahujícího seznam řetězců s mnemonikami povolených instrukcí.

4.4 Testování

Jelikož chování překladače spoléhá výhradně na vstup od uživatele a to dokonce ne na jedné, ale na dvou frontách (soubor se zdrojovým kódem a konzolové argumenty), je důležité zajistit, aby každý krajní případ byl ošetřen.

V případě, že takový stav nastane, by aplikace měla vypsat na standardní chybový výstup zprávu popisující ke které chybě došlo a správně se ukončit.

Námi zvolený programovací jazyk Rust nám poskytne veškerou pomoc, kterou k zajištění hladkého běhu aplikace budeme potřebovat. Jeho striktní typový systém nám nedovolí použít výstupní hodnotu funkcí, pokud neošetříme chyby ke kterým mohlo dojít. To je zajištěno pomocí takzvaných „algebraických datových typů“, též známých jako „tagged union“ nebo „sum types“ v anglicky psané literatuře. Hlavním takovým typem je v Rustu typ `Result<T, E>`, který k typu `T` přidává alternativu, že typ obsahuje chybu popsanou typem `E`. Práce s typem `Result<T, E>` je znázorněna v ukázce 4.8.

```
fn operace_ktera_muze_selhat() -> Result<u32, String> {
  // ...
}

fn main() {
  let vysledek = operace_ktera_muze_selhat();

  // Následující by byla chyba, `vysledek' nemá typ u32
  // let chyba: u32 = vysledek + 1

  match vysledek {
    Ok(hodnota) => {
      println!("Operace uspěla, hodnotou je {}.", hodnota);
    },
    Err(chyba) => {
      println!("Při operaci nastala chyba '{}'.", chyba);
    },
  }
}
```

Ukázka 4.8: Ukázka struktury `Result`

Překladač pro Rust potom disponuje zabudovanou možností testování kódu. Vše co je třeba udělat je přidat do modulu s naším kódem submodul `test`. Všechny funkce, které jsou v něm definované a anotované direktivou `#[test]` se zkompilují a spustí pouze při spuštění projektu v testovacím režimu. Každá funkce je potom samostatně spouštěna a její výsledek je přehledně vypsán na standardní výstup.

```
$ cargo test
    Finished dev target(s) in 0.0 secs
     Running target/debug/deps/assembler

running 6 tests
test compiler::tests::it_respects_whitelist ... ok
test compiler::tests::it_resolves_labels ... ok
test compiler::tests::it_produces_output ... ok
test compiler::tests::it_produces_a_symfile ... ok
test compiler::tests::literals_are_not_terminated ... ok
test compiler::tests::it_resolves_local_labels ... ok

test result: ok. 6 passed; 0 failed; 0 ignored;
```

Ukázka 4.9: Výstup spuštění testů překladače

4.5 Závěr

Díky Rustu a jeho ekosystému byla implementace překladače velmi snadná. Kód je přehledný, krátký a díky typovému systému jsou explicitně pokryty všechny cesty programem. Gramatika je snadno upravitelná a adaptovatelná pro případné změny v jazyce a instrukční sadě. Kód je rozdělen do funkcionálních celků, které jsou samostatně testovány zabudovanou testovací funkcionalitou jazyka. Rust samotný se ukázal jako dobrá volba, při použití knihovny rust-peg se však vyskytly menší problémy způsobené jejím návrhem. Vyžaduje, aby typ chybových hlášek vzniklých při kompilaci byl `&'static str`, tj. retězec jehož životnost je statická. Nelze tedy jednoduše vracet formátované zprávy.

Kapitola 5

Emulátor

Druhou částí práce je emulátor, který umožní spouštět strojový kód vyprodukovaný překladačem jehož implementace je popsána v sekci 4. Návrh emulátoru je blíže popsaný v sekci 2.2.3.

5.1 Možnosti řešení

Jelikož je funkcionalita emulátoru úzce omezena na práci s binárními daty a programovou logiku, bude se počet závislostí projektu blížit nule. Zároveň bude důležité, aby knihovnu bylo co nejsnažší integrovat do jiných projektů, což nám malá stopa závislostí usnadní.

5.1.1 Rust

Stejně jako pro překladač je i pro emulátor možné použít jazyk Rust. Jeho výhody a nevýhody jsou blíže popsány v sekci 4.1.3. Pro užití při psaní emulátoru jsou pro nás obzvláště relevantní nízkoúrovňové operace, které má Rust jako součást standardní knihovny. Jmenovitě pak detekce přetečení při sčítání a odčítání, nebo snadná konverze dat mezi jejich bytovou reprezentací a kanonickým typem.

Další z výhod je opět zabudovaná funkcionalita pro testování kódu, které je v emulátoru kritické – chyby při emulaci, při kterých se emulátor může dostat do stavu nekonzistentního s jeho specifikací, jsou obzvlášť pro začátečeníky naprosto neproniknutelné. Je proto kritické takovým chybám zamezit.

Zároveň by bylo možné při testování využít překladače, který je též napsaný v Rustu, jako knihovny použité pro překlad kódu v testech, což by přispělo k jejich přehlednosti.

5.1.2 C++

Druhou variantou je jazyk C++, který je na implementaci této části práce vhodnější. Hlavním důvodem je kompatibilita s aplikacemi, které budou naši knihovnu využívat. Například s grafickou ladící aplikací navrženou v kapitole 6.

K implementaci emulátoru nejsou zapotřebí téměř žádné knihovny, stávají se proto některé nevýhody jazyka C++ popsané v sekci 4.1.1 irelevantními. Vhod pak přijde extenzivní podpora jazyka C++ pro práci s bitovými poli a binárními soubory. Pro čtení souboru byla použita standardní knihovna `filesystem`, která je v jazyce dostupná od verze C++17. Náš kód tudíž není možné ve starší verzi jazyka přeložit, avšak v případě potřeby lze použít knihovnu **Boost.Filesystem**[13], jejíž rozhraní odpovídá velice blízce knihovně `filesystem`.

K samotné správě projektu a překladu poté můžeme použít nástroj CMake, popsaný v sekci 4.1.1.

5.2 Zvolená technologie

Pro implementaci emulátoru použijeme jazyk C++. Důvodem této volby je primárně snadnější využití knihovny v jiných projektech, jako bude například ladící program popsaný blíže v kapitole 6.

5.3 Technické problémy

Jedním z technických problémů, které nám přineslo užití C++ je detekce přetečení při sčítání dvou bytů. Manuální detekce není nikterak složitá, avšak jak překladačová sada GCC[26], tak překladač Clang[27] disponují zabudovanými funkcemi `__builtin_add_overflow` a `__builtin_sub_overflow`, které návratovou hodnotou indikují, zda při operaci došlo k přetečení, nebo ne.

