VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

BAKALÁŘSKÁ PRÁCE

Brno, 2018 Dominik Friml

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY
DEPARTMENT OF CONTROL AND INSTRUMENTATION

VÝUKOVÝ SIMULÁTOR POČÍTAČOVÉHO SYSTÉMU
COMPUTER SIMULATOR FOR EDUCATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Dominik Friml**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Petr Petyovský, Ph.D.**
SUPERVISOR

BRNO 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní obor **Automatizační a měřicí technika**
Ústav automatizace a měřicí techniky

Student: Dominik Friml ***ID:*** 186060
Ročník: 3 ***Akademický rok:*** 2017/18

NÁZEV TÉMATU:

Výukový simulátor počítačového systému

POKYNY PRO VYPRACOVÁNÍ:

Cílem práce je navrhnout a realizovat interaktivní výukový simulátor počítačového systému.

1. Seznamte se s architekturou, komponentami a principy fungování počítačového systému.
2. Proveďte rešerši existujících řešení pro simulaci počítačových systémů vhodných pro výukové účely.
3. Na základě nastudovaných znalostí definujte výhody a omezení existujících simulačních řešení. Definujte požadavky nutné pro efektivní využití simulátoru ve výuce.
4. Navrhněte modulární koncept výukového simulátoru počítače a zvolte vhodnou architekturu procesoru.
5. Navrhněte vhodnou reprezentaci vnitřních stavů všech komponent simulátoru (paměť, registry, vnější i vnitřní sběrnice, ALU, paměť mikro kódu, zásobník) tak, aby bylo možné tyto stavy zobrazit a modifikovat.
6. Realizujte výukový simulátor počítače a prezentujte jeho správnou funkci.
7. Navrhněte a realizujte alespoň pět výukových úloh demonstrující principy fungování počítače, které bude možné využít v kurzu mikroprocesorová technika.
8. Zhodnoťte dosažené výsledky a navrhněte možná vylepšení.

DOPORUČENÁ LITERATURA:

[1] Knuth E. Donald: Umění programování 1.díl - Základní algoritmy, Computer press 2008, ISBN: 978-80-2-1-2025-5

[2] Dvořák V., Drábek V.: Architektura procesorů, Vutium 1999, ISBN: 80-214-1458-8

Termín zadání: 5.2.2018 ***Termín odevzdání:*** 21.5.2018

Vedoucí práce: Ing. Petr Petyovský, Ph.D.
Konzultant:

doc. Ing. Václav Jirsík, CSc.
předseda oborové rady

ABSTRAKT

Tato bakalářská práce se dělí na několik částí. První obsahuje seznámení s jednotlivými komponentami procesoru a několika jeho periferiemi. Další částí práce je rešerše reálných, výukových a demonstračních prostředků použitelných ve výuce. Z výsledků této rešerše byly vytvořeny požadavky na výukový systém. Podle požadavků byl proveden návrh architektury výukového procesoru pro výuku nejen na UAMT VUT. V práci je také popsán postup, který vedl k vytvoření fungujícího simulátoru navrženého procesoru. Poslední částí práce je návrh několika výukových úloh, které demonstrují principy fungování obecného počítačového systému a problematiku programování ve strojovém kódu a jazyku assembler.

KLÍČOVÁ SLOVA

simulátor, emulátor, počítačový systém, procesor, instrukce, mikroinstrukce, mikroprocesor, architektura

ABSTRACT

This bachelor thesis is divided into several parts. The first part consists of an introduction to individual parts of a processor and some of its peripheries. Next part of thesis is a research of existing educational and demonstrative tools usable in education. Results of the research were compiled into requirements for educational system. Using those requirements, and design of an architecture for educational processor for education, not only on FEEC BUT was created. As a next step, there is described a procedure, that led to a creation of a working simulator of the designed processor. Last part of this thesis is a design of several educational exercises, that demonstrates principles of computers and programming in a machine code and an assembly language.

KEYWORDS

simulator, emulator, computer system, processor, instruction, microinstruction, microprocessor, architecture

FRIML, Dominik. *Výukový simulátor počítačového systému*. Brno, 2018, 82 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedoucí práce: Ing. Petr Petyovský, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Výukový simulátor počítačového systému“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu bakalářské práce panu Ing. Petrovi Petyovskému, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci. Dále děkuji své přítelkyni a rodině za podporu a trpělivost během studia.

Brno ....

podpis autora

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

Tato práce se věnuje pomůckám vhodným k výuce programování mikroprocesorové techniky. V této oblasti se zdá být nedostatek kvalitních, dostatečně aktuálních, vhodných a dostupných výukových pomůcek.

Z těchto důvodů jsem se rozhodl tuto oblast prozkoumat a ověřit, zda je situace opravdu taková, jaká se zdá. Pro výuku se používají jak reálné procesory, tak různé simulátory a demonstrační prostředky. V závěru rešerše budou shrnuty výhody a nevýhody jednotlivých řešení.

Cílem práce je navrhnout a realizovat výukový simulátor počítačového systému. Ze zjištěných vlastností existujících řešení bude vytvořeno několik požadavků na nový výukový systém. Tento krok je klíčový pro celou práci a proto je nutné jej provést velmi důkladně a s dostatečným časovým předstihem.

Jakmile jsou jasně definovány požadavky, je možné navrhnout architekturu procesoru, který bude simulován. Navíc bude pravděpodobně potřeba navrhnout instrukční a mikroinstrukční sadu procesoru a následně jazyk, který by umožnil jednoduchou editaci programu a mikroprogramu. Navržené komponenty by bylo vhodné podrobit několika myšlenkovým testům, které ověří kvalitu návrhu.

Pro navrženou architekturu je následně možné vytvořit simulátor. Dá se předpokládat, že se při tvorbě a testování simulátoru objeví nedostatky navržené architektury, která bude posléze změněna tak, aby byly nedostatky eliminovány. Tento proces tvoření a ladění aplikace a architektury bude pravděpodobně časově nejnáročnější částí celé práce.

Po dokončení a odladění simulátoru je možné navrhnout výukové úlohy. Tyto výukové úlohy otestují navrženou architekturu z hlediska přínosu do výuky. Následně je bude možné využít v kurzu BMIC (kurz mikroprocesorové techniky na UAMT VUT).

Nakonec bude nástroj představen vyučujícím kurzu BMIC, pro které je simulátor primárně určen, a upraven podle jejich požadavků a připomínek.

1 Procesor, mikroprocesor a mikrokontrolér

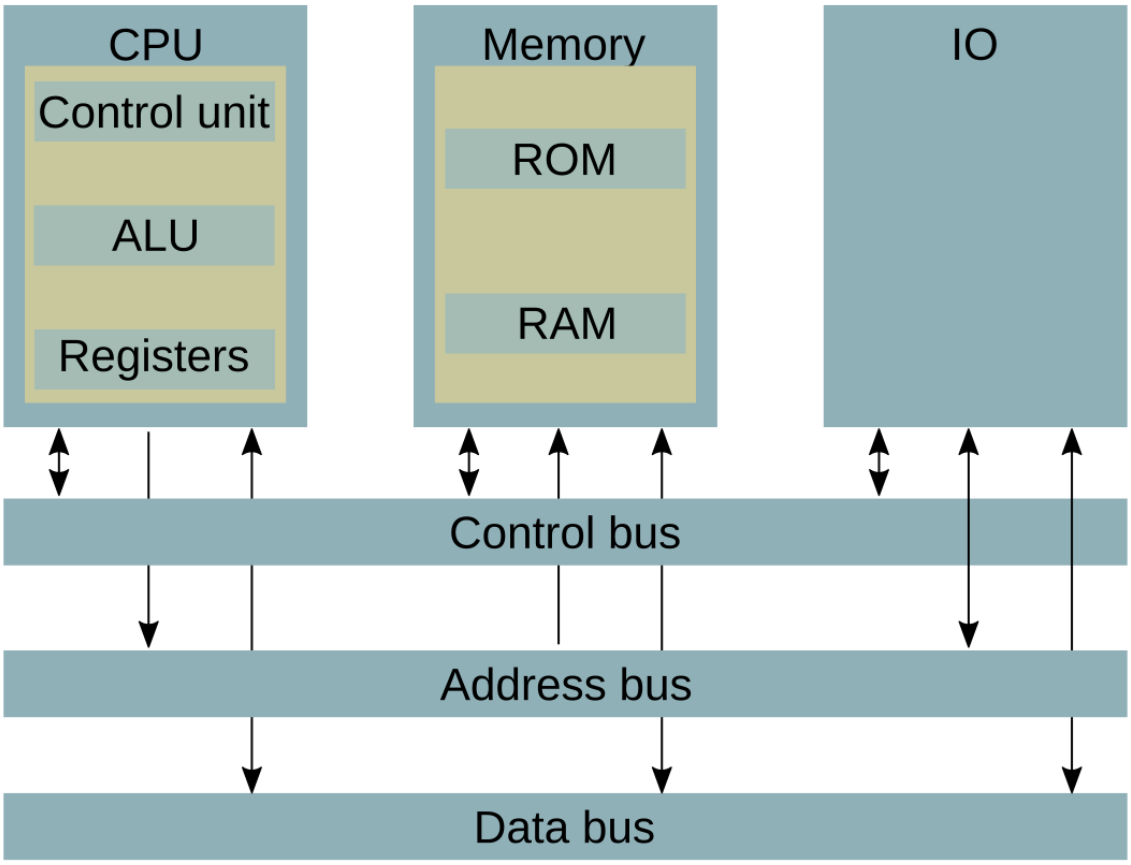
Historie procesoru jak jej známe dnes, sahá mnohem dále do minulosti, než si někteří z nás uvědomují. Procesor, nebo centrální procesorová jednotka (CPU), je součást počítače, která je schopná provádět strojové instrukce. Procesor je základní součást počítače, a tak se nacházel už v nejstarších elektronkových, sálových počítačích.

Postupně byly počítače, včetně CPU, miniaturizovány. Časem zabírala procesorová jednotka pouze jednu přístrojovou skříň, až miniaturizace dosáhla takové úrovně, že bylo možné podstatnou část CPU umístit do jediného integrovaného obvodu. Pro procesor integrovaný do jedné kompaktní součástky vznikl název mikroprocesor.

Integrace ale pokračovala dále. Do čipu byly přidávány další a další funkční bloky počítače, až vznikl integrovaný obvod obsahující kompletní jednoduchý počítač. Takový čip je nazýván jednočipový počítač, nebo také mikrokontrolér.

S postupem času bylo možné do mikrokontroléru obsáhnout více a více periferií, a tak dnes můžeme najít integrované obvody obsahující mnohdy i několik počítačů, hradlová pole, digitální, analogové nebo i rádiové obvody. Takové integrované elektronické systémy nesou název systém na čipu (system on chip, SoC).

2 Seznámení s procesorem a jeho periferiemi



Obr. 2.1: Blokový diagram obecného počítače

Obecný počítač obsahuje centrální procesorovou jednotku (CPU), Sběrnice, programovou paměť (ROM), datovou paměť (RAM), vstupy a výstupy (IO).

V této kapitole se s jednotlivými komponentami seznámíme.

2.1 Procesor

Procesor se skládá z několika funkčních bloků. V následujících kapitolách si jednotlivé funkční bloky projdeme a vysvětlíme si jejich funkci.

2.1.1 Aritmeticko-logická jednotka

Aritmeticko-logická jednotka (dále ALU), je jedna z nejsložitějších komponent procesoru. Jak již napovídá název, jedná se o složitý kombinační obvod, který provádí

aritmetické a logické operace nad celými čísly. Protože procesor spadá pod výpočetní techniku, je jasné, jak důležitá a nezaměnitelná tato komponenta je.

Princip fungování ALU je poměrně jednoduchý. ALU má dva datové vstupy, na které se přivedou vstupní logické hodnoty reprezentující celá čísla pomocí datové sběrnice. Pomocí řídicí sběrnice se zvolí operace, která se má nad vstupy provést. Posledním vstupem může být registr příznaků (speciální registr popsaný v kapitole 2.1.2), který se využívá u některých operací. Až jsou všechny vstupní signály připraveny, hodinový signál zahájí výpočet operace. Výsledkem operace jsou zpravidla dvě binární čísla. Jedno číslo je výsledek operace samotné (například pro sčítání je výsledkem operace samozřejmě součet sčítanců na vstupu), druhé potom číslo uložené v registru příznaků (někdy také nazývaném stavový registr). Toto číslo obsahuje informaci o provedené operaci, která není obsažena ve výsledném čísle (například informaci o přetečení při výpočtu, zda je výsledek nulový, negativní, nebo přenos při výpočtu, který se do výsledného čísla nevešel).

Nejjednodušší aritmeticko-logické jednotky podporují aritmetické operace jako součet (s přenosem i bez), rozdíl (opět s přenosem i bez) a výpočet dvojkového doplňku. Z logických operací například porovnání operandů (větší než, menší než, rovno, ...), negaci (zpravidla čísla z prvního vstupu), konjunkci, disjunkci, ekvivalenci, nonekvivalenci a další.

Dnešní aritmeticko-logické jednotky rozšiřují podporované operace ještě o součin, podíl, posuny a rotace (pokud se o tyto operace nestará jednotka s názvem barrel shifter), rotace přes carry, inkrementace, dekrementace a další.

Všimněme si, že všechny uvedené operace se provádí nad binárními čísly uložené ve formátu celých čísel se znaménkem. Pro operace nad čísly uloženými ve formátu s plovoucí řádovou čárkou bývá dodáván integrovaný matematický koprocesor (floating-point unit, FPU). I když dnes bývá FPU integrována přímo v procesoru, jedná se stále o dva různé funkční bloky.[9][8][10]

2.1.2 Registr

Velmi důležitou součástí procesoru je registr. Můžeme jej najít prakticky v každém procesoru již od jejich vzniku. Jedná se o velice rychlou paměť, která je obvykle schopná uložit jedno číslo v binárním formátu, zpravidla o počtu bitů odpovídajícím šířce datové nebo adresové sběrnice, nebo šířce operandů vstupujících do aritmeticko-logické jednotky.

Jelikož registry používají takřka všechny instrukce, rychlost registrů je klíčová. Proto se pro registry používají ty nejnovější technologie. U běžných procesorů se tedy využívají statické paměti, nejčastěji založené na klopných obvodech typu D.

Registrů v procesoru je větší množství. Některé z nich mají určenou funkci, některé nemůže programátor přepisovat, jiné jsou programátorovi skryty úplně. Podívejme se nyní na některé z nich.[8]

Buffer

Buffer je registr, který je programátorovi skryt. Slouží k uložení vstupní nebo výstupní proměnné pro funkční bloky. Takovýto buffer můžeme nalézt například na vstupu v aritmeticko-logické jednotky, abychom mohli na vstup přichystat jiné proměnné, zatímco ALU počítá.

Registry pro obecné použití

Využívané programátorem jako dočasné úložiště. V některých procesorech můžeme nalézt registry určené pouze pro data (datový registr), pouze pro adresy (adresový registr), nebo speciální registry pro uložení čísla s plovoucí řádovou čárkou (floating-point register). Nejznámějším registrem pro obecné použití, který nechybí takřka v žádném procesoru je akumulátor (někdy nazývaný střadač). Někdy se registry sdružují do párů, což dovoluje uložit číslo o dvojnásobné šířce (například registry H a X u mikrokontroléru HCS08).

Velmi často jsou tyto registry uspořádány do takzvaných registrových bank. Registrová banka je sada registrů, která se dá přepnout na jinou sadu. Podprogram tedy nemusí ukládat všechna data z registrů na zásobník, ale pouze přepne na jinou registrovou banku. Tento přístup je pro programátora mnohem příjemnější než přemýšlet, které registry se využijí a které ne. Doba provedení instrukce přepnutí banky bývá také mnohem kratší než instrukce ukládání dat z registru na zásobník, která se musí pro uložení všech registrů provést vícekrát.

Čítač

Kromě zápisu a čtení uloženého čísla umožňuje i jeho inkrementaci nebo dekrementaci.

Speciálním čítačem je časovač. Rozdíl mezi čítačem a časovačem je ten, že časovač je inkrementován v pravidelných intervalech (například vnitřními hodinami mikrokontroléru), zatímco čítač inkrementuje na základě zjištění sestupné nebo náběžné hrany sledovaného signálu. Tento signál může být jak vnitřní, tak vnější.

Čítač instrukcí

Registr, bez kterého by žádný procesor nemohl nikdy fungovat. Čítač instrukcí (někdy také Program counter, nebo Instruction counter) slouží jako ukazatel na ak-

tuálně prováděnou instrukci v paměti. Program Counter je vlastně čítač, který je inkrementován po každém provedení instrukce (u některých procesorů již po načtení instrukce do instrukčního dekodéru).[11]

Řídicí registry

Dovolují nastavovat řídicí bity. Řídicí bit je bit registru, který má konkrétní význam. Nastavováním těchto bitů můžeme nastavovat chování různých funkcí funkčních bloků. Například povolovat přerušení, nebo měnit mód procesoru.

Registr příznaků

Výstupní registr aritmeticko-logické jednotky. Slouží pro uložení informací o provedeném výpočtu pomocí speciálních bitů zvaných flag. Příkladem takové informace je flag zero, který je nastaven, pokud je výsledkem operace nula a smazán, je-li výsledkem jiné číslo.

Někdy registr příznaků obsahuje také řídicí registry.

Stack pointer

Ukazatel na poslední prvek v zásobníku. Stack pointer se tedy sám dekrementuje při přidání prvku na zásobník a opět se inkrementuje při odebrání prvku. Speciální instrukce dovolují programátorovi mazat nebo tvořit více prvků najednou pouze přičtením nebo odečtením konstanty od zásobníku. Takto vytvořené prvky ovšem nejsou inicializované.

2.1.3 Řadič

Řadič je sekvenční obvod, který pomocí řídicích signálů (binárních hodnot na řídicí sběrnici) ovládá ostatní moduly procesoru. Je to tedy řadič, který říká aritmeticko-logické jednotce, jakou operaci má vykonat, do jakého registru se uloží jaká hodnota a podobně.

Kromě řadiče v podobě sekvenčního obvodu se můžeme setkat i s takzvaným mikroprogramovým řadičem. Mikroprogramový řadič je realizován pamětí, která obsahuje mikroprogram. Mikroprogram je série mikroinstrukcí, které tvoří jednu celou instrukci. Provedení jedné mikroinstrukce zpravidla trvá jeden takt (perioda hodinového impulsu). Počet taktů potřebných k provedení jedné instrukce tedy odpovídá počtu jejich mikroinstrukcí. Příkladem takovéto instrukce může být instrukce sčítání. Její mikroinstrukce mohou být:

- Načíst první sčítanec z registru na sběrnici
- Uložit sčítanec do akumulátoru

- Načíst druhý sčítanec, tentokrát například z paměti programu, na sběrnici
- Připravit ALU k operaci sčítání a operaci provést
- Uložit výsledek do akumulátoru
- Inkrementovat čítač instrukcí

Takový mikrořadič může mít spoustu výhod. Mikroinstrukce mohou být výrazně delší než instrukce, jelikož to neovlivňuje velikost programu v operační paměti. Další výhodou je možnost instrukce upravovat například z důvodu zefektivnění odhalení bezpečnostní chyby v instrukční sadě.

