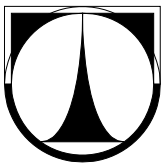
TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií



BAKALÁŘSKÁ PRÁCE

Liberec 2010 **Martin Tichovský**

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: B2612 Elektrotechnika a informatika
Studijní obor: Informatika a logistika

Webový simulátor školního µP pro předmět Číslicové počítače

Web simulator of school µP for subjekt CIP

Bakalářská práce

Autor: **Martin Tichovský**
Vedoucí práce: Ing. Martin Vlasák
Konzultant: Ing. Tomáš Martinec, Ph.D.

V Liberci 10. 5. 2010

Prohlášení

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

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

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

Datum

Podpis

Poděkování

Rád bych poděkoval panu Ing. Martinovi Vlasákovi, který se mnou měl trpělivost při zpracování bakalářské práce, jež se neplánovaně protáhla, a také bych chtěl poděkovat své rodině, která mě podporovala v nejtěžších dnech mého života.

Acknowledgement

I would like to offer thanks mr. Ing. Martin Vlasák for very patience with me and with making this work, which was unplanned enlarge, and so would like to offer thanks my family, which was supported me in the hardest days of my life.

Abstrakt

Cílem práce je nabídnout studentům interaktivní program, v našem případě webovou aplikaci, která bude sloužit pro studijní účely předmětu Číslicové počítače. Vytvořená webová stránka zahrnující JavaScriptové a PHP skripty, má být plnohodnotnou náhradou simulátoru vytvořeného pro studijní účely Ing. Martinem Vlasákem a Ing. Tomášem Martincem, Ph.D. Webový simulátor používá jednoduchý fiktivní mikroprocesor pro to, aby naučil studenty základním asemblerovým operacím a zasvětil je do prostředí práce mikroprocesorů v programovém prostředí. Bakalářská práce se opírá zejména o učební text předmětu Číslicové počítače, který je také hlavním dokumentem pravidel některých instrukcí. V souhrnu bakalářské práce lze vyčíst hlavní myšlenku simulátoru, využívající identifikaci textového řetězce za pomocí prázdných znaků a čárek. Výsledkem bakalářské práce je webová stránka, která nabídne studentům předmětu Číslicové počítače interaktivní prostředí simulátoru především s pokročilou nápovědou, která je specifická pro většinu chyb.

Abstract

The goal is to provide students with an interactive program, in our case a web application that will be used for educational purposes of subject CIP. Creating a Website, including JavaScripts and PHP scripts should fully substitute a simulator designed for educational purposes by Ing. Martin Vlasak and Ing. Tomas Martinec, Ph.D. The web simulator uses a simple fictitious microprocessor to ensure that students learn basic assembler operations and is dedicated to the work environment in the microprocessor programming environment. Bachelor's thesis is based primarily on the textbook of subject CIP, which is also the main document of rules of some instructions. The whole thesis can be seen the main idea of the simulator, using the identification string of the text using blank characters and accent. Result of this work is a website that will offer students of subject CIP primarily for interactive simulator with advanced help that is specific for major mistakes.

Obsah

1 Úvod

V průběhu této bakalářské práce budu probírat jednotlivé části, které je zapotřebí znát pro správné pochopení funkční hodnoty simulátoru, či jeho naprogramování. Probíranou látkou budou základy fiktivního školního mikroprocesoru, typy instrukcí a jejich funkční hodnoty, ale z větší části se budu v práci věnovat programovacím jazykům JavaScript a PHP, které slouží jako základní elementy pro simulování funkcí fiktivního mikroprocesoru. Předpokladem je, že čtenář zná základní převody mezi dvojkovou neboli binární a šestnáctkovou neboli hexadecimální soustavou, protože tyto převody tvoří základ veškerého porozumění pracovní činnosti jednotlivých instrukcí, či například operací se vstupy/výstupy nebo pamětí.

Simulátor se kromě JavaScriptu a PHP skládá také z programovacích jazyků XHTML a CSS, které slouží jako pomůcky pro vizuální interpretaci přes webový prohlížeč a nabízí tak uživateli jednotvárné interaktivní prostředí pro práci se simulátorem a jeho funkcemi.

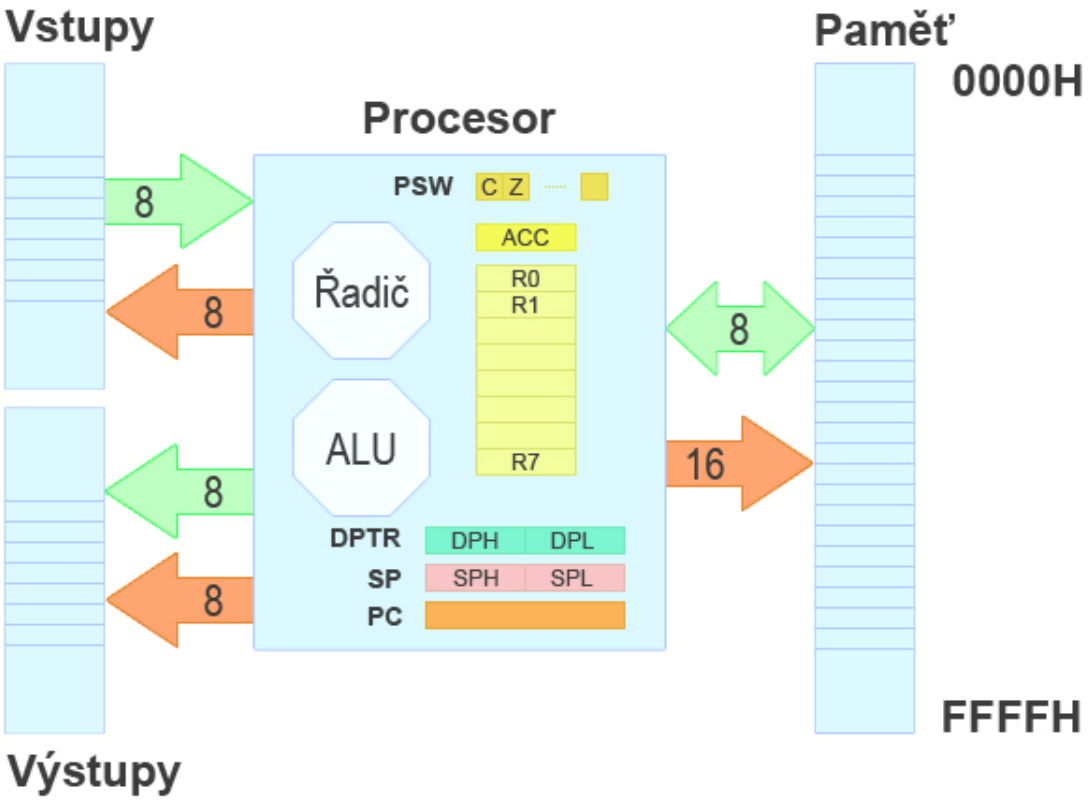
1.1 Fiktivní mikroprocesor pro účely simulace

Mikroprocesor, který byl vybrán pro studijní účely předmětu Číslicové počítače, se dá výborně popsat jednoduchým obrázkem, jehož základní podoba lze vidět na obrázku 1.1. Učební text předmětu Číslicové počítače, mi byl hlavním průvodcem při práci se simulátorem a jeho instrukcemi. Pro účely simulace byly některé vlastnosti mikroprocesoru, jenž má základy například v mikroprocesoru 8051, záměrně zanedbány.

Simulovaný mikroprocesor je osmibitový a disponuje jedním šestnáctibitovým rozhraním, které reprezentuje čtení/zápis do paměti s až 65 536 osmibitovými adresami. Ostatní datové sběrnice komunikují pouze osmibitově. Programová paměť mikroprocesoru je pro účely simulace zanedbána a je dán předpoklad, že vykonání jedné instrukce trvá 1µs.

Mikroprocesor bude také disponovat vstupně/výstupním rozhraním. Toto rozhraní je rovněž osmibitové a v samotné simulaci lze vybrat až z tří možných výstupů, reprezentovaných dvěma řadami osmi diod a jedním osmisegmentovým displejem.

Vstup máme prozatím volen jako řadu osmi spínacích tlačítek, u kterého první tlačítko odshora značí nejnižší bit, tedy 2^0, a poslední tlačítko 2^8.



Obr. 1.1: Základní schéma simulovaného fiktivního mikroprocesoru

Pomocí programového kódu chceme přistupovat například do paměti, či registrů a tak si představíme veškeré bloky a registry simulovaného mikroprocesoru.

Základní bloky:

- Řadič – Řídí činnost jednotlivých instrukcí
- ALU (Arithmetic Logic Unit) – Aritmeticko-logická jednotka slouží pro vlastní matematické operace

Registry:

- R0-R7 – slouží jako rychlá paměť pro matematické operace

- A – nazýván také jako akumulátor nebo ACC, je speciální registr, který je u některých instrukcí povolen jako jediný, většinou slouží spíše jako prostředník pro komunikaci s ostatními registry
- PSW (Program State Word) – je registr, který obsahuje aktuální stav procesoru, pro účely simulace se bude používat pouze bit C (Carry), který nabývá hodnoty 1, pokud došlo při poslední matematické operaci k přetečení, a bit Z (Zerro), který nabývá hodnoty 1 v případě, že je v akumulátoru uložena nula
- DPTR (Data Pointer) – slouží pro nepřímé adresování paměti, je šestnáctibitový, ale přistupujeme k němu osmibitově pomocí registrů DPH a DPL
- SP (Stack Pointer) – jedná se o šestnáctibitový registr, sloužící jako ukazatel na vrchol zásobníku, přistupujeme k němu osmibitově pomocí registrů SPH a SPL
- PC (Program Counter) – šestnáctibitový registr, který obsahuje adresu právě vykonávané instrukce

2 Instrukce

Aby bylo možné naprogramovat simulátor, musí být dáno, jaké instrukce se budou zpracovávat, jaké mají povolené parametry, co přesně mají dělat a jaké funkce mají vykonávat. Vypíšeme si seznam důležitých instrukcí, se kterými bude simulátor mimo jiné pracovat, a jednoduše si popíšeme to, co přesně mají instrukce dělat a s jakými registry, či hodnotami mají pracovat. Pouze po pochopení toho, jakou přesně mají jednotlivé instrukce funkci, můžeme promyslet celou problematiku simulování ve webovém prohlížeči.