Dalším problémem pak byla implementace obslužných funkcí pro vstup a výstup. Ve finální verzi projektu jsou realizovány jako mapa mezi typy `u8` a `IoHandler`. Typ `IoHandler` je potom jednoduchá struktura obsahující dvě funkce – jednu pro čtení a druhou pro zápis. Výchozí hodnota struktury IoHandler poté splňuje chování procesoru pro nepřirazené pozice v paměti, mohou být tudíž vytvořeny výchozím konstruktorem. Definice struktury je demonstrována v ukázce 5.10.

Posledním problémem byla potom samotná krokovací funkce emulátoru. V prvotním návrhu byla logika spouštění jednotlivých instrukcí řešena pomocí kolekce lambda funkcí, indexované pomocí operačního kódu instrukce. Díky

```
#include <functional>

struct IoHandler {
    std::function<u8()> get = []() {
        return 0x00;
    };
    std::function<void(u8)> set = [](u8) {
        return;
    };
};
```

Ukázka 5.10: Definice struktury `IoHandler`

využití pokročilých metaprogramovacích technik které jazyk C++ umožňuje bylo pak možné detekovat automaticky o jaký druh instrukce se jedná, přečíst automaticky její operandy a předat je přímo lambda funkci, jak je znázorněno v ukázce 5.11.

```
this->instruction_map[CALL] = [this] (Address addr) {
        this->push_addr(this->pc);
        this->pc = addr;
};

...

auto opcode = this->read_opcode();
this->instruction_map[opcode].execute();
```

Ukázka 5.11: Registrace instrukcí pomocí lambda funkcí

Ačkoliv se jednalo o elegantní řešení, byl kód nakonec přepsán aby používal dlouhý blok `switch`, jehož větve odpovídají jednotlivým instrukcím. Načítání argumentů ze strojového kódu je implementováno pomocnými funkcemi, které automaticky inkrementují registr `PC`. Toto řešení je sice méně elegantní, ale oproti originálnímu řešení produkuje podstatně jednodušší binární soubor a je tudíž rychlejší.

5.4 Testování

K testování v jazyce C++ můžeme použít například knihovnu Catch[28]. Jedná se o jedinný hlavičkový soubor, který pomocí maker v jazyce C++ implementuje doménově specifický jazyk, ve kterém se testy definují. Knihovna se za nás poté postará o implementaci funkce `main`, která spustí všechny definované testy. Knihovna také podporuje výbornou integraci s vývojovým

prostředím CLion[29] a výsledky testování jsou potom podrobně dostupné přímo v něm.

5.5 Závěr

Přestože jsme první část práce implementovali v Rustu, zvolili jsme pro emulátor jazyk C++. Tato volba se ukázala jako správná, protože usnadnila integraci do ladícího programu.

Utrpělo však testování. Ačkoliv knihovna Catch usnadnila samotné psaní testů, jejich spouštění spoléhá na existenci cesty ke spustitelnému programu překladače v proměnné prostředí `ASSEMBLER`. Je tomu tak hlavně proto, aby testy mohly obsahovat zrojový kód, který se později přeloží do strojového kódu a načte překladačem.

Kapitola 6

Ladící program

Poslední částí práce je ladicí program, který umožní uživatelům naší platformy spouštět, krokovat a ladit programy, které napíšou.

6.1 Návrh

Žádná z aplikací které jsme analyzovali v první kapitole neměla grafické rozhraní, které by splňovalo naše očekávání pro lazení běžících programů. Pohlédneme se proto po inspiraci jinde.

6.1.1 BGB

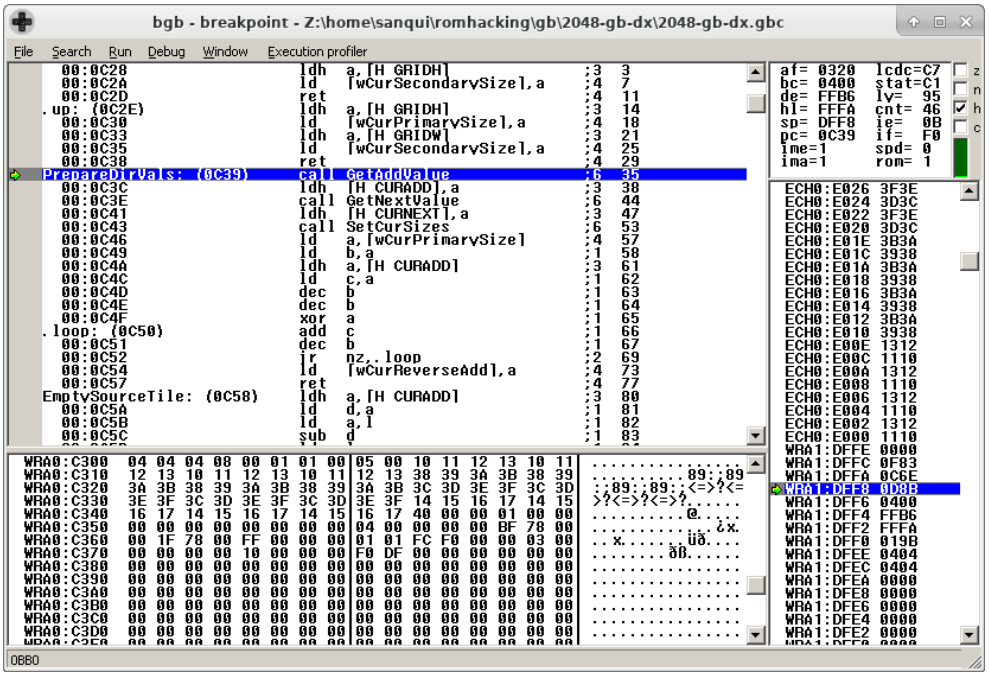
Primární inspirací pro podobu grafického rozhraní ladícího programu byl projekt BGB. Jedná se o velice přesný emulátor platformy Nintendo Game Boy[30], který však také disponuje ladící aplikací. Její rozhraní, byť trochu hutné, je velice funkcionální a plné užitečných informací.

Jak můžeme vidět na obrázku 61, disponuje aplikace oblastí s čitelnou interpretací strojového kódu. Dále potom oblastí umožňující čtení a úpravu paměti, spolu s textovou reprezentací. Další oblastí je pak monitor registrů a příznaků procesoru. Aplikace obsahuje mnoho další funkcionality, ale pro nás bude dostačující, pokud se omezíme na zmíněné tři.

Rozhraní ladící aplikace BGB je jednoduché a přehledné, pokusíme se ho proto replikovat v naší ladící aplikaci.