Často se pojmem řadič nazývá jakýkoliv konečný stavový automat, který řídí činnost i jiných částí počítače (tedy nejen částí procesoru). Příkladem může být autonomní řadič diskové paměti. [8][10]

Dekodér instrukcí

Dekodér instrukcí je součást řadiče, která předpřipravuje instrukce pro řadič. Načtená instrukce z paměti programu putuje po datové sběrnici do instrukčního registru, kde je uložená po dobu jejího dekódování. Dekódování rozdělí instrukci na samotný znak instrukce a její operandy. Tedy například 3-adresová instrukce je rozdělena na operační znak, adresu výsledku, adresu prvního operandu a adresu druhého operandu. Dekodér již při načtení operačního znaku rozhodne, kolik je třeba načíst operandů.

Po dekódování instrukce přiřadí dekodér patřičný stav řadiči, nebo v případě mikroprogramového řadiče nastaví adresu první mikroinstrukce odpovídající mikroprogramu dané instrukce.[8]

2.1.4 Sběrnice

Sběrnice je nejjednodušší, ale klíčová část procesoru. Slouží k distribuci signálů a dat mezi jednotlivými částmi procesoru tak, že zpravidla jedna část data vysílá a jiná část data příjmá. Jedná se o skupinu vodičů, po nichž se přenáší binární informace.

V mikropočítači se nachází sběrnice tři. Datová, adresová a řídicí. Dříve bývaly z důvodu prostorové náročnosti sběrnice multiplexovány, tedy jedna sběrnice může v jednom okamžiku plnit funkci sběrnice datové, a v jiném okamžiku funkci běrnice adresové. [8]

Datová sběrnice

Datová sběrnice slouží, jak název napovídá, k přenosu dat. Šířka datové sběrnice omezuje šířku čísla, které se může přenést za jeden takt. Datové sběrnice bývají dvě. Jedna je vnitřní sběrnice procesoru, která slouží k přenosu dat uvnitř procesoru,

tedy nejčastěji dat z registrů a ALU. Druhou sběrnicí je vnější sběrnice, pomocí které procesor získává data z periferií.

Adresová sběrnice

Adresová sběrnice slouží k výběru adresy paměťové buňky, se kterou chceme pracovat. Toto platí, i když chceme zapisovat například do vstupně-výstupního zařízení. Ty totiž bývají paměťově adresované, dá se k nim tedy přistupovat jako k paměti.

Šířka adresové sběrnice určuje, s jak velkým adresovým prostorem lze pracovat. Tedy například u 32 bitové adresové sběrnice máme k dispozici $2^{32} - 1 = 4294967295$ adresovatelných paměťových buněk, což odpovídá zhruba čtyřem gigabajtům adresovatelné paměti, nebo paměťově adresovaným zařízením.

Řídicí sběrnice

Pomocí řídicí sběrnice je možné ovládat (řídit) komponenty a periferie procesoru. Zatímco po ostatních sběrnicích se šíří většinou paralelní data, po řídicí sběrnici se rozvádí především individuální a nesourodé bity.

2.2 Paměť

Další část, bez které by se počítač neobešel, je paměť. Využívá se jak k ukládání programu, který procesor vykonává, tak mezivýsledků a výsledků jeho práce. Je tedy jasné, jak moc je paměť pro činnost procesoru klíčová. V minulých kapitolách už jsme se zmínili o registrech a registrových sadách. Proto se jim v této kapitole věnovat nebudeme. Je třeba si také uvědomit, že zatímco registry jsou obvykle extrémně rychlé paměti, velmi malé kapacity, které jsou obvykle schopné uložit bitové slovo o maximální šířce sběrnice, paměti, o nichž je tato kapitola, mají za úkol uložit velká množství dat.[10]

2.2.1 Paměti RAM

Zkratka RAM doslova znamená Random Access Memory. Z toho plyne, že RAM je zastřešující termín pro všechny nesekvenčně zapisovatelné a čitelné paměti. Tyto paměti jsou využívány jako operační paměti, tedy k ukládání dat, výpočtů, mezivýsledků a občas i instrukcí. Lze z nich číst i do nich zapisovat. Paměti RAM obvykle neudrží uložená data bez přítomnosti napájecího napětí.

Statická paměť

Statická paměť je tvořena klopnými obvody. Každý klopný obvod ukládá jeden bit. Je tedy potřeba pro každý bit klopný obvod na uložení dat a ovládací logiku vyhodnocující adresní a další signály. Toto uspořádání je tedy značně náročné na počet transistorů, protože klopný obvod je velice složitý typ buňky (pro uložení jednoho bitu), dále kvůli adresním vodičům, kterých je například na 1Mb potřeba 20. Zmíněné nedostatky řeší dynamické paměti.

Dynamická paměť

Tento typ paměti používá k uložení dat elektrický náboj na kapacitě. Binární informace o přítomnosti náboje odpovídá uložené binární informaci. Neideální, reálný kondenzátor se však vybíjí i v době, kdy je paměť připojena ke zdroji napájení. Aby se kondenzátor nevybil pod určitou úroveň, a neztratil tím uložená data, je potřeba data v paměťových buňkách dynamicky obnovovat. Odtud název dynamická paměť. Tento proces bývá v literatuře popsán jako refresh. Kdyby se data pravidelně neobnovovala, dynamická paměť by ztratila všechna data během asi 4 milisekund.

Buňky jsou umístěny do čtvercových matic (nebo co nejvíce čtvercových, pokud z nějakých důvodů není možné použít čtvercovou). Polovina adresových vodičů tvoří adresu řádku a druhá polovina tvoří adresu sloupce. Menší paměťové buňky nám dovoluje vytvořit paměti o velkých kapacitách v malých pouzdrech.

2.2.2 Paměti ROM

Zkratka ROM znamená Read Only Memory, tedy paměti určené pouze pro čtení. Během historie počítačů se postupně pro paměti ROM využívaly všechny elektronické prvky. Byly tak paměti využívající indukčnost, transformátory, feritová jádra, kapacitory, diody, unipolární i bipolární transistory. Některé paměti není možné přepisovat fyzicky, jiné jsou přepisovatelné složitě a oproti pamětí typu RAM zdlouhavě. Nejsou vhodné k častému přepisování, jelikož mají omezený počet přepisovacích cyklů.

Paměti ROM slouží k uložení dat, která musí existovat i po odpojení napájení. Bývají tedy používány pro startovací proces počítače (boot proces).

2.3 Vstupně-výstupní brány

Procesor a jeho činnost má smysl teprve ve spojení s periferními zařízeními, díky kterým získáváme informaci o vnitřních stavech počítače. Z technického hlediska je styk s okolím možný prostřednictvím vstupně-výstupních bran.

Vstupně-výstupní brána je obvod, který zprostředkovává předání dat mezi sběrnicí mikropočítače a periferním zařízením. Vstupně-výstupní brány se ovládají pomocí vlastních řídicích registrů. Z pravidla můžeme nastavovat bránu jako vstupní nebo výstupní. U výstupní brány můžeme měnit výstupní hodnotu. Ta bývá 1, 0, nebo Z. 1 znamená kladné výstupní napětí, 0 znamená uzemnění výstupu a Z znamená přivedení pinu na vysokou impedanci. Vstupní proměnná nám potom dovoluje vyčítat binární hodnoty z brány. Některé brány nám také umožní připojit vstup na vnitřní pull-up, nebo pull-down resistor, nebo měnit rychlost přechodu napětí při změně logické úrovně (slew rate). [10]

3 Existující mikroprocesory a mikrokontroléry

V této kapitole se seznámíme s některými z existujících mikroprocesorů a mikrokontrolérů, které by bylo možné použít ve výuce. Budeme se věnovat hlavně osmibitovým architekturám, jelikož použití více než osmibitových architektur by mohlo být dle mého hlediska kontraproduktivní pro výuku. Dále jejich architekturu zhodnotíme z hlediska vhodnosti použití jako výukové pomůcky.

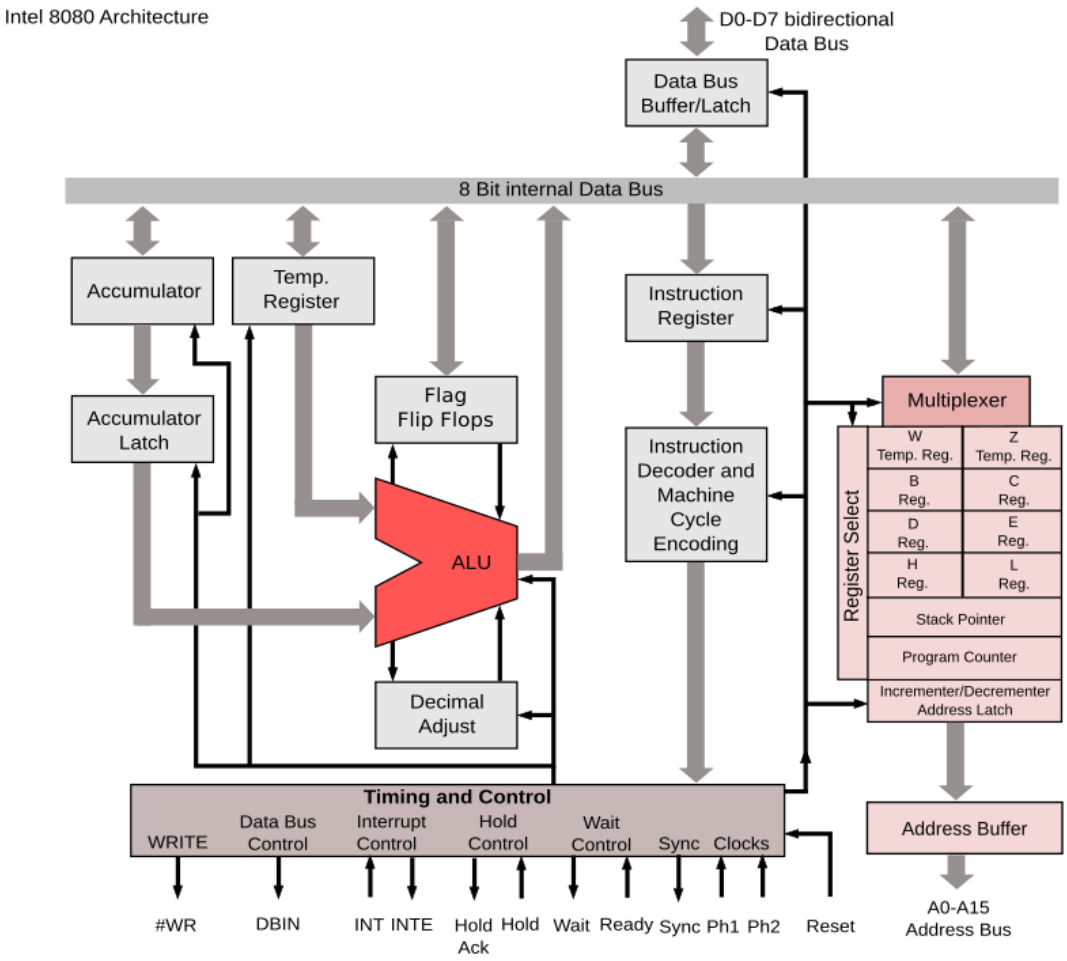
Postupně se podíváme na mikroprocesor Intel 8080, jeden z prvních osmibitových mikroprocesorů, následně na rodinu mikroprocesorů S08 firmy NXP, ze které se používá mikroprocesor HCS08 v kurzu BMIC na UAMT VUT. Následně se podíváme na ARM cortex-M0, který je využíván ve výukovém embedded systému BBC Micro Bit. [28]

3.1 Mikroprocesor 8080(A)

Jedním z prvních monolitických mikroprocesorů byl mikroprocesor 8080 společnosti Intel. Jelikož byla zatěžovací schopnost výstupů příliš malá, byl po několika měsících nahrazen novějším typem 8080A, který se od původního typu lišil takřka jen výkonnějšími výstupními budiči.

3.1.1 Popis architektury 8080(A)

- Délka slova: základní slovo (slabika) je paralelní osmibitové. Zdvojené slovo má 2 slabiky sériově-paralelně. Delší slova je potřeba zajišťovat programově.
- Sběrnice datová: osmibitová, obousměrná, třístavová.
- Sběrnice adresová: šestnáctibitová, výstupní, třístavová.
- Aritmetická jednotka: založena na paralelní osmibitové sčítačce s obvody pro urychlení přenosu, pracuje se dvěma osmibitovými registry, které nejsou programově přístupné a zachovají operandy po dobu výpočtu, s ALU dále souvisejí obvody pro úpravu výsledku při dekadických operacích (binary decimal adjust).
- Indikátory: bity stavového registru (Flag Flip Flops) jsou klopné obvody tvořící registr příznaků, které se při výkonu některých instrukcí nastaví podle výsledku operace. 8080(A) má 5 indikátorů:
  - CY - (carry) nastaví se do stavu logické hodnoty 1, pokud nastal přenos z nejvyššího řádu.



Obr. 3.1: Blokové schéma vnitřního uspořádání mikroprocesoru 8080(A).

  - AC - (auxiliary carry - half carry) nastaví se do logické hodnoty 1, nastal-li přenos z nižší tetrády slabiky do vyšší.
  - Z - (zero) nastaví se do logické hodnoty 1, je-li výsledek aritmetické nebo logické operace nulový.
  - S - (sign) nastaví se do logické hodnoty 1, jestliže výsledek operace je záporný.
  - P - (parity) nastaví se do stavu 1, jestliže parita výsledku je sudá (je-li ve výsledku sudý počet jedniček).

  Některé instrukce nenastavují bity registru příznaků vůbec, jiné instrukce nastavují pouze některé indikátory.
- Zápisníková paměť: je tvořena sedmi osmibitovými registry označenými A, B, C, D, E, H, L. Jedná se o registry univerzální, mají ale následující speciální vlastnosti:

  - A - (accumulator) je střadač pro jednoslabikové operace.
  - HL - dvojice registrů sloužící jako střadač pro dvouslabikové sčítání. Dále umožňuje nepřímé registrové adresování.
  - BC a DE - tyto dvojice také umožňují nepřímé registrové adresování, ale v omezené míře o oproti registrové dvojici HL.
- Zásobník - není součástí mikroprocesoru 8080(A), je potřeba jej vytvořit v hlavní paměti. Pro adresování takto vytvořeného zásobníku je mikroprocesor vybaven šestnáctibitovým ukazatelem zásobníkové paměti (stack pointer).
- Čítač instrukcí - (program counter) je šestnáctibitový, přepsatelný instrukcemi skoků, podmíněných skoků a vyvoláním podprogramu.
- Registr instrukcí - je osmibitový, udržuje operační znak instrukce po dobu dekódování a vykonávání instrukce
- Formáty dat - mikroprocesor uplatňuje princip little-endian, záporná čísla reprezentuje v kódu druhého doplňku. Podporovaný datový typ je pouze celé číslo (integer).
- Instrukce - v instrukčním souboru nalezneme 91 instrukcí. Tento počet je možno redukovat na 78, zahrnutím třinácti instrukcí s nepřímým registrovým adresováním mezi instrukce pracující s registry. [3][4]

3.1.2 Vhodnost použití mikroprocesoru 8080(A) ve výuce

Mikroprocesor je velice jednoduchý a ve své době sloužil jako inspirace pro většinu nových mikroprocesorů. Používal se v mnoha aplikacích, od kalkulaček, přes domácí spotřebiče až po počítače, než byl nahrazen procesorem Z80 společnosti Zilog. Dnes je však mikroprocesor zastaralý a je vhodnější použít alternativu.

3.2 Mikrokontrolér S08

S08 je rodina mikrokontrolerů od firmy NXP. Rodina vychází z rodiny procesorů 68HC08 původně od firmy Motorola, která je modifikací rodiny 68HC05 vytvořené inspirací procesorem 6800. Procesor 6800 byl konkurenčním procesorem pro 8080(A). Můžeme tedy říct, že rodina S08 je potomkem mikroprocesoru 6800.[24] [18]

3.2.1 Vhodnost použití mikrokontroléru S08 ve výuce

Procesory z této rodiny nabízejí mimo jiné i osmibitové mikroprocesory se širokou škálou vlastností, vhodných pro výuku. Procesor má praktické využití v praxi, nabízí background debug controller umožňující ladění přímo samotného mikroprocesoru, má relativně jednoduchou instrukční sadu a šířku slova osm bitů.

O těchto procesorech se dá říci, že jsou vhodné k výuce. Bohužel firma NXP prochází značnými změnami, a tak je jeho budoucnost nejistá.

3.3 Mikrokontroléry architektury ARM cortex-M0

Cortex-M0 je architektura nejjednodušší rodiny mikrokontrolérů architektury ARM, u nichž je žádoucí dosáhnout co nejnižší výrobní ceny, malých rozměrů a malého příkonu (s tím samozřejmě souvisí i příslušně nízký výpočetní výkon). Mikrokontroléry architektury cortex-M0 podporují instrukční sadu Thumb bez tří instrukcí. Dále podporují šest instrukcí z instrukční sady Thumb-2. [12]

Již dnes je možné narazit na výukové pomůcky využívající mikrokontroléry této architektury. Jedním z nejznámějších příkladů může být BBC micro:bit. Jedná se o mikropočítač, který vznikl pro výuku mikroprocesorové techniky ve Spojeném království Velké Británie a Severního Irska. Mikropočítač obsahuje mikrokontrolér Nordic nRF51822 s jádrem ARM cortex-M0, akcelerometr, magnetometr, bluetooth, USB, displej tvořený 25 LED diodami a dvě programovatelná tlačítka. Je jej možné programovat pomocí Microsoft MakeCode, MicroPython, C++, Forth a dalších programovacích jazyků.[28] [17]

Jelikož procesory ARM jsou používané čím dál častěji, je pravděpodobné, že mikroprocesory ARM cortex-M0 jsou zajímavou volbou. [19]

3.3.1 Vhodnost použití ARM cortex-M0 ve výuce

Procesor je relativně jednoduchý, s jednoduchou instrukční sadou, vysokou pravděpodobností výroby i za několik let. Je jisté možné využití v praxi. Nevýhodou je, že architektura je třiceti dvou bitová, takže slova jsou méně přehledná. Dále obsahuje instrukce obtížněji uchopitelné pro začátečníky a studenty.

4 Existující výukové pomůcky

V minulých kapitolách jsme viděli, že použití reálných mikrokontrolérů ve výuce má své nenahraditelné výhody a charakteristické nevýhody.

V zájmu studenta by bylo ideální seznámit se s co nejširší škálou architektur. Na to bohužel často není v rozsahu jednoho kurzu dostatečná časová kapacita, a tak se student student v rámci bakalářského studijního programu setká pouze s jednou konkrétní použitou architekturou.

Další velkou nevýhodou je velice nepřehledné zobrazení vnitřních stavů procesoru v grafickém rozhraní vývojového prostředí, které může být pro studenty matoucí. Vyučující musí často kreslit vysvětlující obrázky pro demonstraci vnitřního toku dat.