Dále je nutné analyzovat instrukce a jejich operandy tak, aby se daly rozpoznat automatickou formou neboli samotným simulátorem a vykonaly přesně to, k čemu instrukce slouží, či co operandy obsahují. Analýza instrukcí bude následující, nejprve budou instrukce rozděleny na čtyři možné typy, které se pak zpracovávají v samotné simulaci nebo při analýze programového kódu ve validátoru. Bude se jednat o typy s různým počtem operandů, tedy například s žádným operandem, se dvěma operandy, apod.

2.1 Základní pravidla programového kódu

Na začátek, než začneme s výčtem některých použitých instrukcí, je nutné zmínit základní pravidla vstupního programového kódu, respektive asemblerového kódu, který bude zadávat uživatel ve webovém rozhraní simulátoru do textového pole. Jedním ze základních pravidel je, že při definici návěští se vždy používá za textovým řetězcem dvojtečka. Návěští nesmí být ve vstupním programovém kódu duplicitní a také musí být definováno vždy, je-li obsaženo jako operand v některé ze skokových instrukcí.

V programovém kódu je povoleno komentovat jednotlivé funkční postupy. Pro zakomentování určité části textu lze používat středník, či dvojitá lomítka. Jako komentář je brán textový řetězec za identifikujícím znakem až do konce řádku, nikoliv do dalšího identifikujícího znaku. V následujícím příkladu jsou vypsána všechna zmíněná pravidla.

Příklad:

```
skok:     ;komentář
INC   A
JMP   skok        //druhý komentář
```

2.2 Instrukce s žádným operandem

2.2.1 NOP (No Operation)

Instrukce NOP slouží jako prázdná instrukce, neprovádí žádný přenos a nemá v podstatě žádný účel. Tato instrukce ovšem zaměstná mikroprocesor na stejnou dobu, jako jakákoliv jiná instrukce.

2.2.2 RET (Return)

Return slouží jako návratová instrukce pro instrukci CALL, kterou naleznete v podkapitole 2.3.1. Instrukce RET nejdříve odebere z paměti vyšší byte na adrese zásobníku, sníží zásobník o jedničku, poté odebere z paměti na adrese zásobníku nižší byte a také sníží zásobník o jedničku. Vyšší a nižší byty jsou reprezentovány horními a dolními osmi bity, ze kterých se složí šestnáctibitová adresa instrukce, které se poté předá řízení.

2.3 Instrukce s jedním operandem

2.3.1 CALL

Zatímco RET odebírá z paměti návratové adresy a snižuje zásobník, instrukce CALL zvýší zásobník o jedničku a do paměti na adresu zásobníku uloží nižší byte adresy následující instrukce za CALL, v dalším kroku znovu zvýší zásobník o jedničku a uloží vyšší byte adresy následující instrukce za CALL. V posledním kroku předá řízení instrukci na adrese uvedené v operandu.

Jako operand skokových instrukcí vždy slouží šestnáctibitová adresa, kterou ovšem u všech skokových instrukcí zadáváme v textové podobě návěští bez dvojtečky.

2.3.2 INC (Increment)

Z anglického slova Increment, tedy zvýšit, je zřejmé, že tato instrukce slouží k zvýšení hodnoty registru uvedeného na místě prvního operandu. Instrukce vezme hodnotu uloženou v registru a zvýší ji o jedničku, výsledek pak opět uloží na místo daného registru.

U akumulátoru, či některého z registrů R se při přetečení své maximální hodnoty začíná od nuly, nesmíme však opomenout, že jde o osmibitové registry, kdežto DPTR jako takový je šestnáctibitový, takže může nabývat hodnoty až 65 535.

Povolené operandy:

INC A

INC R_n

INC DPTR

2.3.3 JMP (Jump)

Jump je základní skoková instrukce, která vykoná skok na zadanou adresu bez jakýchkoliv podmínek, jako tomu je například u instrukcí JZ, JNZ, JC apod. U skokových instrukcí všeobecně lze použít na místě adresy symbol „$“, který znamená, že instrukce skočí sama na svoji adresu, takže v podstatě vytvoří pomyslnou nekonečnou smyčku. V simulátoru tuto nekonečnou smyčku zpracujeme jako úplný

konec simulace, neboť taková nekonečná smyčka by mohla počítači způsobit velkou zátěž, či dokonce zamrznutí prohlížeče.

Povolené operandy:

JMP <adresa16>

2.3.4 PUSH, POP

Instrukce PUSH a POP slouží pro práci se zásobníkem. PUSH nejprve zvýší registr SP o jedničku a uloží obsah uložený v registru zadaném na místě prvního operandu do paměti na adrese zásobníku. Druhá instrukce slouží pro vybírání hodnoty uložené v paměti na adrese zásobníku, kterou uloží do zadaného registru a v posledním kroku sníží registr SP o jedničku.

Povolené operandy:

PUSH A	POP A
PUSH R_n	POP R_n
PUSH PSW	POP PSW

2.4 Instrukce se dvěma operandy

2.4.1 MOV (Move)

Tato instrukce je nejzákladnější instrukcí asembleru a slouží ke zkopírování určité hodnoty zadané programátorem, či uložené v registru, do registru, který určíme na místě prvního operandu.

Povolené operandy:

MOV A, R_n	MOV R_n, A	MOV <adresa8>, A
MOV A, @R_n	MOV R_n, <data8>	MOV DPH, A
MOV A, DPH	MOV @R_n, A	MOV DPL, A
MOV A, DPL	MOV DPTR, <data16>	MOV SPH, A
MOV A, SPH	MOV @DPTR, A	MOV SPL, A

MOV A, SPL MOV SP, <data16> MOV A, @DPTR

MOV A, <data8> MOV A, <adresa8>

2.4.2 ORL

Příkladem jedné z logických instrukcí je instrukce ORL, která slouží k logickému součtu prvního a druhého operandu po jednotlivých bitech. Podobné logické instrukce značí, že se musí při programování simulátoru počítat i s logickými operacemi, kterých nelze docílit použitím běžně dostupných matematických operací programovacího jazyku, ve kterém se bude samotný běh simulace vytvářet.

Povolené operandy:

ORL A, R_n

ORL A, @R_n

ORL A, <data8>

ORL A, <adresa16>

Příklad:

ACC	0	0	1	0	1	0	0	1
#167	1	0	1	0	0	1	1	1
ORL A, #167	1	0	1	0	1	1	1	1

2.4.3 OUT

Pro zápis na výstup slouží instrukce OUT, která jednoduše zkopíruje hodnotu uloženou v akumulátoru na adresu výstupu. Akumulátor je jediným povoleným operandem na druhém místě. V simulátoru používáme tři adresy výstupů s označením 0 (vrchních osm diod), 1 (spodních osm diod), 2 (osmisegmentový displej).

Povolené operandy:

OUT <adresa8>, A

2.5 Instrukce se třemi operandy

2.5.1 CJNE (Compare and jump if not equal)

Instrukce CJNE porovnává hodnotu registru se zadaným číslem, pokud se hodnota registru liší od zadaného čísla, pak se provede skok na zadanou adresu, v opačném případě se pokračuje další instrukcí.

Je-li použit jako první operand akumulátor, pak platí, že když bude hodnota akumulátoru menší, nežli porovnávané číslo, nastaví se jednobitový registr C na hodnotu 1, v opačném případě na hodnotu 0.

Povolené operandy:

CJNE A, <data8>, <adresa16>

CJNE R_n, <data8>, <adresa16>

2.6 Seznam ostatních instrukcí

Instrukce pro matematické operace:

CPL, ANL, XRL, RR, RRC, RL, RLC, DEC, ADD, ADDC, SUBB

Vstupní instrukce:

IN

Skokové instrukce:

JZ, JNZ, JC, JNC, DJNZ

3 Princip simulace

Pro vytvoření simulátoru fiktivního mikroprocesoru s programovým prostředím je nutné znát základní pravidla, která jsou určena v předešlých kapitolách. Jedná se zejména o hodnoty, které se budou přenášet, ukládat, či vypočítávat. Také je důležité

vědět, kdy a kde dochází k přetečení dat, která přesáhla svojí maximální hodnotu a provedení vhodného přeložení, protože na některých místech v simulaci bude zapotřebí nejdříve provést matematickou operaci, až pak přeložit číslo a na jiných místech je tomu přesně naopak.

Úplným základem pro sestrojení simulátoru je sestavení plánu, jak bude taková simulace probíhat.