6.1.2 Qt

Knihovna Qt je jednou z největších a nejčastěji používaných GUI knihoven pro jazyk C++. Mezi projekty, které ji aktivně používají patří například grafické



Obrázek 61: Ladící rozhraní emulátoru BGB

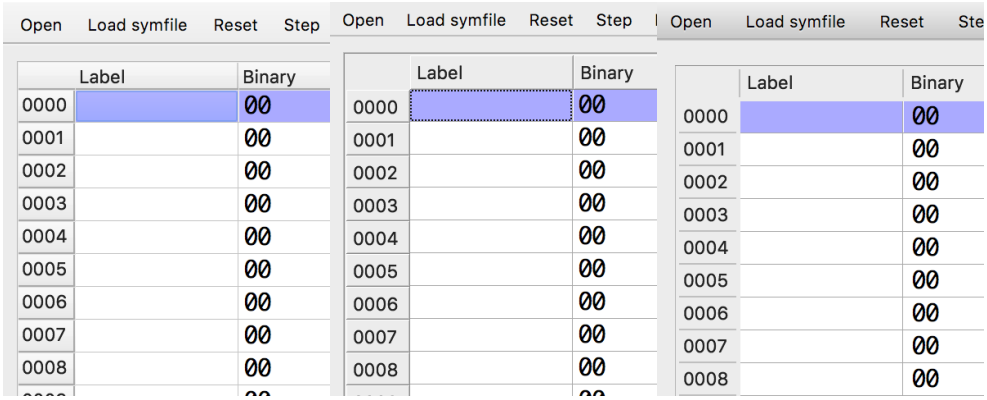
rozhraní pro CMake, aplikace Spotify nebo aplikace QtCreator[31] – vývojové prostředí přímo od tvůrců projektu Qt.

Hlavní výhodou knihovny Qt je kolik práce za nás udělá. Poskytuje velmi rozsáhlou knihovnu ovládacích prvků jako jsou tlačítka, textová pole nebo tabulková zobrazení. Zajistí pro nás hlavní smyčku programu, distribuci událostí a chování ovládacích prvků. V neposlední řadě potom zajistí, že naše aplikace bude následovat jak grafické, tak ovládací idiomy konkrétní platformy na které naše aplikace poběží. Adaptace grafických prvků pro různé platformy je znázorněna na obrázku 62.

Mezi její nevýhody patří fakt, že nabízí alternativní implementaci velké části knihovny STL (např. třídy `QString`, `QQueue` nebo `QList`). Z toho důvodu většina rozhraní knihovny Qt očekává tyto typy, pro které ne vždy existuje snadná konverze ze standardních typů knihovny STL. Kód proto nabývá o vrstvu typových konverzí.

6.1.3 ImGui

Jednou ze zajímavějších voleb je knihovna Dear ImGui[32] (dále jen „ImGui“). Jedná se jednoduchou knihovnu na tvorbu uživateského rozhraní napsanou



Obrázek 62: Vzhled naší aplikace na třech různých platformách

v jazyce C++. Narozdíl od knihovny Qt, která zachovává stav a rozložení rozhraní mezi snímky, knihovna ImGui skládá rozhraní v každém snímku znovu. To činí výsledný kód značně jednoduchým, jak je patrné z ukázky 6.12.

```
if (window_open) {
    ImGui::Begin("Okno", &window_open);

    ImGui::Text("Příklad");
    ImGui::SliderFloat("Hodnota", &hodnota, 0.0f, 1.0f);
    ImGui::ColorEdit3("Barva", &barva);

    if (ImGui::Button("Zavřít")) {
        window_open = false;
    }

    ImGui::End();
}
```

Ukázka 6.12: Využití knihovny ImGui

Jméno knihovny ImGui vychází z anglického sousloví „Immediate GUI“, které naznačuje, že knihovna operuje v bezprostředním řežimu – souslovím zapůjčeném z obdobného konceptu běžného při grafickém programování. Ačkoliv původního autora myšlenky bezprostředního uživatelského rozhraní není snadné dohledat, jedním z prvních zdrojů popularizující tuto myšlenku je video, které Casey Muratori zveřejnil v roce 2005[33].

Výhodou i nevýhodou knihovny ImGui je její nezávislost na vykreslovacím kódu. Samotná knihovna vykreslovací logiky obsahuje jen nezbytné minimum

Obrázek 63: Grafické rozhraní knihovny ImGui

a dodat kód, který bude rozhraní vykreslovat, je povinností uživatele knihovny. To dělá z ImGui skvělého kandidáta jak pro komplexní grafické aplikace, tak prostředí, kde je k dispozici pouze omezená grafická funkcionalita. Dělá to však ImGui těžkopádnou volbou pro grafickou aplikaci na běžné osobní počítače. Mimo nutnost vykreslovat rozhraní ručně poté rozhraní nekopíruje vzhledově hostující systém, jako například knihovna Qt. Uživatelé potom nemohou spoléhat na idiomy, které se naučili ve svém operačním systému používat.

Ačkoliv se tedy jedná o velice zajímavou volbu, nevyhrává nad knihovnou Qt, která poskytne uživateli aplikace daleko větší komfort.

6.2 Zvolené technologie

Pro realizaci grafického rozhraní byla tedy zvolena knihovna Qt. Vzhledem k tomu, že knihovna pro emulaci je napsána v C++ a knihovna Qt také, byl jazyk C++ jasnou volbou pro vývoj ladícího programu.

Namísto standardního nástroje QMake bude ke správě projektu bude použit nástroj CMake. Díky komplexitě knihovny Qt není překlad projektů, které ji využívají tak přímočarý, jako jiné projekty v jazyce C++. CMake však nabízí přímou integraci s Qt a udělá za nás nezbytné kroky které by jinak prováděl

QMake. CMake byl zvolen primárně kvůli zachování konzistence s emulátorem (navrženým v kapitole 5), který jej také používá.

6.3 Realizace

Ačkoliv Qt poskytuje způsob, jak navrhovat grafické rozhraní vizuálně pomocí aplikace Qt Designer[34], jedná se o poměrně těžkopádný způsob návrhu. Proto je v projektu použit manuální návrh pomocí manipulace rozložení (v angličtině známých jako „layouts“) přímo v kódu.