V této kapitole se podíváme na existující projekty využitelné ke školení v této oblasti a shrneme si jejich výhody a nevýhody při použití ve výuce.

4.1 Megaprocessor

Megaprocessor je velmi velký mikropočítač. Myšlenka, která vedla k jeho vytvoření byla jednoduchá. Nelze-li zmenšit člověka tak, aby se vešel do mikropočítače a mohl vidět, jak mikropočítač funguje, musíme zvětšit mikropočítač. Tak James Newman z Cambridge postavil mikropočítač o rozměrech $10m$ na $2m$. [20]

Jelikož hlavní myšlenkou bylo vysvětlit, jak mikropočítač funguje na úrovni transistorů, je postaven z rekordního množství diskrétních součástek, viditelně orientovaných do logických hradel, která jsou opět viditelně orientována do logických bloků mikropočítače. Navíc je každý důležitější signál opatřen LED diodou indikující stav signálu. Některé stavy, jako například data v registrech, jsou také zobrazovány v šestnáctkové soustavě na sedmisegmentových displejích.

Na první pohled nejzajímavější na celém projektu je paměť RAM, u které je každý bit opatřen LED diodou indikující uložený stav. Paměťové bity jsou rozmístěny do rovnoměrného pole, a tak vzniká jednoduchý displej. Tímto způsobem je paměť RAM megaprocesoru často používána.

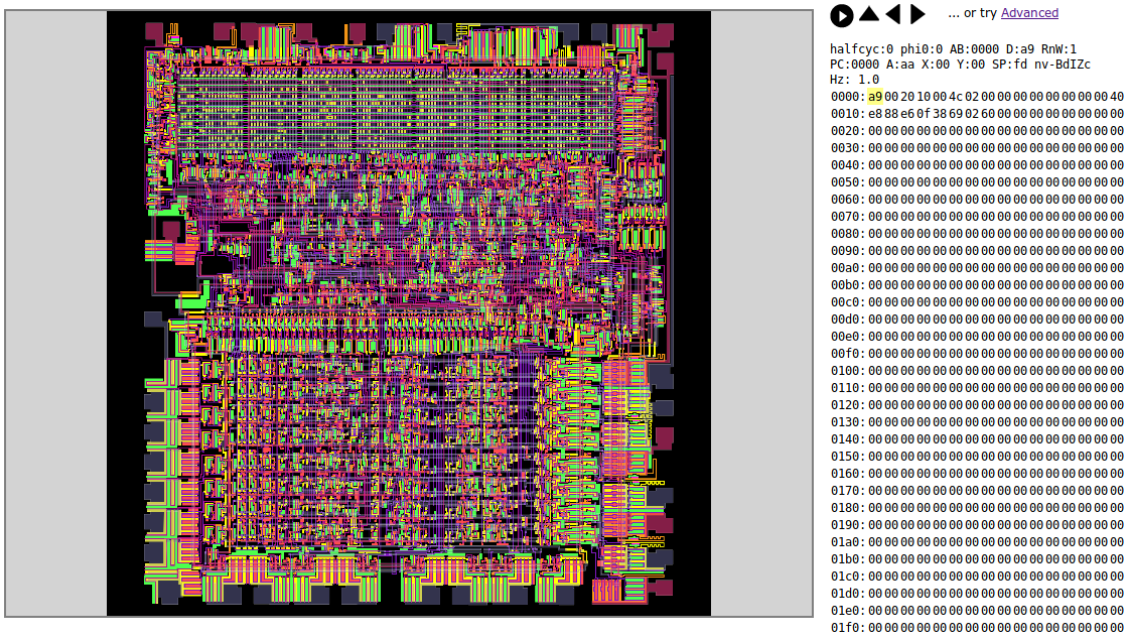
Tento šestnáctibitový procesor je možné programovat, procházet jeho kód po jednotlivých cyklech, nebo měnit rychlost hodinových cyklů při normálním běhu programu, a tak sledovat, co se v procesoru zrovna děje. Procesor se programuje pomocí počítačového programu, který dovoluje psát kód v jazykce assembler. Tento kód je možno spustit, krokovat a emulovat stav reálného procesoru. Dále je možno nahlédnout, jak vypadají data v paměti, a dokonce si prohlédnout i graficky zobrazenou videopaměť. [20]

4.1.1 Vhodnost použití projektu megaprocessor ve výuce

Megaprocesor by se při výuce na univerzitě nebo jiné škole dal využít jen velmi těžko. Pomineme-li, že se na světě nachází jen jeden, takže by bylo nutné jej postavit, že váží půl tuny a že je velice prostorově náročný, narazili bychom jistě na cenu. Pořizovací cena součástek použitých na mikroprocesor se pohybuje kolem 40000 liber. [20]

Přestože je megaprocesor vhodný pro demonstraci logických obvodů a jejich využití, pro demonstraci a vysvětlení funkce mikroprocesoru je pohled na změť tranzistorů a LED diod pro začátečníka matoucí. Megaprocesor je sice opatřen popisky jednotlivých součástek, náčrty sběrnic a jejich funkcí, samotná bitová šířka instrukce, která činí 16 bitů, je pro vysvětlení a demonstraci fungování procesoru zbytečná.

4.2 The Visual 6502



Obr. 4.1: Grafické rozhraní Visual 6502.

The Visual 6502 je webová aplikace napsaná v html a programovacím jazyku javascript. Jak název napovídá, aplikace vizuálně simuluje procesor 6502, a to na úrovni křemíku.

Při prvním spuštění nás uvítá velmi barevný obrázek obvodů integrovaných v čipu. Barvami jsou totiž oddělené jednotlivé vodivé vrstvy jako například napájení, zem, signál, kovové vodiče a podobně. Dále můžeme na obrazovce vidět jednoduchou mapu bitů v paměti a tlačítka pro start, krok vpřed, krok vzad a restart.

Při krokování můžeme pozorovat, jak jednotlivé signály na procesoru mění svůj stav a jak se mění data v paměti.

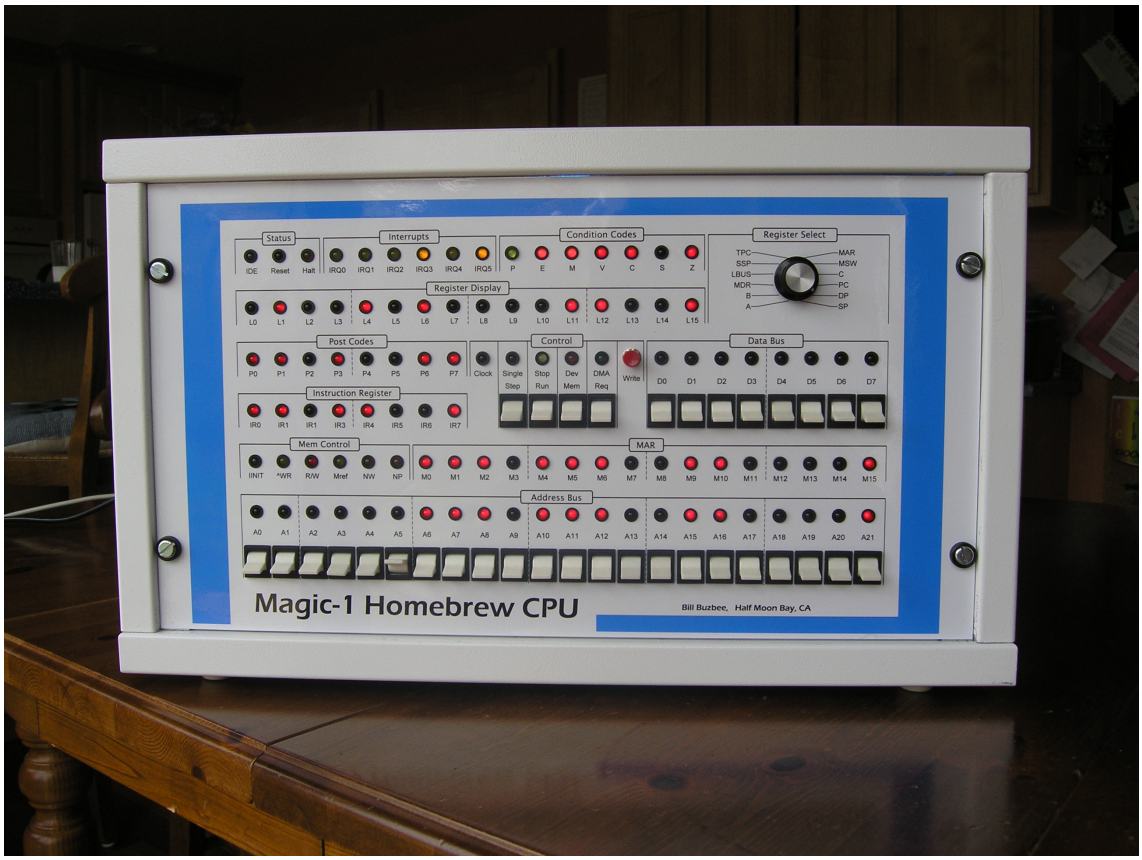
Dále máme možnost se přepnout do pokročilého módu, kde můžeme vypínat a zapínat zvýraznění jednotlivých vodičů, psát program v binárním kódu (nebo odpovídajících klíčových slovech v asembleru), ten potom překládat a trasovat.

Tato aplikace je velmi názorná pro vysvětlení fyzického zapojení logických obvodů, distribuce signálů procesorem, způsob vyvedení signálů z pouzdra čipu nebo uspořádání funkčních bloků v čipu. [22]

4.2.1 Vhodnost použití simulátoru Visual 6502 ve výuce

Pro výuku fungování procesoru je tento model příliš složitý a nepřehledný. Čtení stavu procesoru není jednoduché ani pro člověka znalého procesoru 6502, protože je nutné studovat přímo signály na daných vodičích.

4.3 Magic-1 Homebrew CPU



Obr. 4.2: Přední panel Magic-1.

Magic-1 je velice zajímavý mikropočítač navržený a sestrojený Davidem Brooksem. Zajímavostí na tomto počítači je totiž hned několik. [21]

Počítač je navržený na TTL logických obvodech série 74. Všechny tyto čipy (zhruba 200) jsou popropojovány metodou ovíjených spojů, tedy na výrobu nebyla potřeba ani trocha pájecího cínu.

Pro takto po domácku navržený počítač je možno psát programy nejen ve strojovém kódu a assembleru, ale také v ANSI C díky cross-compileru LCC. Toto umožnilo na mikroprocesoru zprovoznit port operačního systému Minix 2, kompletní TCP/IP stack a stovky programů, jako webový server, telnet server nebo hry jako Eliza, nebo Conway's life. Na stránkách autora [21] můžeme najít způsob, jak se pomocí telnetu na počítač připojit a tyto programy vyzkoušet.

Když se ale podíváme na fyzickou přední stranu přístroje, uvidíme velké množství LED indikátorů a přepínačů. Indikátory nám ukazují stavy mikroprocesoru, tedy stavy registrů, sběrnic, přerušení,... Přepínače nám dovolují měnit data na sběrnicích a hlavně zapnout nebo vypnout běh mikroprocesoru a krokovat program po jednotlivých cyklech.

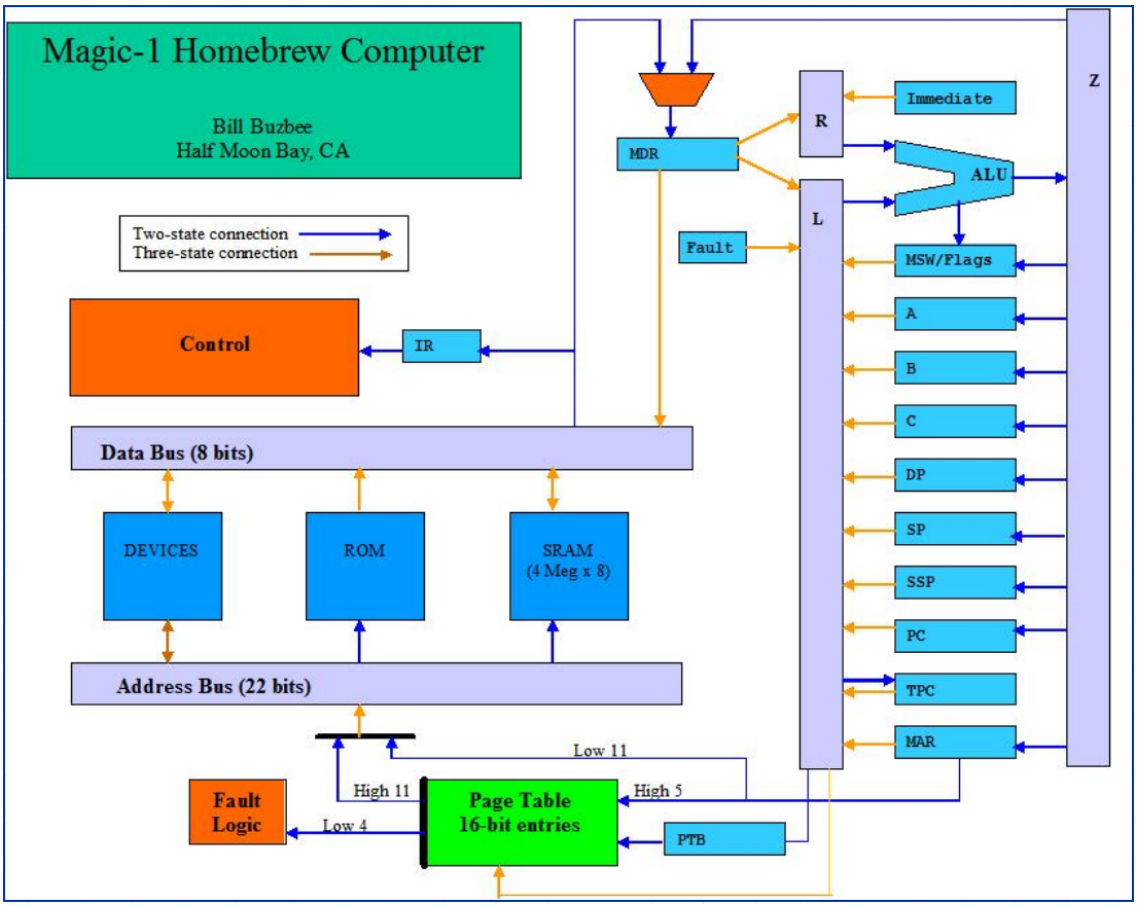
4.3.1 Popis architektury Magic-1

Základní délka slova je 8 bitů, což z tohoto procesoru dělá osmibitový procesor. Paměťový prostor je 22. bitový, ale jednotlivé procesy mohou používat vždy jen 16. bitové virtuální adresy.

Procesor má dále následující registry:

- A - akumulátor. Může být adresován jako osmi, nebo šestnácti bitový. Akumulátor je výchozí registr pro výsledek většiny operací. Jinak používaný jako obecný registr.
- B - obecný registr, zdroj operandu pro ALU
- C - speciální čítač pro bitové posuny.
- MSW - stavový registr ALU. Obsahuje následující stavové bity:
  - C - carry
  - Z - zero
  - S - sign
  - V - overflow

  Dále funguje jako řídicí registr obsahující následující bity:
  - M - mode (0 pro supervisor, 1 pro uživatele) Tato podpora umožňuje spustit jednoduchý operační systém s ochranou jednotlivých procesů a jádra operačního systému.
  - P - Paging - povoluje stránkování a přerušení



Obr. 4.3: Blokové schéma vnitřního uspořádání mikroprocesoru Magic-1.

  - D - Data - bit, který indikuje, zda chyba paměti nastala při adresování programové nebo datové části paměti.
- DP - data pointer - globální ukazatel na počáteční datovou adresu
- SP - stack pointer
- SSP - supervisor stack pointer (používán v supervisor módu)
- PC - program counter - programový čítač
- PTB - program table base - ukazatel do stránkovací tabulky pro proces. V supervisor módu nastaveno na $0x0000$.

Instrukční sada využívá pouze šest adresovacích módů.

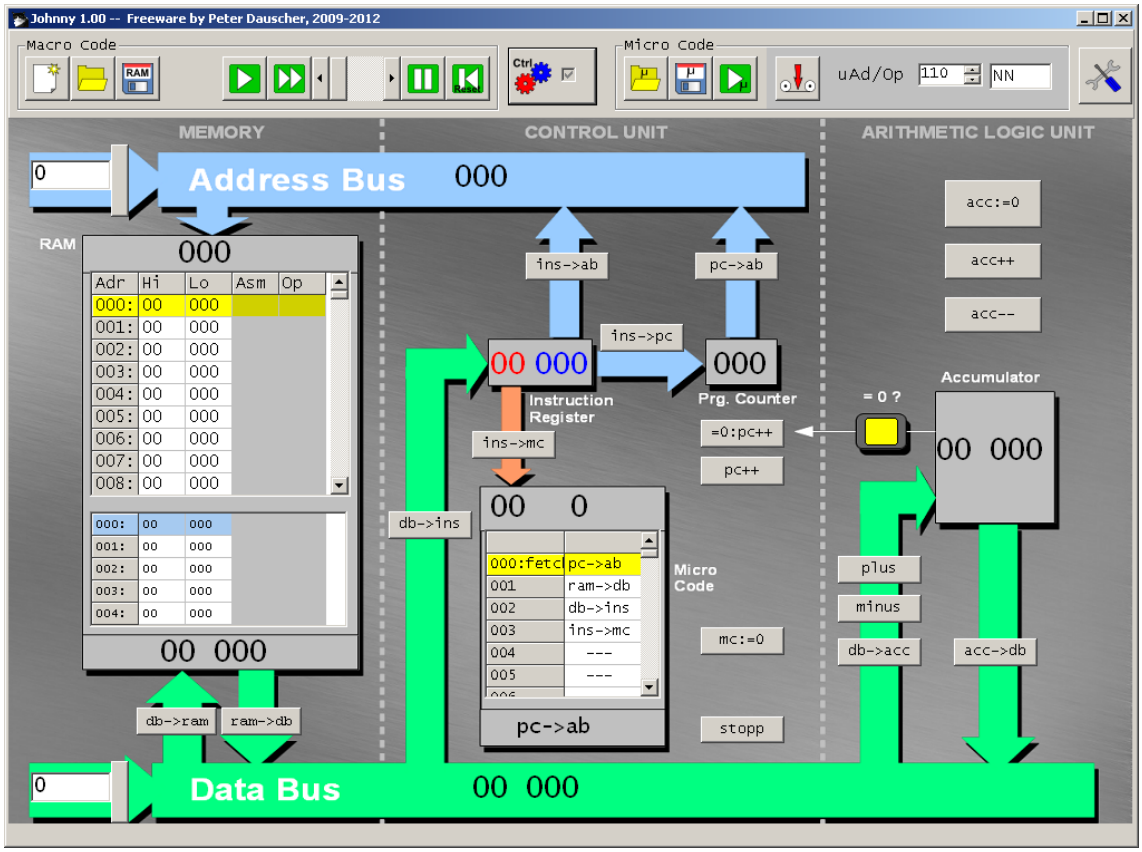
- Register Indirect with offset - uint8(A) and uint8(B)
- Frame local with offset - uint8(SP) and uint16(SP)
- Global with offset - uint16(DP)
- Immediate - (PC++)
- Push - (−−SP)
- Pop - (SP++)

Samotné instrukce jsou tří, dvou, jedno, nebo bezoperandové. Samotných instrukcí je potom 47 až 239 podle toho, jak se počítají. Dekodér instrukcí je mikrokódový. [21]

4.3.2 Vhodnost použití počítače Magic-1 ve výuce

Magic-1, podobně jako megaprocesor je počin jednoho nadšence, a přesto že jsou dostupné všechny potřebné zdrojové soubory, výroba vlastního školního přípravku by zabrala mnohem více času a úsilí, než nákup existujícího mikroprocesoru na vývojářské desce.