Návrh plánu:

1. Výběr prostředí a sestavení webové stránky
   - výběr programovacího jazyka
   - návrh designu webové stránky
   - sestavení webové stránky za pomocí vybraného programovacího jazyka
2. Promyšlení pracovní činnosti simulátoru
   - promyslet jak bude simulátor fungovat, které programovací jazyky se použijí, kam se budou ukládat funkční hodnoty pro pozastavení, krokování, atp.
   - rozdělení na různé funkční části (validace, simulace)
3. Návrh validace
   - rozdělení práce mezi skripty
   - identifikace instrukcí a proměnných
   - generování chybových hlášení
4. Návrh simulace
   - ukládání proměnných
   - ovládání simulace (start, krokování, pozastavit, stop)
   - logika procházení jednotlivých instrukcí

3.1 Výběr prostředí

Jelikož se jedná o webový simulátor, vybere se vhodný programovací jazyk, ve kterém se nám bude nejlépe programovat. Základem webové stránky je HTML kód, v našem případě to bude novější verze v podobě XHTML 1.0 Transitional, což je přechodná verze mezi HTML 4.01 a XHTML 1.0. Výběr programového prostředí je

otázkou zkušeností autora, a jelikož nám doba diktuje jít kupředu, je vhodnou formou použít novější programovací jazyk XHTML, i když mnoho prohlížečů není připraveno podporovat MIME typ *application/xhtml+xml*, který by se měl podle pravidel konsorcia W3C správně používat u dokumentů psaných novější verzí XHTML.

Při programování v XHTML se bude používat stylovací jazyk CSS pro přiřazení barev, rozestavení bloků, funkci tlačítek a mnoho dalších funkcí. K jednotlivým vlastnostem CSS se budeme přistupovat především pomocí skriptovacího jazyka JavaScript a objektového modelu (DOM).

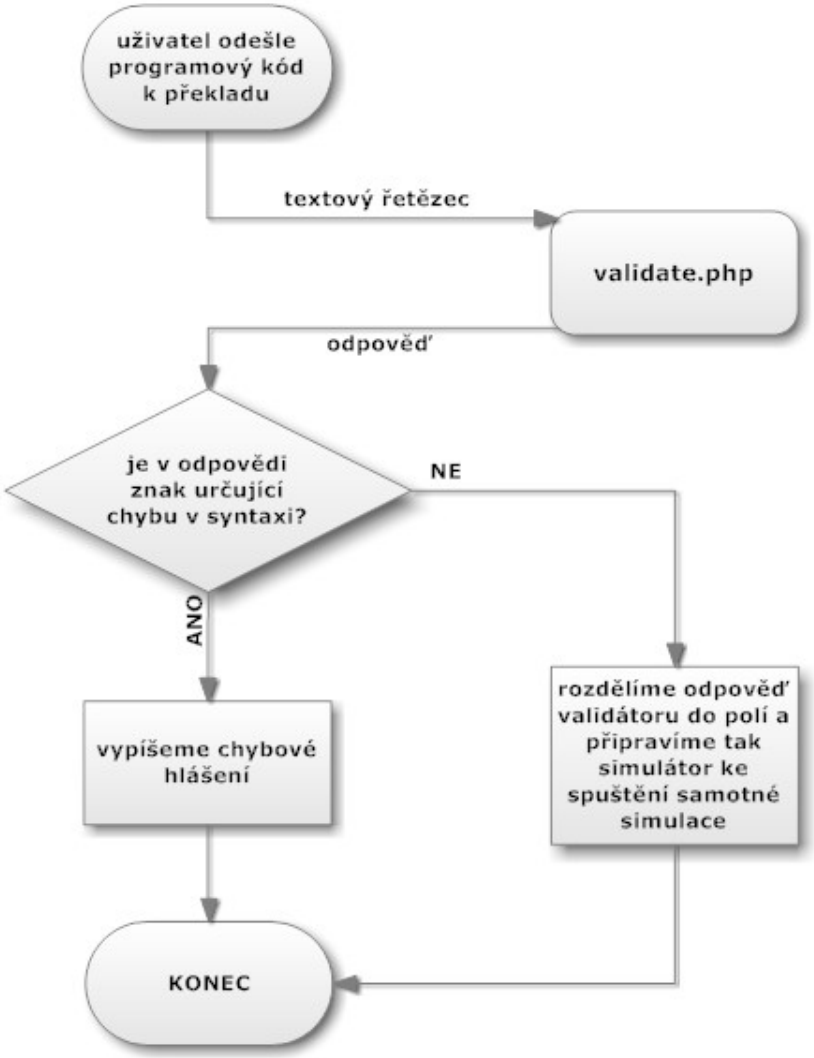
3.2 Návrh pracovní činnosti simulátoru

Nyní bychom se měli dostat do důležitého bodu, kdy rozhodneme, jak bude simulátor a v podstatě celá simulace pracovat. Pro účely bakalářské práce jsem navrhl běh simulace následovně. Po zadání programového kódu do editoru odešleme text pomocí AJAXu vytvořenému PHP skriptu, který slouží jako validátor. PHP skript rozloží programový kód podle kritérií, jejichž základy byly definovány v předchozí kapitole a budou se probírat i v kapitole zabývající se samotnou validací i simulací.

Po rozložení kódu se mohou vrátit dva typy odpovědí. První odpovědí může být chyba s popisem chyby ve validovaném programovém kódu, a druhou odpovědí může být řetězec rozdělený oddělovacími znaky, který bude rozebrán JavaScriptem pro běh simulace. Vývojový diagram na obrázku 3.2 přesně vystihuje to, co se bude dít po odeslání programového kódu k přeložení, respektive validaci.

Druhá část návrhu se týká provedení simulace samotné. Provedení simulace obstarává JavaScript, který si uloží rozdělenou odpověď validátoru do polí. V poli jsou instrukce uloženy postupně podle toho, jak jdou za sebou, což řeší budoucí problém se skokovými instrukcemi.

Samotná simulace bude probíhat tak, že poběží v pomyslném cyklu, ve kterém se volají funkce pro přednastavené instrukce. Při zpětném volání těla simulace dochází ke zpožděnému volání, kvůli možnosti nastavení rychlosti simulace a také proto, že nemůžeme jednotlivé instrukce volat bez zpoždění, neboť poté dochází k zahlcení procesoru počítače. V průběhu simulace se může měnit řídící proměnná pomyslného cyklu, čímž docílíme skoků na potřebné instrukce.



Obr. 3.2: Vývojový diagram návrhu simulace

4 Interaktivní prostředí simulátoru

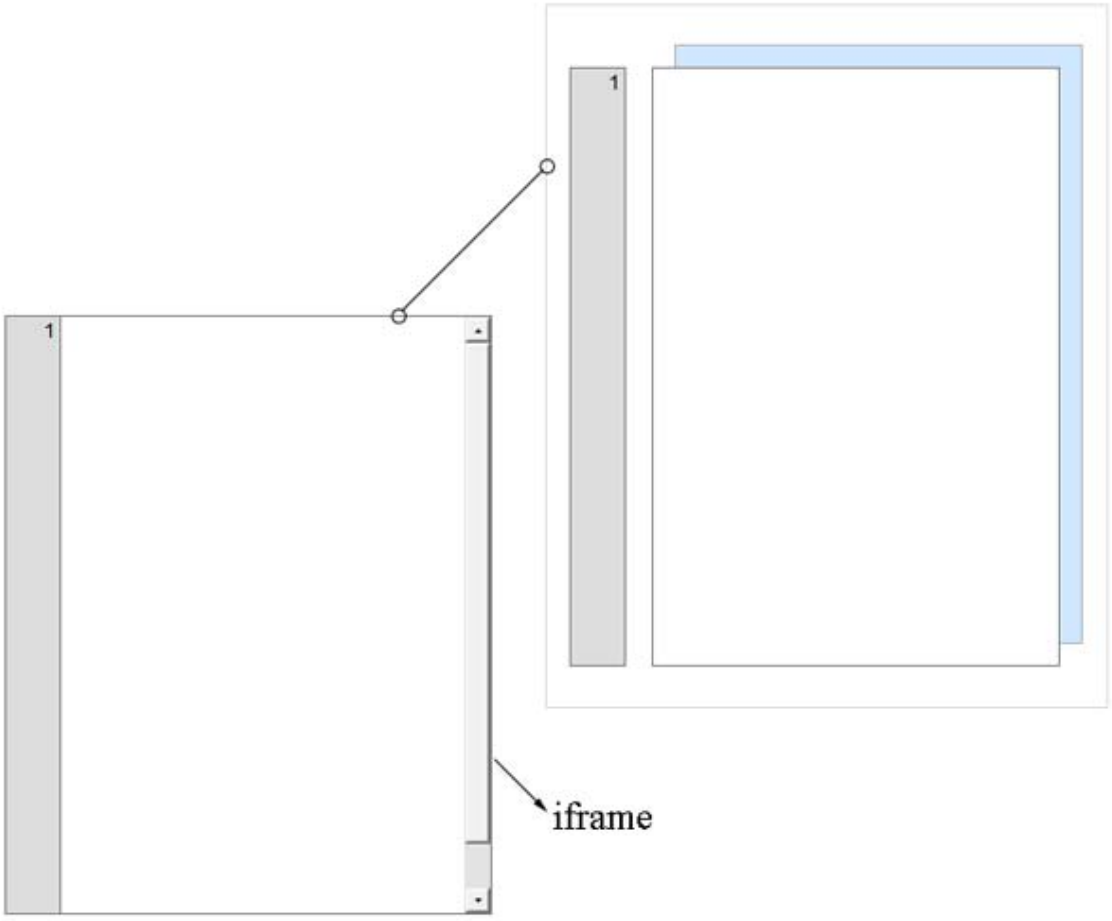
Do základních vlastností webového rozhraní, které bude komunikovat s uživatelem, zahrnujeme jednoduchý editor s možností řádkování a označením řádků, interaktivní tlačítka, nastavení, otevření souboru, uložení souboru či možnosti paměti.