Pro samotné zobrazení většiny dat použijeme třídu `QTableView`, která poskytuje tabulkové rozhraní nad námi definovaným modelem s rozhraním `QTableModel`. Implementace samotného modelu je poté přímočará – vydědíme třídu `QTableModel` a reimplementujeme následující metody:

rowCount Počet řádek v tabulce

columnCount Počet sloupců v tabulce

data Přístup k datům jednotlivých buněk

setData Nastavování dat jednotlivých buněk

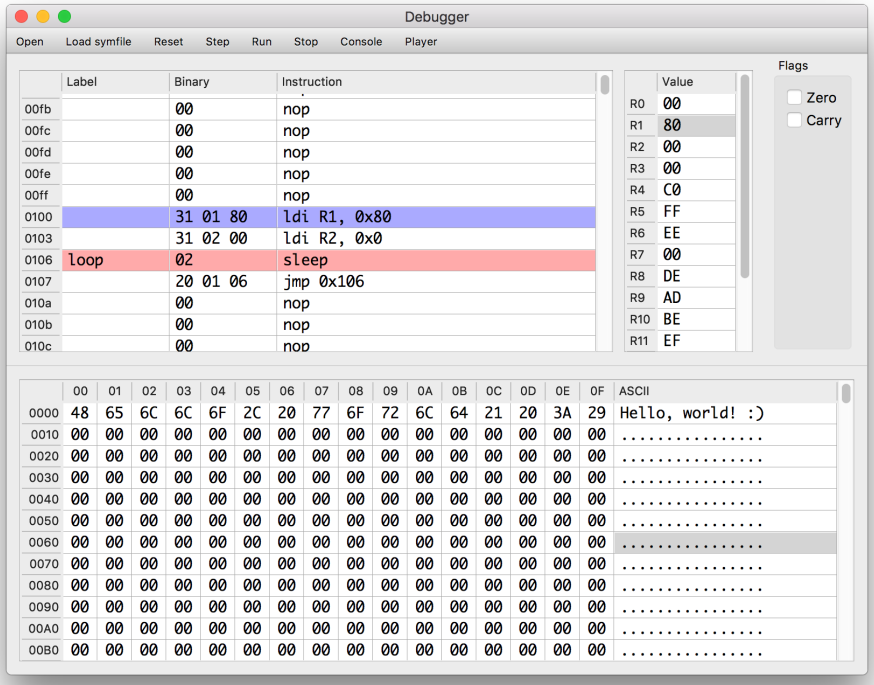
headerData Přístup k datům v hlavičkách řádků a sloupců

flags Příznaky ovlivňující chování buněk

Metody přistupující k datům berou parametr popisující roli metody, tj. na jaká data o buňce se ptáme. To je trochu nešťastný přístup, neboť v první úrovni každé z funkcí musíme podle role diferencovat a implementujeme vlastně vícero funkcí zároveň. Kvůli tomu je pak kód méně čitelný.

Komunikaci mezi aplikací a emulátorem zajišťuje třída `McuState`, využívající navrhový vzor Singleton[35]. Kromě udržování instance emulátoru vystavuje ještě několik signálů a slotů pro knihovnu Qt (6.1.2) a udržuje ostatní data potřebná ke správnému běhu aplikace.

Nejdůležitější funkcí třídy `McuState` je však reinterpretace strojového kódu zpět do textové podoby. Vzhledem k tomu, že překlad je ztrátová operace, nebude reinterpretovaný kód stejný jako kód před překladem. Samotný překlad zpět poté není o nic těžší než čtení strojového kódu. Zásadní slabinou je instrukce `db`, kterou není možné při zpětném překladu rozpoznat a celý proces většinou znemožní. Z podobného důvodu pak ladící aplikace není připravena na manuální skok doprostřed instrukce.



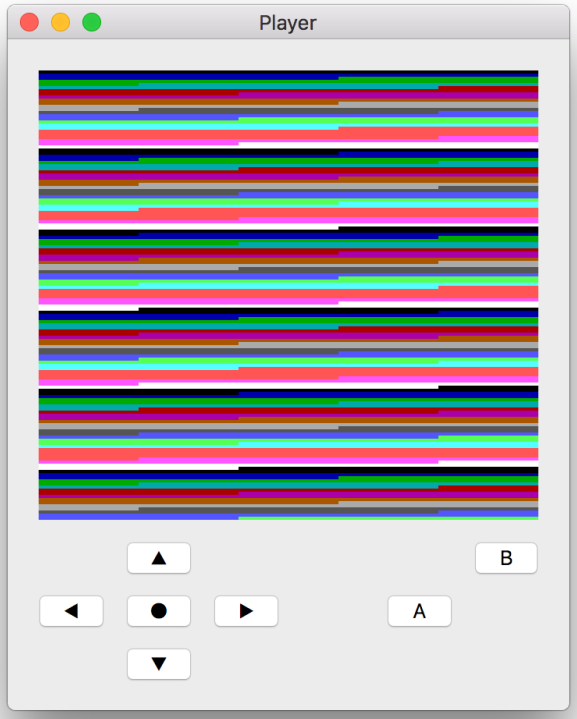
Obrázek 64: Hlavní okno aplikace

Pokud jsme si při překladu nechali vyprodukovat soubor s adresami, použijeme ho při zpětném překladu k doplnění návěstí do instrukcí obsahující adresy. Doplníme je také přímo do tabulky obsahující reinterpretované instrukce.

Při automatickém běhu programu uměle omezujeme rychlost spouštění instrukcí na 16Mhz. K tomu nám napomáhá třída `QTimer`, která spouští námi definovanou funkci v intervalu odpovídajícím třiceti snímkům za sekundu. Abychom si ušetřili práci s časováním na každé instrukci, spouštíme je proto v dávkách odpovídajícím jednomu snímku displeje, který je součástí emulace.

Nedílnou součástí grafického rozhraní je pak grafická reprezentace displeje a tlačítek, pro která máme připravené rozhraní. Ta je realizována pomocí separátního okna, které se rozložením podobá klasické herní konzoli Nintendo Game Boy. Tato podoba byla zvolena primárně pro svou univerzální rozpoznatelnost mezi potenciálními uživateli našeho programu.

Poslední součástí grafického rozhraní je potom sériová konzole. Je tvořena



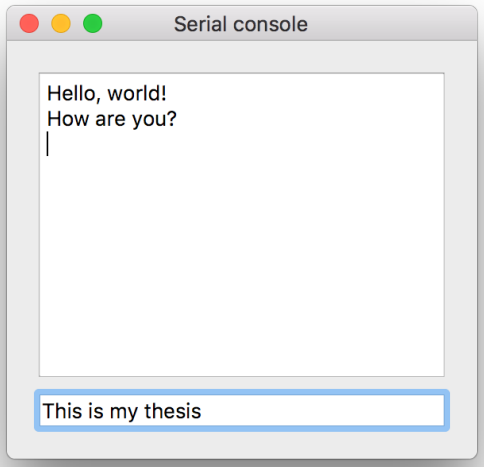
Obrázek 65: Okno obsahující displej a tlačítka

dvěma grafickými prvky: editovatelným textovým polem na zadávání zpráv k odeslání po sériové lince a needitovatelným textovým polem, které zobrazuje zprávy příchozí. Odeslání zprávy je možné klávesou Enter, což také textové pole vyprázdní. Reset mikroprocesoru potom vyprázdní obě textová pole a uvede je tak do stavu, ve kterém se nacházeli původně.