4.4 Johnny simulator



Obr. 4.4: Grafické rozhraní simulátoru Johnny.

Johnny simulator je velice jednoduchý výukový simulátor procesoru von-Neumannovy architektury. Skládá se ze tří hlavních bloků, a to z paměti, ovládací jednotky a aritmeticko-logické jednotky.

Paměťový prostor není rozdělen na několik pamětí různých typů, popřípadě periferií. Celý paměťový prostor, vyjádřený trojciferným desítkovým číslem, je obsazen jedinou pamětí typu RAM. Na každé adrese můžeme uložit pěticiferné číslo, potažmo instrukci, opět v desítkové soustavě. Paměť je připojená na adresovou a datovou sběrnici, kterou komunikuje s ostatními bloky procesoru.

Aritmeticko-logická jednotka je také velmi zjednodušená. Obsahuje jeden registr, akumulátor, který umí inkrementovat, dekrementovat, nulovat a odečíst nebo sečíst s číslem, které je aktuálně na datové sběrnici. Dále ALU obsahuje jeden stavový registr o jednom bitu indikujícím, zda v akumulátoru je číslo nula.

Nejsložitějším blokem je řídicí jednotka. Ta se dá pro extrémní zjednodušení skrýt a procesor je použitelný i bez ní. Řídicí jednotka obsahuje dva registry, instrukční registr a čítač instrukcí. Dále zde najdeme paměť pro mikroinstrukce.

Na celé ovládací obrazovce najdeme spoustu tlačítek. Každé jedno tlačítko odpovídá jedné mikroinstrukci, tím pádem uživatel nemusí vůbec psát kód do paměti a může ovládat celý mikroprocesor podle své libosti. Pokud ale uživatel do paměti instrukce zapíše, může se pohybovat v programu po jednotlivých instrukcích nebo po mikroinstrukcích, může během programu i volat jednotlivé mikroinstrukce manuálně, a ovlivňovat tak běh programu. Uživateli je také dovoleno definovat vlastní instrukce a instrukční sady.[23]

4.4.1 Vhodnost použití simulátoru Johnny ve výuce

Johnny je jedním z nejlepších výukových a demonstračních simulátorů, na které se dá narazit. Bohužel ale má také spoustu nedostatků, které brání demonstraci jak komplexnějších operací, tak operací naprosto základních.

Celý mikroprocesor používá desítkovou soustavu, což by mohlo být pro studenty velice matoucí. Zakrývá se tím celá boolovská logika, na které jsou všechny dnešní procesory postavené. Dále se například nedají pomocí tohoto simulátoru vysvětlit a demonstrovat bitové operace, pro nízkoúrovňové programování tak typické, nebo vyjádření záporných čísel dvojkovým doplňkem a počty s ním.

Každá instrukce musí být definovaná maximálně deseti mikroinstrukcemi, což je pro implementaci komplexnějších instrukcí také velice omezující.

5 Požadavky na výukový systém

Výukový systém by měl být přenositelný mezi různými platformami, jednoduše použitelný, všestranný, ale jednoduchý. Měl by být alespoň teoreticky aplikovatelný, rozšiřitelný a pokud možno aktuální. To je relativně hodně požadavků a je potřeba pokusit se splnit všechny co nejlépe. V následujících kapitolách se podíváme na jednotlivé požadavky a rozebereme si, jak bychom mohli dosáhnout co nejlepších kompromisů, tedy co nejlepšího výukového systému.

5.1 Požadavek na přenositelnost

V dnešní době, kdy každé druhé zařízení používá jiný operační systém, je důležité, aby simulátor byl spustitelný na co největším počtu takových zařízení.

Jednou cestou implementace multiplatformní aplikace je volba takového jazyka, který je možno kompilovat, popřípadě interpretovat na všech cílených platformách. Takových možností je více (C, python, java, ...), ale toto řešení by znamenalo starat se o podporu všech použitých knihoven, spravovat jednotlivé předkompilované balíčky pro všechny platformy aktuální, opravovat nebo obcházet vzniklé nekompatibility a podobně. Tato možnost je tedy velice časově náročná.

Zkusme se tedy na problém podívat z druhé strany. Nehledejme platformu, která různými způsoby podporuje všechna cílená zařízení. Hledejme platformu, která je podporovaná všemi cílenými zařízeními stejně. Touto platformou jsou webové aplikace. Tato možnost se přímo nabízí.

Stále se ale musíme rozhodnout nad webovou aplikací spuštěnou na serveru, nebo aplikací interpretovanou pomocí internetového prohlížeče klienta. Jelikož výukový simulátor by měl být jednoduchý, můžeme se rozhodnout zvolit aplikaci běžící na straně klienta. Tento přístup nás částečně omezí, například budeme mít omezený prostor pro uložená data. Odpadnou nám tím však starosti se správou serveru, případná potřeba pořídit serverů více při velkém zájmu o aplikaci. Další výhoda bude, že uživatel může teoreticky aplikaci ve webovém prohlížeči načíst a od té doby může pracovat offline.

Řešením požadavku přenositelnosti je tedy vytvoření simulátoru jako webové aplikace spouštěné na straně klienta.

5.2 Požadavek na jednoduchost

U jakékoliv výuky je vysvětlování nebo demonstrování vždy klíčové a velice těžké se rozhodnout, co všechno učit, vysvětlovat nebo demonstrovat, co by mělo být jako

vstupní znalost a co všechno by se mělo zanedbat a nechat na individuálním doučení. Naštěstí většina procesorů má velice podobnou vnitřní stavbu, která funguje stejným způsobem. Většinou se procesory odlišují počtem registrů, komplexností aritmeticko-logické jednotky, nebo bitovou šířkou sběrnice. Z těchto různých vlastností tedy můžeme vybrat prostě jedno řešení a prohlásit, že reálný procesor, na kterém si mohou studenti ověřit znalosti získané studiem výukové pomůcky, může mít například více či méně registrů než výukový simulátor.

Jiná situace ale nastane v případě funkčních bloků, které se v některých procesorech nachází a v některých ne. Pro příklad se můžeme zamyslet nad tím, zda by výukový procesor měl, nebo neměl mít barrel shifter, obvod umožňující provést bitový posun o jakýkoliv počet bitů během jednoho kroku. Při dnešní převaze procesorů architektury ARM, která tento obvod používá, by se mohlo zdát, že simulátor by měl jistě takovouto možnost nabídnout. Na druhou stranu se jedná o obvod, jenž nemají zdaleka všechny procesory a který se dá nahradit několika bitovými posuny pomocí ALU. Konečné rozhodnutí je tedy barrel shifter neimplementovat, zjednodušit tak architekturu výukového procesoru a při výuce se zmínit o existenci tohoto obvodu, jeho použití a výhodách.

Řešením požadavku jednoduchosti je tedy implementovat pouze obvody nutné k vysvětlení základních principů a další funkční bloky, které výuková pomůcka nemá, vysvětlit teoreticky. Popřípadě navrhnout mikroinstrukční sadu tak, aby bylo možné funkční blok odsimulovat na úrovni mikroinstrukcí.

5.3 Požadavek na použitelnost

Nevýhodou spousty výukových programů je, že software je příliš složitě ovladatelný a proto se složitě používá jak studentům, tak vyučujícím. Možná je to záměr, možná to má mladé programátory připravit na složitá IDE, se kterými se budou setkávat, možná se autoři těchto programů soustředili příliš na realistický vjem. Tímto problémem takřka netrpí výukový simulátor Johnny, o kterém jsem se zmiňoval v kapitole 4.4. Ten totiž nabízí pouze to, co je nutné. Všechny možnosti jsou rozprostřené rozumně po pracovním prostředí (modelu procesoru) místo toho, aby byly schované v nabídkových lištách. To pomáhá studentovi zapamatovat si, která akce procesoru dělá co. Dokonce člověk, který neví o procesoru nic, je schopen uhádnout, k čemu které tlačítko slouží. Tento přístup je velice vhodný, dá se však ještě více zdokonalit. V Johnny simulátoru totiž najdeme tlačítka uložená na korespondujících funkčních blocích. Co by bylo ještě názornější, je udělat z celého funkčního bloku jedno tlačítko, takže student nemá pocit, že kliká na tlačítka na funkčním bloku, ale že supluje řadič a řídicí sběrnici a přímo aktivuje jednotlivé bloky.

Řešením problému použitelnosti je tedy vytvořit grafické rozhraní s minimem tlačítek a responsivním schématem simulátoru.

5.4 Požadavek na všestrannost

Bylo by vhodné, aby byl simulátor využitelný k různým jak výukovým, tak demonstrativním účelům. To znamená, že musí být dostatečně jednoduchý pro demonstrativní účely a zároveň dostatečně složitý, aby se na něm daly vysvětlit všechny potřebné vlastnosti procesorů. Bohužel každý uživatel může mít trochu jiné účely a trochu jiné požadavky. Například pro extrémně jednoduchou demonstraci fungování procesoru stačí velice omezená instrukční sada, zatímco pro výuku obecného programování pomocí jednotlivých instrukcí nebo assembleru je potřeba instrukční sady mnohem mocnější.

Tento problém se dá vyřešit více instrukčními sadami, ale stále můžou někteří uživatelé narazit na to, že mu instrukční sada nebude pro jeho účely vyhovovat. Nejjednodušším způsobem, jak celý tento problém vyřešit je, nechat uživatele navrhnout si vlastní instrukční sadu. To bude velice jednoduché v momentě, kdy se rozhodneme udělat řadič procesoru mikroprogramový a necháme uživatele mikroprogramovou paměť přepisovat. Každý uživatel si tedy bude moci navrhnout instrukční sadu podle vlastních potřeb.

Řešením problému všestrannosti je tedy implementování mikroprogramového řadiče procesoru a možnost přepsání mikroprogramové paměti.

5.5 Požadavek na aplikovatelnost

Nejeden výukový simulátor má tu nevýhodu, že je pouze výukový simulátor. Studenti potom mají pocit, že to, co se učí, nemají kde využít. Proto se často vyučuje konkrétní reálný hardware a student místo toho, aby kurz opouštěl s dostatečně obecnými vlastnostmi ohledně dané látky, řekněme procesorů, si odnese informace pouze o jednom konkrétním procesoru, který byl v kurzu použit.

Simulátor sice díky vyřešení problému všestrannosti dovoluje simulovat více reálných procesorů a je tedy možnost pomocí simulátoru odsimulovat daný procesor, poté jej naprogramovat a ukázat studentům, že jejich program skutečně funguje i na reálném procesoru a je tedy možnost jejich program integrovat do nějakého zařízení. To však vyžaduje, aby mikroarchitektura a mikroinstrukční sada byly dostatečně komplexní.

Dále by bylo zajímavé umožnit i takový postup, že by si uživatel vytvořil hardwarovou implementaci simulátoru například pomocí programovatelného hradlového

pole. Samotný návrh hardwarové implementace simulátoru není součástí práce, ale pokud simulátor bude navržen tak, že hardwarová implementace bude možná, mohli bychom se jednou dočkat i jí v rámci jiné bakalářské či diplomové práce. Další výhoda takového návrhu bude, že student získá pocit, že simulace je na nižší úrovni, tedy že je blíže reálnému hardwarovému procesoru.

Řešením problému s reálnou aplikovatelností tedy je navrhnout simulátor tak, aby byl teoreticky hardwarově proveditelný. Dále je potřeba navrhnout dostatečně komplexní mikroinstrukční sadu a dovolit tak simulovat instrukční sady reálných procesorů.

5.6 Požadavek na modifikovatelnost

Přestože návrh procesoru počítá s různými požadavky uživatelů, mohla by se najít vlastnost, která by větší části uživatelů scházela. Z tohoto důvodu by bylo vhodné dát uživatelům možnost takovéto změny navrhovat. To by ale znamenalo, že by se někdo musel o software neustále starat a žádané vlastnosti doplňovat. Mnohem jednodušší je, nechat uživatele přímo nahlédnout do zdrojových kódů simulátoru, měnit je a sdílet mezi ostatními uživateli.

Řešením problému s modifikovatelností by tedy bylo zveřejnit software pod open-source licencí GNU GPL v.3 [6].

6 Konceptuální návrh výukové pomůcky

Dříve než se začne reálný simulátor programovat, je dobré mít alespoň hrubou představu o výsledném produktu. V této kapitole se pokusím nastínit, jaká rozhodnutí byla provedena, aby byly splněny požadavky na výukový simulátor popsané v kapitole 5. Konkrétně se podíváme na to, jakou architekturu bude mít výsledný simulovaný procesor, jaká bude mikroinstrukční sada, jaká bude instrukční sada a jakým způsobem se bude tvořit samotná aplikace.

6.1 Název výukové pomůcky

Název simulátoru je první věc, které si uživatel zpravidla všimne. Musí být lehce zapamatovatelný a musí popisovat hlavní účel a myšlenky simulátoru. Jelikož by měl název oslovit co nejvíce lidí, je vhodné použít jazyk, kterému rozumí co nejvíce potencionálních uživatelů. Tímto jazykem je angličtina.

Nejdříve je dobré vymyslet nějaká klíčová slova. Poté z nich vytvořit nějaký anagram nebo akronym. Pro náš simulátor by mohla přicházet v úvahu následující klíčová slova:

- Educative
- Simulator
- Processor
- Microprocessor
- Demonstrative
- ...

Šikovným poskládáním počátečních písmen některých z těchto slov získáme nedokonalý akronym **EDEM**, tedy **E**ducational **DE**monstrative **M**icroprocessor jako název simulovaného procesoru.

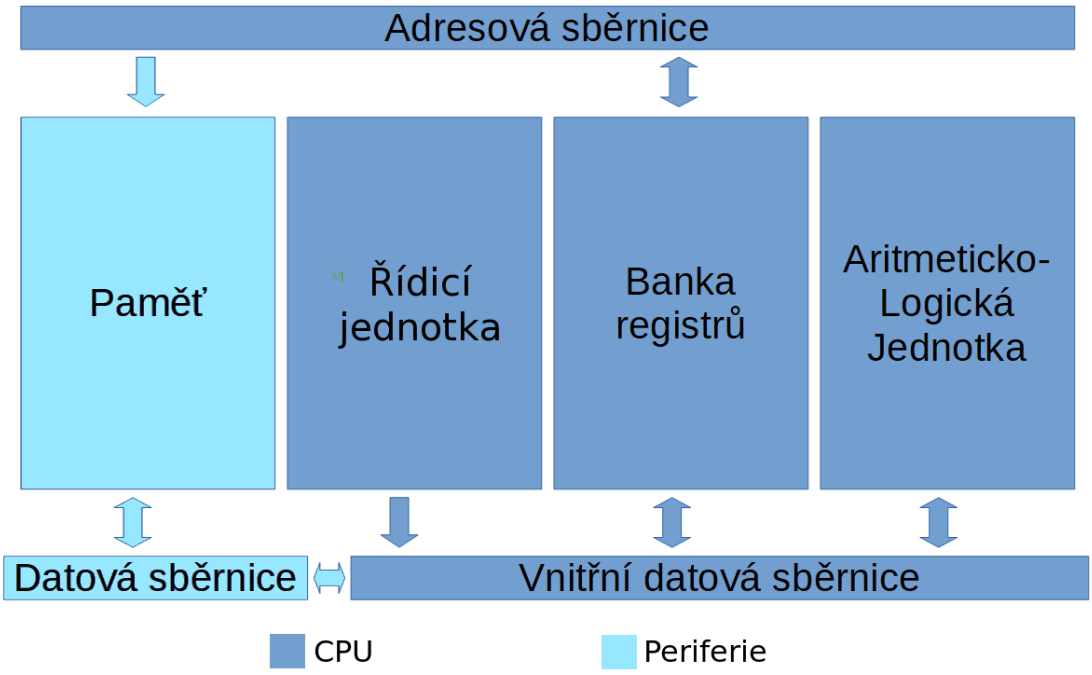
Chceme-li název pro simulátor, stačí přidat slovo „simulator“ na konec. Název simulátoru je **EDEMS**, tedy **E**ducational **DE**monstrative **M**icroprocessor **S**imulator.

6.2 Architektura procesoru

Jedním z nejdůležitějších rozhodnutí, které se týká procesoru, je jeho architektura. Na výběr máme mezi architekturou Harvardskou a Von Neumannovou, protože ostatní architektury se v dnešních procesorech a procesorech takřka nepoužívají. V předchozí kapitole bylo rozhodnuto, že procesor by měl mít mikroprogramový řadič. Tím jsme získali možnost demonstrovat něco velice podobného Harvardské architektuře, protože programovou paměť pro mikroprogram tvoří mikroprogramová paměť a datovou paměť zase paměť procesoru.

Pokud je tedy procesor na mikroprogramové úrovni postaven na Harvardské architektuře, měl by být procesor z důvodu demonstrace obou dvou zmíněných architektur postavený na architektuře Von Neumannově.

Nyní je možno vytvořit hrubý náčrt architektury, který je zobrazen na obrázku 6.1



Obr. 6.1: Obecný náčrt architektury simulátoru procesoru.

Na obrázku 6.1 můžeme vidět čtyři hlavní bloky procesoru. Těmi jsou řídicí jednotka, dále banka registrů a aritmeticko-logická jednotka. Na vnější datové sběrnici je poté umístěna programová paměť.

Zatímco chování procesoru je možné silně změnit zvolením jiné instrukční sady (přepsáním mikrokódu), architektura je nezměnitelná. Je tedy nutné ji vytvořit dostatečně silnou a obecnou.

6.2.1 Registry procesoru

Navržený procesor má celkem 16 registrů. Každý registr má své jméno, své číslo (adresu), kterým může být adresován. Některé registry mají zvláštní funkce. Registry jsou orientovány do párů, je tedy možné je používat jako registry šestnáctibitové. Obrázek číslo 6.2 zobrazuje jednotlivé registry, jejich orientaci do párů a jejich čísla.

Obr. 6.2: Přehled registrů a jejich adres.

Registry A, B, C, D, E, S, P

Tyto registry je možné použít jako registry pro obecné použití, popřípadě může být jejich funkce definována instrukční sadou

Registrové páry BC, DE, SP

Pouze tyto registrové páry je doporučeno používat jako 16b registry, přesto že je možné takto používat i jiné registrové páry (například registrový pár FA).

Registrový pár SP, jak název napovídá, je předpokládaný pro využití jako ukazatel na stack. Jeho využití je ale definováno instrukční sadou.

Registry TMP0, TMP1, TMP2

Registry pro obecné použití v mikrokódu. Registr TMP0 je zároveň jedním ze vstupů do ALU.

Registrový pár TMP1TMP2

Registrový pár pro obecné použití v mikrokódu.

Registry PCH, PCL

Registry tvořící registrový pár PC, tedy programového čítače.

Registry UPCL, UPCH

Registry tvořící registrový pár UPC, tedy mikroprogramového čítače. Mikroprogramové adresy jsou pouze 11b, takže 5 nejvyšších bitů UPCH není využito a není možné je modifikovat.

Registr F

Registr příznaků aritmeticko-logické jednotky. Jeho název je odvozen od anglického slova flags, kterým se označují jednotlivé bity stavového registru. Následující obrázek číslo 6.3 ukazuje jména jednotlivých bitů a jejich pořadí v registru příznaků.

X	Q	H	P	V	N	Z	C
F[7]	F[6]	F[5]	F[4]	F[3]	F[2]	F[1]	F[0]

Obr. 6.3: Přehled bitů registru F.

Registr příznaků se mezi procesory často liší pořadím jednotlivých stavových bitů. Funkce stavových bitů ale bývají často podobné. Například stavový bit přenosu do vyššího řádu, identifikátor nulového nebo negativního výsledku operace najdeme v každém procesoru. V procesoru EDEM můžeme nalézt následující bity:

- C - carry - přenos do vyššího řádu
- Z - zero - výsledná operace je nulová
- N - Negative - výsledná operace je záporné číslo
- V - Two's complement overflow - přetečení při převodu do druhého doplňku
- P - Parity - indikace liché parity
- H - Half carry - přenos do čtvrtého bitu (počítáno od nuly)
- Q - není ovlivněn ALU, použití je definováno instrukční sadou
- X - není ovlivněn ALU, použití je definováno instrukční sadou

Registr OP

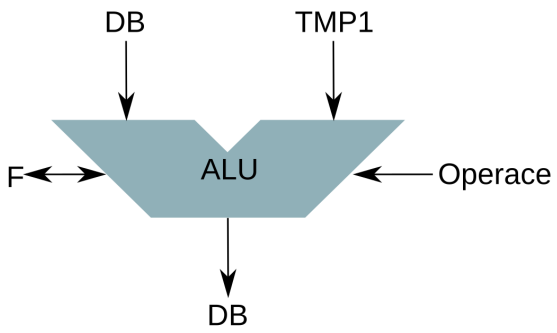
Registr OP je speciální registr sloužící pro nepřímé adresování registrů a operací aritmeticko-logické jednotky.

Když porovnáme většinu dnešních instrukčních sad, zjistíme, že některé instrukce jsou velice podobné. Zpravidla se liší cíleným registrem. Tudíž by bylo zbytečné psát mikrokód pro každou z těchto instrukcí. Je lepší zavést pojem mikrooperand, což je operand obsažený přímo v operačním kódu mikroinstrukce, který může ovlivnit její funkci jako operand běžné instrukce.

Příkladem může být instrukce načtení z paměti s operačním kódem 0b0000000000XX, kde poslední dva bity určují cílený registr (například B,C,D,E,..). Mikrokód instrukce poté přeloží poslední dva bity jako mikrooperand a pomocí něj adresuje cílený registr. Tedy například pokud je použita mikroinstrukce pro načtení z datové sběrnice do registru, která bude mít jako operand registr OP, použije se registr, jehož adresa je v registru mikrooperandu uložena.

Tato technika dovolí vytvořit více instrukcí se stejným mikrokódem, čímž zmenšíme potřebnou mikrokódovou paměť a velikost mikrokódu. Dále zjednoduší a zefektivní tvorbu výukových instrukčních sad.

6.2.2 Aritmeticko-logická jednotka procesoru



Obr. 6.4: Náčrt aritmeticko-logické jednoty, jejích vstupů a výstupů.

Aritmeticko-logická jednotka má u většiny procesorů tři vstupy a dva výstupy. Dva vstupy fungují jako operandy zvolené operace a třetí vstup jako volba operace. Výstup je výsledek operace. Druhým výstupem se ovlivňují bity stavového registru.

Tento návrh by měl tedy být respektován i ve výukovém procesoru. Pokud ale chceme umožnit operace, které mají jako operand bit přenosu do vyššího řádu ve stavovém registru (například rotace přes carry), musíme před jeden ze vstupů přidat multiplexor, který vybere, zda bude vstupem registr, nebo bit stavového registru.

Podporované operace navržené ALU tedy jsou:

0. ADD - Součet DB a TMP1
1. SUB - Rozdíl DB a TMP1
2. NEG - Dvojkový doplněk DB
3. NOT - Negace DB
4. AND - Bitový součin DB a TMP1
5. ORR - Bitový součet DB a TMP1
6. XOR - Bitová nonekvivalence DB a TMP1
7. SHR - Bitový posun vpravo
8. SHL - Bitový posun vpravo
9. ROR - Bitová rotace vpravo
10. ROL - Bitová rotace vlevo
11. RCR - Bitová rotace přes carry vpravo
12. RCL - Bitová rotace přes carry vlevo
13. ASR - Aritmetický posun vpravo
14. ASL - Aritmetický posun vlevo
15. BSR - BCD posun vpravo
16. BSL - BCD posun vlevo
17. EQU - Bitové porovnání DB a TMP
18. OOP - Operace definovaná registrem OP

Každá operace má svůj index (0-18), kterým se dá vybrat operace pomocí registru OP.

6.2.3 Řídicí jednotka procesoru

Řídicí jednotka obsahuje instrukční registr, dekodér a mikroprogramový řadič s mikroprogramovou pamětí.

Instrukční registr je osmibitový registr, který má po dobu provádění instrukce uložený její operační kód. Tohoto se využívá zejména u mikroinstrukce `COOP`, pomocí níž je vypočítán správný obsah pro registr OP.

Dekodér je schopný rozdělit operační slovo mikroinstrukce na její operační kód a argumenty. Následně provede operaci popsanou mikroinstrukcí. Vykonání každé mikroinstrukce trvá vždy pouze jeden instrukční cyklus.

6.2.4 Mikroinstrukce procesoru

Mikroinstrukce jsou 12b široká slova. Operační slovo se skládá z operačního kódu identifikujícího mikroinstrukci a jednoho až dvou operandů. Výsledné operační slovo vzniká součtem operačního kódu a operandu.

Například navržená mikroinstrukce `SVR`, která vymění obsah dvou registrů, má dva čtyřbitové operandy, které určují cílené registry. Chtěli-li bychom tedy vytvořit mikroinstrukci, která vymění obsah registrů `A` a `P`, musíme si zjistit tři informace. Operační kód instrukce `SVR`, který je 0x100, a dále čísla registrů A a P. Tato čísla jsou 4 a 7. Zapíšeme-li čísla 4 a 7 jako čtyřbitová čísla za sebe do hexadecimálního zápisu, získáme 0x47. Toto číslo následně přičteme k operačnímu kódu 0x100 a vyjde nám operační slovo 0x147, které odpovídá mikroinstrukci `SVR A P`. Tyto operace za nás ovšem obstará překladač.

Tabulky 6.2.4 a 6.2.4 obsahují kompletní seznam podporovaných mikroinstrukcí. Na obrázku 6.5 je poté zobrazena mapa operačních slov mikroinstrukcí.

Název	Operační kód	Operand	Funkce
DB <C	0x500	8b	Přesune konstantu operandu na datovou sběrnici.
DB <R	0x7C0	4b	Přesune hodnotu z registru definovaného operandem na datovou sběrnici.
AB <R	0x7B0	4b	Přesune hodnotu z registru definovaného operandem na spodních 8b adresové sběrnice. Horních 8b je vynulováno.
AB <W	0x7A0	4b	Přesune hodnotu z registrového páru definovaného operandem na adresovou sběrnici.
DB>R	0x790	4b	Přesune hodnotu z datové sběrnice do registru definovaného operandem.
AB>W	0x780	4b	Přesune hodnotu z adresové sběrnice do registrového páru definovaného operandem.
DB <O	0x7F0		Přesune obsah registru OP na datovou sběrnici.
DB>O	0x7F1		Přesune obsah datové sběrnice do registru OP.
SVR	0x100	4b, 4b	Vymění obsah registrů definovaných prvním a druhým operandem.
SVW	0x200	4b, 4b	Vymění obsah registrových párů definovaných prvním a druhým operandem.
ALU	0x000	5b	Alu provede operaci definovanou operandem. Například 0x01 je ADD, 0x02 je operace SUB, protože index této operace je 2.
SETB	0x300	4b, 4b	Nastavit bit definovaný prvním operandem v registru definovaném druhým operandem na 1.
RETB	0x400	4b, 4b	Nastavit bit definovaný prvním operandem v registru definovaném druhým operandem na 0.

Tab. 6.1: Tabulka mikroinstrukcí procesoru EDEM, 1. část

Název	Operační kód	Operand	Funkce
INCB	0x770	4b	Inkrementuje obsah registru definovaného operandem.
DECB	0x760	4b	Dekrementuje obsah registru definovaného operandem.
INCW	0x750	4b	Inkrementuje obsah registrového páru definovaného operandem.
DECW	0x740	4b	Dekrementuje obsah registrového páru definovaného operandem.
READ	0x7F4		Čtení z paměti.
WRT	0x7F5		Zápis do paměti.
JOI	0x730	4b	Přeskočí následující mikroinstrukci, pokud obsah registru definovaného operandem je 0x00.
JON	0x720	4b	Přeskočí následující mikroinstrukci, pokud obsah registru definovaného operandem není 0x00.
JOFI	0x710	4b	Přeskočí následující mikroinstrukci, pokud hodnota bitu F[operand] je 0b. OP se chová jako normální registr pro tuto mikroinstrukci.
JOFN	0x700	4b	Přeskočí následující mikroinstrukci, pokud hodnota bitu F[operand] není 0b. OP se chová jako normální registr pro tuto mikroinstrukci.
JMP	0x800	11b	Zapíše hodnotu operandu do UPC, tedy mikroprogramový skok.
COOP	0x600	8b	Naplní mikrooperandový registr správnou hodnotou. OP=IR-(operand).
END	0x7F2		Konec instrukce. Signál pro řadič k načtení další instrukce. Inkrementuje se programový čítač, jeho obsah je vystaven na adresovou sběrnici a do registru IR je načtena hodnota z programové paměti. Automaticky se provede skok mikroprogramu na adresu obsaženou v registru IR.

Tab. 6.2: Tabulka mikroinstrukcí procesoru EDEM, 2. část

Obr. 6.5: Mapa operačních slov mikroinstrukcí procesoru EDEM.

6.2.5 Paměť procesoru

Procesor EDEM obsahuje dva různé paměťové bloky. Prvním blokem je paměť mikroprogramová, která je ukryta v bloku řídicí jednotky. Mikroprogramová paměť je přímo adresována registrovým párem UPC, tedy mikroprogramovým čítačem. Protože tento registrový pár je celkem jedenáctibitový, můžeme pomocí něj adresovat

až 2048 slov. Jednotlivá slova v paměti jsou v tomto případě dvanáctibitová.

Druhým paměťovým blokem je paměť programová. Tato paměť je adresována šestnáctibitovou adresovou sběrnicí. Celý adresovatelný prostor je pro zjednodušení obsazen jednou pamětí typu RAM s osmibitovými slovy.

6.3 Instrukční sada procesoru

Plánované využití procesoru EDEM je takové, že si každý vyučující vytvoří vlastní instrukční sadu, aby mohl demonstrovat své učivo způsobem, na který je zvyklý. Přesto bude procesor dostupný se základní, minimální instrukční sadou, která slouží pro demonstraci vlastností procesoru.

Základní instrukční sada disponuje instrukcemi popsanými v tabulce 6.3

Popsaných instrukcí je celkem 14. Všimněme si ale, že téměř každá instrukce využívá mikrooperand pro umožnění provedení své operace na více registrech. Celkem tedy instrukční sada disponuje 97 instrukcemi.

Jméno	**Operand**	**Strojový kód**	**Funkce**
LD{reg}	16b addr	0x00 - 0x07	Načte hodnotu z paměti do registru. {reg} = MEMORY[16b addr]
ST{reg}	16b addr	0x08 - 0x0F	Uloží hodnotu z registru do paměti. MEMORY[16b addr] = {reg}
ADD{reg}	16b addr	0x10 - 0x17	Přičte hodnotu z paměti do registru. {reg} = {reg} + MEMORY[16b addr]
SUB{reg}	16b addr	0x41 - 0x48	Odečte hodnotu z paměti do registru. {reg} = {reg} - MEMORY[16b addr]
AND{reg}	16b addr	0x49 - 0x20	Provede logický součin s hodnotou z paměti a registru. {reg} = {reg} & MEMORY[16b addr]
OR{reg}	16b addr	0x51 - 0x58	Provede logický součet s hodnotou z paměti a registru. {reg} = {reg} | MEMORY[16b addr]
XOR{reg}	16b addr	0x59 - 0x60	Provede logickou nonekvivalenci s hodnotou z paměti a registru. {reg} = {reg} ^ MEMORY[16b addr]
INC{reg}		0x18 - 0x1f	Inkrementuje hodnotu registru. {reg} = {reg} + 1
INC{reg*}		0x20 - 0x23	Inkrementuje hodnotu registrového páru. {reg}{reg+4} = {reg}{reg+4} + 1
DEC{reg}		0x24 - 0x2b	Dekrementuje hodnotu registru. {reg} = {reg} - 1
DEC{reg*}		0x2c - 0x2f	Dekrementuje hodnotu registrového páru.{reg}{reg+4} = {reg}{reg+4} - 1
JP	16b addr	0x30	Provede nepodmíněný skok na adresu.
JPF{flag}	16b addr	0x31 - 0x38	Provede skok na adresu, pokud je hodnota uvedeného bitu stavového registru rovna nule.
JP{reg}	16b addr	0x39 - 0x40	Provede skok na adresu, pokud je hodnota uvedeného registru rovna nule.

Tab. 6.3: Minimální instrukční sada procesoru EDEM

{reg} značí název registru (F,A,B,C,...), {reg*} značí název registrového páru (BC,DE,...), {flag} značí název bitu stavového registru (C, Z, N, ...)

7 Návrh aplikace a vnitřních stavů

Dříve než se začne s vývojem samotného simulátoru procesoru, je třeba učinit několik rozhodnutí ohledně struktury aplikace, programovacího jazyka a případných knihoven. V neposlední řadě je také potřeba zhodnotit způsob ukládání vnitřních stavů procesoru. Popis a důvody těchto rozhodnutí jsou v následujících podkapitolách.

7.1 Výběr programovacího jazyka

Ač by se mohlo zdát, že aplikace spouštěné v prohlížeči se dají psát jen v jazyku javascript, není tomu tak. V této kapitole si kromě jazyku javascript i další možnosti. [16]

7.1.1 Programovací jazyk javascript

Javascript je programovací jazyk, pomocí kterého je možné dynamicky vytvářet a měnit obsah webových stránek mimo jiné pomocí DOM (Document Object Model) API a vytvářet tak webové aplikace.

Ač historie tohoto jazyku zasahuje až do druhé poloviny devadesátých let minulého století, nejedná se tedy o žádný nový jazyk, je javascript velmi často spojován se zvláštním chováním, způsobeným zejména velice dynamickým typováním. Jazyk se stále vyvíjí a tyto chyby jsou opravovány. Bohužel novinky jsou prohlížeči, moduly a transpilery (software kompilující jazyk javascript do a z jiných jazyků) podporovány pouze postupně. Například nejpoužívanější transpiler používaný pro kompresi kódu, UglifyJS 3 zatím nepodporuje ani standard ECMAScript 2015 i přesto, že aktuálním standardem je již ECMAScript 2017. [26]

Na druhou stranu je javascript jazyk, jehož interpret má každý ve svém osobním počítači a dokonce i v mobilním telefonu, a to od momentu, kdy zařízení pořídil. Proto je tento jazyk dnes používaný na takřka každé internetové stránce a pokud si dá vývojář pozor na některé nástrahy, může být javascript nejlepší řešení. [5]

7.1.2 Ostatní programovací jazyky s využitím transpilace

Další možností, jak psát webové aplikace, je napsat je v jiném jazyce a následně transpilovat do jazyka javascript. Vznikají transpilery ze skriptovacích jazyků, jako jsou Python (Pyjamas, Skulpt), Perl (Perlito) nebo Ruby (Opal, RubyJS). Dále je možné transpilovat i staticky typované, kompilované jazyky jako Java (GWT, Java2Script) nebo C# (Script#). Dokonce vznikají nové jazyky, které přidávají do jazyka javascript nové vlastnosti jako statické typování (TypeScript, Flow), nebo lepší syntaxi (CoffeeScript). [16]

Mohlo by se zdát, že transpilace například z jazyka TypeScript by velice zjednodušila práci díky zavedení statických proměnných, díky kterým bychom se vyhnuli dvojsmyslným kódům. Příkladem takového kódu je nedostatečné ošetření vstupní proměnné, která vede na součet dvou proměnných jiných typů. Musíme ale myslet na to, že transpilovaný kód bude opět pouze javascript a odstraňování chyb nebo optimalizace kódu je tedy mnohem složitější, jelikož je potřeba rozumět kódu jak před transpilací, tak po ní.

Dalších výhod transpilace, jako je například automatická optimalizace kódu, je navíc možné dosáhnout i u programu napsaném v programovacím jazyku javascript pomocí javascript-javascript transpilerů.

Pro simulátor jsem tedy jako hlavní programovací jazyk zvolil javascript, který může každý rozšiřovat jakýmkoli programovacím jazykem transpilovaným do jazyka javascript.

7.1.3 Programovací jazyk webassembly

Další možností, která přichází v úvahu, je velice nový programovací jazyk WebAssembly. Jedná se o jazyk binárních instrukcí, do nějž je možné kompilovat z jazyků C, C++ a Rust. Výsledný kód je oproti jazyku javascript mnohem menší, takže je rychleji stažen a spuštěn, a rychlejší, protože nemusí probíhat ani interpretace, ani kompilace za běhu jako u jazyku javascript.

Přestože jde o velice nový projekt, je podporovaný všemi nejpoužívanějšími webovými prohlížeči, jako jsou Mozilla Firefox, Google Chrome, Safari nebo Microsoft Edge.

Tento jazyk ale nepodporuje, ani nebude podporovat úpravy webových stránek pomocí DOM API. WebAssembly se totiž nesnaží nahradit javascript, jeho účelem je vylepšit jej možností vytváření rychlejšího a efektivnějšího kódu. [27]

Pro tvorbu simulátoru by tedy bylo potřeba využít obou programovacích jazyků. Protože ani s jedním jazykem nemám velké zkušenosti a simulátor nemusí simulovat procesor v reálném čase, rozhodl jsem se nevyužít programovací jazyk WebAssembly.

7.2 Výběr modulů z npm

V minulé kapitole jsme si určili, že aplikace bude psaná v jazyce javascript. Nyní se podíváme, jaké moduly a balíčky dostupné pomocí Node Package Manager (NPM), správce a repozitář balíků pro Node (virtuální stroj pro spouštění programů napsaných v programovacím jazyku javascript), budou použity v programu simulátoru.

7.2.1 Moduly pro vývoj

Vývoj aplikace se dá zjednodušit a zefektivnit použitím modulů. Největším problémem při psaní aplikací v jazyku javascript je fakt, že prohlížeče nepodporují možnost používání modulů. Je tedy nutné použít některý z balíků, který umožní použití modulů z npm a poskytne možnost zpřehlednit kód rozdělením do modulů.

- Browserify - balík umožňující používání modulů, který kompiluje všechny moduly do jednoho čitelného zdrojového souboru.
- Browserify-watchify - modul pro browserify, jenž spustí překlad při každém uložení změn do některého z modulů.
- Browserify-notify - modul pro browserify, který vytváří upozornění na probíhající operaci browserify a její výsledek.
- Jest - jednoduchý balík umožňující vytváření unit testů.

7.2.2 Moduly aplikace

Aplikace samotná by měla být pokud možno nezávislá na kódu třetích stran. To ale není důvod vytvářet něco, co už někdo vytvořil mnohem lépe. Moduly používané v aplikaci byly vybírány podle počtu aktivních uživatelů, aktivity autorů a velikosti modulu.

- JQuery - modul zjednodušující a zefektivňující práci s HTML/DOM elementy, manipulací s HTML a mnoho dalšího.
- ace - modul umožňující jednoduše vytvořit editory textu se zvýrazněním syntaxe. V programu byla použita verze modulu ace zvaná brace, která je kompatibilní s modulem browserify.
- clusterize.js - modul umožňující efektivní umístění velkého množství objektů na jednu webovou stránku tak, že jsou renderovány pouze viditelné objekty.

7.3 Datový typ proměnných v aplikaci

Javascript má pouze tři základní datové typy. String, Number a Boolean. Jelikož v celém procesoru se operuje s osmi, maximálně šestnácti bitovými čísly, můžeme si dovolit používat převážně proměnnou typu Number.

7.3.1 Reprezentace registrů a sběrnice

Proměnná typu number se v rozmezí čísel -9007199254740991 až 9007199254740991 chová jako celé číslo a pokud číslo přeteče do vyšších hodnot, je proměnná typu number uložena do paměti jako číslo s plovoucí desetinnou čárkou. Pokud tedy zamezíme

přetečení osmibitového nebo šestnáctibitového čísla, je použití tohoto datového typu vhodné. [14]

Protože by ale bylo příliš pracné a neefektivní při každé operaci s číslem zajišťovat upravení proměnné typu number na správnou hodnotu, vytvořil jsem třídu nazvanou BinNumber. Tato třída umožňuje vytvoření n-bitové proměnné, která může být provázána s jinou a vytvořit tak pár. Této vlastnosti je využito hlavně u registrových párů. Metody této třídy jsou:

- constructor() - konstruktor třídy
- val() - setter, který je schopný nastavit hodnotu proměnné z daného vstupu. Podporuje hodnoty jak číselné, tak v podobě řetězce reprezentujícího číslo v desítkové, dvojkové a šestnáctkové soustavě.
- valPair() - setter nastavující hodnotu proměnné v páru.
- hex() - getter, jehož návratová hodnota je hodnota třídy v šestnáctkové soustavě.
- hexPair() - getter, jehož návratová hodnota je hodnota páru v šestnáctkové soustavě.
- bin() - getter, jehož návratová hodnota je hodnota třídy ve dvojkové soustavě.
- binPair() - getter, jehož návratová hodnota je hodnota páru ve dvojkové soustavě.
- dec() - getter, jehož návratová hodnota je hodnota třídy v desítkové soustavě.
- decPair() - getter, jehož návratová hodnota je hodnota páru v desítkové soustavě.
- incr() - metoda pro inkrementaci proměnné
- incrPair() - metoda pro inkrementaci páru
- decr() - metoda pro dekrementaci proměnné
- decrPair() - metoda pro dekrementaci páru
- setBit() - metoda pro nastavení konkrétního bitu proměnné na 1
- resBit() - metoda pro nastavení konkrétního bitu proměnné na 0

7.3.2 Reprezentace paměti

Paměti, jako mikroinstrukční a programová, jsou pouze orientované seznamy osmibitových hodnot. Proto by se nabízelo použití seznamu proměnných typu number. Tento datový typ je ale příliš paměťově náročný na to, aby se proměnné uložily do LocalStorage, úložného prostoru, do kterého nám dovolí webový prohlížeč zapisovat. Velikost LocalStorage se liší, ale v době psaní aplikace (2018) se nejmenší LocalStorage (2MB) vyskytovalo u prohlížeče AndroidBrowser pro Android 4.3. [15]

Aplikace tedy v rámci podporování co největšího počtu zařízení nemůže do LocalStorage ukládat více než 2MB dat.

Paměti budou většinu času zobrazovány v šestnáctkové soustavě. Proto budou v LocalStorage uloženy jako řetězec čárkou oddělených hodnot v šestnáctkovém formátu, které budou během běhu programu překládány na dočasné proměnné typu number (například pro načtení proměnné z adresy v paměti na datovou sběrnici).

7.3.3 Reprezentace aritmeticko-logické jednotky

Aritmeticko-logickou jednotku je možné napsat jako funkci nebo metodu, jenž má jako vstupní proměnné adresy na proměnné, se kterými se operuje (F, DB, TMP1) a index operace, která bude provedena (ADD, SUB, NEG,...)

Jelikož vstupní proměnné F a DB jsou zároveň výstupními proměnnými, může být výstupní proměnnou informace o chybě při operaci funkce.

7.4 Struktura aplikace

Aplikaci jako takovou můžeme velice snadno rozdělit na dvě hlavní sekce, dva různé stavební bloky aplikace.

Sekce grafického rozhraní a sekce simulátoru procesoru. Struktura aplikace je taková, že sekce grafického rozhraní je přímo závislá na sekci simulátoru. Sekce simulátoru ale funguje i bez sekce grafického rozhraní. Je tak možné vytvořit pro simulátor jiné grafické rozhraní (například s použitím node, interpreta jazyka javascript, a knihovny electron vytvořit multiplatformní desktopovou aplikaci[25]), popřípadě jiný ovládací prvek, například rozhraní pro příkazovou řádku.

7.4.1 Sekce grafického rozhraní

Grafické rozhraní je ze svého principu událostmi řízená aplikace. Producent událostí je webová stránka napsaná pomocí html a css. Události jsou odposlouchávány zejména pomocí knihovny jquery. Grafické rozhraní tedy slouží jako transportní kanál. Zpracovatel události je poté sekce simulátoru.[7]

7.4.2 Sekce simulátoru

Sekce simulátoru je aplikace vytvořená podle objektově orientovaného paradigmatu. Skládá se z jednotlivých objektů, modulů, z nichž každý zapouzdřuje konkrétní funkce, kterou simulátor poskytuje. Máme tedy následující moduly:

- binNumber - definice nového datového typu
- globals - objekt obsahující všechny stavy procesoru
- alu - objekt umožňující provádění operací aritmeticko-logické jednotky
- controlUnit - objekt umožňující provádění operací řadiče

- clock - objekt umožňující provádění operací generátoru hodinových cyklů
- memoryCompiler - objekt překládající jazyk symbolických instrukcí kódu simulátoru do strojového kódu
- microcodeCompiler - objekt překládající jazyk symbolických instrukcí mikrokódu simulátoru do strojového kódu

7.5 Grafické rozhraní aplikace

Grafické rozhraní aplikace přímo odpovídá blokovému diagramu procesoru. Je navržené tak, aby bylo co nejpřehlednější (jednotlivé funkční bloky jsou barevně odděleny) a uživatelsky nejpřívětivější.

Krom funkčních bloků procesoru jsou na stránce pouze tři další bloky, a to blok pro zaslání zpětné vazby, odkaz na dokumentaci a blok pro změnu módu grafického rozhraní. Tyto módy jsou dva. První je mód „advanced“, tedy pokročilejší, kompletní, zobrazení celého procesoru, které slouží pro demonstraci funkce celého procesoru na úrovni mikroarchitektury.

Obr. 7.1: Grafické rozhraní simulátoru EDEMS advanced.

Druhý mód je nazvaný „basic“, tedy zjednodušené zobrazení, které skryje řídicí jednotku a registry využívané pouze mikrokódem. Toto zobrazení se více podobá pohledu na reálný procesor například při ladění pomocí pokročilejšího IDE.

Obr. 7.2: Grafické rozhraní simulátoru EDEMS basic.

Na obrázcích 7.1 a 7.2 si můžeme prohlédnout grafické rozhraní obou módů. Kromě funkčních bloků uvedených výše zde přibyl ještě jeden, a to blok generátoru hodinových cyklů. Tento blok umožňuje měnit frekvenci hodinových cyklů, spustit a zastavit generování hodinových cyklů a dále krokovat po mikroinstrukcích a po instrukcích.

Většina mikroinstrukcí má v grafickém rozhraní svůj funkční prvek, nebo své tlačítko. Nejlépe viditelnými jsou například zvýrazněné šipky vedoucí z a do bloku programové paměti. Intuitivně šipka směrem do paměti provede zápis, mikroinstrukci `WRT`, a šipka směrem z paměti provede čtení, tedy mikroinstrukci `RD`. Operandy mikroinstrukcí, například výběr registru se provedou pouhým kliknutím na cílený registr, čímž se grafický blok registru, popřípadě registrového páru, zvýrazní (na obou obrázcích je zvýrazněn registr A).

Dále můžeme vidět, že oba paměťové bloky disponují editorem, který umožňuje jednoduše měnit jejich obsah. O tomto více v kapitole 7.6.

Každý editor dále disponuje možností exportování, popřípadě importování kódu. Dále je možno paměť smazat (naplnit nulami) a přeložit obsah editoru a výsledek zapsat do paměti.

7.6 Programovací jazyk v simulátoru

Jelikož je třeba nabídnout uživateli jednoduchou možnost, jak měnit obsah pamětí (programové i mikroprogramové), grafické rozhraní disponuje textovým editorem. Tento editor umožňuje vytvořit textovou interpretaci dat, syntakticky podobnou jazyku symbolických adres, která budou následně přeložena do strojového kódu a zapsána do paměti.

Velikým rozdílem od jazyka symbolických adres je absence právě oněch symbolických adres, po kterých je jazyk pojmenován. Toto zjednodušení vede ke složitějšímu tvoření kódu, ale jasnější demonstraci funkce procesoru. Zavedení volitelných symbolických adres je jedno z prvních vylepšení, které je pro simulátor EDEMS plánováno.

Po překladu programu je zobrazen ladicí pohled, ve kterém je vidět listing a data jsou zapsána do paměti. Text v ladicím okně není možné editovat. Vždy je zvýrazněn řádek obsahující kód v paměti, na který právě ukazuje programový čítač (popřípadě mikroprogramový čítač pro mikroprogramový listing). Tímto způsobem je možné trasovat program i mikroprogram.

Podporovaná klíčová slova zjednodušeného jazyka assembler používaného simulátorem EDEMS se u jednotlivých editorů (mikroprogramového a programového), mírně liší a budou probrána v dalších kapitolách. Ostatní podporované prvky jsou ale pro oba překladače stejné.

Zápis konstant je podporován jak v desítkovém zápisu (číslo zapsané bez prefixu, tedy například číslo 42 je zapsáno ve tvaru `42`) a šestnáctkovém zápisu (číslo je zapsáno s prefixem `0x`, tedy číslo 42 je zapsáno ve tvaru `0x2a`), tak v zápisu binárním (číslo je zapsáno s prefixem `0b`, tedy číslo 42 je zapsáno ve tvaru `0b101010`). U příliš velkých čísel dojde k jejich přetečení do daného rozsahu.

Dále oba překladače podporují komentáře. Komentář začíná znakem „`;`“ a končí novým řádkem.

7.6.1 Zjednodušený assembler pro mikroprogram

Zjednodušený assembler pro mikroprogram slouží k popisu mikroprogramu a definici instrukcí. Klíčová slova přímo odpovídají názvu mikroinstrukcí. Nalezneme tu tedy například klíčová slova RD, WRT, DB>R, INC, DEC, COOP a END.

Jelikož je potřeba přímo adresovat jednotlivé registry, disponuje tento jazyk klíčovými slovy adresujícími registry (A,B,PCH, PCL,...), což zjednodušuje čitelnost kódu. Tato klíčová slova se mohou vyskytnout pouze jako operand mikroinstrukcí. Klíčové slovo vždy odpovídá názvu registru.

Dalšími klíčovými slovy jsou možné operandy mikroinstrukce ALU, tedy operace aritmeticko-logické jednotky. Tato klíčová slova přímo odpovídají názvům jednotlivých operací, tedy například ADD, SUB, AND, EQU a OOP.

Posledním podporovaným klíčovým slovem je pseudoinstrukce .DEF (tečka značí, že se nejedná o instrukci, ale o pseudoinstrukci). Tato tříoperandová pseudoinstrukce slouží k definování nové instrukce. Prvním operandem je adresa první mikroinstrukce definované instrukce. Celá pseudoinstrukce je poté přeložena jako skok na onu mikroinstrukci. Dalším operandem je název instrukce a posledním operandem je šířka operandu definované instrukce. Definice instrukce LDA (načtení hodnoty z paměti do registru A) je definována pseudoinstrukcí `.DEF 0x61 LDA 2B`, protože operand je dvoubajtový a první mikroinstrukce leží na adrese `0x61`.

7.6.2 Zjednodušený assembler pro program

Zjednodušený assembler pro program slouží k popisu programu, který bude procesor při spuštění vykonávat. Všechny instrukce definované pomocí zjednodušeného jazyka assembler pro mikroprogram jsou přístupné jako klíčová slova.

Dalším klíčovým slovem je pseudoinstrukce .CONST, pomocí které můžeme na dané místo v paměti uložit konstantu uvedenou jako operand pseudoinstrukce.

Posledním klíčovým slovem je pseudoinstrukce .ORG, kterou nastavíme odsazení následujících instrukcí.

Spojením pseudoinstrukcí .CONST a .ORG tedy můžeme například definovat proměnné na konkrétních adresách. Chceme-li například definovat proměnnou na adrese `0x20` s hodnotou 42, napíšeme následující kód.

main.asm

```
.org 0x20                                                    1
.const 42                                                    2
```

8 Návrh výukových úloh

V následujících kapitolách se podíváme na výukové úlohy navržené pro demonstraci možností simulátoru EDEMS a pro seznámení uživatele se základy programování procesorů.

Úlohy předpokládají teoretickou znalost logických obvodů a základních funkčních bloků procesoru.

Každá navržená úloha má 3 části. První částí je zadání, které by mělo být studentovi předloženo na začátku hodiny. Další částí jsou tipy, jimiž může vyučující postupně pomáhat studentovi ke zdárnému a správnému vyřešení. Poslední částí je samotné řešení, se kterým se může řešení studenta porovnat. Je třeba mít na paměti, že studentovo řešení nemusí být stejné. I přesto je možné toto řešení považovat za správné.

8.1 Úloha první, návrh instrukce

Cílem úlohy je seznámit uživatele s architekturou procesoru, s grafickým rozhraním a s ovládáním procesoru.

Po vypracování úlohy by měl student rozumět tomu, co jsou registry a jaké jsou jejich speciální funkce (jako například mikroprogramový čítač, programový čítač, registr příznaků, registr mikrooperandu).

Dále by měl být student schopen simulátor ovládat a to jak manuálně (pomocí tlačítek mikroinstrukcí), tak krokováním (pomocí bloku generátoru hodinových cyklů).

Student by měl po vypracování úlohy rozumět rozdílu mezi instrukcemi a mikroinstrukcemi.

8.1.1 Zadání první úlohy

Založte si proměnnou uloženou na adrese `0x20`.

S pomocí manuálu vytvořte mikrokód implementující jedinou instrukci, která načte proměnnou z adresy uvedené jako operand a uloží její negaci do registru `A`. Definujte její název jako „LDNA“. Instrukce má jeden operand o šířce 16b. Endianita operandu je little-endian.

Máte-li dostatek času, pokuste se implementovat instrukce `LDNB 16b`, `LDNC 16b`, `LDND 16b`, `LDNE 16b`, `LDNS 16b`, `LDNP 16b` tak, aby všechny instrukce měly společný mikrokód.

Spočítejte počet hodinových cyklů potřebných pro provedení instrukce. Odpovídá počet potřebný počet cyklů u simulátoru EDEMS počtu cyklů, který potřebuje reálný procesor? Jaké optimalizace využívá reálný procesor?

8.1.2 Tipy k první úloze

Následující tipy by měly studenta vést ke zdárnému vyřešení úlohy.

- Vytvořte instrukci `NOP`, která nemá operandy a nic nedělá.
- Vytvořte instrukci `NOP 16b`, která má jeden operand a stále nic nedělá.
- Založte novou proměnnou na adrese, která nekoliduje s výsledným binárním kódem programu.
- Negujte hodnotu registru pouze pomocí grafického rozhraní aplikace.
- Negujte hodnotu mikrokódem.
- Načtěte hodnotu do registru pouze pomocí grafického rozhraní aplikace.
- Načtěte hodnotu do registru mikrokódem.
- Sjednoťte kód.
- Pokuste se kód optimalizovat tak, aby zabral pokud možno co nejméně mikroinstrukcí.
- Nastudujte si mikrooperand a mikroinstrukci `COOP`

8.1.3 Řešení první úlohy

Jenodušší verze zadání implementující jedinou instrukci `LDNA 16b`.

Výpis 8.1: Programová část řešení úlohy „Návrh instrukce“

```
ldna 0x0020
.org 0x0020
.const 0x1
```

Výpis 8.2: Mikroprogramová část řešení úlohy „Návrh instrukce“

```
.DEF 0x1 LDNA 2B

; load low address
INCW PCH
AB<W PCH
RD
DB>R TMP2
; load high address
INCW PCH
AB<W PCH
RD
```

```
DB>R TMP1                                                    12
; load value to register                                    13
AB<W TMP1                                                    14
RD                                                           15
; create a negation of the value                            16
ALU NOT                                                      17
DB>R A                                                       18
END                                                          19
```

Složitější verze zadání implementující instrukce `LDNA 16b`, `LDNB 16b`, `LDNC 16b`, `LDND 16b`, `LDNE 16b`, `LDNS 16b`, `LDNP 16b` sdílející jediný mikrokód.

Výpis 8.3: Programová část řešení složitější verze úlohy „Návrh instrukce“

```
ldnb 0x0020                                                  1
ldna 0x0021                                                  2
ldnd 0x0022                                                  3
.org 0x0020                                                  4
.const 0x1                                                   5
.const 0x42                                                  6
.const 0x88                                                  7
```

Výpis 8.4: Mikroprogramová část řešení složitější verze úlohy „Návrh instrukce“

```
.DEF 0x7 LDNB 2B                                             1
.DEF 0x7 LDND 2B                                             2
.DEF 0x7 LDNS 2B                                             3
.DEF 0x7 LDNA 2B                                             4
.DEF 0x7 LDNC 2B                                             5
.DEF 0x7 LDNE 2B                                             6
.DEF 0x7 LDNP 2B                                             7
                                                             8
; count right COOP value                                    9
COOP 0xff                                                    10
; load low address                                          11
INCW PCH                                                     12
AB<W PCH                                                     13
RD                                                           14
DB>R TMP2                                                    15
; load high address                                         16
INCW PCH                                                     17
AB<W PCH                                                     18
RD                                                           19
DB>R TMP1                                                    20
```

```
; load value to register
AB<W TMP1
RD
ALU NOT
DB>R OP
END
```

8.2 Úloha druhá, paměťově mapované periferie

Cílem úlohy je seznámit uživatele se základními postupy vytváření programů v programovacím jazyku assembler.

V minulé úloze jsme poznali architekturu procesoru, jeho funkční bloky a jejich funkce. V této úloze si vytvoříme náš první program.

8.2.1 Zadání druhé úlohy

Ve většině programovacích jazyků bývá prvním programem program s názvem „ahoj, světe“, který vypíše na obrazovku text „ahoj, světe“ a skončí. Mikrokontroléry obrazovky nemají, a tak je standardním postupem blikání LED diodou připojenou na vstupně-výstupní bránu. Simulátor EDEMS ale žádné vstupně výstupní brány nemá, a proto místo blikání LED diodou budeme pouze měnit konkrétní proměnnou.

Představme si, že jednotlivé bity uložené na adrese `0x20` přímo odpovídají stavům vstupně-výstupní brány mikrokontroléru (1 odpovídá logické 1 a 0 logické 0). Napište program, který bude v nekonečném cyklu měnit hodnotu všech bodů vstupně-výstupní brány z 0 na 1 a zase zpět. Jakým způsobem by bylo vhodné připojit LED diody k pinům mikrokontroléru?

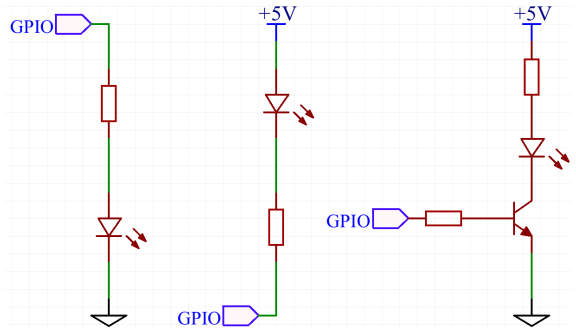
8.2.2 Tipy k druhé úloze

- Seznamte se se všemi instrukcemi, které procesor EDEMS nabízí.
- Použijte krokování po hodinových cyklech, abyste zjistili, jak která instrukce funguje, jaké registry ovlivní a kolik hodinových cyklů je potřeba k jejímu provedení.
- Zamyslete se nad strukturou programu.
- Založte proměnné.
- Založte konstanty.
- Napište program.
- Otestujte správnou funkčnost.
- Optimalizujte program.

8.2.3 Řešení druhé úlohy

Výpis 8.5: Programová část řešení úlohy „Ahoj, světe“

```
;variable definitions                                    1
.org 0x20                                                2
.const 0xa ; output    = 0x20                            3
.const 0x0 ; 0         = 0x21                            4
.const 0xff; 127       = 0x22                            5
                                                         6
;code definition for Hello World                        7
.org 0x0                                                 8
lda 0x21                                                 9
ldb 0x22                                                 10
                                                         11
sta 0x20                                                 12
stb 0x20                                                 13
jp 0x06                                                  14
```



Obr. 8.1: Různá zapojení LED diody na výstupní pin mikrokontroléru.

8.3 Úloha třetí, výpočet rovnice

Cílem úlohy je seznámit uživatele s analýzou zadání a následným přepisem do jednotlivých instrukcí.

Student si v této úloze musí poradit s nedokonalostí instrukční sady a vymyslet implementaci násobení pomocí jiných instrukcí. Dále si student musí poradit s nedostatečným počtem registrů, což je častý problém programování procesorů.

8.3.1 Zadání třetí úlohy

Založte si proměnné `X`, `A`, `B`, `C`, `D` s libovolným obsahem. Pomocí základní instrukční sady napište program, který vypočítá rovnici 8.1 a výsledek uloží na adresu `0x35`

$$X = 4 \cdot (A + (D + B - C) - D) + (D - 4) \tag{8.1}$$

Instrukce skoku nesmí být použita.

Máte-li dostatek času, upravte program tak, aby obsah registrů `B`, `C`, `D`, `E`, `S` a `P` nebyl programem ovlivněn.

8.3.2 Tipy ke třetí úloze

- Nebojte se používat poznámky.
- Začněte vytvořením proměnných.
- Analyzujte rovnici a pokuste se ji zjednodušit.
- Napište program, který vypočítá výsledek první závorky.
- Zamyslete se, odkud můžeme do programu nahrát konstantu 4.
- Napište program, který vypočítá výsledek druhé závorky.
- Vymyslete, jak se dá násobit bez skoků a bitových posunů.
- Sjednoťte kód
- Pokuste se změnit kód tak, abyste použili pouze registr `A`

8.3.3 Řešení třetí úlohy

Výpis 8.6: Programová část řešení úlohy „Výpočet rovnice“

```
;variable definitions                                    1
.org 0x30                                                2
.const 1    ; A       0x30                               3
.const 1    ; B       0x31                               4
.const 5    ; C       0x32                               5
.const 1    ; D       0x33                               6
.const 4    ; 4       0x34                               7
.const 0xff ; result  0x35                               8
                                                         9
;code definition 4*(A+(D+B-C)-D)+(D-4)                   10
                                                         11
.org 0x0                                                 12
;B-C                                                     13
lda 0x31                                                 14
suba 0x32                                                15
```

```
                                                                    16
;(A+(D+B-C)-D)                                                      17
adda 0x30                                                           18
                                                                    19
;4*(A+(D+B-C)-D)                                                    20
sta 0x35                                                            21
adda 0x35                                                           22
adda 0x35                                                           23
adda 0x35                                                           24
sta 0x35                                                            25
                                                                    26
;4*(A+(D+B-C)-D)+(D-4)                                              27
lda 0x33                                                            28
suba 0x34                                                           29
adda 0x35                                                           30
sta 0x35                                                            31
```

8.4 Úloha čtvrtá, instrukce skoku

Cílem této úlohy je seznámit uživatele s instrukcí podmíněných a nepodmíněných skoků.

Student je postaven před problém přepsání programu z programovacího jazyka C do programovacího jazyka Assembler, což je situace, se kterou se může student setkat při optimalizaci programů. Řešení úlohy vyžaduje analýzu programu napsaného v jazyce C a podporuje přemýšlení nad kódem v jazyce C jako nad zjednodušeným zápisem strojového kódu.

Student se také naučí se nejdříve nad problémem zamyslet a analyzovat jej s pomocí papíru a tužky, a teprve poté programovat jeho řešení.

8.4.1 Zadání čtvrté úlohy

V jazyce C je napsán program pro počítání lichých číslic. Program počítá čísla od nuly do hodnoty proměnné `upTo` (hodnota proměnné `upTo` není kontrolována). Výstupní proměnnou je proměnná `counter`. Kód v jazyce C vypadá následovně:

Výpis 8.7: Zadání úlohy „Instrukce skoku“

```
const uint8_t upTo = 10;
uint8_t counter = 0;

for(uint8_t i=0; i<upTo; ++i){
  if(i%2 == 1)
    counter++;
}
```

Napište program, který bude mít stejnou funkci. Zajistěte, aby po skončení programu nepokračoval procesor v provádění nenaprogramované části paměti. Program co nejvíce optimalizujte na rychlost.

8.4.2 Tipy ke čtvrté úloze

- Nejdříve program pořádně analyzujte. Zjistěte, co můžete v assembleru přímo naprogramovat.
- Instrukční sada procesoru EDEMS nám dovoluje podmíněně skákat jen pro hodnotu proměnné rovné 0. Jak můžeme změnit `for` smyčku tak, abychom tohoto využili?
- Instrukční sada procesoru EDEMS postrádá instrukci odpovídající klíčovému slovu `for`. Jak můžeme program změnit, abychom se tomuto vyhnuli?
- Instrukční sada procesoru EDEMS nemá instrukci pro výpočet zbytku po dělení (operátor `%`). Můžeme místo toho použít jinou instrukci?
- Pokuste se program v jazyce C přepsat podle předešlých tipů. Váš program by mohl být podobný následujícímu kódu:

Výpis 8.8: První přepis zadání úlohy „Instrukce skoku“

```
const uint8_t upTo = 10;
uint8_t counter = 0;

uint8_t i = upTo;
do{
  i--;
  uint8_t tmp = i & 1;
  if(tmp != 0)
    counter++;
} while (i != 0);
```

- Instrukční sada nedovoluje ani smyčku `do, while`. Jakého klíčového slova v jazyce C bychom mohli využít?

- Místo klíčového slova `do, while` použijte klíčové slovo `goto`.
- Podmíněně skákat můžeme, pouze pokud je proměnná rovna nule. Jak můžeme tuto skutečnost obejít?
- Přepište program v jazyce C podle předešlých tipů. Váš program by mohl připomínat následující kód:

Výpis 8.9: Druhý přepis zadání úlohy „Instrukce skoku“

```
const uint8_t upTo = 10;
uint8_t counter = 0;
uint8_t i = upTo;

loop:
    i--;
    uint8_t tmp = i & 1;
    if (tmp == 0)
      goto notIncrement;
    counter++;
notIncrement:

    if (i == 0)
      goto notLoop;
    goto loop;
notLoop:
```

- Přepište program z programovacího jazyka C do programovacího jazyka assembler.
- Zajistěte, aby procesor nezpracovával nenaprogramovanou část paměti.
- Optimalizujte program. Jaký je rozdíl mezi optimalizací na rychlost a optimalizací na velikost kódu?

8.4.3 Řešení čtvrté úlohy

Výpis 8.10: Programová část řešení úlohy „Instrukce skoku“

```
;variable definitions
.org 0x30
.const 0xa ; upTo      = 0x30
.const 0x0 ; counter   = 0x31
.const 0xff; tmp       = 0x32
.const 0x1 ; 1         = 0x33

;code definition for odd counter
.org 0x0
lda 0x31
ldc 0x30

;i--
decc

;tmp = i&1
stc 0x32
ldb 0x32
andb 0x33

;if(tmp == 0) goto notIncrement
jpb 0x14

;counter++
inca

;if (i==0) goto notLoop
jpc 0x1a
jp 0x06

;save counter to its variable
sta 0x31

;jump to itself
jp 0x1D
```

8.5 Úloha pátá, PID regulátor

Cílem úlohy je seznámit studenta s časováním. Student si musí poradit s požadavkem na přesné vzorkování vstupní proměnné.

8.5.1 Zadání páté úlohy

Naprogramujte PSD regulátor, jehož parametry jsou $K_p = 1$, $K_s = 1$, $K_d = 1$, $\Delta t = 1s$, $W = 1$. Vstupní proměnná Y je na adrese `0x51`, výstupní proměnná X je na adrese `0x52`. Perioda hodinových cyklů je $2ms$.

Jakou vlastnost, kterou bychom mohli dosáhnout přesné vzorkování, procesor EDEMS postrádá? Můžeme se bez ní obejít?

8.5.2 Tipy k páté úloze

- Odvoďte algoritmus diskrétního regulátoru.

$$x\left(t\right) = \left(K_p \cdot e\left(t\right) \, + \, K_s \cdot \int_0^t e\left(t\right) dt \, + \, K_d \cdot \frac{d}{dt} e(t)\right)$$

- Zkuste se zbavit operací, pro které procesor EDEMS nemá instrukci. (Není potřeba se zbavovat násobení ani dělení, protože všude se násobí a dělí konstantou 1.)

$$x\left(t\right) = \left(e\left(t\right) \, + \, \int_0^t e\left(t\right) \, + \, \left(e(t) - e(t{-}1)\right)\right)$$

- Jak zajistíme vzorkovací periodu 1s bez přerušení? (Bez přerušení se můžeme obejít pomocí zpoždění hlavní smyčky.)
- Přepište algoritmus do imperativního programovacího jazyka (například do programovacího jazyka C).

Výpis 8.11: Přepis zadání úlohy „PID regulátor“

```
volatile uint8_t w;
volatile uint8_t y;
uint8_t e;
uint8_t S = 0;
uint8_t D = 0;
uint8_t e_old = 0;
uint8_t X;

while(1){
  e=w - y;
  S=S + e * dt;
  D=(e - e_old) / dt;
```

```
    X=(Kp * e) + (Ki * I) + (Kd*D);                    13
    e_old = e;                                         14
                                                       15
    delay(1000-whileLoopTime)                          16
}                                                      17
```

- Nahraďte smyčku `while` klíčovým slovem `goto`
- Napište program v jazyce Assembler
- Vypočítejte hodnotu proměnné `whileLoopTime`

Příklad výpočtu hodnoty proměnné whileLoopTime

Předpokládejme, že naše řešení se shoduje s uvedeným řešením. Spočítejme tedy počet mikroinstrukcí, které jsou potřeba pro provedení smyčky.

Nejkratší bude smyčka, když hodnota proměnné `delay` bude 1. V tom případě se delay smyčka provede pouze jednou a na návěští `delay_over` se skočí při první příležitosti. Je tedy třeba provést následující instrukce:

- 1x instrukci DEC $(1 \cdot 4 = 4)$
- 15x instrukci LD nebo ST $(15 \cdot 14 = 210)$
- 5x instrukci ADD nebo SUB $(5 \cdot 17 = 85)$
- 2x instrukci JP $(2 \cdot 6 = 12)$
- 1x instrukci JPreg (delší varianta) $(1 \cdot 15 = 15)$

Nyní sečteme počet hodinových cyklů, který tato smyčka zabere. Zjistíme, že cyklus trvá 326 hodinových cyklů. Pokud je perioda hodinových cyklů $2ms$, hlavní smyčka zabere $652ms$. Abychom zajistili vzorkovací periodu $1s$, musíme hlavní smyčku zdržet o $348ms$, neboli 174 hodinových cyklů.

Pro provedení jedné zpožďovací smyčky je potřeba:

- 1x instrukci DEC $(1 \cdot 4 = 4)$
- 1x instrukci JPreg (kratší varianta) $(1 \cdot 6 = 6)$
- 1x instrukci JP$(1 \cdot 12 = 12)$

Provedení zpožďovací smyčky tedy zabere 22 hodinových cyklů. Potřebujeme-li hlavní smyčku zpozdit o 174 hodinových cyklů, necháme zpožďovací smyčku proběhnout 8x. Tím dosáhneme zpoždění o $22 \cdot 8 = 176$ hodinových cyklů.

Z předchozích výpočtů plyne, že pokud bude hodnota proměnné `delay` rovna 8, bude hlavní smyčka probíhat $1,004s \doteq 1s$.

8.5.3 Řešení páté úlohy

Výpis 8.12: Programová část řešení úlohy „PID regulátor“

```
;;variable definitions                                    1
.org 0x50                                                 2
.const 0x01 ; W       = 0x50                              3
.const 0x00 ; Y       = 0x51                              4
.const 0x00 ; X       = 0x52                              5
.const 0x00 ; TMP     = 0x53                              6
.const 0x00 ; e_old   = 0x54                              7
.const 0x08 ; delay   = 0x55                              8
                                                          9
;;code definition for PID controller                     10
.org 0x0                                                  11
ldb 0x54 ; S = 0                                          12
ldc 0x54 ; D = 0                                          13
lde 0x54 ; e_old = 0                                      14
                                                          15
;loop:                    ;0x09                          16
;;e=w-y                                                   17
ldd 0x50                                                  18
subd 0x51                                                 19
                                                          20
;;I=(I+e)*dt                                              21
std 0x53                                                  22
addb 0x53                                                 23
                                                          24
;;D=(e-e_old)/dt; e_old = e                               25
ldc 0x53                                                  26
lde 0x53                                                  27
ste 0x53                                                  28
subc 0x53                                                 29
                                                          30
;;X = Kp*e + Ki*I + Kd*D                                  31
std 0x53                                                  32
lda 0x53                                                  33
stb 0x53                                                  34
adda 0x53                                                 35
stc 0x53                                                  36
adda 0x53                                                 37
sta 0x52                                                  38
```

```
                                                                    39
;;delay loop                                                        40
lda 0x55                                                            41
;delay:                    ;0x39                                    42
deca                                                                43
jpa 0x40                  ;to delay_over                            44
jp 0x39                   ;to delay                                 45
                                                                    46
;delay_over:               ;0x40                                    47
ste 0x54                                                            48
;;Jump to loop                                                      49
jp 0x09                   ;to loop                                  50
```

9 Zhodnocení a další vývoj simulátoru EDEMS

Simulátor EDEMS je aktuálně připraven pro použití ve výuce jako pomůcka pro vysvětlení základů mikroprocesorových technologií. Z důvodu požadavku na jednoduchost pomůcky nemůže simulátor EDEMS nahradit reálný mikrokontrolér a reálné vývojové prostředí, které je třeba neopomenout. Výrazně ale pomůže při vysvětlování funkce mikrokontrolérů, toku dat a problematiky programování v jazyce assembler.

Dá se očekávat, že vyučujícím nebo studentům bude některá vlastnost simulátoru připadat nedokonalá, nebo bude chybět úplně. Z tohoto důvodu je přímo v simulátoru možnost kontaktovat autora.

Již během tvorby programu jsem dostal zpětnou vazbu od vyučujících kurzu BMIC (mikroprocesorová technika) z VUT, na kterou jsem zareagoval.

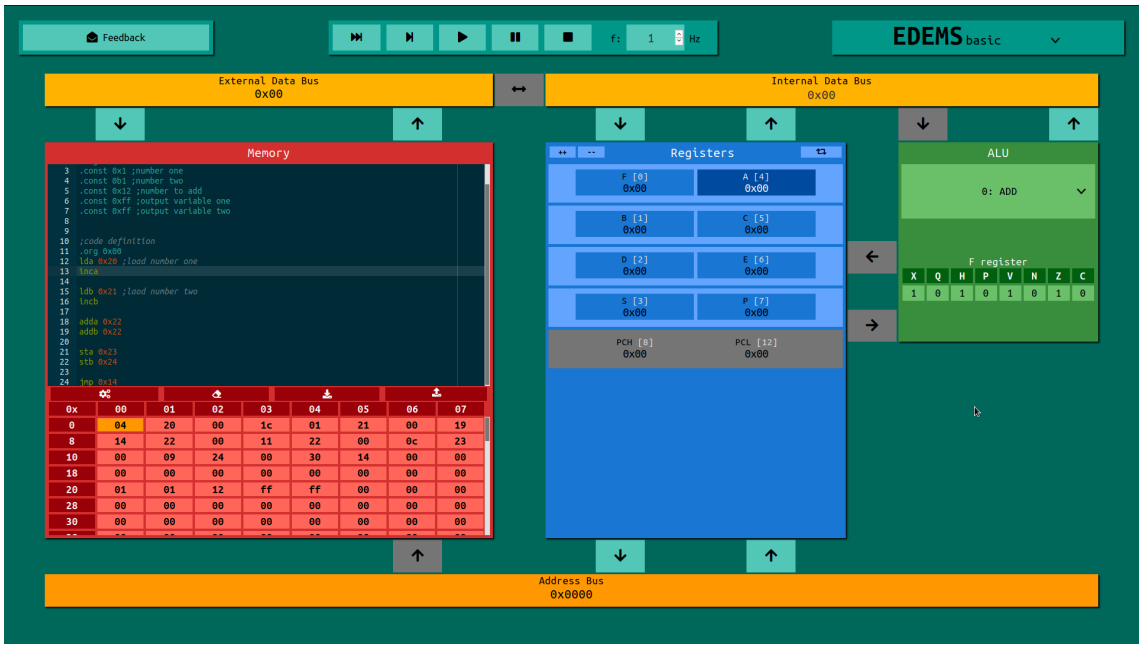
Nejlépe jsou změny vidět na grafickém rozhraní aplikace před 9.1 a po 9.2 zpětné vazbě. Můžeme si povšimnout, že sběrnice byly přemístěny tak, že adresová sběrnice je nyní u horního okraje blokového diagramu, jak to bývá u blokových diagramů procesorů zvykem.

Další velkou změnou je možnost zobrazit ladicí pohled na paměť a listing. Tato změna umožnila zvětšit pole pro editor a velmi zpřehlednila trasování kódu.

Následně se změnila barva pozadí šipek zobrazujících tok dat tak, aby nevyvolávaly pocit, že se jedná o deaktivovaná tlačítka mikroinstrukcí. Možnost používat mikroinstrukce v módu basic byla odebrána úplně.

Do bloku aritmeticko-logické jednotky byl přidán náhled na operandy. Dále přibylo tlačítko odkazující na dokumentaci. Přibylo také zobrazení vysvětlení funkce při setrvání kurzoru na funkčním bloku.

Celé grafické rozhraní se těmito změnami zjednodušilo a zpřehlednilo. Předpokládám, že se během používání programu ve výuce na VUT objeví více požadavků, na které bude třeba dále reagovat.



Obr. 9.1: Grafické rozhraní simulátoru EDEMS basic před zpětnou vazbou.

Obr. 9.2: Grafické rozhraní simulátoru EDEMS basic po zpětné vazbě.

10 Závěr

Cílem této práce bylo navrhnout a realizovat interaktivní výukový simulátor počítačového systému. Tohoto výsledku bylo zdárně dosaženo dokonce na takové úrovni, že výsledek této práce vyhrál druhé místo studentské konference EEICT.

Tomu ovšem předcházelo několik velice důležitých kroků. V kapitole číslo 1 jsme si ujasnili, jaký je rozdíl mezi procesorem, mikroprocesorem, mikrokontrolérem a systémem na čipu. To bylo velice důležité, jelikož tyto pojmy jsou v práci hojně používány.

V následující kapitole jsme si postupně prošli všechny funkční bloky procesoru. Podívali jsme se na obecné vlastnosti aritmeticko-logické jednotky (2.1.1) a seznámili jsme se s různými registry (2.1.2), které se v procesorech často objevují. Nakonec je popsán sekvenční i mikroprogramový řadič (2.1.3) a sběrnice (2.1.4).

Protože procesor není jediný funkční blok, který je nutný pro vytvoření jednoduchého počítačového systému, bylo dalším krokem seznámení s pamětmi (2.2) a vstupně-výstupní bránou (2.3).

V kapitole Existující mikroprocesory a mikrokontroléry (3) jsme se seznámili s některými reálnými mikroprocesory a mikrokontroléry, které by mohly být vhodné k použití ve výuce. Postupně jsme si představili mikroprocesor Intel 8080A, rodinu mikrokontrolérů NXP S08 a rodinu mikrokontrolérů s procesory architektury ARM cortex-M0. Zjistili jsme, že mikroprocesor 8080A je příliš zastaralý (3.1.2), rodina S08 má nejistou budoucnost (3.2.1) a mikrokontroléry s procesory architektury ARM cortex-M0 jsou zbytečně složité pro výuku základů mikroprocesorové techniky (3.3.1). Navíc všechny procesory trpí nepřehledným zobrazením dat v IDE při ladění, a tak se mi zdá vhodné základy mikroprocesorové techniky vyučovat pomocí jiné výukové pomůcky.

Z tohoto důvodu byla provedena rešerše existujících výukových pomůcek a projektů, které vznikly pro účel demonstrace fungování procesorů. Výsledky průzkumu jsou uvedeny v kapitole 4, nazvané Existující výukové pomůcky. Byly zde představeny projekty Megaprocessor (4.1) a The Visual6502 (4.2), které jsou sice dobré demonstrační pomůcky, ale nejsou vhodné pro výuku na vysoké škole. Důvody nevhodnosti jsou uvedeny v kapitolách 4.1.1 a 4.2.1. Dále zde byl uveden projekt Magic-1 Homebrew CPU (4.3), který není do výuky vhodný z důvodů popsaných v kapitole 4.3.2. Poslední a nejlepší představenou výukovou pomůckou je Johnny simulator (4.4). Tento projekt je příkladem dobrého výukového simulátoru obecného procesoru. Bohužel i zde najdeme výrazné nedostatky. Například pohled na aritmeticko-logickou jednotku je až příliš zjednodušený. Proto ani tento projekt není dostatečně vhodnou výukovou pomůckou.

Provedení rešerše existujících procesorů, výukových i demonstračních pomůcek

a zhodnocení jejich výhod a nevýhod, umožnilo definovat šest požadavků, které by měla splňovat ideální výuková pomůcka pro výuku mikroprocesorové techniky. Tyto požadavky byly popsány v kapitole Požadavky na výukový systém (5). Definované požadavky jsou přenositelnost (5.1), jednoduchost (5.2), použitelnost (5.3), všestrannost (5.4), aplikovatelnost (5.5) a modifikovatelnost (5.6). V jednotlivých kapitolách jsou popsány detaily požadavků a vlastnost, která z požadavku plyne a bude implementována do vytvořené pomůcky.

V následující kapitole Konceptuální návrh výukové pomůcky (6) je proveden konceptuální návrh výukové pomůcky. Byl definován název EDEMS, tedy educational demonstrative microprocessor simulator, jako jednoduchý, výstižný a zapamatovatelný název. Dále byla definována architektura procesoru, který bude simulován. Tuto architekturu je možné prozkoumat na obrázku 6.1. Pro tento procesor byly také definovány mikroinstrukční (6.2.4) a instrukční (6.3) sady.

V kapitole 7 (Návrh aplikace a vnitřních stavů) je popsán proces, který vedl ke zdárnému vytvoření výukového simulátoru počítačového systému. Je zde popsán výběr programovacího jazyka (7.1) a výběr použitých modulů (7.2) jazyka javascript. Dále je zde popsána reprezentace vnitřních stavů jednotlivých komponent procesoru (7.3) a struktura celé výsledné aplikace (7.4). V kapitole 7.5 je definováno grafické rozhraní, které je možné si prohlédnout na obrázcích 7.1 a 7.2. Poslední této kapitoly je definice zjednodušeného jazyka assembler (7.6) pro program i mikroprogram.

Pro navržený a vytvořený simulátor byly následně vytvořené výukové úlohy, které studenta seznámí jak s vlastnostmi a funkcí obecného procesoru, tak s programováním v jazyce assembler. Jednotlivé úlohy na sebe logicky navazují a v každé úloze si student osvojí novou znalost. V první úloze, popsané v kapitole 8.1 se student seznámí s procesorem, jeho funkcí a jeho periferiemi. Následně si navrhne vlastní instrukci pomocí mikroinstrukční sady. V druhé úloze (8.2) si student vytvoří první jednoduchý program v jazyce assembler. Třetí úloha (8.3) studenta seznámí s omezeními, které jednodušší procesory mají a naučí jej, jak je obejít. Čtvrtá úloha (8.4) naučí studenta, jak přepsat kód z programovacího jazyka C do programovacího jazyka assembler. V páté úloze (8.5) musí student využít všech znalostí z předešlých úloh k vytvoření velice jednoduchého PID regulátoru. Tato úloha seznámí studenta s pozicí mikrokontrolérů v oblasti regulační techniky. Ke všem úlohám je uvedeno řešení i tipy, kterými se dá pomoci studentům dojít ke správnému řešení.

V poslední kapitole, která je nazvaná Zhodnocení a další vývoj simulátoru EDEMS (9), byly zhodnoceny výsledku projektu a navržen další vývoj simulátoru. Je zde uvedena skutečnost, že simulátor byl již představen vyučujícím kurzu BMIC, pro které je tato pomůcka hlavně vytvořena. Jsou zde také popsány změny, které byly provedeny po obdržení zpětné vazby. Tyto změny byly převážně kosmetické, přesto vedly k výraznému vylepšení simulátoru. Stav simulátoru před a po úpravách je možné

porovnat na obrázcích 9.1 a 9.2.

Jelikož se jedná o projekt spadající pod otevřenou licenci, je možné jej najít ve veřejném repozitáři na adrese (`github.com/frimdo/EDEMS`). Z toho plyne, že simulátor i navržený procesor je možné dále vylepšovat.

Jako první vylepšení se nabízí implementace symbolických adres a s ní podpora návěští. Tato funkce totiž v simulátoru z důvodu demonstrace práce s adresami není. Další vylepšení, kterým by se dal simulátor zdokonalit je vytvoření virtuálních periferií, se kterými by mohl procesor pracovat. Příkladem mohou být vstupně-výstupní bloky, čítače, časovače a generátory PWM signálů.

Jelikož navržený procesor splňuje podmínku aplikovatelnosti (5.5), je možné simulátor implementovat do hradlového pole a vytvořit tak další velice názornou výukovou pomůcku, pomocí které si student ověří funkci svého kódu na reálném hardwaru. Pro takto vytvořenou součástku se nabízí umožnit uživateli programování v jiném jazyce než je assembler, například pomocí cross-compileru LCC vytvořit podporu jazyka C. Všechna tato vylepšení jsou nad rámec této práce. Proto si myslím, že by bylo vhodné se jim věnovat v rámci dalších závěrečných prací.

Výsledkem mojí práce je výuková pomůcka, která bude jistě užitečná při výuce na UAMT VUT. Tento simulátor počítačového systému pomůže nejen při demonstraci funkce a toku dat uvnitř obecného procesoru, ale také připraví studenta na programování v jazyce assembler.

Literatura

[1] Knuth E. Donald: Umění programování 1.díl - Základní algoritmy, Computer press 2008, ISBN: 978-80-2-1-2025-5

[2] Dvořák V., Drábek V.: Architektura procesorů, Vutium 1999, ISBN: 80-214-1458-8

[3] VALÁŠEK, P. *Monolitické mikroprocesory a mikropočítače*. 1989th ed. 1989. ISBN 9788003000562.

[4] LIČEV, L. *Procesory: architektura, funkce, použití: kompletní průvodce procesory a souvisejícími hardwarovými komponentami*. 1999th ed. 1999. ISBN 9788072261727.

[5] EISENMENGER, Richard. *JavaScript: kompletní kapesní průvodce*. Praha: Grada, 1999. ISBN 80-7169-383-9.

[6] GNU Operating System. *GNU General Public License* [Online] 2007. https://www.gnu.org/licenses/gpl-3.0.en.html (accessed May 15, 2018).

[7] Paykin, J.; et al. The Essence of Event-Driven Programming [online]; niversity of Pennsylvania, Philadelphia, USA, pp 1–2. https://www.cl.cam.ac.uk/ nk480/essence-of-events.pdf (accessed April 08, 2018).

[8] TIŠNOVSKÝ, P. Pohled do nitra mikroprocesoru. *root.cz* [online]. 20.3.2008 [cited 2018-01-01]. Available from https://www.root.cz/clanky/pohled-do-nitra-mikroprocesoru/ .

[9] TIŠNOVSKÝ, P. Od logických obvodů k mikroprocesorům. *root.cz* [online]. 20.3.2008 [cited 2018-01-01]. Available from https://www.root.cz/clanky/od-logickych-obvodu-k-mikroprocesorum/ .

[10] TIŠNOVSKÝ, P. Jak pracuje počítač?. *root.cz* [online]. 20.3.2008 [cited 2018-01-01]. Available from https://www.root.cz/clanky/jak-pracuje-pocitac/ .

[11] TIŠNOVSKÝ, P. Činnost mikroprocesoru, aneb jde to i bez trpaslíků. *root.cz* [online]. 20.3.2008 [cited 2018-01-01]. Available from https://www.root.cz/clanky/cinnost-mikroprocesoru-aneb-jde-to-i-bez-trpasliku/ .

[12] TIŠNOVSKÝ, P. Architektura mikrořadičů s jádry ARM Cortex-M0 a ARM Cortex-M0+. *root.cz* [online]. 2015 [cited 2018-05-15]. Available from https://www.root.cz/clanky/architektura-mikroradicu-s-jadry-arm-cortex-m0-a-arm-cortex-m0/ .

[13] UTKARSHBHATT12 Number. *MDN Web Docs* [online]. 2017 [cited 2018-01-01]. Available from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number .

[14] Number. *MDN web docs* [online]. 2017 [cited 2018-03-30]. Available from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number .

[15] VIEIRA, L. HTML5 Local Storage. *Sitepoint* [online]. 2015 [cited 2018-03-30]. Available from https://www.sitepoint.com/html5-local-storage-revisited/ .

[16] List of languages that compile to JS. *The CoffeeScript Wiki* [online]. 2018 [cited 2018-03-30]. Available from https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS .

[17] JRT BBC micro:bit přichází. *http://robodoupe.cz* [online]. 2017 [cited 2018-05-15]. Available from http://robodoupe.cz/2017/microbit-prichazi/ .

[18] PASTORIBOT Freescale 68HC08. *Wikipedie* [online]. 2014 [cited 2018-01-01]. Available from https://cs.wikipedia.org/wiki/Freescale_68HC08 .

[19] COFFEE Arm Cortex-M. *Wikipedie* [online]. 2017 [cited 2018-01-01]. Available from https://en.wikipedia.org/wiki/ARM_Cortex-M#Cortex-M0 .

[20] NEWMAN, J. Home. *The Megaprocessor ....* [online]. 2016 [cited 2018-01-01]. Available from http://www.megaprocessor.com/index.html .

[21] BROOKS, D. Home. *Homebrew CPU* [online]. 2016 [cited 2018-01-01]. Available from http://www.homebrewcpu.com/ . - visual 6502: http://www.visual6502.org/JSSim/index.html

[22] Visual Transistor-level Simulation of the 6502 CPU. *visual6502* [online]. 2011 [cited 2018-01-01]. Available from http://visual6502.org/ .

[23] DAUSCHER, P. *Johnny 1.00* [online]. 2012th ed. GPL, 10.7.2012 [cited 01 Jan 2018].

[24] NXP S08FL: 8-bit Cost-Effective FL16/8 MCUs. *NXP* [online]. [cited 2018-01-01]. Available from https://www.nxp.com/products/processors-and-microcontrollers/additional-processors-and-mcus/8-16-bit-mcus/8-bit-s08-mcus/8-bit-cost-effective-fl16-8-mcus:S08FL .

[25] Electron Documentation. *Electron* [online]. [cited 2018-04-08]. Available from https://electronjs.org/docs/tutorial/about .

[26] UglifyJS 3. *UglifyJS 3* [online]. 2018 [cited 2018-03-30]. Available from https://github.com/mishoo/UglifyJS2 .

[27] WebAssembly High-Level Goals. *WebAssembly Docs* [online]. [cited 2018-03-30]. Available from http://webassembly.org/docs/high-level-goals/ .

[28] Features. *MicroBit* [Online]. http://microbit.org/guide/features/ (accessed March 30, 2018).

Seznam symbolů, veličin a zkratek

ALU	Aritmeticko-logická jednotka – arithmetic-logic unit
CPU	Procesor – central processing unit
ROM	Paměť pouze pro čtení – read only memory
RAM	Paměť pro čtení i zápis – random access memory
IO	Vstup/Výstup – input output
FPU	Matematický koprocesor pro výpočty s plovoucí čárkou – floating-point unit
DB	Datová sběrnice – data bus
AB	Adresová sběrnice – address bus
GUI	Grafické rozhraní – graphical user interface
BCD	Dvojkově reprezentované dekadické číslo – binary coded decimal
IDE	Vývojové prostředí – integrated development environment
BMIC	Bakalářský kurs mikroprocesorové techniky na UAMT VUT
DOM	objektově orientovaná reprezentace XML nebo HTML dokumentu – document object model
API	rozhraní pro programování aplikací – application programming interface
NPM	správce a repozitář balíků pro node (virtuální stroj pro spouštění programů napsaných v programovacím jazyku javascript) – node package manager

Seznam příloh

A Obsah přiloženého CD

Aplikace byla vyvíjena pomocí verzovacího systému git. Díky tomu vznikly dvě různé větve aplikace. První větví je větev EDEMS. V této větvi byla aplikace vyvíjena. Obsahuje všechny zdrojové kódy jak aplikace samotné, tak modulů pro vývoj i aplikaci samotnou.

Druhou větví je větev EDEMS_RELEASE, která, jak název napovídá, obsahuje pouze tu část aplikace, která je potřebná k nasazení aplikace na reálný server.

- / ..... kořenový adresář přiloženého CD
  - EDEMS ..... vývojová větev aplikace EDEMS
    - documents ..... dodatečná dokumentace
    - EDEMS ..... aplikace EDEMS
      - img ..... obrázky aplikace
      - js ..... zdrojové soubory v jazyce javascript
        - ace ..... zdrojové soubory pro mudul ace
        - browser ..... zdrojové soubory grafického rozhraní
      - manual ..... zdroje manuálové stránky
      - node_modules ..... zdrojové soubory modulů
      - style ..... zdrojové soubory kaskádových stylů
      - edems.js ..... transpilované zdrojové soubory v jazyce javascript
      - package.json ..... soubor pro správu modulů
      - index.html ..... zdrojový soubor aplikace
      - style.css ..... zdrojový soubor kaskádových stylů
    - README.md ..... dokumentace aplikace
  - EDEMS_RELEASE ..... release větev aplikace EDEMS
    - EDEMS ..... aplikace EDEMS
      - img ..... obrázky aplikace
      - manual ..... zdroje manuálové stránky
      - node_modules ..... zdrojové soubory pro modul clusterize
      - style ..... zdrojové soubory kaskádových stylů
      - edems.js ..... transpilované zdrojové soubory v jazyce javascript
      - index.html ..... zdrojový soubor aplikace
      - style.css ..... zdrojový soubor kaskádových stylů
    - README.md ..... dokumentace aplikace
  - tex ..... zdrojové soubory textu práce
  - thesis.pdf ..... práce v elektronickém formátu