4.1 Editor

Jednoduchý editor slouží v simulátoru jako hlavní komunikátor s uživatelem, neboť se zde budou zobrazovat řádky s chybou, či právě vykonávané řádky. Navíc je v editoru vypsáno řádkování pro lepší orientaci. V bakalářské práci se budeme o vstupu textu zmiňovat jako o editoru, pravdou ale je, že se jedná spíše o textové pole

s přidanými stylovými vylepšeními. Jde spíše o takovou jednoduchou formu editoru pro práci s textovým vstupem, ovšem nejsou zde možnosti měnit styl textu.

Pole pro zápis programového kódu je navrženo v XHTML formou rámu, respektive iframe. Do rámu se načítá jednoduchá HTML stránka, která obsahuje textové pole textarea, jehož styl řídíme pomocí JavaScriptu. Abychom lépe pochopili, jak editor v simulátoru funguje, představte si objekt, který má šířku 30 pixelů, napravo od něj je umístěno textové pole s šířkou 250 pixelů, které má nastaveno průhledné pozadí a v pozadí za textovým polem je stejně veliký objektový element, který bude svojí funkcí působit dojmem označení řádků.



Obr. 4.1: Rozdělení editoru do několika vrstev

Na přesnou formu rozdělení editoru se můžete podívat na obrázku 4.1, který znázorňuje rám editoru, jenž je rozdělen na tři objekty. Světle modrá část za editorem je v podstatě vrstva, do které se duplicitně zapisují řádky z přední, světlejší vrstvy. Bílá

vrstva má nastaveno transparentní pozadí, takže text je vlastně přenesen dopředu a veškerý obsah jak grafický tak stylistický je promítán přes zadní vrstvu. Tento způsob nám umožní přesně identifikovat jednotlivé řádky, ovšem zde musí být zachováno pravidlo stejně vysokého řádku v obou vrstvách a také stejná délka elementů.

V editoru si lze všimnout menšího nedostatku při označování řádků. Pokud do jednoho řádku zadá uživatel příliš dlouhý text, je barevné označení vůči řádku krátké. Pokud nastavíme větší délku barevného elementu, jenž nám dává vizuální vjem označení řádku, pak se rám rozšíří a ve spodní části se zviditelní skrolovací lišta. Jelikož je asemblerový kód vesměs krátký, není nutné kazit vzhled simulátoru o zbytečnou skrolovací lištu, která posouvá text. V tomto případě se zohledňuje předpoklad, že uživatel nebude zadávat příliš dlouhý text.

Princip řádkování editoru je v tom, že při psaní se kontroluje textové pole (na obrázku 4.1 bílá vrstva), které je rozděleno do pole podle znaku „\n“, značící konec řádku, čímž můžeme zapsat jednotlivá čísla řádků. Pokud dojde ke spuštění překladu vstupního programového kódu, pak se po řádcích do pozadí textového pole zapíší další elementy s pomyslnou výškou řádku (světle modrá vrstva na obrázku 4.1), kterým se při simulaci, či chybě, bude měnit barva pozadí.

4.2 Tlačítka a nastavení

Všechna tlačítka ve webovém prostředí jsou vytvářena pomocí stylovacího jazyka CSS. Jedná se vlastně o blokové elementy s pozadím ve formě obrázku. Interakce je pak řízena JavaScriptem, pomocí něhož se změní buďto pozadí elementu, či třída neboli class. Tlačítka mají různě nastavené akce podle toho, jakou zastávají funkci. Některá tlačítka se chovají jako obyčejná s běžnou funkcí po stisknutí tlačítka myší, ale například tlačítka vstupů lze ponechat aktivovaná, přesuneme-li při aktivaci tlačítka kurzor myši mimo objekt tlačítka.

Nastavení slouží pro výběr předurčených akcí, je zde také zajímavý doplněk a to možnost řídit rychlost simulace za běhu pomocí posuvníku, který je tvořen pomocí blokového elementu a kaskádových stylů. Při interakci s posuvníkem se zjišťuje poloha kurzoru, která se odečte od vzdálenosti levého okraje okna a blokovému elementu se nastaví odsazení od levé strany. Tím tvoříme dojem, že posuvník ovládáme a posouváme s ním.

Všechna okna otevřená v simulátoru jsou závislá na přímém vypnutí uživatelem. Pomocí JavaScriptu řídíme, aby se ukazovalo pouze jediné okno, ale dokud jej uživatel nezavře, bude stále viditelné. Tento systém oken je dělán záměrně proto, aby měl uživatel možnost například řídit rychlost simulace, či měnit výpis paměti, aniž by musel po každé změně znovu otevírat nastavení.

4.3 Otevření a uložení souboru

Okna, která se objeví po interakci s tlačítky značící otevření, či uložení, jsou skryté rámy, respektive externě načtené HTML dokumenty. Tlačítko otevření nabídne v rámu výběr souboru, a pokud uživatel vybere jiný, než podporovaný soubor s koncovkou txt, či asm, vypíše se varování a neprovede se žádná akce. Pokud naopak uživatel vybere soubor s podporovanou koncovkou, je pomocí JavaScriptu obsah souboru pro otevření zapsán z potomka rámu do editoru, respektive potomka hlavního dokumentu. Díky tomu, že se u všech souborů používá jako identifikace konce řádku znak „\n“, mělo by být zaručeno bezproblémové vložení i s překladem řádků do editoru webového simulátoru. V editoru je speciálně pro tento nárazový způsob vložení delšího textu upraven výpis řádkování formou podmínky pro opakovanou kontrolu.

Uložení souboru pracuje na principu přepsání HTTP hlavičky. Po kliknutí na uložení souboru, je zkontrolováno, zdali není obsah prázdný, či zda bylo zadáno jméno vstupního souboru. Pokud jsou všechny náležitosti splněny, je obsah editoru vypsán do prázdného HTML dokumentu v příslušném rámu, kterému se změní hlavička HTTP na „Content-Disposition: attachment“, ve které se definuje název souboru, díky čemuž se nám nabídne soubor ke stáhnutí s názvem, jaký jsme si určili. Ukládání souborů je kompatibilní se simulátorem od pana Ing. Martina Vlasáka a pana Ing. Tomáše Martince, Ph.D, díky čemuž lze přenášet programový kód mezi oběma simulátory.

4.4 Paměť

Element paměti je také usazen do rámu, neboť u rámů se dá definovat pevná velikost a v jejím obsahu lze jak vertikálně, tak horizontálně skrolovat. Díky objektovému modelu JavaScriptu a možnosti přistupovat k funkcím a vlastnostem stránky s definicí potomka, respektive vložené externí stránky, která musí být ze stejné domény jako nadřazená stránka, čili rodič, je zaručen volný prostor k úpravám různých vlastností paměti z nadřazeného HTML dokumentu.

Můžeme tak paměť zobrazovat od různých hodnot, respektive se přepíše obsah objektového elementu a nahrají se do něj nové řádky, značící ve webovém rozhraní adresy paměti. V pozadí ovšem ukládáme do proměnných možnost procházet paměť, neboť se liší zápis pro adresy paměti vypsané v rámu a ty, které nebyly v rámu zobrazeny.

Při práci s pamětí musí být vlastnosti zachovány tak, abychom uživateli v paměti zobrazili i hodnoty uložené na adresách, které jsme nově vypsali.

5 Validátor

Tato kapitola se zaobírá nejdůležitější částí simulátoru, kterou tvoří validátor, nazývaný také jako překladač, či parser. V našem případě validátor zastává nejdůležitější funkci, neboť zkontroluje vložený kód a pokud nalezne chybu v syntaxi, vypíše na výstupu znak určující chybu a přidá informaci o chybě, plus stručnou nápovědu. Pokud vložený programový kód projde validací, na výstup se vypíší instrukce a operandy oddělené znaky, které pak JavaScript rozdělí do polí a bude je používat pro běh simulace.

5.1 Ústřední skript pro validaci

Hlavním souborem pro validování vloženého textového kódu je skript validate.php, se kterým komunikuje JavaScript při asynchronních operacích volaných přes XmlHTTPRequest objekt. Validate.php nejdříve načte všechny potřebné skripty přes include a v načtených skriptech už k žádnému dalšímu načítání skriptů nedochází.

Jedním z načtených skriptů je například _rstring.php, který obsahuje vytvořené funkce pro práci s řetězci, jež budou uplatněny při samotné identifikaci validovaného programového kódu a tvoří tak pro tento projekt primární funkce, bez kterých by nebylo možné správně procházet a identifikovat textový řetězec.

Ve zmíněném skriptu jsou například dvě funkce pro výběr části textu. Obě funkce mají stejný princip, tedy najít hledaný textový řetězec a vrátit všechny znaky, které proces funkce prošel při hledání. Zbylé znaky, respektive ty které se neprošly, budou ignorovány, protože se končí nalezením hledaného textového řetězce. Odlišnost obou funkcí je ve směru prohledávání. Jedna funkce postupuje od začátku a druhá od

konce prohledávaného textového řetězce. Další funkce obsažené ve skriptu _rstring.php slouží pro identifikaci obsahu, respektive nám vrátí, kolikrát se hledaná proměnná vyskytuje v prohledávaném textovém řetězci.

Ve spuštěném skriptu validate.php se po načtení všech potřebných skriptů rozdělí validovaný text do pole podle řádků. V tomto kroku je důležité vědět, že text obsahuje i neviditelné znaky, které oznamují konec řádku, návrat vozíku, tabulátor apod. Znak „\n“ v textovém řetězci znamená konec řádku, což bude hlavní separátor textového řetězce pro rozdělení do polí podle řádků. Po tomto rozdělení vstupního textu se spustí tělo procesu vyhledávání.