Stav obou oken je zachován i po jejich zavření a obnoven při jejich opakovaném otevření.

6.4 Závěr

V této kapitole jsme prozkoumali existující ladící aplikaci BGB a zreplikovali část jejího rozhraní v knihovně Qt. Implementovali jsme emulační funkcionalitu pomocí knihovny navržené v kapitole 5 a přidali k ní možnost reinterpretovat strojový kód zpět do čitelné podoby.



Obrázek 66: Okno se sériovou konzolí

Tím je aplikační část této práce hotova a v následující kapitole ji otestujeme na několika uživatelích, abychom ověřili kvalitu našeho návrhu.

Kapitola 7

Uživatelské testování

Abychom ověřili, jak dobře se s námi navrženými nástroji funguje, podrobíme je testování s reálnými uživateli. V této kapitole budeme tedy testovat práci jako celek v ohledu používání uživatelem. Testování spolehlivosti překladače a emulátoru jsme již provedli v kapitolách 4.4 a 5.4 respektive.

7.1 Metodika

Testování bude probíhat ve třech krocích a bude časově omezeno na hodinu od začátku řešení úloh (tj. nevčetně přípravy).

V prvním kroku obeznámíme uživatele s problematikou, návrhem a možnostmi procesoru. Provedeme ho technickou dokumentací, ve které bude moci později vyhledávat patřičné informace. Také mu poskytneme mu počítač, na kterém jsou nainstalovány všechny aplikace potřebné k začátku testu.

V druhém kroku bude uživatel samostatně řešit několik úloh, které jsou navrženy se zvyšující se obtížností, aby uživatele seznámily s vlastnostmi a funkcemi procesoru postupně. Bude mít k dispozici plnou dokumentaci procesoru i aplikacím, které bude během řešení úloh používat. Na jeho otázky budeme odpovídat, avšak budeme se snažit aby co nejvíce problému vyřešil bez naší pomoci.

V třetím kroku provedeme s uživatelem krátkou reflexi, kde zjistíme které úlohy, funkce mikroprocesoru nebo části aplikace pro něj byly problematické. Ověříme, zda jeho řešení úloh je korektní a zda plně pochopil koncepty, které úlohy měly uživateli představit.

7.2 Persony

Při uživatelském testování je užitečné načtrnout si takzvané persony – archetypy uživatelů, které budou naši aplikaci používat.

7.2.1 Ondřej

Ondřej je absolventem humanitních studií a nikdy neprogramoval. Jeho koníčkem je stavění modelových železnic. Při svém posledním projektu se rozhodl automatizovat zvedání závor podle časovače a chce se proto naučit programovat pro mikrokontrolery. Neví však kde začít a chce proto nejdříve získat základní znalosti, než bude investovat do drahé vývojové sady.

Jeho definující vlastnosti jsou neznalost základních programovacích konceptů a orientace na externí rozhraní, která plánuje používat.

7.2.2 Martin

Martin je zkušený programátor, který začal programovat webové stránky a na vysoké škole se naučil C++. Jeden z povinných předmětů na jeho škole vyučuje základy programování pro mikrokontrolery a jeho vyučující se rozhodl využít naši platformu pro domácí úkoly.

Mezi jeho definující vlastnosti patří spoléhání na komfort vysokoúrovňových jazyků a dobrá znalost programovacích konceptů.

7.2.3 David

David je zkušený programátor, který byl v mládí fascinován starými herními konzolemi. Jeho zájem v tomto oboru ho přivedl k programování pro staré počítače se kterými se setkával v dětství. Jeho koníčkem je psaní jednoduchých her pro mikroprocesory a naši virtuální platformu se rozhodl vyzkoušet pro svou novou hru.

Mezi jeho definující vlastnosti patří vysoká náročnost na funkcionalitu překladače a kvalitu ladící aplikace.

7.3 Úlohy

Při testování aplikace s uživateli je užitečné mít konzistentní sadu úloh, kterou budou uživatelé při testování řešit. Sada nemusí nutně pokrýt celou funkcionalitu aplikace, ale měla by pokrýt funkcionalitu relevantní pro všechny persony. Zároveň by sada měla rozsahově odpovídat délce testování, pro které je v našem případě vyhrazena jedna hodina. Samotný uživatel potom ne-

musí úlohy splnit všechny. Počet vyřešených úloh je sám o sobě užitečným indikátorem přívětivosti naší aplikace.

Proto pro uživatelské testování aplikace bylo navrženo šest úloh. Jsou navrženy se vzrůstající obtížností a pokrývají základní operace v jazyce symbolických adres, základní stavební bloky programu a užívání rozhraní pro vstup a výstup. Některé úlohy používají možnost omezit instrukční sadu přijímanou překladačem, aby omezili prostor možných řešení pro danou úlohu.

Samotné zadání úloh je možné najít v příloze A.

7.4 Výsledky testování

Výsledky testování se výrazně lišily dle persony pod kterou uživatel náleží, jsou proto rozděleny do skupin podle patřičné persony.

7.4.1 Persona Ondřej

Uživatelé odpovídající personě Ondřej (definované v 7.2.1) se nejvíce potýkali se samotným návrhem logiky programu. Jejich prvotní návrhy byly povětšinou deklarativní a procedurální způsob myšlení nad problémy začali používat až po drobných radách. Z testování vyplynulo, že takové přemýšlení je nejefektivnější jim připodobnit k zadáváním příkazů robotovi nebo operacím na výrobní lince. Práce s pamětí a adresace byla potom nejlépe vysvětlena připodobněním k tabulkovému dokumentu.

Po překonání prvotního ostychu, vysvětlení základních programovacích pojmů a nastínění způsobu, kterým je dobré problémy řešit však naprostí začátečníci řešili úlohy samostatně a bez větších nesnází. Reflexe s nimi ukazuje, že tomu tak je hlavně proto, že samotný jazyk symbolických adres je jednoduché obsáhnout jako celek. Nezkušenost, která byla na začátku testování hendikepem, je nyní osvobozovala od tápání po chybějící funkcionalitě z vysokoúrovňových jazyků a umožnila jim lépe se soustředit na řešení samotného problému. Také možnost překladače omezit přijímanou instrukční sadu pomohla rychleji navést uživatele na správnou cestu. Přesto však začátečníci splnili během vymezené hodiny menší množství úloh. Řešení úloh také kvalitativně strádalo oproti účastníkům, kteří mají s programováním předchozí zkušenosti a často se v řešení nacházely neošetřené mezní případy.