5.2 Tělo validace

Skript control.php tvoří pomyslné tělo celé validace. Je řešen formou třídy, protože se budou v průběhu skriptu ukládat potřebné informace do veřejných proměnných, ke kterým chceme mít přístup, a nemusí se dále u funkcí předávat proměnnými.

Proces celé validace se řeší formou cyklu, kde jako ústřední proměnná slouží počet řádků, takže v podstatě procházíme pole, ve kterém se vyskytují jednotlivé řádky validovaného textu. Podmínkou je, že validovaný text, čili vložený programový kód, splňuje pravidlo jedné instrukce na řádek. Výjimku připouštíme pouze při definici návěští, za kterým může být na stejném řádku jedna instrukce.

První úpravou u textového řetězce je odstranění komentářů. Komentáře se odstraňují formou regulárních výrazů a funkce *ereg_replace*, která vyhledá znaky „;“ a „//“, za nimiž odstraní veškeré znaky. Po tomto kroku se provádí první z mnoha volaných funkcí pro odstranění prázdných znaků ze začátku a konce řetězce.

5.2.1 Návěští

Dalším krokem předcházející samotné identifikaci instrukcí a proměnných je vyhledání návěští, které se vždy identifikuje dvojtečkou, ale může za ním pokračovat jedna instrukce. Není dovoleno, aby textový řetězec na jednom řádku obsahoval více identifikačních znaků v podobě dvojité tečky a tak pokud identifikujeme na jednom řádku více dvojitých teček, vygeneruje se chyba.

Další pravidlo, na které se dává pozor, je text definující návěští, který nesmí obsahovat mezery, musí být tedy spojen v jeden řetězec, proto pokud se nalezne před dvojitou tečkou mezera, či tabulátor, vygeneruje se chyba. Jelikož mohou za návěštím pokračovat instrukce, nalezne-li se za návěštím jakýkoliv textový znak, uloží se návěští do potřebných proměnných, vymaže se definice návěští z textového řetězce a upravený text se postoupí dalšímu prohledávání stejně, jako kdyby na řádku žádné návěští nebylo.

5.2.2 Princip prohledávání textového řetězce

Řídíme se tím, že za instrukcemi vždy musí následovat prázdný znak, ovšem ten může mít podobu jak mezery, tak tabulátoru, na což se klade velký důraz. Při prohledávání textového řetězce se prochází cyklus, pokud se narazí na prázdný znak, vystaví se nový cyklus, nebo tomu také můžeme říkat nová vrstva, ve které se předá proměnná a začne se prohledávat zbytek textového řetězce. Jednoduše bychom mohli říci, že se nejdříve hledá instrukce, pak první parametr, druhý parametr a třetí parametr. Pokud cyklus došel na konec textového řetězce, vyhodnotí se výsledek vůči dané vrstvě.

Instrukce jsme si rozdělili na instrukce bez operandů, s jedním operandem, se dvěma operandy a se třemi operandy. První dva spojuje podobný příznak a tím je absence čárky. Tento příznak nechť je definicí i pro identifikaci instrukcí se dvěma, či třemi operandy. V praxi to znamená to, že pokud v prohledávaném řádku identifikujeme čárku, bude se volat funkce pro práci s více operandy, respektive parametry, a pokud se v řádku neidentifikuje čárka, volá se funkce pro práci s maximálně jedním parametrem.

Příklady nevalidního kódu:

1. INCA
2. INC AA
3. INC A A

První bod příkladu se identifikuje jako instrukce bez parametru a chyba se identifikuje při prohledávání povolených instrukcí. Abychom mohli ve skriptu identifikovat, které instrukce jsou povolené, jsou všechny instrukce uloženy v určité proměnné. Zmíněný první bod příkladu se vyhodnotí jako správný zápis pro instrukci bez parametru, protože je textový řetězec spojen a není rozdělen žádným prázdným znakem, podle kterého by

se identifikovalo, že za instrukcí pokračují parametry. U kontroly povolených instrukcí ovšem instrukce INCA neprojde, protože se nejedná o žádnou známou instrukci.

Druhý bod příkladu je také důkazem správného zápisu, ovšem špatného parametru. Chyba se bude identifikovat při kontrole povolených parametrů, které jsou stejně jako instrukce uloženy v určité proměnné.

Třetí bod příkladu se vyhodnotí jako špatná syntaxe, neboť neznáme žádný příklad, kde jsou za operandy další znaky, aniž by byly odděleny čárkou. Pokud se mělo jednat o komentář, chybí potřebné specifické znaky pro rozpoznání komentáře a pokud se mělo jednat o zápis instrukce s více parametry, chybí oddělovací znak v podobě čárky.

Dalšími druhy programového kódu jsou instrukce s více jak jedním parametrem, které jsou vždy odděleny čárkou.

Příklad nevalidního kódu:

1. MOV,
2. MOV,A, #5
3. ,MOV
4. MO,V
5. MOV A,
6. MOV A, , #5
7. MOV A A, #5
8. MOV A, #5 A
9. MOV A, #5, #10 A

U prvního bodu příkladu se identifikuje chyba, protože za instrukcí musí nutně být mezera nebo tabulátor, takže pokud následuje specifický oddělovací znak v podobě čárky, jedná se o chybu. První bod příkladu je spojen s druhým, neboť pokud narazíme na první chybu, nemá cenu pokračovat v další identifikaci textového řetězce.

Třetí chybový bod bude identifikován hned na začátku prohledávání díky specifické podmínce zaměřené přesně na identifikaci čárky na první pozici textového řetězce.

Čtvrtý bod se tváří jako instrukce bez parametru až na to, že obsahuje oddělovací znak, takže se v procesu na konci prvního taktu identifikuje jako chyba, protože se při prohledávání bude hledat mezera a místo mezery přijde konec prohledávaného textového řetězce.

Pátý a šestý bod příkladu mají stejné identifikace, neboť po oddělovacím znaku musí být definován další parametr, pokud následuje konec řádku nebo oddělovací znak, jedná se o chybu bez ohledu, co následuje dále.

Zbývající body v příkladu mají stejné, či podobné identifikace jako předchozí zmíněné. V praxi validátoru se jedná o vrstvení prohledávání, identifikaci míst, kde dochází k chybě podle výše zmíněných pravidel a míst, kdy je syntaxe v pořádku a textový řetězec se postoupí finální kontrole povolených instrukcí a operandů. Pro takové případy se musí na konci každého taktu připravit podmínka pro identifikaci stavu validace a to samozřejmě platí i pro vystavování dalších vrstev.

5.3 Prohledávání instrukcí s maximálně jedním parametrem

K prohledávání textového řetězce, respektive polí, se používají dva skripty, které pouze kontrolují syntaxi, nikoliv povolené instrukce, jenž se vždy prohledávají až na konci procesu. Prvním zmíněným skriptem je research_instructions_one_param.php, který je přizpůsoben na prohledání instrukce, u které se díky absenci identifikačního znaku předběžně identifikoval pouze jeden, či žádný operand, respektive parametr.

Tento skript je velmi jednoduchý, neboť identifikuje textové znaky, které se postupně ukládají po znaku do proměnné, a jakmile se dosáhne konce prohledávaného řetězce, aniž by se zaregistroval jakýkoliv prázdný znak, identifikované znaky se uloží do proměnné a ta se postoupí dalšímu porovnávání.

V prohledávání se také může narazit na prázdný znak, za kterým jsou textové znaky, v takovém případě předpokládáme, že ony znaky patří do prvního parametru. Pro zajištění bezproblémového identifikování instrukcí a parametrů prochází textový řetězec úpravami. V jedné z takových úprav se odstraňují prázdné znaky ze začátku i konce prohledávaného textu nebo stejně tak se odstraňují i komentáře a případné návěští, takže formát prohledávaného textového řetězce vždy končí tisknutelným znakem, nikoliv mezerou, či tabulátorem.

5.4 Prohledávání instrukcí s více parametry

Skript pro prohledávání instrukcí s více parametry je takovou více vrstvenou verzí skriptu z předchozí kapitoly. Textový řetězec se v něm prochází tou samou logikou, jen se bere v potaz oddělovací znak v podobě čárky a je zde také samozřejmě o mnoho více míst, kde dochází k chybě v syntaxi.

Logikou celého vyhledávání je vyhledávání prázdných znaků, či oddělujícího znaku v podobě čárky. Musí se brát v potaz to, že mezera, čili prázdný znak, musí být pouze za instrukcí, pak už se můžou parametry oddělovat pouze čárkou bez nutnosti další mezery. I když nám textové pole prohlížeče, v našem případě jednoduchý editor, nedovolí použít tabulátor ve webovém prohlížeči, protože po stisknutí tabulátoru přeskočí akce na jiný element, musíme brát v potaz také to, že umožníme uživatelům kopírovat a načítat obsah programového kódu z externích zdrojových souborů, které mohly být původně vytvořeny v profesionálním, či amatérském textovém editoru, kde už nastíněné omezení neplatí.

V jednotlivých vrstvách prohledávání se vždy zjišťuje, zdali se nedošlo na konec textového řetězce a podle situace se vyhodnotí buďto chyba, či validní syntaxe a uložené výsledky se postoupí dalším kontrolám. Zde platí, že na konci každého procesu se předá instrukce a proměnné ke zjištění, zdali patří mezi povolené instrukce a parametry, či nikoliv.

5.5 Identifikace povolených instrukcí

Pro porovnávání textových řetězců, zdali jsou mezi povolenými instrukcemi, či parametry, se rozlišují tři funkce a to opět podle počtu operandů, respektive parametrů. Při vyhledávání, zdali je případná instrukce mezi povolenými, se volá funkce *ident_instr()* ve skriptu control_instructions_parameters.php, která vrací několik různých pevně definovaných hodnot.

Pokud nám funkce vrátí číslo -1, pak nebyla instrukce nalezena v seznamu povolených instrukcí, pakliže dostaneme jako odpověď číslo -2, byla instrukce nalezena v seznamu se třemi parametry. Jelikož instrukce se třemi parametry je pouze jedna, neprohledáváme ji cyklem, ale určujeme ji pevně definovaných ukazatelem na pole. U dalších instrukcí se vrací číslo v rozmezí 0–99 pro instrukce se dvěma parametry, v rozmezí 100–199 pro instrukce s jedním parametrem a číslo větší jak 200 pro

instrukce bez parametrů. Tento systém nám umožní lépe zjistit, do jaké kategorie spadá daná instrukce a relevantně pro ni zobrazit chybovou zprávu a především nápovědu.

5.6 Identifikace povolených parametrů

Parametry máme uložené ve čtyřech polích podle důležitosti. U definice parametrů je vidět, že porovnávané textové řetězce musí být vždy malými písmeny, tzn., že musí být zajištěn překlad prohledávaného textu na malá písmena. Registr ACC lze při psaní programového kódu vkládat formou A, i ACC a proto je v porovnávání zajištěn přepis na požadovaný formát, respektive parametr „a“ se přepisuje na „acc“, jenž se pak porovnává s povolenými parametry.

Podobné pravidlo jako u porovnávání instrukcí je uplatněno i při zjišťování povolených parametrů, kde se vracejí hodnoty -1 pro nenalezený parametr, 0–99 pro parametry z prvního pole $allowed_param1, 100–199 pro parametry z druhého pole $allowed_param2, 200–299 pro parametry z pole $allowed_param3 a číslo větší než 300 pro parametr nalezený v poli $allowed_param4. S těmito údaji se bude pracovat při zjišťování povolených operandů u daných instrukcí. To znamená, že se při prohledávání textového řetězce volá příslušná funkce podle toho, do jaké kategorie spadá nalezená instrukce, a v té se zjišťují další okolnosti.

Jelikož lze zadávat i parametry ve tvaru čísla, počítá se i s překladem různých numerických formátů.

Příklad různých numerických formátů:

1. MOV A, #17
2. MOV A, #00001001b
3. MOV A, #11h
4. MOV A, 17

Zde se rozlišují numerické formáty v podobě čísla a adresy. Čísla všeobecně mají na prvním místě uvedenou mřížku, kdežto adresy jsou zadány bez mřížky. Logika zjištění numerického formátu je, že pokud dostaneme z identifikace parametrů návratovou hodnotu -1, pak se zjišťuje, zdali není parametr zadán v numerickém formátu. Pokud na

prvním místě parametru zjistíme znak #, pak se jedná o numerický formát v podobě čísla, v opačném případě se bude jednat o adresu.

Identifikace znaku # nám rozděluje prohledávání numerického formátu na dvě části podle čísla a adresy, ovšem jejich průběh je naprosto shodný, neboť se v prvním případě odstraní zmíněný znak a pokračuje se stejně jako u předpokladu numerického formátu adresy. V dalších krocích se zjišťuje přes regulární výrazy, zdali byl zadán správný decimální, hexadecimální nebo binární tvar a podle situace vrátíme potřebnou návratovou hodnotu, která značí podobu numerického formátu a také, zdali se jedná o číslo nebo adresu, podle čehož vypíšeme v případě chyby relevantní chybové hlášení a nápovědu.

5.7 Identifikace chyby ve validovaném programovém kódu

Pokud například v těle prohledávání identifikujeme instrukci bez parametru, zavolá se funkce *control_instructions_without_param()*, které se předá proměnná s instrukcí. Zmíněná funkce si zavolá o identifikaci instrukce a uloží si návratovou hodnotu, která následně slouží jako identifikátor chyby. V tomto případě je vše v pořádku, pokud se nám vrátí jako návratová hodnota číslo větší nebo rovno než 200, jakákoliv jiná hodnota je v tomto případě známkou chyby a díky tomu, že máme instrukce rozděleny, můžeme například při zadání instrukce MOV vypsat hlášení, že tato instrukce pracuje se dvěma parametry a zadává se pouze ve tvaru MOV parametr1, parametr2. To samé platí u dalších instrukcí.

Chceme-li porovnávat instrukce s parametry, postupuje se stejným způsobem, jako v předchozím případě, pouze v bodě, kdy je instrukce v pořádku se volá pro ověření parametru nebo parametrů a zjišťují se vazby jednotlivých parametrů. Zde v jednotlivých podmínkách identifikujeme instrukce a pod touto podmínkou identifikujeme parametry. To znamená, že máme přesně daný úsek u funkce pro práci se dvěma parametry, který patří pouze instrukci MOV, kde se zjišťují parametry na prvním a druhém místě s návratovou hodnotou parametrů. Podle pravidel stanovených v kapitole 2 se i zde porovnají zadané parametry s výsledky návratových hodnot, a pokud nejsou dané kombinace operandů povoleny, vypíše se adekvátní chybové hlášení s nápovědou, jak problém odstranit. Takovým chybovým hlášením může být například:

„Nelze zapisovat do @r1 numerickým formátem. Používejte výhradně ACC.“

U každé instrukce je použito tolik chybových zpráv, kolik to jen povoluje logika. Pokud máme například instrukci IN, tak zde lze vypisovat jen dvě různá chybová hlášení podle případu. V prvním případu víme, že instrukce IN povoluje na prvním místě parametru pouze registr ACC, takže jakákoliv jiná kombinace nutně vede k chybovému hlášení. V druhém případě víme, že jako druhý parametr je povolen pouze numerický formát v podobě adresy, takže pokud jako návratovou hodnotu u druhého parametru dostaneme číslo různé od -40, které nám vrátí funkce pro identifikaci numerického formátu v případě, že se jedná o adresu, pak se také jedná o chybu.

S generováním chybového hlášení u ostatních instrukcí je jednáno se stejnou logikou, respektive jsou striktně dodržována pravidla, která jsou dána v kapitole 2 a samozřejmě i pravidla ostatních instrukcí, které nebyly v kapitole zmíněny, ale lze je vyčíst v učebním textu předmětu Číslicové počítače.

5.8 Závěrečné operace validátoru a generování výstupu

Jakmile dospějeme ke konci prohledávání, respektive byly prohledány všechny řádky a nedošlo k chybě, připraví validátor poslední operace. Takovou poslední operací, kterou validátor provede, je prohledání použitých adres návěští a zjištění, zdali byly na některém místě validovaného programového kódu adresy definovány. V tomto procesu se prohlíží pole, do kterého se ukládaly použité návěští, a toto pole se porovná s polem, do kterého se ukládaly definované návěští. Mohou být definována návěští, která nebyla použita, ale nemohou být použita návěští, která nebyla definována, neboť by simulátor nevěděl, na který řádek má skočit. Pokud se při použití návěští nenajde jeho definice, vygeneruje se chyba.

Pokud i poslední operace projde bez vygenerování chyby, bude se generovat výstup. Při generaci výstupu se předpokládá, že validovaný programový kód je v naprostém pořádku a nedošlo k žádné chybě, v tomto kroku se budou na výstup vypisovat upravené instrukce a parametry. Taková úprava tkví například v překladu textu na malá písmena, přepisu instrukce „a“ na „acc“ a také přepis numerického formátu, který se vrací vždy v dekadické podobě, pouze rozlišen znakem na prvním místě textu, zdali se jedná o číslo, či adresu. Například adresa 15 se na výstup zapisuje ve tvaru „m15“.

Výstupní text se rozlišuje znaky „&“ a „|“, díky kterým si JavaScript uloží výstup do pole. Znak „&“ odděluje bloky a znak „|“ řádky. Generovaný výstup má vždy

pět bloků značící postupně identifikátor validnosti, instrukce, první parametry, druhé parametry a třetí parametry. To znamená, že délky pole jsou spárované a pokud si na čtvrtém místě voláme o instrukci MOV, musí na čtvrtém místě v poli s prvním, či druhým parametrem být přesně parametry pro tuto instrukci, zadány uživatelem do editoru simulátoru.

6. Simulace

Běh simulace zastává programovací jazyk JavaScript s využitím objektových modelů, naopak pro přeložení vstupního programového kódu slouží asynchronní JavaScript neboli AJAX, díky kterému pošleme text na přeložení validátoru. Po stisknutí tlačítka „Přeložit“ se nejdříve odešle vstupní programový kód z editoru, a validátor vrátí určitý řetězec, podle kterého se bude postupovat buď chybou, nebo přípravou na samotné spuštění simulace.

Řešení simulace přes programovací jazyk JavaScript nese jisté výhody ve zpracování v podobě tříd. Jeden ze spousty JS skriptů obsahuje třídu nazvanou *upc*, u které je definováno několik vlastních proměnných tak, že opravdu tvoří dojem, jako by samotná třída byla jistým ekvivalentem simulovaného mikroprocesoru. U třídy je definována například paměť v podobě pole, kde každý ukazatel na pole odpovídá adrese a hodnota uložená na místě ukazatele odpovídá hodnotě uložené v paměti na dané adrese.

Dalším takovým případem může být například seznam registrů, kde jsou názvy malými písmeny a každý registr má svoji určenou hodnotu, přesně jako na webovém rozhraní simulátoru. Díky této třídě je přístup k potřebným hodnotám na místech registrů, či paměti velmi usnadněn a především řeší problém s uchováním návratového textového řetězce validátoru při přerušení simulace a jejím opětovném spuštění, či krokování.

6.1 Rozlišení návratového řetězce

Aby se dalo rozlišit, zda validátor vygeneroval chybu nebo shledal validovaný programový kód validním, určili jsme si, že při rozdělení návratového řetězce do pole

pomocí znaku „&“, bude na nulté adrese pole vždy identifikátor validnosti. Pokud bude tento identifikátor různý od znaku „?“, jedná se o chybu ve validovaném textu.

V případě chyby následuje na první pozici ukazatele do pole řádek, na kterém došlo k identifikaci chyby, na druhém místě ukazatele do pole následuje sdělení a na třetím místě následuje identifikační číslo chyby. Tyto údaje se vypíší uživateli přes varovné okno v závislosti na typu chyby a samotné spuštění simulace nebude umožněno.

Pokud dostaneme z validátoru kladnou odpověď, rozdělí se získaný textový řetězec do polí instrukcí a parametrů, které se uloží do třídy simulátoru, kde se zachovají po dobu aktuálního načtení stránky. Důležitým krokem je nastavení několika prioritních parametrů, které se v průběhu simulace ověřují kvůli předčasnému ukončení. Zde dochází k zápisu řádků do pozadí editoru podle posloupnosti instrukcí, aby mohla být jednotlivá pozadí řádků měněna v závislosti na právě vykonávané instrukci a tvořit tak vizuální dojem vykonávaných řádků.

6.2 Průběh simulace

Při předpokladu, že všechna předchozí pravidla byla splněna a nedošlo k vygenerování chyby, lze simulaci spustit. Po spuštění se zavolá funkce *go()*, která vynuluje proměnné značící krokování, zastavení, či pauzu a ověří se, zdali je programový kód v editoru stále označen jako validní a případně se spustí samotná simulace.

Průběh simulace je založen na procházení řádků vloženého programového kódu od nuly po délku všech vrácených instrukcí validátorem, ovšem pořadí se může měnit v závislosti na skokových instrukcích. Provádí se pouze ty řádky, které mají identifikovaný proces v podobě nějaké instrukce, zde dochází také k barevnému označení řádku pro vizuální ilustraci provedení daného řádku. Proces označení má dva kroky, nejdříve je odznačen předchozí řádek a následně je označen právě se vykonávající řádek. Tento systém odznačování předchozích řádků je důležitý, protože pokud bychom si neukládali do proměnné číslo označeného řádku, v dalším kroku bychom nevěděli, který řádek byl označen. V potaz se bere dynamické označování díky skokovým instrukcím, a proto nelze jednoduše zrušit označení řádku v matematické souvislosti mínus jeden řádek. Kvůli zachování systému se zapisují řádky do pozadí

editoru od nuly, i když reálné vykonávání je až od jedničky, a proto je také nula výchozí hodnotou nastavenou pro předchozí odznačení řádku.

Následně se volají funkce přesně podle toho, která instrukce má být vykonána jako další. To znamená, že máme pevně dán úsek, kde se rozpozná, zdali se jedná například o instrukci MOV nebo o instrukci IN a pro simulování procesu dané instrukce se zavolá stejnojmenná funkce, respektive pro instrukci MOV je to funkce *instrMov()*, které se předají dvě proměnné v podobě prvního a druhého parametru.

Na každém konci funkce, která má simulovat určitou instrukci, se vykonává pro simulaci důležitý kód.

Příklad kódu:

```
upc.unmark = upc.row;
upc.row++;
if (!upc.step)
  window.setTimeout("getSimulation()", upc.speed);
```

Tento kód značí, že se nejdříve uloží řádek, kterému má být v následujícím kroku odstraněno označení, zvýší se číslo řádku o jeden k tomu, abychom mohli pokračovat v následujícím kroku další instrukcí, a v poslední řadě se zjistí, zdali je spuštěno krokování, či nikoliv. Pokud není spuštěno krokování, vykonávají se instrukce automaticky za sebou, v opačném případě řídí jejich vykonávání uživatel kliknutím na ikonku skoku. Volá se ústřední funkce pro běh simulátoru, tedy *getSimulation()*, která se volá přes funkci *window.setTimeout* kvůli možnosti nastavení rychlosti simulátoru a také proto, že pokud bychom volali funkci přímo, tedy bez zpoždění, docházelo by v prohlížeči k moc rychlému překladu, ne-li až ke zhroucení samotného prohlížeče při použití skokových instrukcí ve vloženém programovém kódu, které by vykonávaly funkci nekonečného cyklu. Na konci každého prováděného řádku se ještě volají funkce pro inicializaci registrů ZF a PSW, které zjistí hodnoty v registru ACC a podle situace vygenerují zápis do třídy a na webové rozhraní simulátoru.

6.3 Simulace instrukce

Aby byla zřejmá představa, jak se simulují některé instrukce, uvedu zde několik příkladů důležitých funkcí. Vesměs každá typová instrukce má kvůli roztřídění svůj skript. Některé instrukce mají velmi podobný průběh, ale jinou funkční hodnotu a tak jsou spojeny do jednoho názvu skriptu a funkce, ve které jsou pak rozlišeny podmínkami.

6.3.1 Instrukce MOV

Simulace instrukce MOV je skryta ve skriptu instrMov.js pod funkcí *instrMov()*. Základní instrukce pro kopírování hodnot se zpracovává postupně vždy podle prvního parametru, který je identifikátorem toho, co budeme u instrukce kontrolovat na straně druhého parametru.

V prvním kroku se načte element, do kterého se bude zapisovat výstup na webové rozhraní. V tomto případě jsou všechny elementy v HTML kódu pojmenovány podle parametrů, tedy až na ukazatele registrů, které mají v reálu na prvním místě zavináč a jelikož pravidla HTML nepovolují pojmenovávat elementy zavináči, je tento znak přejmenován na „at_“. Takže pokud identifikujeme v prvním znaku prvního parametru znak „@“, pak se načítá element s názvem „at_“ za kterým pokračuje název ukazatele, tedy například „at_r1“.

Další výjimka je u registrů SP, DPTR, které jsou šestnáctibitové a jejich ukazateli SPH, SPL a DPH, DPL, které jsou naopak pouze osmibitové. SP a DPTR skládáme a rozdělujeme pomocí funkcí, které vrací dolních, či horních osm bitů nebo složí z dolních a horních osmi bitů šestnáctibitové číslo. Funkce pro výběr dolních, či horních osmi bitů pracují tak, že se číslo převede do binárního formátu pomocí další funkce, která používá stejný systém převodu, jenž se učí na hodinách Číslicových počítačů. Funkce pro překlad decimálního čísla do binární podoby nám vrátí textový řetězec v podobě binárního čísla, protože textový řetězec můžeme postupně procházet jako text a jednodušeji identifikovat jednotlivé bity. Funkce pro návrat dolních, či horních osmi bitů pak na číslo v binární podobě jednoduše aplikuje funkci *substr()*, která vrátí část textového řetězce. V posledním kroku máme osmibitové číslo v binární podobě, takže si zavoláme o přeložení čísla z binární do decimální podoby.

Jelikož je registr ACC ústředním komunikátorem pro práci s ostatními registry, je jasné, že bude mít nejvíce kombinací, ale také jeho zápis do webového rozhraní je

vůči ostatním zápisům atypický. Všechny ostatní řádky mají vesměs stejný formát ve tvaru „hexadecimální_číslo (dekadické_číslo)“, kdežto registr ACC má tvar „hexadecimální_číslo (dekadické_číslo) (binární_číslo)“. Díky této problematice má registr ACC vždy pouze své identifikační místo a je rozlišen zápis do webového rozhraní zvlášť pro tento registr a ostatní.

Přesouvaná, respektive kopírovaná hodnota se získává ze třídy *upc* tak, že jednotlivé registry jsou uloženy v poli, kde jejich ukazatel má stejný název jako registr, proto když chceme získat hodnotu uloženou například v registru r1, stačí zavolat *upc.variable[“r1“]*. To ovšem neplatí pro SPH, SPL, DPH, DPL a adresy paměti. Díky jednotným příznakům lze určit, že získáme hodnotu registru výše popsaným způsobem pro registr ACC a některý z registrů R, v opačném případě se rozdělují identifikace do několika podmínek podle přístupu k SP, či DPTR a paměti.

Registry SPH, SPL, DPH a DPL se identifikují v podmínce podle celého názvu parametru nebo se identifikuje, zdali se jedná o adresu paměti. Adresa paměti se zjišťuje tak, že máme ve validátoru určeno, aby všechny adresy vracel jako číslo, kde na první místo bude přidáno písmeno „m“. Takže pokud nalezneme na prvním místě druhého parametru písmeno „m“, jedná se o adresu paměti. Další podmínkou se zjišťuje, zdali není druhý parametr nepřímým ukazatelem do paměti, v takovém případě má parametr na prvním místě znak „@“. Podle situace se pak volá třída *upc* a z určené adresy se získá uložená hodnota.

Na konci každého přepisu registru, či adresy paměti se provádí zápis do webového rozhraní. U registrů je zápis jednoduchý, protože každý registr má svůj element téhož jména, respektive až na nepřímé ukazatele. Paměť je ale jinak strukturována, protože se ve webovém rozhraní zobrazuje pouze 256 adres a v nastavení webového simulátoru je možnost tento výpis měnit. Proto existují dvě proměnné, do kterých se ukládá aktuální rozsah paměti ve webovém rozhraní v podobě první a poslední adresy. Při práci s adresami se pak zjišťuje, zdali je adresa, do které se má zapisovat, v rozsahu první a poslední uložené adresy a pokud ano, pak dojde k zápisu na webové rozhraní, v opačném případě se zápis přeskočí. Pokud by se nezjišťovalo, zdali je adresa v zápisu a má-li vypsáno v rozhraní paměti svůj element, hlásil by prohlížeč chyby a JavaScript by nemusel pracovat správně. Také nelze vypsat všech 65536 adres, neboť při opakovaném načítání, přepisovaní a nulování webového rozhraní by uživatel čekal na nahrání webové stránky příliš dlouho.

6.3.2 Instrukce IN a OUT

U vstupní instrukce lze dobře vysvětlit, jak se identifikují akce ve webovém rozhraní. Při identifikování například tlačítek, zdali byla stisknuta, je velký výběr možností, jak tak učinit. V našem případě se stisknuté tlačítko zjišťuje pomocí třídy elementu. Při každém stisknutí levého tlačítka myši nad elementem tlačítka, dojde k přepisu třídy tlačítka. Tlačítko může nabývat dvou tříd znamenající normální a aktivovaný stav, ve kterých se liší obrázek pozadí, jenž navozuje dojem stisknutí tlačítka.

Tlačítka simulátoru jsou seřazena postupně od spodu jako jednotlivé bity od nejvyššího po nejnižší bit, proto se prochází tlačítka cyklem od nejvyššího bitu a do proměnné se ukládají jednotlivé bity v podobě čísla. Pokud se u tlačítka zjistí třída značící aktivovaný stav, přidá se do proměnné jednička, v opačném případě nula. Jakmile je prohledáno všech osm tlačítek, provede se překlad proměnné, která má výslednou podobu jednobytového čísla, jež se uloží do registru ACC v dekadické podobě a stejně jako u všech zbylých instrukcí dojde ke konci procesu k zápisu do webového rozhraní a k příslušnému konečnému volání těla simulace pro případné pokračování.

Výstupní instrukce pracuje na stejném principu, jako instrukce vstupní, ale s tím rozdílem, že se identifikuje přesný opak. Nejdříve se přeloží do binární podoby číslo z registru ACC, které se má uložit na výstup a pak se prochází cyklem. Podle toho, jaká hodnota je na místě bitu nalezena, se změní třída elementu značící ve webovém rozhraní výstup a tvoří tak dojem aktivního, či neaktivního zapojení.

6.3.3 Skokové instrukce

Skokové instrukce jsou v podstatě velmi jednoduché. Nejjednodušší skokovou instrukcí je samozřejmě JMP, který má v odpovědi validátoru tvar JMP <číslo řádku>, takže jakmile se identifikuje v těle simulace, je automaticky na místo upc.row uložena hodnota parametru u instrukce JMP a zavolá se samotné tělo simulace znovu, takže se bude v následujícím kroku vykonávat řádek, který byl určen v onom parametru.

Další skokové instrukce pracují stejně, respektive z validátoru vždy dostáváme místo parametru adresy číslo řádku, takže není samotný skok žádným problémem. Rozdíl ve skokových instrukcích je dán pouze operacemi, které předcházejí samotnému skoku, což jsou vesměs vždy podmínky.

Takové instrukce CALL a RET navíc pracují s pamětí a zásobníkem obdobným způsobem, jako instrukce MOV. Tyto operace se ve všech simulacích instrukcí často opakují, pouze s jinými příznaky. V tomto případě se volá funkce nazvaná *instrCallRet()*, ve které se podmínkami odliší provedení instrukce CALL a RET. Je-li volána instrukce CALL, pak se zvýší číslo uložené v registru SP a dojde k jeho přeložení, respektive se zjistí, zdali přesáhlo číslo šestnáctibitovou hranici, pokud ano, je od čísla odečtena hodnota 65536. V dalším kroku se rozloží adresa řádku následující instrukce, jejichž spodních osm bitů bude uloženo do paměti na adresu registru SP. Dále už se pracuje s podmínkovými zápisy do paměti a celý krok se provede ještě jednou, počínaje zvýšením registru SP, jeho přeložením, apod., ovšem tentokrát se uloží do paměti horních osm bitů. Na konci celého procesu se uloží do *upc.row* číslo řádku předané prvním parametrem. Instrukce RET pracuje přesně opačně.

6.3.4 Ostatní operace

U instrukcí ADD, ADDC a SUBB se pracuje s jednobitovým registrem C, který je ovlivněn výsledkem matematické operace. Zde například nejdříve dochází k matematické operaci a až poté dochází k přeložení čísla, jestliže přeteklo svoji povolenou hodnotu.

Instrukce, které pracují s číslem v binární podobě, si nejdříve musí jednotlivá čísla přeložit pomocí funkce *decToBin()* a poté se projde cyklem. Při procházení čísla v binární podobě se identifikují znaky 0,1 a podle příznaků se generují výsledky. V posledním kroku dojde ke zpětnému přeložení z binární podoby do decimální.

Zbylé instrukce používají vesměs jednoduché systémy identifikací a matematických operací, kde se odečítá, či přičítá, a tak nejsou nezbytnou součástí pro jejich vypsání. Je to sled ne příliš náročných podmínek, které se ovšem v některých skriptech opakují.

Do webového rozhraní se vypisuje celkový čas, po který webový simulátor pracuje. Zde se uplatní naše omezení 1µs na jednu instrukci. Čas se zapisuje do proměnných ve třídě *upc*, které obsahují rozdělení na hodiny, minuty, sekundy, milisekundy a mikrosekundy. Kvůli velice častému zapisování a překládání čísel je nejjednodušší zvyšovat základní proměnnou, od které se odvíjí ostatní výsledné hodnoty.

7. Závěrečné poznatky webového simulátoru

Webový simulátor nedisponuje takovou rychlostí jako simulátor navržen panem Ing. Tomášem Martincem, Ph. D a panem Ing. Martinem Vlasákem, a to především díky relativně pomalému zpracovávání webovým prohlížečem. Takovému jevu se nedá divit, neboť samotnému simulátoru předchází zpracování minimálně dalších tří programovacích jazyků. Pro jednoduché účely simulace ovšem postačí výsledek, jakého bylo dosáhnuto i při maximálním nastavení rychlosti, tedy nulovém zpoždění volání jednotlivých instrukcí. Ovšem pro účely delšího pracovního procesu, kdy se odečítají od čísel jednotky v poměru 2^8 kvůli zaměstnání mikroprocesoru na určitou dobu k dosažení určité prodlevy, je webový simulátor naprosto nevyhovující, neboť by uživatel čekal příliš dlouho, než by dostal výsledek, jenž by mohl být i záporný, takže by musel uživatel vystavit opravený kód a čekat další cyklus.

Z hlediska programátora webových aplikací je nutné upozornit na příšernou kompatibilitu jednotlivých prohlížečů, které si dělají, co chtějí a mnohdy se nedrží ani základních pravidel webového konsorcia W3C. Nejhůře je na tom Internet Explorer, pro který se musí vždy nacházet v psaném HTML kódu i JavaScriptu nějaký kompromis. Nekompatibilitou webových prohlížečů jsou biti především uživatelé, protože mohou přicházet o prvky, jež nelze kvůli zajištění stejného překladu u všech prohlížečů zajistit.

S uspokojením lze říci, že webový simulátor splňuje validitu jak podle anglického, tak podle českého validátoru, na což byl brán důraz, jelikož by byla ironie kázat v prohlížeči na validnost asemblerového kódu a nemít v pořádku vlastnoručně psaný programový kód v XHTML. Webový simulátor by mohl být z mého hlediska přínosem při výuce, neboť by si uživatelé nemuseli stahovat do počítače žádný program a zkoušet si ho přímo na internetu. O to menší bude přínos pro uživatele, kteří budou chtít s webovým simulátorem pracovat v režimu offline, a nebudou mít k dispozici internetové připojení. Díky validátoru v podobě PHP je také zaručeno, že se nedá webový simulátor provozovat jinými institucemi, než Technickou univerzitou v Liberci, jenž bude mít zdrojové skripty k dispozici.

Při generování chyb jsem bral důraz na co největší zastoupení chybových hlášení s možnou nápovědou, jak kód opravit, což by mohlo být pro studenty velkým přínosem při učení. Do budoucna se dá tato nápověda vylepšit s průnikem učebního textu předmětu Číslicové počítače.

Seznam použité literatury

[1] *Učební text k předmětu Číslicové počítače – Technická univerzita v Liberci, Fakulta Mechatroniky*

[2] Škultéty, *Rastislav. JavaScript – Programujeme internetové aplikace, 2. aktualizované vydání.* Brno 2004. 224 stran.

[3] *Webová příručka w3schools.com*. URL: http://www.w3schools.com/

[4] Kosek, Jiří. *PHP – Tvorba interaktivních webových aplikací.* Praha 1998. 492 stran.

[5] *PHP manuál*. URL: http://cz.php.net/manual/en/index.php

[6] Prokop, Marek. *CSS kaskádové styly pro webdesignéry.* Brno 2005. 288 stran.

[7] *Jak psát web*. URL: http://www.jakpsatweb.cz/

[8] Darie, Cristian. Brinzarea, Bogdan. Cherecheş-Toşa, Filip. Bucica, Mihai. *AJAX a PHP tvoříme interaktivní webové aplikace PROFESIONÁLNĚ.* Brno 2006. 320 stran.