Výsledky testování se začátečníky byly až na drobné vyjímky veskrze pozitivní. Samotní uživatelé potom cítili dobrý pocit z odvedené práce a z pokroku, který během hodiny učinili. Část z nich projevila zájem věnovat se programování i po konci testu, většinou k dokončení rozpracované úlohy.

7.4.2 Persona Martin

Uživatelé odpovídající personě Martin (definované v 7.2.2) se z počátku potýkali s některými principy programování v nízkoúrovňovém jazyce symbolických adres. Hledali marně koncepty z vyšších jazyků, jako jsou funkce, proměnné nebo standardní knihovna. Po překonání počáteční „jazykové bariéry“ potom řešili úlohy samostatně.

Výsledky testování se zkušenými programátory byly spíše pozitivní, omezené možnosti jazyka v nich však často vyvolávali rozpaky. Nejevili přílišný zájem o pokračování v programování, ocenili však ladící program a jeho relativní jednoduchost v porovnání s nástroji na které jsou zvyklí.

7.4.3 Persona David

Uživatelé odpovídající personě David (definované v 7.2.3) se v programování pro mikrokontroler zorientovali velice rychle. Úlohy řešili samostatně, bez potřeby častější z naší strany. Často konzultovali dokumentaci a místy i samotný zdrojový kód emulátoru.

Největším problémem pak bylo jejich spoléhání na funkcionalitu, na kterou jsou zvyklí z pokročilejších nástrojů a v naší práci nejsou přítomny. Primárně potom jde o makra v jazyce symbolických adres, víceúčelové výstupní porty nebo například lepší nástroje pro spouštění již hotového programu, který není třeba ladit.

Výsledky testování s pokročilými uživateli odhalili několik funkcionálních nedostatků v aplikaci, jako je absence pokročilých krokovacích režimů. Jedná se však povětšinou o funkcionalitu, která začátečníky příliš neovlivní. Ohlasy byly pozitivní, uživatelé se při testování bavili a několik zaujala možnost snadno integrovat emulátor do vlastního projektu.

7.5 Vyhodnocení

Celkem jsme aplikaci testovali s jedenácti uživateli. Výsledky testování byli pozitivní, naprostí začátečníci reagovali na jednoduchost návrhu i jazyka velmi pozitivně. Testování s pokročilejšími uživateli odhalilo pár nedostatků v naší sadě nástrojů, které by v budoucnu bylo vhodné vyřešit:

- pokročilé krokovací funkce,
- makra v překladači,
- chybějící knihovna standardních definic a funkcí.

Závěr

V bakalářské práci jsme se zabývali návrhem a implementací překladače a emulátoru pro instrukční sadu jednoduchého mikrokontroleru, kterým bychom mohli nahradit platformu AVR pro naprosté začátečníky.

V první kapitole jsme analyzovali problém, provedli výzkum mezi studenty předmětu BI-SAP na FIT ČVUT a shledali, že současné řešení problému je nedostačující. V další kapitole jsme pak podrobněji analyzovali problém a rozhodli, které problémové body se pokusíme naším řešením ošetřit.

V následujících kapitolách jsme se zabývali postupně návrhem samotného mikroprocesoru, překladače pro jazyk symbolických adres, emulátoru a ve finále ladící aplikace, pomocí které mohou uživatelé naší platformy spouštět a krokovat své programy.

V poslední kapitole jsme potom výsledný produkt otestovali s několika uživateli. Poznatky z testování jsme vyhodnotili, zkonstatovali které části řešení byly úspěšné a načtrtli možnosti, jak aplikaci na základě výsledků testování vylepšit.

Výsledkem této práce je tak dvojice aplikací. První z nich je překladač z jazyka symbolických adres do strojového jazyka. Druhou je potom multiplatformní grafická ladící aplikace, která umožňuje uživatelům spouštět a krokovat programy, během čehož mohou sledovat obsah paměti a registrů. Ačkoli aplikace splňuje zadání a je použitelná, uživatelské testování odhalilo drobné nedostatky. Pro ideální využití aplikace, například při výuce v předmětu BI-SAP, by bylo vhodné tyto nedostatky vyřešit.

Literatura

[1] Ronacher, A.: Click. 2017. Dostupné z: http://click.pocoo.org/5/

[2] Heule, S.: How Many x86-64 Instructions Are There Anyway? 2016-03-07. Dostupné z: https://stefanheule.com/blog/how-many-x86-64-instructions-are-there-anyway/

[3] emu8086.com: Complete 8086 instruction set. 2005. Dostupné z: http://www.gabrielececchetti.it/Teaching/CalcolatoriElettronici/Docs/i8086_instruction_set.pdf

[4] Kohlík, M.: 8. Laboratoř: Aritmetika a řídicí struktury programu. Dostupné z: https://edux.fit.cvut.cz/courses/BI-SAP/labs/08/start

[5] Microchip: Atmel Studio. Dostupné z: https://www.microchip.com/webdoc/GUID-ECD8A826-B1DA-44FC-BE0B-5A53418A47BD/index.html?GUID-8F63ECC8-08B9-4CCD-85EF-88D30AC06499

[6] Pollet, M.: simavr. 2018-04-24. Dostupné z: https://github.com/buserror/simavr

[7] Arduino Software (IDE). Dostupné z: https://www.arduino.cc/en/Main/Software

[8] Avra: Assember for the Atmel AVR microcontroller family. Dostupné z: http://avra.sourceforge.net/

[9] GNU Project: Avr-GCC. Dostupné z: https://gcc.gnu.org/wiki/avr-gcc

[10] Atmel: ATtiny11/12 Datasheet. Dostupné z: http://www.allspectrum.com/semiconductors/micro-controllers/attiny12/doc1006.pdf

[11] Albatross: History of C++. Dostupné z: http://www.cplusplus.com/info/history/

[12] The CMake Project: Overview. Dostupné z: https://cmake.org/overview

[13] Dawes, B.: Filesystem Library Version 3. 2015-10-25. Dostupné z: https://www.boost.org/doc/libs/1_67_0/libs/filesystem/doc/index.htm

[14] CppReference: Filesystem library. 2018-02-04. Dostupné z: http://en.cppreference.com/w/cpp/filesystem

[15] The Art of C++: PEGTL. 2015-10-21. Dostupné z: https://github.com/taocpp/PEGTL

[16] PyPa: pip. Dostupné z: https://pip.pypa.io/en/stable/

[17] Python: Unit testing framework. Dostupné z: https://docs.python.org/3/library/unittest.html

[18] Rose, E.: Parsimonious. 2017-10-17. Dostupné z: https://github.com/erikrose/parsimonious

[19] The Rust Programming Language: Frequently Asked Questions. 2018. Dostupné z: https://www.rust-lang.org/en-US/faq.html#project

[20] Stack Overflow: Developer Survey Results. 2016. Dostupné z: https://insights.stackoverflow.com/survey/2016#technology-most-loved-dreaded-and-wanted

[21] Stack Overflow: Developer Survey Results. 2017. Dostupné z: https://insights.stackoverflow.com/survey/2017#most-loved-dreaded-and-wanted

[22] Stack Overflow: Developer Survey Results. 2018. Dostupné z: https://insights.stackoverflow.com/survey/2018#most-loved-dreaded-and-wanted

[23] Walton, P.: C++ design goals in the context of Rust. 2010-12-05. Dostupné z: http://pcwalton.blogspot.com/2010/12/c-design-goals-in-context-of-rust.html

[24] Mehall, K.: rust-peg. 2018-04-24. Dostupné z: https://github.com/kevinmehall/rust-peg

[25] kbknapp: clap-rs. 2018-04-11. Dostupné z: https://github.com/kbknapp/clap-rs

[26] GCC: Built-in Functions to Perform Arithmetic with Overflow Checking. Dostupné z: https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html

[27] LLVM: Clang Language Extensions. Dostupné z: https://clang.llvm.org/docs/LanguageExtensions.html

[28] Catch: test framework. 2018-05-10. Dostupné z: https://github.com/catchorg/Catch2

[29] Nash, P.: To Catch a CLion. 2017-03-02. Dostupné z: https://blog.jetbrains.com/clion/2017/03/to-catch-a-clion/

[30] BGB: Game Boy emulator/debugger. 2017-12-06. Dostupné z: http://bgb.bircd.org/

[31] Qt Creator Manual. Dostupné z: http://doc.qt.io/qtcreator/

[32] Cornut, O.: dear imgui. 2018-05-11.

[33] Muratori, C.: Immediate-Mode Graphical User Interfaces. 2016-05-12. Dostupné z: https://caseymuratori.com/blog_0001

[34] The Qt Company: Qt Designer Manual. Dostupné z: http://doc.qt.io/qt-5/qtdesigner-manual.html

[35] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994.

Seznam použitých zkratek

AVR Alf (Egil Bogen) and Vegard (Wollan)'s RISC Processor.

FIT Fakulta Informačních Technologií.

GPIO General-purpose Input/Output.

RAM Random-Access Memory.

RISC Reduced Instruction Set Computing.

SD Secure Digital.

Příloha **A**

Úlohy k testování

Úlohy pro testování

Úloha 1: Rovnice

Napište program, který řeší následující rovnici a ověřte správnost fungování programu na hodnotách $R1 := 20, R2 := 11$.

$$R0 = R1 + 2 \times R2$$

Úloha 2: Ahoj, světe!

Nastudujte z dokumentace procesoru, jak funguje sériový vstup a výstup. Napište program, který na seriálový výstup vypíše řetězec "Ahoj, svete!". Řetězec by měl být uložen v programové paměti procesoru.

Úloha 3: Zásobník

Použijte zásobník k prohození obsahu registrů $R0$ a $R1$.

Povolené instrukce: **ldi**, **push**, **pop**

Úloha 4: Podprogramy

Napište podprogram, který ze zásobníku přečte postupně vyšší byte a nižší byte adresy řetězce v operační paměti programu. Tento řetězec poté převeďte tak, aby na konci podprogramu byl psán pouze velkými písmeny. Nepísmenné znaky by však měly zůstat nezměněny.

Rada: Ke změně malého písmena na velké stačí odsnatavit 6. bit v jeho reprezentaci.

Úloha 5: Vstup a výstup

Nastudujte z dokumentace procesoru, jak funguje seriálový vstup a výstup.

(a) Napište program, který do adresy pro seriálový vstup a výstup zapíše hodnotu 42 a ihned ji zase přečte do registru $R0$. Zkuste předpovědět obsah registru $R0$ a porovnejte své očekávání se skutečným výsledkem.

(b) Napište program, který každou příchozí zprávu po seriálové lince přepošle ihned zpět. Procesor by v odbobí nečinnosti neměl provádět žádnou práci.

Úloha 6: Grafický výstup

Nastudujte z dokumentace procesoru, jak funguje grafický výstup procesoru. Napište program s co nejzajímavějším grafickým výstupem – fantazii se meze nekladou. Změna grafického výstupu by měla být synchronizována s obnovovací frekvencí displeje.

Příloha B

Dokumentace mikroprocesoru a instrukční sady

Dokumentace jednoduchého procesoru

Toto je dokumentace jednoduchého procesoru, jehož omezení jsou pečlivě vybrána, aby pomohla začátečníkům naučit se programovat v jazyce assembly. Doufám, že vám pomůže udělat první krok ke složitějším platformám :)

:: Technická specifikace

Displej: 160x144, fixní paleta s 16 barvami
Vstup: 7 tlačítek, sériový
Velikost pamětí (program/operační/rozhraní): 65536B/65536B/256B
Frekvence: 16Mhz

:: Syntaxe

Základním stavebním kamenem zdrojového kódu je řádka v následujícím tvaru:

```
    [návěstí:][mnemonika [operandy]][; Komentář]
```

Operandem může být:

- Návěstí (např. `Main')
- 8/16-bitová konstanta (např. 'a', 42, 0xDEAD, 0b1010, ...)
  - 8-bitovou konstantu lze získat i z adresy pomocí funkcí `hi' a `lo'. (např. `hi(Main)', `lo(loop)')
- Pro instrukci `db' pak ještě ASCII řetězec (např. "Hello!")
- Registr (např. r0, R15, ...)

Pro přehlednost bude dále používat následující zkratky:

- imm8/16: 8/16-bitová hodnota
- STR: Řetězec
- A: Adresa (návěstí nebo imm16)

:: Pseudoinstrukce

Překladač nabízí následující pseudoinstrukce pro ovlivnění překladu:

- `include <file>': Vloží obsah souboru <file> na místo užití
- `org imm16': Nastaví pozici v binárním souboru
- `db [imm8, STR]*': Vloží binární podobu operandů
- `ds imm16`: Posune kurzor o určitý počet bytů

:: Registry

Procesor disponuje následujícími registry:

- R0-R15: Víceúčelový registr
- X: Dvojregistr R12:R13, cíl instrukce `st`
- Y: Dvojregistr R14:R15, cíl instrukcí `ld` a `lpm`
- SP: Ukazatel na vrchol zásobníku
- PC: Čítač programu
- SR: Registr příznaků

Pouze registry R0-R15 lze používat jako operandy.

:: Příznaky

Registr příznaků obsahuje následující příznaky:

- I: Povolení přerušení
- Z: Nulový příznak
- C: Přenos do vyššího řádu

Všechny příznaky jsou nastavitelné pomocí patřičných instrukcí. Příznaky Z a C jsou nastaveny automaticky každou aritmetickou operací a lze je využívat jako podmínky pro skok.

:: Přerušení

Před načtením instrukce může dojít k následujícím přerušením, které skočí na patřičný vektor:

- VBlank: Obnovení obrazovky
- Button: Stisknutí tlačítka
- Serial: Příchozí zpráva na sériovém vstupu.

:: Rozdělení paměti

Procesor disponuje třemi druhy paměti. Každá paměť je adresována separátně. Adresní prostor každé paměti je popsán níže:

[Programová paměť]
  0x0000 – 0x0010: Reset vektor
  0x0010 – 0x0020: VBlank vektor
  0x0020 – 0x0030: Tlačítkový vektor
  0x0030 – 0x0040: Nevyužito
  0x0040 – 0x0050: Sériový vektor
  0x0050 – 0x0100: Nevyužito
  0x0100 – 0xFFFF: Uživatelský program

[Operační paměť]
  0x0000 – 0x8000: Uživatelská data
  0x8000 – 0xDA00: VRAM
  0xDA00 – 0xFFFF: Uživatelská data, zásobník

[Paměť rozhraní]
  0x02: Stisknutá tlačítka / –
  0x03: Fronta sériového vstupu / sériový výstup

:: Sériová linka

Procesor disponuje sériovou linkou, po které je schopen přijímat a odesílat zprávy v kódování ASCII. Pokud dojde k přerušení 'Serial', lze příchozí data číst byte po bytu z příslušné adresy rozhraní.

:: Displej

Emulátor disponuje malým displejem s rozměry 160x144 pixelů, jehož obsah odpovídá adresnímu prostoru operační paměti označenému jako 'VRAM'. Spodní čtyři bity každé hodnoty v rozsahu určují barvu pixelu jako index do fixní palety s 16 barvami. Jednotlivé pixely jsou namapovány po řádkách počínaje levým horním rohem.

:: Tlačítka

Emulátor disponuje sedmi tlačítky. V paměti rozhraní je na patřičné adrese uložen jejich stav. Při stisknutí tlačítka se provede přerušení 'Button'.

:: Instrukční sada

V této sekci si představíme celou instrukční sadu procesoru, včetně její binární podoby. Pro zkrácení budeme operandy instrukcí indikovat následovně:

- rX: Registr X
- i8/16: 8/16-bitová hodnota
- A: 16-bitová adresa

:: Strojový kód

Instrukce jsou reprezentovány vždy svým operačním znakem a binární podobou operandů, která je pro registry rozhodnuta následovně:

- Registr rX: 0x0X
- Registry rX, rY: 0xXY

Ostatní hodnoty a adresy jsou reprezentovány svou kanonickou binární podobou.

:: Instrukce

Instrukce jsou nadepsány ve formátu:

  [operační znak] mnemonika [operandy]: <popis>

[0x00] nop: Neprovede žádnou operaci.
[0x02] sleep: Přeruší chod procesoru do dalšího přerušení.
[0x03] break: Indikuje pozici breakpointu pro ladící program.

[0x04] sei: Nastaví příznak I.
[0x05] sec: Nastaví příznak C.
[0x06] sez: Nastaví příznak Z.
[0x07] cli: Odnastaví příznak I.
[0x08] clc: Odnastaví příznak C.
[0x09] clz: Odnastaví příznak Z.

[0x10] add rA, rB: Přičte k registru A hodnotu registru B.
[0x11] adc rA, rB: Přičte k registru A hodnotu registru B a přenos.
[0x12] sub rA, rB: Odečte od registru A hodnotu registru B.
[0x13] sbc rA, rB: Odečte od registru A hodnotu registru B a přenos.

[0x14] inc rA: Zvýší registr A o 1.
[0x15] dec rA: Sníží registr A o 1.

[0x16] and rA, rB: Přidá k registru A hodnotu registru B operací AND.
[0x17] or rA, rB: Přidá k registru A hodnotu registru B operací OR.
[0x18] xor rA, rB: Přidá k registru A hodnotu registru B operací XOR.

[0x19] cp rA, rB: Nastaví C pokud rX < rY, nastaví Z pokud rX == rY.
[0x1A] cpi rA, i8: Nastaví C pokud rX < i8, nastaví Z pokud rX == i8.

[0x20] jmp A: Nastaví PC na A.
[0x21] call A: Umístí PC na zásobník a nastaví PC na A.
[0x22] ret: Načte PC ze zásobníku.
[0x23] reti: Načte PC ze a nastaví příznak I.

[0x24] brc A: Nastaví PC na A pokud je nastaven příznak C.
[0x25] brnc A: Nastaví PC na A pokud je odnastaven příznak C.
[0x26] brz A: Nastaví PC na A pokud je nastaven příznak Z.
[0x27] brnz A: Nastaví PC na A pokud je odnastaven příznak Z.

[0x30] mov rA, rB: Zkopíruje do registru A hodnotu registru B
[0x31] ldi rA, i8: Uloží do registru A hodnotu i8

[0x32] ld rA: Načte do registru A z operační paměti hodnotu na adrese Y.
[0x36] lpm rA: Načte do registru A z programové paměti hodnotu na adrese Y.
[0x33] st rA: Uloží do operační paměti na adresu X hodnotu registru A.

[0x34] push rX: Umístí hodnotu rX na zásobník a inkrementuje SP.
[0x35] pop rX: Umístí do rX vrchol zásobníku a dekrementuje SP.

[0x3A] in rA, IO: Načte do registru A hodnotu z adresy IO paměti rozhraní.
[0x3B] out rA, IO: Uloží do paměti rozhraní na adresu IO hodnotu registru A.

Příloha C

Obsah přiložené SD karty

- `documentation.txt` ..... dokumentace procesoru v textové podobě
- `readme.txt` ..... stručný popis obsahu SD karty
- `src`
  - `assembler` ..... zdrojové kódy překladače
  - `debugger` ..... zdrojové kódy ladící aplikace
  - `emulator` ..... zdrojové kódy emulátoru
  - `paper` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF