VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

VIZUALIZACE PRÁCE CPU

VISUALISING CPU ACTIVITY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MARIÁN ĎURČO**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/19853/2016/xdurco00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Ďurčo Marián**
Obor: Informační technologie
Téma: **Vizualizace práce CPU**
Visualising CPU Activity
Kategorie: Operační systémy

Pokyny:

1. Seznamte se s vnitřní strukturou mikroprocesorů, a to jak čipů s mikroprogramovým řadičem, tak i procesorů s architekturou RISC. Zaměřte se zejména na způsob postupné aktivace jednotlivých funkčních bloků mikroprocesoru.
2. Navrhněte datovou strukturu pro reprezentaci bloků jednoduchého mikroprocesoru propojených RISCovou linkou zřetězeného zpracování a datovou strukturu pro popis činnosti strojových instrukcí takového procesoru.
3. Vytvořte aplikaci, která provádění jednotlivých instrukcí uvažovaného typu mikroprocesorů či jejich sekvencí vhodným způsobem vizualizuje.
4. Shrňte a diskutujte dosažené výsledky a možnosti jejich dalšího rozvoje do budoucnosti.

Literatura:

- Dandamudi, S.P.: Guide for RISC Processors: For Programmers and Engineers, Springer, 2004.
- Patterson, D.A., Hennesy, J.L.: Computer Organization and Design: The Hardware/Software Interface, Morgan Kaufmann, 2013.
- Patterson, D.A., Sequin, C.H.: RISC I: A Reduced Instruction Set VLSI Computer, In: Proc. of ISCA'81, IEEE, 1981.
- Sequin, C.H., Patterson, D.A.: Design and Implementation of RISC I, technical report, University of California, Berkley, 1982.
- Berger, A.S.: Introduction to Modern Computer Architectures, Hardware and Computer Organization, Elsevier, 2005.

Pro udělení zápočtu za první semestr je požadováno:

- První bod zadání a alespoň začátek práce na bodě druhém.

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

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

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

Vedoucí: **Vojnar Tomáš, prof. Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

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

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Táto práca má slúžiť, ako doplnok výučby na tému RISC pipeline. Samotná práca je tvorená, ako webová aplikácia. Po preskúmaní rôznych nástrojov a knižníc vhodných na túto prácu sme zvolili hlavné dve knižnice React a Redux. Vytvorené riešenie umožňuje podľa vstupu inštrukcií zobraziť inštrukčný tok v RISC pipeline a zároveň stavy registrov a pamäte. Umožňuje jednoduchým spôsobom vykonávanie prechodov medzi jednotlivými časťami vizualizácie. Na základe danej vizualizácie je možné základné pochopenie princípov RISC pipeline a jednotlivých inštrukcií asembleru.

Abstract

This thesis is intended to be a complement for learning about the RISC pipeline. Product of this thesis is a web application. After reviewing various tools and libraries suitable for this work, we have chosen two main libraries React and Redux. The created solution allows the instruction flow to be displayed in the RISC pipeline as well as states of the registers and the memory. It makes easy to perform transitions between the various parts of the visualization. This visualization allows a basic understanding of the RISC pipeline principles and also individual assembly instructions.

Kľúčové slová

RISC, vizualizácia, prúdové spracovanie inštrukcií, pipeline, asembler, webové aplikácie, SPA, React, Redux

Keywords

RISC, visualisation, pipelining, pipeline, asembler, web applications, SPA, React, Redux

Citácia

ĎURČO, Marián. *Vizualizace práce CPU*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Tomáš Vojnar, Ph.D.

Vizualizace práce CPU

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Prof. Ing. Tomáš Vojnar, Ph.D. Ďalšie informácie mi poskytol Ing. Pavel Tišnovský Ph.D. zo spoločnosti Red Hat. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Marián Ďurčo
15. mája 2017

Poďakovanie

Chcel by som sa poďakovať Prof. Ing. Tomášovi Vojnarovi, Ph.D. za jeho odbornú pomoc, rady a profesionálny prístup pri tvorbe textovej časti. Ďalej chcem poďakovať Ing. Pavlovi Tišnovskému Ph.D. zo spoločnosti Red Hat, ktorý ma viedol v implementačnej časti a za jeho cenné postrehy a poučné rady.

Obsah

Kapitola 1

Úvod

Dizajn procesorov s redukovanou inštrukčnou sadou RISC je v súčastnosti na vzostupe. Začína byť používaný v každodennom živote napríklad v mobilných telefónoch alebo autách či dokonca aj v superpočítačoch. Táto práca sa z toho dôvodu zameriava na vizualizáciu RISC-ového procesoru. Hlavné zameranie práce je na prúdové spracovanie inštrukcií pomocou klasickej RISC pipeline. Práca má za účel vysvetliť a zároveň znázorniť, akým spôsobom procesor pracuje na základe vstupu, ktorým je kód symbolických inštrukcií. Jeho využitie je hlavne zamerané na výučbu a pre lepšie pochopenie princípov a postupov, ktoré takýto procesor používa. Pre jednoduchý prístup je táto práca vytvorená ako webová aplikácia. Zobrazuje všetky stavy, v ktorých sa program v asembleri môže nachádzať, pomocou niečoho, čo by sa dalo nazvať časová os. Tiež je možné sa na tejto osi pohybovať a tým skúmať stavy samotnej vizualizácie.

Práca je rozdelená do päť na seba nadväzujúcich kapitol. V kapitole 2 porovnávame 2 dizajny, ktoré sú dnes používané, ich princípy a vzájomné rozdiely. Na konci druhej kapitoly sa venujeme popisu pipeline, ako základ pre prúdové spracovanie inštrukcií, jej výhody, ale definujeme aj možné komplikácie, ktoré môžu byť spôsobené jej použitím.

V nasledujúcej kapitole 3 preberáme problematiku takzvaných „single page applications“, ktoré sú v dnešnej dobé veľmi rozšírené. Spomenuté sú rôzne frameworky a knižnice, pomocou ktorých je takéto aplikácie možno tvoriť. Ich porovnaním chceme vysvetliť prístup, ktorý bol nakoniec v tejto práci použitý.

Kapitola 4 opisuje niektoré z už existujúcich riešení, ktoré sa zameriavajú na podobnú tému. Z uvedených existujúcich riešení sme čerpali inšpiráciu pri vytváraní vlastného programu, ktorý je súčasťou predkladanej práce. Pri vlastnom riešení sme prihliadali na dobré aj zlé vlastnosti už existujúcich riešení.

Predposledná kapitola 5 obsahuje základné informácie pre akú cieľovú skupinu je táto práca určená. Je tu popísaná inštrukčná sada, ktorá je použitá zároveň s popisom častí procesora, ako je pipeline, registre a pamäť.

Posledná kapitola 6 je už zameraná na tvorbu samotnej aplikácie. V kapitole 6 taktiež definujeme, ktoré nástroje boli použité pri jej vývoji aplikácie a z akého dôvodu. Obsahuje presnejší popis jednej z hlavných knižníc, ktorá hrá veľkú rolu v samotnej aplikácií. V kapitole 6 je vysvetlený základný princíp, na akom celá aplikácia funguje spolu s jej hlavnými časťami. Na konci kapitoly 6 je popísané, ako sa webové aplikácie testujú spolu s konkrétnym príkladom, ktorý je použitý v tejto práci.

Cieľom práce je poukázať na možnosti využitia nami vytvorenej aplikácie pre učebné účely zo zameraním sa na RISC pipeline pri vykonávaní jazyka symbolických inštrukcií.

Kapitola 2

Architektúra procesorov

Počítače sú komplexné systémy. Ako sa vieme v takom zložitom systéme zorientovať? Ako sa vysporiadať so systémami, ktoré majú tak vysokú mieru komplexnosti? Najčastejší spôsob je ten, že si vytvoríme hierarchickú štruktúru celého systému, pomocou ktorej rozdelíme komplexný systém na menšie časti. Pokračujeme v rozvetvovaní štruktúry až dovtedy, kým sa nedostaneme na úroveň, ktorá je ľahšie pochopiteľná a tiež riešiteľná. Každou nižšou úrovňou si niektoré časti zjednodušujeme či vynechávame. Táto metóda je známa pod názvom **abstrakcia**.

Túto abstrakciu je možné vidieť aj pri každodennom používaní počítača. Dobrým príkladom je používanie internetového prehliadača. Ten používa istú abstrakciu z dôvodu, aby užívateľ mal prístup k informáciám internetu aj bez znalosti ako daný prehliadač funguje. Z pohľadu technicky zručnejšieho užívateľa napríklad programátora, tento level abstrakcie závisí napríklad od toho, aký programovací jazyk využíva pre svoju prácu. Tie by mohli byť rozdelené na dve skupiny:

- Vyššie programovacie jazyky: C/C++, Java, ...
- Nižšie programovacie jazyky: Asembler

Toto základné rozdelenie je správne v prípade, že vynecháme strojový jazyk. Predpokladáme, že nikto neprogramuje v jednotkách a nulách. O level vyššie sa nachádza asembler, jazyk symbolických adries alebo inštrukcií. Tento jazyk používa inštrukčnú sadu (ISA) a odvíja sa od značky a typu procesora. ISA špecifikuje, ako procesor funguje, ako bude konkrétna inštrukcia vykonaná a čo bude jej výsledkom. Jej dizajn je možné rozdeliť na dve filozofie: RISC a CISC [7, str.3-5]. Každý s týchto dizajnov má svoje špecifiká. Jednotlivé filozofie a ich špecifickosť rozoberieme samostatne.

2.1 CISC

CISC je v skratke „Complex instruction set computing“. Ako už názov naznačuje systém používa komplexné inštrukcie. Najprv si však ujasnime, čo to znamená. Napríklad súčet dvoch celých čísel môžeme považovať za jednoduchú inštrukciu. Inštrukcia, ktorá presunie prvok z jedného pola do druhého a automaticky obe polia aktualizuje, by mohla byť považovaná za komplexnú. Táto komplexnosť bola zavedená hlavne z dôvodu uľahčenia práce programátorom pri tvorbe programu. Často opakujúce sa konštrukcie obsahujúce desiatky inštrukcií alebo komplikované výpočty boli nahradené jednou inštrukciou. Medzi veľkú výhodu spomínaného prístupu patrí, že prekladač z vyššieho programovacieho jazyka dokáže

vykonať preklad do asembleru aj pri malom úsilí. Keďže program je relatívne malý, vďaka komplexným inštrukciám, je jeho uloženie v pamäti ešte efektívnejšie a šetrnejšie. Čo v časoch keď DRAM pamäte boli veľmi drahé hralo veľkú rolu. Na druhú stranu tým, že sú inštrukcie komplikovanejšie a prepracovanejšie, ich dĺžka nie je rovnaká, čo môže spôsobiť viac stráveného času pri jej dekódovaní [6].

Adresovanie nie je viazané len na komunikáciu registra s registrom, ako je to v RISC architektúre, o ktorej si povieme viac v sekcií 2.2. Je možné pristupovať k hodnotám priamo v pamäti aj bez toho, aby boli pred tým nahraté do niektorého z registrov.

2.1.1 História

Počiatky používania CISC dizajnu začínajú v roku 1962, keď firma IBM chcela vyriešiť základný problém. Kód totiž nebol vôbec prenositeľný. Na počítači od jedného výrobcu fungoval, no na inom počítači od druhého výrobcu, už nie. Z tohoto dôvodu bol vytvorený referenčný počítač pomenovaný System/360. Ten pomocou emulácií umožňoval používať jednotnú inštrukčnú sadu.

Rokom 1970 prišiel nástup integrovaných obvodov a s nimi mikroprocesory. Jeden z prvých bol Intel 4004. O pár rokov neskôr Intel predstavil 8-bitový 8008, ktorý bol začiatkom v rade, ktorá sa stala predchodcom dnešnej rodiny x86. 8008 bol používaný vo veľkej miere či už v termináloch, tlačiarňach alebo priemyselných robotoch.

V roku 1980 výskumníci z univerzity Berkeley a IBM zistili, že prekladač používa iba malú podmnožinu inštrukcií CISC a plytvalo sa tak výkonom procesora. Uvedomili si, že vytvorenie jednoduchšieho počítača by mohlo mať za následok zvýšenie rýchlosti a zníženie jeho ceny. Táto myšlienka začala teóriu o RISC dizajne, o ktorom bude viac povedané v sekcií 2.2.

Koncom 80-tych rokov, v dobe, keď sa začalo s využívaním paralelizmu vďaka prúdovému spracovaniu inštrukcií, vzniká tu tzv. „Superskalárna architektúra“. Táto architektúra sa snaží zvýšiť výkon procesorov tým spôsobom, že umožňuje v jednom takte spracovať viacero strojových inštrukcií zároveň. V tomto má navrch RISC oproti CISC, hlavne z toho dôvodu, že obsahuje jednoduché inštrukcie. Avšak Intel ukázal, že aj tento koncept môže byť aplikovaný aj na CISC, ale samozrejme za svoju cenu.

2.1.2 Použitie dnes

V dnešnej dobe je možné vidieť prevažne procesory rodiny x86 vo väčšine osobných počítačov. Aj keď prvý mikroprocesor tejto rodiny bol predstavený v roku 1978, je obdivuhodné, že táto rodina je stále používaná, ako aj jej niektoré princípy. Medzi najznámejších výrobcov patrí AMD a Intel. Intel so svojimi viacerými najnovšími generáciami mikroprocesorov rady Intel Core i3, i5 a i7 vlastní značnú časť súčastného trhu.

Všetky tieto procesory stále patria pod CISC architektúru, aj keď v mnohých faktoroch táto architektúra oproti RISC zaostáva. Môžeme sa pýtať prečo, ju teda úplne nevyradíme a nenahradíme. Čo by sa stalo keby Intel zrazu zmenil inštrukčnú sadu? Software by bol nekompatibilný a to čo doteraz fungovalo by prestalo. Preto je potrebné zvoliť iný prístup, alternatívou je napríklad implementácia RISC funkcií do CISC procesorov. S týmto prístupom pracuje aj Intel. Má to za následok, že dnešné procesory pracujú s CISC inštrukciami, ktoré sa snažia jednu komplexnú inštrukciu rozložiť na viacero jednoduchých inštrukcií a tým celkový výpočet zrýchliť. Podobne, ako to robí RISC. Tým je umožnené vývojárom pracovať ako doteraz, ale pritom profitovať z RISC-ových techník [20].

2.2 RISC

V tejto časti si niečo povieme o RISC (Reduced instruction set computing) dizajne. Keďže aj samotná vizualizácia bude zobrazovať tento typ architektúry zameriame sa naň viac do hĺbky. Zhrnieme si jej hlavné princípy, vlastnosti, pozrieme sa na to, ako sa táto myšlienka vyvíjala a aká je situácia dnes.

2.2.1 História

Ako už bolo spomenuté v sekcií 2.1.1, dizajnéri začiatkom 80-tych rokov sa začali obzerať po jednoduchšej inštrukčnej sade. Keďže tieto sady obsahovali menšie počty inštrukcií, tým dostali názov „Reduced Instruction Set Computing“. Aj keď hlavným cieľom nebolo zmenšiť počet inštrukcií, ale skôr zmenšiť ich komplexnosť.

Všetko sa to začalo vo výskumnom centre IBM. Malý počítač, ktorého hlavným cieľom bolo kontrolovať telefónnu ústredňu sa napokon stal mikro-počítačovým dizajnom. John Cocke z IBM si všimol, že len veľmi malá časť inštrukcií už spomínaného IBM System/360 bola používaná väčšinu času a práve to malo najväčší podiel na výpočetnom čase. Cocke a jeho spolupracovníci z IBM chceli tento proces zefektívniť dosiahnutím v priemere jedného taktu procesora na inštrukciu. To sa im mohlo podariť, len za pomoci zreťazeného spracovania, o ktorom si viac povieme v sekcií 2.3. Na týchto konceptoch začali pracovať dve veľké univerzity. Kalifornská Berkeley na čele s Davidom Pattersonom a Carlom Heinrichom Séquinom, ktorí začali s projektom RISC-1. Druhou univerzitou bola Stanfordská univerzita, kde vznikol projekt MIPS na čele s Johnom Leroy Hennessy [9].

RISC-1

Hlavné ciele pre túto architektúru boli:

- Vykonať jednu inštrukciu za jeden takt procesora.
- Všetky inštrukcie majú mať rovnakú veľkosť.
- Pre prístup do pamäte sú iba inštrukcie *LOAD* a *STORE*. Ostatné operácie sú vykonávané len cez registre. Toto obmedzenie uľahčuje dizajn. Bez komplexného adresovania je tiež možné jednoduchšie reštartovať inštrukcie.
- Podpora vyšších programovacích jazykov.

Tieto pravidlá sú rozšírené aj dnes a tvoria základ RISC dizajnu.

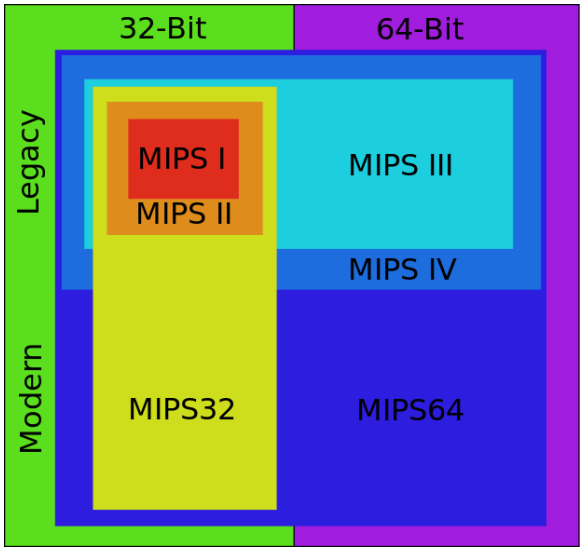
Inštrukčná sada RISC-1 obsahovala len zopár operácií (aritmetické, logické a posuvy). Tie pracovali nad registrami. Inštrukcie, dáta, adresy a tiež registre boli 32-bitové [13].

Ešte v 80-tych rokoch vyšla novšia verzia nazvaná RISC-2. Najnovšia je RISC-V. Jej cieľ znie: „Stať sa ISA štandardom pre všetky výpočetné jednotky“ [4].

MIPS

Vyvíjaný bol v svojich začiatkoch ako časť výskumného programu VLSI. Keďže Stanfordská výskumná skupina mala silné základy v prekladačoch, to ich viedlo k tvorbe architektúry, ktorá smerovala viac k hardvéru.

Rovnako bola vytvorená jednoduchá a malá inštrukčná sada. Táto sada obsahovala približne stovku inštrukcií, ktorej každá inštrukcia dokázala byť vykonaná v jednom takte procesora. Bolo používaných 32 registrov, 32-bitov dlhých.



Obr. 2.1: Diagram ukazujúci vzťahy medzi všetkými verziami inštrukčných sád MIPS[1]

Od roku 1988, keď sa MIPS začal používať do súčastnosti bolo vytvorených niekoľko inštrukčných sád. Jedným s hlavných faktorov bol aj prechod na 64-bit inštrukčnú sadu. Tieto sady sú pekne zobrazené na obrázku 2.1. Je vidieť, ako rôzne sady spolu súvisia a ktoré patria pod 32 či 64-bitovú architektúru.

POWER architektúra

Počiatky Power architektúry boli navrhnuté spoločnosťou IBM. Táto architektúra sa zameriava na vysoko výkonné RISC mikroprocesory pre pracovné stanice a servery. Medzi jej hlavné produkty patria procesory POWER a PowerPC. Aliancia AIM spoločností Apple, IBM a Motorola sa zameriavala na upravené verzie POWER procesoru pre masový trh (PowerPC). Neskôr v roku 2006 bola táto architektúra pomenovaná Power ISA, ktorá je stále používaná a v roku 2015 vyšla zatiaľ najnovšia verzia 3.0. [8].

POWER mikroprocesory majú využitie hlavne v serveroch a superpočítačoch. Séria týchto mikroprocesorov siaha až do súčastnosti. Aktuálne najnovší je POWER8, ktorý poskytuje frekvenciu až 4 GHz spolu s 12 jadrami. Jeho veľkou výhodou je aj rýchla spolupráca s dedikovanými akcelerátormi (GPU, FPGA). Už teraz je, ale plánované vydanie POWER9 v roku 2017. Mikroprocesor POWER9 by mal používať inštrukčnú sadu Power ISA v3.0., ako sme už spomínali vyššie [19].

2.2.2 Využitie v súčastnosti

K najlepšiemu pochopeniu RISC je potrebné ho považovať za koncept v dizajne procesorov. Hoci počiatočné RISC procesory mali menej inštrukcií v porovnaní s CISC, dnes je situácia odlišná. Nová generácia procesorov má stovky inštrukcií, pričom niektoré z nich sú komplexné ako CISC-ové. Takéto systémy je možné považovať za akési hybridy. Každopádne sú tu isté princípy, ktoré väčšina RISC dizajnov nasleduje. Viac sa o nich dozvieme v sekcií 2.2.3. Aj keď súčasný trend sa vyznačuje istou snahou presadiť RISC, viac do popredia

[1]Obrázok prebratý z: https://en.wikipedia.org/wiki/File:MIPS_instruction_set_family.svg

na úkor CISC, je tento prechod je veľmi komplikovaný. Hlavne z dôvodu veľkosti pokrytia Intelu na dnešnom trhu a jeho rodiny x86 v osobných počítačoch a serveroch.

Napriek týmto faktom sú RISC procesory veľmi rozšírené naprieč širokej škále platforiem. Medzi najznámejšie patria mobilné telefóny, tablety, či dokonca aj aktuálne najrýchlejší superpočítač sveta *Sunway TaihuLight*. V mobilných technológiách dominuje ARM architektúra, ktorá je používaná v Androidových systémoch, v mobilných zariadeniach od spoločnosti Apple, Windows Phone, RIM zariadeniach a tiež v hernom priemysle spoločnosťou Nintendo. Architektúra MIPS je rozšírená v herných konzolách PlayStation. Mikrokontrolér od spoločnosti Atmel rodiny AVR je použitý v konzole Xbox a tiež v autách BMW. Ako je možné vidieť použiteľnosť RISC procesorov je naozaj veľmi široká a určite má tendenciu rásť aj v budúcnosti.

2.2.3 Vlastnosti a princípy

Ako bolo naznačené v sekcií 2.2.1 ohľadne počiatočných cieľových vlastností RISC architektúry, tie sa dodnes považujú ako relevantné a základné body tohto dizajnu. Avšak tieto plus ďalšie, ktoré boli pridané neskôr je dobré si lepšie ujasniť.

Fixná inštrukčná dĺžka

Variabilná inštrukčná dĺžka môže spôsobovať implementačné a výpočetné problémy. Napríklad nie je možné určiť, či je potrebné načítať ďalší byte, po tom, čo spracujeme prvý. Spolu s fixnou dĺžkou je RISC formát inštrukcií jednoduchší.

Inštrukcie je možné rozdeliť na 4 skupiny podľa ich využiteľnosti:

- Aritmeticko-logické: jedná sa o štandardné výpočetné inštrukcie ako *ADD, AND, ...*
- Pristupujúce do pamati: typické *LOAD, STORE* inštrukcie pre prístup k dátam z/do pamäte.
- Inštrukcie vetvenia: hlavne sa jedná o volanie podprogramov a podmienené a nepodmienené skoky *CALL, JMP, RETURN, ...*
- Špecializované: ostatné inštrukcie, ktoré nepatria ani do jednej z kategórií vyššie.

Register-register operácie

Typické CISC inštrukcie umožňujú v inštrukcií použiť na mieste operandu buď konkrétny register alebo adresu v pamäti. RISC procesory používajú inštrukcie komunikujúce medzi registrami. Jediná možnosť ako pristúpiť k pamäti je cez špecializované inštrukcie *LOAD* a *STORE*. Pomocou nich sú požadované hodnoty nahrané do jedného z registrov, odkiaľ je s nimi možné ďalej pracovať.

Veľký počet registrov

Keďže sú používané výlučne operácie medzi registrami, je potrebné mať veľký počet registrov. To môže prispieť k príležitostiam pre prekladač, aby mohol ich využitie optimalizovať. Ďalšou veľkou výhodou je, že je možné minimalizovať réžiu spojenú s volaním procedúr a ich návratom. Pre zrýchlenie je použitý iný princíp, hlavne čo sa jedná predávania parametrov volanej procedúre. Miesto uloženia týchto premenných na zásobník, sú vložené

do špeciálnych registrov. Všetky registre sú rozdelené na tzv. *HIGH*, tie obsahujú parametre z volajúcej (vyššej) procedúry, *LOCAL* pre lokálne premenné a *LOW* pre parametre, ktoré majú byť predané volanej (nižšej) procedúre.

Jednoduché operácie

Cieľom je vytvorenie jednoduchých inštrukcií, z ktorých každá môže byť vykonaná za jeden takt procesora. Táto vlastnosť prispieva k jednoduchšiemu procesorovému dizajnu. Treba poznamenať, že za jednu otáčku je potrebné vykonať nahratie operandov z registrov, výpočet samotnej operácie a nakoniec uloženie výsledku do registra.

Nasleduje malá ukážka, ako by sa postupovalo pre rovnaký typ úlohy, súčet dvoch čísel z pamäte, na adresách 32 a 64 a výsledok uložiť na adresu 32 pri RISC a CISC [7, s.43-44].

CISC

ADD 32, 64

Výhoda tohoto, ako môžeme vidieť je, že kód je veľmi malý. To znamená aj menej pamäte potrebnej na uloženie kódu. Taktiež prekladač vykoná veľmi málo práce pri preklade vyššieho programovacieho jazyka do asembleru.

RISC

LOAD A, 32
LOAD B, 64
ADD A, B
STORE 32, A

Tento prístup sa môže na prvý pohľad zdať ako horšia, menej efektívna varianta, pretože je tu viac riadkov kódu a tým aj viac pamäte potrebnej na jeho uloženie. Prekladač musí vynaložiť väčšie úsilie, aby tento kód vytvoril.

Avšak RISC stratégia prináša veľmi zásadnú výhodu. Každá inštrukcia vyžaduje iba jeden takt procesora, čo v súčte zaberie menej času, ako CISC varianta. Najviac sa táto výhoda prejavý pri komplexných inštrukciách a ich väčšom počte. Taktiež nie je potrebné toľko tranzistorov, a tým sa šetrí miesto napríklad pre registre [6].

2.3 Pipeline

K dosiahnutiu cieľa vykonávať v priemere jednu inštrukciu za jeden takt procesora je možné len za použitia prúdového spracovanie inštrukcií (pipelining). Pipelining patrí k štandardu všetkých RISC a CISC procesorov. Procesor dokáže spracúvať naraz viacero inštrukcií. Východiskom tejto technológie je skutočnosť, že spracovanie inštrukcie možno rozložiť na (spravidla) päť jednoduchších úkonov, ktoré na seba nadväzujú.

Každý úkon sa vykonáva v samostatnom funkčnom bloku, prípadne sa rozloží na ešte jednoduchšie operácie a vykonáva sa v niekoľkých funkčných blokoch. Funkčný blok je skupina logických obvodov, ktoré vykonávajú spoločnú prácu (napr. aritmeticko-logická jednotka). Pri klasickom spracovaní údajov by sa ďalšia inštrukcia začala spracovávať až po ukončení spracovania predchádzajúcej inštrukcie. Pri takomto spôsobe práce by funkčné bloky väčšinu času zaháľali a čakali na príchod ďalších údajov. Prúdové spracovanie údajov

prebieha tak, že funkčné bloky sú zoradené logicky za sebou a tvoria kanál (pipe). Akonáhle spracovanie inštrukcie postúpi z prvého stupňa (funkčného bloku) do druhého, môže prísť ďalšia inštrukcia a vstúpiť do prvej fázy svojho spracovania. Tým sa dosiahne, že až na niekoľko výnimiek, sa v každom hodinovom cykle ukončí jedna inštrukcia [10].

2.3.1 Fázy prúdového spracovania

- Prenos inštrukcie z pamäte do procesora (Instruction Fetch): Inštrukcia, na ktorú odkazuje programový čítač (PC), čo je jeden z registrov je nahratá do inštrukčného registra v procesore z pamäte. Následne je hodnota v PC registri zvýšená o toľko bajtov, koľko zaberá jedna inštrukcia. Táto inštrukcia prechádza do ďalšieho taktu, kde bude dekódovaná.
- Dekódovanie inštrukcie (Instruction Decode): Inštrukcia sa nachádza v inštrukčnom registri procesora, ale ešte nevie, čo je to za inštrukciu. Preto je potrebné danú inštrukciu dekódovať. Tým že RISC inštrukcie majú fixnú dĺžku, je to pre procesor o niečo jednoduchšie.
- Vykonanie inštrukcie (Execute): Nasleduje samotné vykonanie výpočtu prípadne prebieha výpočet adresy. O to sa často stará aritmeticko-logická jednotka (ALU). Tá je zodpovedná za vykonanie logických operácií ako *AND, OR, XOR, ...* alebo prípadne aritmetických operácií *ADD, SUB, ...* V prípade skokových inštrukcií je adresa vypočítaná v tejto fáze, a bude sa ďalej s ňou pracovať až v nasledujúcej fáze.
- Prístup k pamäti (Memory access): Ak je potrebné pristúpiť k pamäti, či už pre načítanie alebo zápis hodnoty, je to vykonané v tejto fáze. Inštrukcie, ktoré k pamäti pristupujú v tejto fáze sú napríklad *LOAD, STORE*. Ostatné inštrukcie, ktorých sa prístup do pamäte netýka, v tejto fáze nevykonávajú nič a len pokračujú ďalej. Všetky skokové inštrukcie sú vykonané práve v tejto fáze.
- Zápis výsledku (Writeback): V poslednej fáze sa výsledky, ktoré boli vypočítané v tretej alebo predchádzajúcej fáze zapíšu do určených dátových registrov.

2.3.2 Dátové hazardy

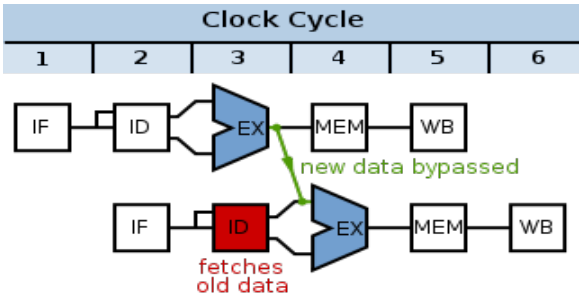
Niekedy sa môže narušiť plynulosť inštrukčného toku. Tieto situácie sa označujú pojmom (pipeline hazards). Nižšie popisujeme detailnejšie rôzne formy týchto hazardov.

Dátová závislosť

Dátová závislosť nastane, keď inštrukcia závisí na výsledku predchádzajúcej inštrukcie. Konkrétna inštrukcia potrebuje pre svoj výpočet dáta v registri, ktoré sa tam ešte nenachádzajú. Predchádzajúca inštrukcia ich tam ešte nestihla zapísať. Najlepšie je si to ukázať na príklade.

A=10, B=8
ADD A, B A=2, B=8
SUB B, A A=2, B=6

Prvá inštrukcia sčíta hodnoty z registrov A a B a výsledok uloží do registra A. Druhá inštrukcia odčíta hodnotu z registra B s hodnotou v registri A a výsledok zapíše do registra



Obr. 2.2: Riešenie dátového hazardu[2]

B. Pri použití pipeline by, ale pri druhej inštrukcií v registri A nebola zapísaná hodnota, ktorú očakávame, tj. výpočet by pracoval s hodnotou 10 namiesto 2.

Jedno z riešení je možné vidieť na obrázku 2.2.

Dáta z prvej inštrukcie sú predané priamo do výpočtu druhej, tj. nemusí sa čakať až na poslednú fázu, kde by boli dáta zapísané do registra. Je dôležité spomenúť, že tento presun dát je možné vykonať, len dopredu v čase.

Ďalšie z riešení je, že medzi tieto dve inštrukcie sa vloží jedna nezávislá inštrukcia, ktorá neovplyvní výsledok. Iným riešením je, že sa samotný výpočet oddiali vložením prázdnych inštrukcií *NOP*.

Konflikty riadenia

Nastávajú pri inštrukciách vetvenia, kedy sa môže prerušiť sekvenčné vykonávanie inštrukcií a preniesť riadenie na vzdialené miesto v programe. Do pipeline však prichádzajú inštrukcie ešte predtým, ako sa skok vykoná. Ak sa v okamihu vyhodnotenia podmienky zistí, že pipeline obsahuje inštrukcie z tej časti programu, ktorá sa nebude vykonávať, je potrebné pipeline vyprázdniť a načítať inštrukcie zo správnej vetvy. Riešením môže byť napríklad použitie prediktoru skoku alebo, ak vieme už pri programovaní určiť, ktorá vetva programu je pravdepodobnejšia, umiestnime ju za skokovú inštrukciu.

Štrukturálne konflikty

Vyskytujú sa v prípade, že viac inštrukcií súčasne požaduje rovnaký prostriedok procesora, napr. súčasný prístup ku cache pamäti. V boji o prostriedok dostáva prednosť tá inštrukcia, ktorá je viac rozpracovaná (je dlhšie v pipeline), zatiaľ čo druhá inštrukcia (a všetky, ktoré za ňou nasledujú v pipeline) musí čakať, kým sa prostriedok uvoľní.

[2]Obrázok prebratý z: https://commons.wikimedia.org/wiki/File:Data_Forwarding_(One_Stage).svg

Kapitola 3

Web a Webové aplikácie

Táto kapitola je zameraná na web a webové technológie z dôvodu, že výsledkom tejto práce je webová aplikácia. V tejto kapitole opisujeme techniky a nástroje pre tvorbu webových stránok, ktoré sú dnes rozšírené a používané.

WWW predstavuje skratku pre *The World Wide Web*. Originálne bol navrhnutý, ako interaktívny svet pre zdieľanie informácií, cez ktoré mohli ľudia komunikovať medzi sebou alebo inými strojmi. Je to teda akýsi abstraktný priestor, ktorý je tvorený stránkami obsahujúce text, obrázky, animácie, odkazy na iné stránky a mnoho ďalších prvkov. Pri vzniku webu sa týmto otvorili dvere s veľkým informačný potenciálom [5].

Dnes, keď internet je dostupný, nie len na počítačoch, ale aj mobiloch a iných mobilných zariadeniach, webové aplikácie majú veľký potenciál a využitie. Hlavným dôvodom je nezávislosť na cieľovej platforme. To je veľmi pozitívne a pohodlné z pohľadu vývojára, lebo sa môže zamerať na vývoji jednej aplikácie, namiesto vytvárania aplikácie pre každú cieľovú platformu osobitne.

Web ako taký, sa ale nezastavil. Začal sa používať aj v rôznych iných zariadeniach ako sú napríklad televízory, autá, hodinky, atď. Doteraz sa stránky tvorili na základe pevných veľkostí prvkov, napríklad bolo presne definované koľko pixelov bude mať menu alebo sa predefinovala šírka lišty. Tento prístup, ale nevyhovoval pri rôznych veľkostiach obrazoviek. Bolo potrebné meniť veľkosti prvkov na stránke podľa veľkosti obrazovky. Začalo sa používanie takzvaného „responzívneho dizajnu“, ktorý práve bral ohľad aj na mobilných používateľov [14].

3.1 Tvorba webových stránok

Na počiatku webové stránky boli zväčša textové a vytvárané výlučne pomocou jazyka HTML spolu s CSS pre rôzne štýly a úpravy kontextu. Tento prístup bol však veľmi časovo náročný hlavne pri väčších webových portáloch, kde bolo potrebné vytvoriť každú stránku zvlášť.

Neskôr sa k tomuto pridal JavaScript, ktorý dodal webu možnosť dynamicky meniť obsah. Väčšina užívateľských interakcií, ako napríklad stlačenie tlačítka pre žiadosť dát zo servera môže byť obslúžené príslušnou JavaScript-ovou funkciou, pomocou ktorej sa mohol zmeniť obsah stránky. Interakcie, ktoré pracovali s dátami často, museli byť poslané z klienta na server, kde boli ďalej spracované a potom poslané späť. Avšak väčšina tejto komunikácie bola „blokujúca“, čo znamená že klient od odoslania čakal na odpoveď a tým nemohol vykonávať inú funkciu. Príkladom je vykreslenie webovej stránky, čím degradoval

užívateľskú interakciu a celkovo klientov prehliadač pôsobil v tej chvíli, ako „zamrznutý“. Kvôli tomu boli vytvorené asynchrónne serverové volania tiež známe pod skratkou AJAX. Asynchrónne volania riešia práve tento problém, že klient aktívne nečaká na odpoveď servera, ale môže sa venovať iným záležitostiam. Po príchode odpovede sa zavolá špeciálna funkcia, ktorá dostane odpoveď zo strany servera ako parameter.

Keď prišiel AJAX do praxe, stal sa veľmi populárnym webovým frameworkom jQuery. Patrí stále jedným z najrozšírenejšie používaných frameworkov. Jedným z hlavných prvkov je vyhľadanie a vybratie príslušného elementu z DOM[1] a možnosť pripojenia udalostí k týmto prvkom. Jednoduché ovládanie je pre vývojárov veľmi pohodlné a možnosť rýchleho vývoja je neporovnateľný s použitím čistého JavaScript-u. Framework samotný je veľmi obsiahly a má rozsiahlu sadu funkcií. Dnes je používaný hlavne pri jednoduchých typoch webov, ktoré nepotrebujú, tak vysokú mieru interaktivity, prípadne medzi začiatočníkmi [2].

3.2 SPA

Asynchrónne volania nám síce umožňujú posielať si dáta medzi klientom a serverom bez nutnosti aktívne čakať na odpoveď, ale väčšina užívateľských dotazov stále potrebuje čakať na odpoveď. Z toho dôvodu používanie webových stránok v porovnaní napríklad s desktopovou aplikáciou nevyzerá, tak plynulo. Je to spôsobené mnoho komponentami, ktoré sa môžu nachádzať vo viacerých stavoch. Vykresľovanie na strane servera sa pre túto variantu veľmi komplikovane implementuje, keďže malé zmeny sa ťažko mapujú medzi URL[2] a akuálnym stavom na stránke.

Z toho dôvodu sa začali používať takzvané „Single Page Applications“. Tie sú známe svojou schopnosťou prekresliť časť webovej stránky bez toho, aby musel byť poslaný dotaz na webový server. Ako aj názov napovedá, web sa v podstate skladá z jednej stránky. Celý web je teda tvorený väčšinou z jedného JavaScript-ového súboru. Ten je zo strany servera poslaný len raz. Veľká časť logiky leží v tomto súbore a tým, že je načítaný v pamäti prehliadača možná rýchla interakcia [15]. Táto webová architektúra si získala veľkú popularitu v posledných rokoch a preto bolo vytvorených mnoho knižníc a frameworkov implementujúc tieto princípy.

3.2.1 Node.js

Skôr, ako budú si predstavíme frameworky a knižnice pre SPA, je potrebné spomenúť ich neoddeliteľnú časť, ktorá tu hrá veľkú rolu. Node je program prevažne napísaný v jazyku C++ a umožňuje spúšťať JavaScript aj bez webového prehliadača. K tomu, aby toto bolo možné, používa JavaScript-ový interpret V8. Tým je možné vytvárať v podstate akýkoľvek program v JavaScript-e, ktorý nemusí byť spúšťaný vo webovom prehliadači. Node obsahuje moduly, pomocou ktorých je možné pracovať napríklad so súborovým systémom, sieťou a ich rozhraniami alebo dátovými prúdmi. Hoci primárnym cieľom bolo vytvorenie webových serverov, ktoré by podporovali asynchrónnu technológiu, neskončilo to len pri nich. Po čase webový vývojári našli mnoho benefitov s používaním Node k nástrojom a manažmentu závislostí a vytvorili projekty, ako napríklad:

[1] Document Object Model, reprezentuje HTML dokument do stromovej štruktúry

[2] Uniform Resource Locator je reťazec znakov, ktorý slúži k presnej špecifikácií zdrojov informácií napríklad na internete

- Grunt
- Gulp
- Browserify
- Webpack

K poslednému nástroju sa ešte vrátime v sekcií 6.1. Node sa stal veľmi populárny, vývojári či organizácie napísali preň mnoho skriptov, ktoré sú zväčša zamerané na web. Je jasné, že celá komunita profituje z týchto skriptov. To, aby bolo jednoduché tieto skripty alebo aj balíčky zdielať, Node prichádza s balíčkovým manažérom zvaným Npm [11].

Npm

Najľahší spôsob, ako zdielať svoj JavaScript-ový kód medzi ostatnými vývojármi a celkovo každým, je použitie balíčkového manažéra npm. Každý linuxový operačný systém má dnes svoj vlastný balíčkovací systém. Tak isto ho má aj jazyk JavaScript alebo iné, ako napríklad Python či Ruby. Síce, každý z nich je vytvorený inak, ale podstata je rovnaká. Jednoducho, väčšinou jedným príkazom umožňuje užívateľovi stiahnuť si vybraný balíček z nejakého verejného repozitáru a zároveň si ho aj nainštalovať. Tým je možné používať kódy iných a stavať na už vytvorených kódoch svoj systém bez toho, aby bolo potrebné implementovať už existujúce riešenie. Tento prístup so sebou prináša veľký komfort či už zo strany udržovania balíčkov na aktuálnych verziách, ale aj ich prípadným odinštalovaním.

3.2.2 AngularJS

Prvý framework, ktorý si predstavíme je AngularJS. Bol vytvorený v roku 2009 spoločnosťou Google. Bol jedným z prvých, ktorý sa začal používať pri tvorbe SPA. Je používaný dodnes. Jeho aktuálna verzia Angular2, avšak pomaly upadá kvôli narastajúcej popularite ostatných knižníc alebo frameworkov.

Pracuje s HTML, ako so šablónou, ktorej syntax rozširuje, aby bolo možné s komponentami pracovať, a tým vytvárať pomocou statického HTML kódu webové aplikácie. Táto syntax je rozšírená o takzvané „direktívy“.

```
1 <p>Enter name:
2   <input type="text" ng-model="name"><br>
3   Hello <span ng-bind="name"></span>!
4 </p>
```

Výpis kódu 3.1: Príklad využitia direktívy a synchronizácie dát

Kód v 3.1 zobrazí v prehliadači vstupné textové pole pre užívateľa a pod tým jeho obsah. Vďaka dvoj-cestnej synchronizácií dát sa tento text prepisuje zároveň pri zmene textu vo vstupnom poli. Jedná sa o deklaratívny spôsob implementácie. Toto isté sa dá vykonať i za pomoci jQuery alebo čistého JavaScript-u, ale výsledný kód by bol o niečo dlhší a práve toto jedna z výhod [12].

AngularJS zjednodušuje vývoj aplikácií pomocou vyššej abstrakcie pre vývojára. Ako s každou abstrakciou prichádza aj cena a tá je v podobe flexibility. To znamená, že AngularJS nie je vhodný pre všetky typy aplikácií. Bol hlavne vytvorený pre takzvané CRUD[3]

[3]Create, Read, Update, Delete

typy aplikácií. Nehodí sa napríklad pre hry alebo grafické editory, ktoré vyžadujú častú a komplikovanú manipuláciu s DOM.

Tento framework nebol použitý pre túto prácu hlavne z dôvodu ústupu od tohto prístupu, kvôli lepším a rýchlejším alternatívam.

3.2.3 Vue.js

Jedným z ďalších frameworkov je Vue.js. Bol vytvorený v roku 2014 vývojárom Evan You, ktorý pracoval aj na už spomínanom AngularJS. Je to aj zrejme jeden z dôvodov, prečo je mu v niektorých častiach tak podobný. Celkovo je zameraný na vizuálnu časť MVC[4] modelu. Keďže tento framework je pomerne nový, autor sa snažil použiť to najlepšie s už existujúcich riešení.

- AngularJS verzia 1:
  - komplexnosť: Vue je omnoho jednoduchší, v smere API[5] a dizaju
  - dátová synchronizácia: AngularJS používa dvojsmernú synchronizáciu, pričom Vue iba jednosmernú medzi komponentami. Toto umožňuje jednoduchšiu predstavu dátového toku v aplikáciách.
  - výkon: AngularJS je pomalší z dôvodu, že pri zmenách musí táto zmena prejsť všetkými pozorovateľmi. Tí môžu spustiť inú zmenu. Tento prístup je ťažké optimalizovať. Naopak Vue používa nezávislých pozorovateľov spolu s asynchrónnou frontou. Všetky zmeny sú vykonávané nezávisle.

  V prípade AngularJS verzie 2 je situácia o niečo odlišná, kvôli rozdielnosti týchto dvoch verzií. Hlavne v rýchlosti a API. Pre viac informácií viz. [16].

- React:
  - využívanie virtuálneho DOM
  - poskytovanie reaktívnych komponentov
  - pozornosť je upriamená na hlavnú knihovňu, ostatné okrajové funkcie sú dodané pomocou ďalších knižníc
  - rýchlosť hrá v prospech Vue, kde React zaostáva kvôli implementácií virtuálneho DOM, taktiež pri opakovanom vykresľovaní nižších komponentov sa zakaždým znovu vykreslí celý podstrom, aj keď to nie je nevyhnutné. Rýchlostný test je zobrazený v tabuľke 3.1. Tento test je dostupný online[6].

Tabuľka 3.1: Tabuľka popisujúca rozdiely v rýchlosti vykreslovania pre React a Vue

	Vue	**React**
Najrýchlejšie	23ms	63ms
Medián	42ms	81ms
Priemer	51ms	94ms
Najpomalšie	343ms	453ms

[4] Model view controller
[5] Application programming interface
[6] https://github.com/chrisvfritz/vue-render-performance-comparisons

Deklaratívne vykreslovanie je základom tohto frameworku.

```
<div id="app">
  {{ message }}
</div>
```

Výpis kódu 3.2: HTML dokument pre Vue aplikáciu

```
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

Výpis kódu 3.3: JavaScript-ový Vue kód

Tento kód v 3.2 definuje jeden DIV element obsahujúci šablónu. Následne vo výpise kódu 3.3 sa na tento DIV element odkazuje pomocou identifikátora *app*, kde potom do šablóny vložíme požadované dáta. Ako výsledok je v prehliadači vidieť „Hello Vue!“ v príslušnom DIV elemente. Dáta a DOM sú „reaktívne“, to znamená, že pri zmene dát sa následne zmení aj DOM, v tomto prípade daný text [17].

Tento framework má veľký potenciál a to sa prejavuje aj v jeho stále väčšej popularite medzi vývojármi, avšak je to stále pomerne nová technológia. To je jedným z hlavných dôvodov nepoužitia spomínaného frameworku pre túto prácu. Diagram zobrazujúci popularitu webových nástrojov obrázok 3.1.

3.2.4 React

Táto knižnica bola vytvorená v roku 2013 spoločnosťou Facebook. Ďalej v tejto práci je v niektorých častiach použitá špecifikácia JavaScript-u ECMAScript 2015 (ES6)[7].

React je JavaScript-ová knižnica pre reaktívne vykresľovanie. Tento prístup oddeľuje stav od vizuálnej časti. Stav sú interné dáta, ktoré definujú aplikáciu v daný okamih. V akom stave sa práve nachádza aplikácia, to je zobrazené v DOM elementoch. Pri zmene tohto stavu sa automaticky zmení aj DOM. Tým je možné deklaratívnou cestou povedať, ako sa majú jednotlivé komponenty správať a vyzerať, keď sa zmenia ich dáta. Tento prístup sa nazýva *Reaktívne vykresľovanie*. Ako bolo už spomínané v sekcií 3.2.3 React využíva virtuálny DOM k tomu, aby nebolo potrebné vykresľovať celý reálny DOM pri každej zmene. Zmeny v reálnom DOM sú totiž veľmi pomalé a preto hlavná výhoda virtuálneho DOM je, že práca s ním je rýchla a efektívnejšia, na rozdiel od spomínaného reálneho DOM elementu. Pri zmene stavu sa teda rýchlo vykoná porovnanie a zvolí sa najefektívnejšia cesta, ako zmeniť reálny DOM, za čo najmenšiu cenu.

Táto knižnica je vybratá pre túto prácu hlavne z dôvodu zaujímavého konceptu. Jednak sa jedná použitie jazyka JSX, aj keď to nie je zásada, ale vo veľkej miere to uľahčí pracné vytváranie HTML elementov. Umožňuje totiž písať HTML kód v JavaScript-e, tým možnosť vytvárať takzvané „komponenty“. Tento prístup rieši „oddelenie zodpovednosti“[8] pri vytváraní webových aplikácií. V porovnaní s prístupom, ktorý je skôr oddelenie technológií než zodpovedností. Myslí sa tým prístup, kde HTML rieši štruktúru dokumentu, CSS jeho štýl a JavaScript správanie. React zlučuje dokopy logiku a formátovanie a vytvára tak diskrétny, znovu použiteľný a dobre zapúzdrený celok [3, str.2-3].

[7] Bližšie informácie tejto špecifikácií na http://es6-features.org
[8] Separation of Concerns

Tieto komponenty predstavujú stavebné kamene samotného webu. Sú flexibilné a majú možnosť byť opakovateľne použiteľné kdekoľvek, dokonca aj v inom projekte. Komponenty v React aplikácií môžu byť rôzne kombinované. Tým je možné z viacero jednoduchých komponentov vytvoriť jeden komplikovaný. React má taktiež veľkú podporu z pohľadu už existujúcich knižníc a balíčkov, ktoré pomáhajú pri práci s touto knižnicou.

Ďalej je možné vidieť, ako vyzerá triviálne „Hello world“ s knižnicou React.

HTML kód 3.4 React aplikácií, konkrétne telo vyzerá väčšinou takto. Jeden DIV element s identifikátorom a odkaz na JavaScrtip-ový kód. Ten vo výpise 3.5 obsahuje načítanie React knižnice. Ďalej nasleduje funkcia v ES6, takzvaná „arrow function“, ktorá je zaujímavá tým, že vracia JSX[9]. Táto funkcia predstavuje React komponentu. Nakoniec, len do daného DIV elementu túto komponentu vložíme a vykreslíme [3, str.4].

```
<!doctype html>
<html>
  <body>
    <div id="app"></div>
    <script type="text/javascript" src="index.js"></script>
  </body>
</html>
```

Výpis kódu 3.4: Základný html súbor pre React aplikáciu

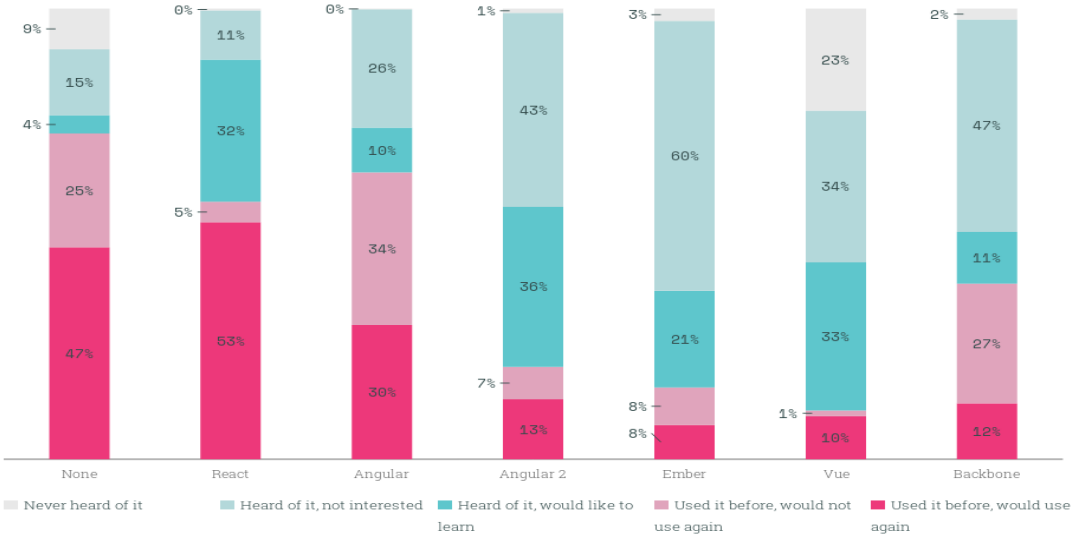
```
import React from 'react'
import ReactDOM from 'react-dom'

const Hello = () => (
  <h1>Hello World</h1>
)

ReactDOM.render(<Hello />, document.getElementById('app'))
```

Výpis kódu 3.5: JavaScript-ový React kód

Na záver tejto kapitoly sa nachádza obrázok 3.1 zobrazujúci frameworky pre webové aplikácie a ich rozšírenosť a znalosť medzi vývojármi. Je možné vidieť, že až 53% tých, čo použili React by ho použili znova a všetci už o ňom počuli. To nasvedčuje o jeho kvalitách a rozšírenosti. Pri Vue je možné si všimnúť druhé najvyššie percentuálne zastúpenie pre tých, ktorý by sa ho chceli naučiť. Vue je pomerne nový a tým až 23% vývojárov o ňom ešte nepočulo.

[9] https://jsx.github.io/

[10] Obrázok prebratý z: http://stateofjs.com/2016/frontend/



Obr. 3.1: Diagram zobrazujúci popularitu dnešných webových frameworkov[10]

Kapitola 4

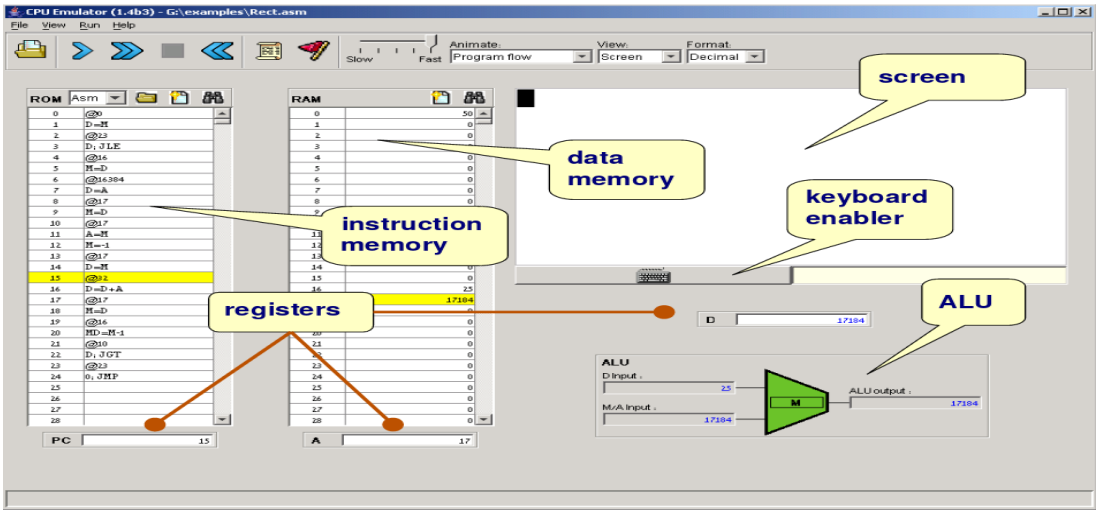
Podobné existujúce riešenia

Táto kapitola bude v krátkosti opisom už existujúcich programov, ktoré sa nejakým spôsobom zameriavajú na vizualizáciu, prípadne simuláciu procesora. Nie je najľahšie takéto programy nájsť. Buď tieto programy vlastnia univerzity na svoje vzdelávacie účely, pričom ich nezverejňujú alebo sú takéto programy proprietárne a vlastnia ich len určité firmy a spoločnosti. V takýchto prípadoch, ale už ide o veľmi presné a komplikované simulátory, simulujúce reálne existujúce, prípadne práve vyvíjané procesory.

Tie programy, čo boli dostupné, a z ktorých bola sčasti čerpali inšpiráciu, sú spomenuté nižšie.

4.1 CPUEmulator

Prvým z programov je CPUEmulator. Tento program je používaný ako výučbový, ktorý pracuje s 16-bitovým procesorom. Patrí medzi vekovo staršie programy. Hlavným zameraním je emulácia preddefinovaného počítačového systému, hlavne za účelom testovania a tvorby programov v tomto systéme.



Obr. 4.1: Obrázok popisujúci časti v CPUEmulatore[1]

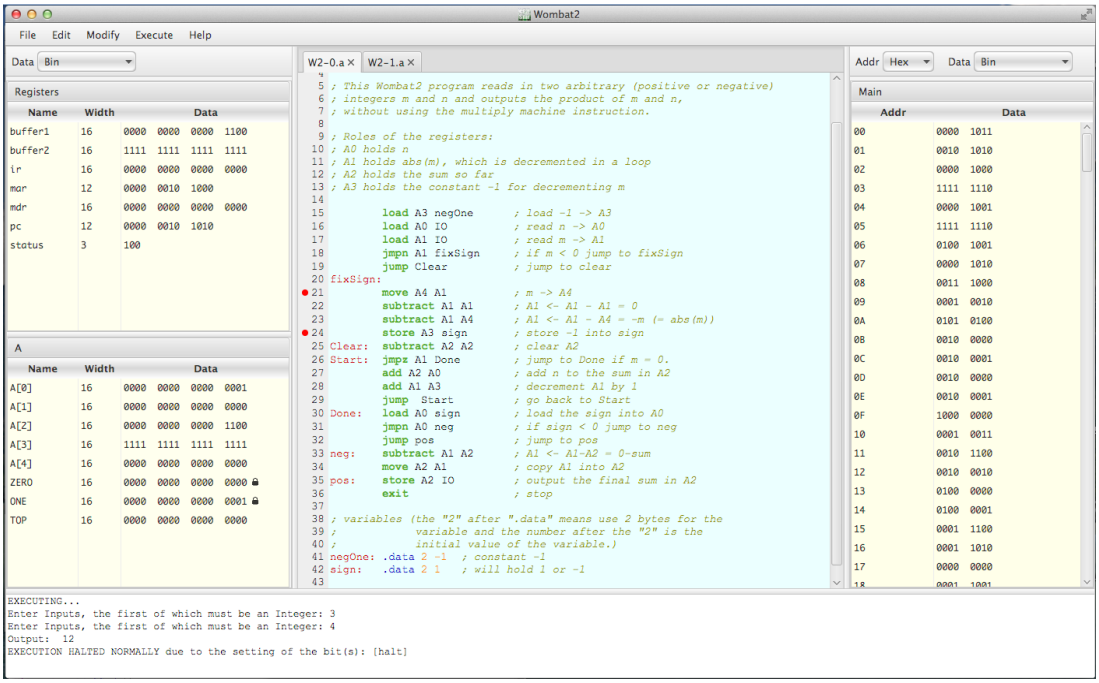
[1]Obrázok prebratý z: http://nand2tetris.org/tutorials/PDF/CPU%20Emulator%20Tutorial.pdf

Na obrázku 4.1 je možné vidieť, že CPUEmulator pracuje len s tromi registrami. Obsahuje ALU, ktorá zobrazuje aktuálny výpočet. Inštrukcie sú do programu možné vložiť osobitne zo súboru alebo ručne napísať v programe. Konkrétne príklad uvedený na obrázku 4.1 vykreslil čierny obdĺžnik v ľavom hornom rohu obrazovky.

Tým, že program je už starší, je jeho inštalácia komplikovanejšia a na niektorých systémoch sa ani nemusí podariť. Pri inštalácií je potrebné použiť nejaký emulátor alebo virtuálny stroj. Podobným problémom sme sa chceli vyhnúť pri návrhu programu pre túto prácu.

4.2 CPU Sim

Druhým programom je CPU Sim. Taktiež aj tento program je výučbový. Patrí však medzi aktuálnejšie programy. CPU Sim je aplikácie napísaná v Jave. Používateľom umožňuje simuláciu viacerých architektúr, ako napríklad RISC-ové.



Obr. 4.2: Hlavné okno programu CPU Sim[2]

Na obrázku 4.2, ktorý v strede zobrazuje časť asembleru, podľa ktorého je simulácia vykonávaná. Podobný spôsob zobrazenia sme sa rozhodli použiť aj v našom vlastnom programe. Po okrajoch je ďalej možné vidieť aktuálny stav registrov a pamäte.

Inštalácia tejto aplikácie je rozhodne jednoduchšia, ako v predchádzajúcom príklade. Stačí mať len aktuálnu verziu Javy na svojom počítači. Pohyb v simulácií je možný pomocou breakpointov nastavených v kóde, ako to býva zvyčajne v rôznych debugovacích nástrojoch. Na tento prístup sme sa aj v tejto práci snažili pozrieť z inej perspektívy a pokúsiť sa zachytiť každý stav, v ktorom sa simulácia môže vyskytovať, namiesto presne užívateľom zvolených častí. To dáva možnosť lepšie pochopiť aktuálny tok programu.

[2]Obrázok prebratý z: http://www.cs.colby.edu/djskrien/CPUSim/

Kapitola 5

Návrh

Táto kapitola sa venuje návrhu častí tejto práce, ako inštrukčnej sade, registrom a pamäti. Definujeme aj cieľovú skupinu tejto aplikácie.

Účelom našej aplikácie je vizualizovať určitú časť RISC-ového procesoru, s hlavným zameraním na prúdové spracovanie inštrukcií. Vizualizácia procesora môže byť chápaná z viacerých pohľadov a môže sa k nej pristupovať rôzne. Môže byt zameraná na nízku úroveň abstrakcie, až ku konkrétnym bitom a ich zobrazením, alebo naopak za použitia vysokej úrovne abstrakcie, napríklad pre zobrazenie virtuálnej pamäti alebo princípov vyrovnávacej pamäti.

Táto práca sa skôr zameriava na jednoduché zobrazenie RISC-ovej pipeline, dátových registrov a pamäti pri vykonávaní kódu v asembleri. Nejedná sa o simuláciu už existujúceho reálneho procesora. Princípy funkčnosti samozrejme vychádzajú z všeobecne známych poznatkov týkajúcich sa tejto témy. Používame klasickú RISC pipeline pre zreťazené spracovávanie inštrukcií, ako aj jej základné princípy. Inštrukčná sada je umelo vytvorená, len pre účely tejto aplikácie, ale je inšpirovaná z inštrukčnej sady MIPS I [1].

5.1 Cieľová skupina

Zmyslom tejto práce je formou nezameriavajúcou sa na úplné detaily v implementácií procesora vysvetliť funkčnosť prúdového spracovania inštrukcií. Tým je hlavne určená pre študentov a užívateľov, ktorí už majú určitú vedomosť o tom, ako vyzerá kód asembleru a akým spôsobom sa v ňom tvorí jednoduchý program. Toto je základná a jediná podmienka potrebná pre používanie tejto aplikácie.

Môže byť zložité v niektorých prípadoch si konkrétne predstaviť beh programu spolu s použitím prúdového spracovania. Vyskytuje sa tu istá forma paralelizmu, pričom pri kóde sme zvyknutí na sekvenčnosť, kde inštrukcia sa začne vykonávať, až keď sa úplne vykonala predchádzajúca inštrukcia. Teraz sa pomocou prúdového spracovania spracováva viacero inštrukcií naraz.

5.2 Procesor

Táto sekcia je popisom inštrukčnej sady pomocou, ktorej pracuje použitý procesor. Popísané sú aj jeho jednotlivé časti, ako samotná pipeline, registre a pamäť.

5.2.1 Inštrukčná sada

Inštrukčná sada je inšpirovaná sadou MIPS I. Z nej sme prebrali hlavne poradie a počty operandov, ktoré pre jednotlivé inštrukcie sú potrebné. V tabuľke 5.1 sa nachádza použitá inštrukčná sada.

Inštrukcie sú rozdelené do 4 skupín podľa ich použitia. V tabuľke 5.1 sú tieto skupiny od seba rozdelené dvoma horizontálnymi čiarami. V prvej časti sa nachádzajú aritmeticko-logické inštrukcie. Všetky pracujú s troma operandmi, ktoré sú dátové registre. Prvý operand je miesto, kam sa zapíše výsledok inštrukcie vypočítaný zo zvyšných dvoch registrov.

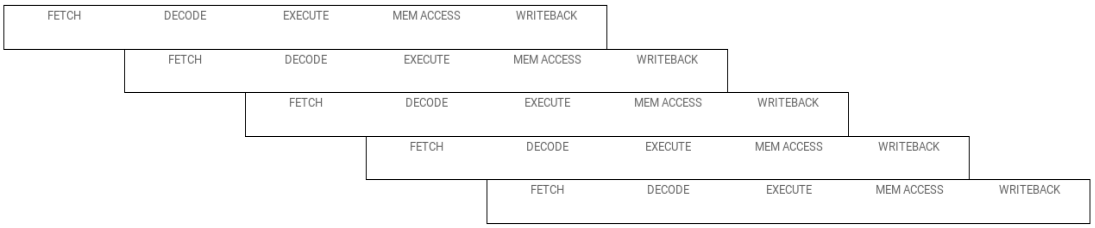
Ďalšia časť obsahuje inštrukcie skokové. Tie sa delia ešte na dve podskupiny a to podmienené a nepodmienené. Medzi nepodmienená patrí iba inštrukcia JMP, ktorá prijíma iba jeden operand. Tento operand je cieľ skoku, čo v tejto aplikácií reprezentuje poradové číslo riadku, z textového editoru. Je možné skákať aj na prázdne riadky, ale po skoku sa vykonávanie presunie na inštrukciu, ktorá sa nachádza pod týmto riadkom najbližšie. Pri podmienených skokoch je podmienka medzi prvými dvoma operandmi, ak táto podmienka je pravdivá, skok je vykonávaný na riadok určený tretím operandom.

Inštrukcie tretej skupiny sú špecifické, lebo zapisujú alebo čítajú dáta z pamäte procesora. Pre adresovanie pamäti sa používa konštanta označujúca pamäťovú bunku procesora.

V poslednej skupine sú špecializované inštrukcie. Medzi špecializované inštrukcie patrí aj MOV. Táto inštrukcia má za úlohu priradenie číselnej konštanty do pamäte procesora. Je vytvorená, len pre účely tejto vizualizáce. Takýto typ inštrukcie sa v sade MIPS nevyskytuje, ale bez nej by nebolo možné nejakým spôsobom dostať požadované hodnoty do pamäte skrz kód.

5.2.2 Pipeline

Pipeline je hlavným prvkom vizualizácie a aj samotného procesora. Môže byť zobrazovaná viacerými spôsobmi. V tejto práci je použitá varianta pipeline, ktorá obsahuje päť fáz. Tým je pre zobrazenie vhodné použiť päť riadkov, pričom každý z nich je odsadený o jednu fázu, pre lepší vizuálny dojem toku inštrukcií. Takýto typ pipeline je možné vidieť na obrázku 5.1. Inštrukcie sú vo vizualizácií vkladané do pipeline zdola nahor. Inštrukcia sa



Obr. 5.1: Klasická vizuálna reprezentácia RISC pipeline s piatimi fázami

hýbe smerom nahor, ako sa vykonávajú jej jednotlivé fázy, až kým nie je po poslednej fáze z pipeline odstránená. Po vykonaní skokovej inštrukcie sa všetky inštrukcie z pipeline zahodia a následne sa do nej vloží prvá inštrukcia, na ktorú bol skok vykonaný.

5.2.3 Registre a pamäť

Podporované sú dátové a stavové typy registrov. Medzi stavovými registrami je potrebný, len PC register, ktorý obsahuje odkaz na poslednú inštrukciu, ktorá sa začala spracovávať.

Tabuľka 5.1: Tabuľka popisujúca inštrukčnú sadu, ktorá je použitá vo webovej aplikácií. Stručnejšiu verziu je možné vidieť aj na samotnom webe.

Syntax	**Operácia**	**Popis**
ADD Ra Rb Rc	Ra = Rb + Rc	Súčet
SUB Ra Rb Rc	Ra = Rb - Rc	Rozdiel
AND Ra Rb Rc	Ra = Rb & Rc	Bitový súčin
OR Ra Rb Rc	Ra = Rb | Rc	Bitový súčet
NOR Ra Rb Rc	Ra = ~(Rb | Rc)	Negovaný bitový súčet
XOR Ra Rb Rc	Ra = Rb ^ Rc	Bitová neekvivalencia
SHIFTL Ra Rb Rc	Ra = Rb « Rc	Aritmetický posun vľavo o Rc bitov
SHIFTR Ra Rb Rc	Ra = Rb » Rc	Aritmetický posun vpravo o Rc bitov
JMP X	skok na X	Nepodmienený skok na riadok v editore, X = register/konštanta
BEQ Ra Rb X	ak Ra == Rb skok na X	Podmienený skok, ak sú hodnoty registrov rovnaké, X = register/konštanta
BNE Ra Rb X	ak Ra != Rb skok na X	Podmienený skok, ak nie sú hodnoty registrov rovnaké, X = register/konštanta
BGT Ra Rb X	ak Ra > Rb skok na X	Podmienený skok, ak hodnota Ra je väčšia, ako Rb, X = register/konštanta
BLT Ra Rb X	ak Ra < Rb skok na X	Podmienený skok, ak hodnota Ra je menšia, ako Rb, X = register/konštanta
LOAD Rn Mn	Rn = Mn	Nahranie hodnoty z pamäte do registra
STORE Mn Rn	Mn = Rn	Uloženie hodnoty z registra do pamäte
NOP	-	Prázdna inštrukcia
MOV Mn N	Mn = N	Priradenie konštanty do pamäte

Tento odkaz je číslom riadku, na ktorom sa daná inštrukcia v textovom editore nachádza. Napríklad pri podmienených skokoch nie je potrebné mať ďalší stavový register pre uloženie príznaku o výsledku podmienky. Podmienka nie je totiž individuálna inštrukcia, ako napríklad inštrukcia *CMP* v niektorých inštrukčných sadách. Podmienka skoku je totiž súčasťou skokovej inštrukcie, ako bolo spomínané v sekcií 5.2.1.

Medzi ďalšie registre, ktoré je potrebné mať sú dátové. Vo vizualizácií ich je k dispozícií 16. Označenie týchto registrov je R1, R2, . . . , R16. Toto značenie sa používa v inštrukciách.

Pamäť, má v tejto vizualizácií slúžiť, iba na uchovávanie číselných konštánt. Nezameriava sa na ukladanie asembler kódu ani rôznych iných častí programu. Vkladanie už spomínaných konštánt je možné pomocou špecializovanej inštrukcie MOV, ktorá už bola spomínaná v sekcií 5.2.1. Adresovanie jednotlivých pamäťových buniek je pomocou indexu konkrétnej bunky. Vizualizácia obsahuje 32 takýchto buniek označených indexami od 0 po 31. Pri inštrukciách, ktoré prijímajú pamäťový index sa používa len daný index, ako konštanta, napríklad *MOV 0 42* priradí do pamäte na index 0 hodnotu 42.

Kapitola 6

Implementácia a testovanie

Kapitola je zameraná na aplikovanie použitých technológií a nástrojov, spolu s implementačným riešením. Nakoniec bude ukázané aj testovanie danej aplikácie.

Ako bolo už spomínané táto práca je tvorená ako webová aplikácia. Hlavným dôvodom, prečo bola práve zvolená práve táto alternatíva, sú nasledujúce výhody:

- Nezávislosť na cieľovej platforme: či sa jedná o rôzne operačné systémy, ale aj samotné zariadenia. Dnešné webové aplikácie je možné zároveň vytvárať tiež, ako mobilné aplikácie. Pre používateľa je toto veľmi výhodné, keďže môže aplikáciu používať skoro kdekoľvek.
- Žiadna inštalácia: keďže tieto aplikácie bežia na serveri ako web, nie je potrebná žiadna inštalácia na strane užívateľa. Stačí mať len internetový prehliadač a pripojenie.
- Veľká komunita: aj keď sa na prvý pohľad toto nemusí javiť, ako veľká výhoda, ale z pohľadu vývojára je to veľmi prospešné. Či už sa jedná o pomoc s nejakou chybou, alebo prístup k veľkým množstvám knižníc a nástrojov, ktoré mu uľahčia prácu.

Nielen pre tieto dôvody bol zvolený daný prístup, ale aj kvôli zaujímavému prostrediu, ktorý JavaScript, ako jazyk poskytuje. Dianie na scéne okolo webového vývoja je veľmi živé a každým dňom sa posúva rýchlymi krokmi. Toto tempo núti vývojárov vzdelávať sa, objavovať nové nástroje a prístupy.

6.1 Webpack, Babel

Existujú dva hlavné prístupy pri vkladaní externých súborov na webu. Prvý, kde všetky externé súbory (štýly, skripty, atď.) sú vkladané na každú stránku zvlášť, pričom pri väčšom počte sa tento prístup stáva veľmi ťažko spracovateľný a udržiavanie všetkých závislostí je veľmi komplikované. Druhý spôsob je zameraný na čo najmenší počet HTTP žiadostí. Používa sa iba jeden súbor (napr. „app.js“) ktorý obsahuje všetky stránky. Tým sa ale celková veľkosť potrebná pre načítanie zväčšuje.

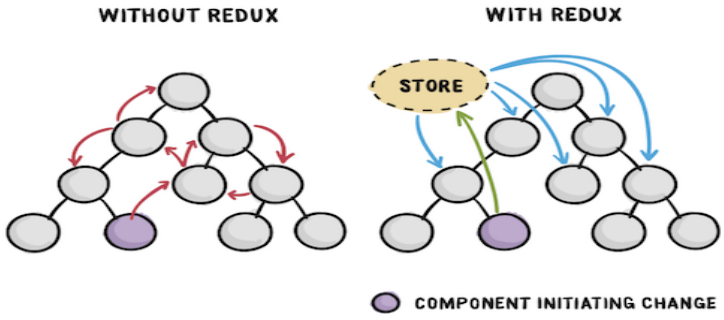
Z tohoto dôvodu je používaný Webpack. Webpack dokáže všetky štýly, obrázky, skripty spracovať a vytvoriť z nich moduly. To je umožnené pomocou takzvaných „loaderov“. Webpack pri správnej konfigurácií dokáže moduly rozkúskovať na menšie časti a tým umožní ich načítanie jednotlivo podľa potreby a nie naraz. Ďalšou veľkou výhodou je, že webpack umožňuje tvorbu a správu závislostí všetkých modulov. Táto správa sa netýka len JavaScript-u, ale môže byť použitý pre CSS, obrázky alebo štýl písma. V konečnom dôsledku pomocou

tohoto nástroja môžeme z viacerých obrázkov, štýlov, skriptov vytvoriť jeden súbor, ktorý je vložený do webu, pričom rieši za nás všetky závislosti a tiež optimalizáciu [18].

Webové aplikácie sú tvorené pre webové prehliadače, väčšina z nich dokáže vykonávať základný JavaScript-ový kód, ale čo ak chceme použiť novšiu verziu, ktorá prináša mnoho vylepšení, ale prehliadač túto verziu ešte nepodporuje? Riešením je použite JavaScript-ového prekladača. Najpoužívanejší a najznámejší je Babel. Jedná sa napríklad o preklad z verzie ES6, ktorý bol spomínaný v sekcií 3.2.4, do základného JavaScript-ového kódu, ktorý je už spustiteľný. Oba tieto nástroje sú v tejto práci používané a tvoria základ, na ktorom aplikácia funguje.

6.2 Redux

Redux je JavaScript-ová knižnica inšpirovaná architektúrou Flux[1]. Účelom je nahradiť už starší model MVC. Obrázok 6.1 zobrazuje rozdiel medzi týmito dvoma prístupmi. Naľavo je použitý MVC napravo Redux. Pri MVC je možné vidieť, že jedna zmena spôsobila spôsobila zmenu vo viacerých komponentoch. Problémom je, že po vykonaní tejto jednej zmeny mohlo napríklad nastať k zacykleniu. Takýto problém sa veľmi ťažko pre vývojára hľadá a celkovo systém sa stáva nepredvídateľným. Na druhú stranu pri redux-e je každá zmena spracovaná osobitne a prechádza cez akúsi centrálnu jednotku, ktorá tieto zmeny spracúva a upravuje ostatné komponenty. O tejto centrálnej jednotke a jej častiach bude viac povedané ďalej v tejto sekcií.



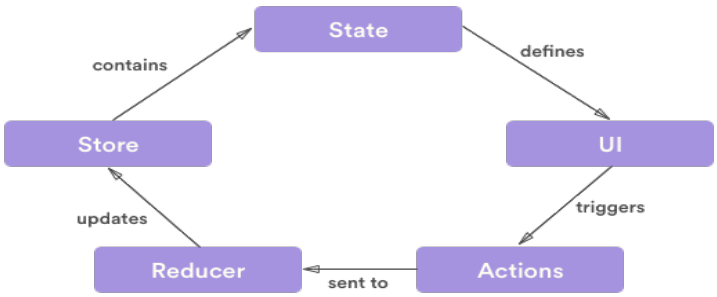
Obr. 6.1: Ako sa prejavujú zmeny pri MVC modely (vľavo) a pri použití redux knižnice (vľavo)[3]

Redux používa takzvaný „jednosmerný dátový tok“ pri zmenách. Diagram na obrázku 6.2 zobrazuje časti, z ktorých sa redux skladá a komunikáciu medzi nimi. *UI* predstavuje webovú stránku povedzme, že obsahuje tlačítko a počítadlo, ktoré zobrazuje, koľko krát bolo tlačítko stlačené. Po stlačení tlačítka sa vyvolá *akcia* inkrementuj čítač, ktorá je poslaná do takzvaného *reducera*. Ten podľa toho, že prijal akciu inkrementácia čitača, vie že má hodnotu čítača v objekte *store* zvýšiť. Tento objekt store obsahuje *stav* v akom sa aktuálna *UI* nachádza. V našom prípade obsahuje iba jednu položku, hodnotu čitača. Webová stránka zobrazuje len stav, ktorý objekt *store* definuje. Ak sa tento *stav* zmení napríklad inkerementuje počítadlo svoju hodnotu, web je znova vykreslený, už z novou hodnotou v počítadle.

[1] https://github.com/facebook/flux

[3] Obrázok prebratý z: https://www.smashingmagazine.com/wp-content/uploads/2016/06/redux-example-css-tricks-opt.pg_.png

Jednotlivé zmeny, ktoré reducer vykonáva by nemali byť mutáciou objektu store, ale upravenou kópiou tohto objektu. Je to z dôvodu, aby jednotlivé stavy boli na sebe nezávislé. To potom umožňuje aj takzvané „cestovanie časom“, kde je možné prepínať medzi stavmi, v akých sa web nachádzal a tým meniť aj samotný zobrazovaný kontext. Tento princíp je vo veľkej miere použitý aj v našej práci, viac o tom v sekcií 6.5.



Obr. 6.2: Diagram zobrazujúci Redux a jeho všetky časti[4]

6.2.1 Formát a štruktúra store objektu

Redux sme použili ako hlavnú časť „časovej osi“ pri prepínaní stavov samotnej vizualizácie, bližšie o tom bude povedané v sekcií 6.5. Teraz si priblížime samotnú štruktúra store objektu, ktorá je v tejto práci použitá.

```
const default_state = {
  pipe: [INITVAL, INITVAL, INITVAL, INITVAL, INITVAL],
  registers: {
    R1: {value: 0, order: 1, lock: 0},
    R2: ... ,
    ...,
    PC: {value: 0, order: 99, lock: 0}
  },
  memory: [0,0,0,...],
  ui: {
    mem_changes: [],
    notifications: [],
    reg_changes: [],
    state_line_msg: []
  }
}
```

Výpis kódu 6.1: Ukážka počiatočného stavu objektu pre Redux Store

V kóde 6.1 je možné vidieť, že store objekt je naozaj len obyčajný JavaScript-ový objekt. Pod kľúčom *pipe* sa nachádza pole piatich hodnôt, kde každý index predstavuje jednu pipeline fázu.

Registre sú rozdelené podľa ich názvov, pričom obsahujú objekt, ktorý hovorí o aktuálnej hodnote v danom registri, poradové číslo pre správne zobrazenie na webe a kľúč *lock*, ktorý slúži na detekciu možných dátových hazardov.

[4]Obrázok prebratý z: https://hackernoon.com/thinking-in-redux-when-all-youve-known-is-mvc-c78a74d35133

Pod posledným *ui* kľúčom sa nachádza objekt s viacerými poľami, ktoré sa starajú o vizuálne zobrazenie zmien v pamäti alebo registroch a tiež pre text popisujúci, čo sa v jednotlivých stavoch vizualizácie udialo.

6.3 Rozdelenie a použitie jednotlivých komponentov

Pri tvorbe webových aplikácií s knižnicou React je dobré rozdeliť komponenty do dvoch skupín:

1. Kontajnery:
   - sú vedomé, ako „veci pracujú“,
   - môžu obsahovať prezentačné komponenty, ako aj iné kontajnery,
   - neobsahujú žiadne štýly,
   - vedia o existencií Redux store, volajú príslušné akcie,
   - dáta a tiež funkcie sú predávané nižším komponentom.
2. Prezentačné komponenty:
   - vedia, „ako veci vyzerajú“,
   - nevedia o existencií Redux-u,
   - zobrazujú len dáta, ktoré sú im sprístupnené od vyšších komponentov,
   - sú nezávislé na aplikácií.

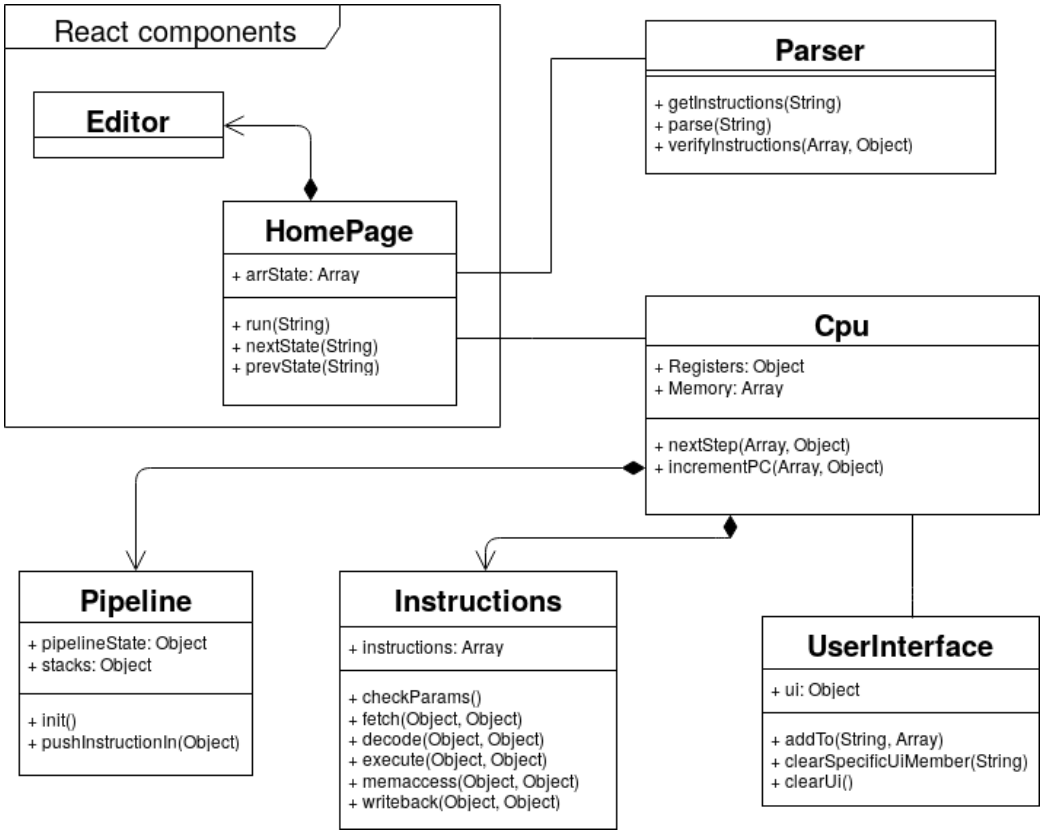
V tejto práci sú hlavné 3 kontajnery. Jedným je hlavný kontajner *App*, ktorý zastrešuje všetky ostatné kontajnery a obsahuje jeden prezentačný komponent, hlavné menu aplikácie. Dôvod je, aby rovnaké menu bolo viditeľné z každej cesty aplikácie. Kontajnery tejto aplikácie sú rozdelené podľa URL ciest, ktoré zobrazujú. Medzi ďalšie dva kontajnery patrí neplatná cesta pre zobrazenie chybovej hlášky a najhlavnejšia koreňová cesta samotnej aplikácie, kde je celá aplikácia vytvorená. Tomuto kontajneru z hlavnej časti budú venované nasledujúce sekcie a popis ich jednotlivých komponentov, ktoré spravuje.

6.4 Ako to funguje

Ako táto aplikácia funguje z pohľadu spracovania inštrukcií asembleru, až po generovanie stavov vizualizácie je zobrazené na diagrame 6.3. Tento diagram nezobrazuje celú aplikáciu, nezameriava sa na prezentačné komponenty, ale skôr na princíp, akým sa z inštrukcií z textového editora vyvárajú jednotlivé stavy vizualizácie. Bližšie popísanie jednotlivých častí diagramu 6.3 bude ďalej v tejto sekcií.

6.4.1 Textový editor

Textový editor je dôležitá časť vizualizácie. Dá sa povedať, že ovláda celú vizualizáciu, keďže podľa jeho obsahu sa všetko vykonáva. Obsahom sú inštrukcie z tabuľky 5.1. Text, ktorý obsahuje predá na požiadanie komponente HomePage, ktorá bude rozobratá v sekcií 6.4.2



Obr. 6.3: Diagram zobrazujúci časti aplikácie, ktoré sa starajú o spracovanie inštrukcií a vytvorenie všetkých stavov pre vizualizáciu

Pre implementáciu editora je použitá knižnica Ace[5], konkrétne verzia, ktorá bola upravená pre React a vytvára samostatnú komponentu. Knižnica poskytuje plnohodnotný textový editor, ktorý je plne rozšíriteľný a umožňuje nástroje od vyznačovania syntaxe, po automatické doplňanie textu.

Pri editore je možnosť predvyplnenia kódu, na základe viacerých ukážkových príkladov, ktoré môžu pomôcť užívateľovi sa lepšie zorientovať a dať mu inšpiráciu, ako vytvoriť svoj vlastný kód.

6.4.2 HomePage

HomePage je react kontainer, ktorý obsahuje celú vizualizáciu. Najdôležitejšou metódou je *run*. Tá si vyžiada aktuálny text z editoru, ten je skontrolovaný *Parserom*, či obsahuje nejaké inštrukcie a či sú napísané správne. V prípade, že všetko prebehlo v poriadku *Parser* vráti inštrukcie v špecifickom formáte pre *Cpu*. *Cpu* podľa týchto inštrukcií vygeneruje všetky stavy vizualizácie a tie predá objektu *HomePage*, ktorý si ich uloží v poli *arrState*. Po tomto kroku je vizualizácia vygenerovaná a aktuálny stav je nastavený na počiatočný. Prepínanie medzi týmito stavmi slúžia metódy *nextState* a *prevState*. Jednotlivé prvky z pola *arrState* je možné nastavovať, ako redux store objekt. Teda pre prepínanie medzi jednotlivými stavmi, v ktorých sa vizualizácia môže nachádzať. Stačí vybrať požadovaný index z pola *arrState* a prenastaviť ho do objektu store.

[5]https://ace.c9.io/

6.4.3 Parser

Užívateľský kód je pred spustením samotného generovania stavov potrebné skontrolovať. Samotná kontrola je jednoduchá a pozostáva z dvoch hlavných krokov:

- Kontrola správnosti mena inštrukcie a počtu jej operandov.
- Kontrola správnosti jednotlivých operandov.

Pri kontrole mien a počtu operandov sa prezerá každý riadok v textovom editore. Samotné inštrukcie nie sú *case-sensitive*, ale operandy ako register *R1* už je.

Kontrola jednotlivých operandov inštrukcie si vykonáva každá inštrukcia samostatne, každý objekt reprezentujúci inštrukciu totiž obsahuje funkciu pre kontrolu jej operandov. Bližšie bude implementácia jednotlivých inštrukcií priblížená v sekcií 6.4.5.

6.4.4 Cpu

Cpu predstavuje hlavnú časť, kde sa z inštrukcií tvorí samotná vizualizácia. Na začiatku sa vytvoria objekty, ktoré sú vizualizované. Sú to registre, pamäť, pipeline a userInterface. Časť z nich bude popísaná v sekcií 6.4.5. Hlavná myšlienka je, že každý z týchto objektov má svoj vlastný stav. Vo výpise 6.1 je možné vidieť počiatočné stavy týchto všetkých objektov. Tieto stavy sa postupne menia a zároveň ukladajú, tým ako inštrukcie prechádzajú cez pipeline. Stavy objektov sú menené samotnými inštrukciami, ktoré sú vykonávané. Ak napríklad inštrukcia ADD je v poslednej pipeline fáze a zapíše hodnotu výpočtu do cieľového registra, zmení tým stav objektu obsahujúci registre. Iba stav objektu *Pipeline* je menený tu v *Cpu*. Metóda *nextStep* vkladá inštrukcie do pipeline a volá metódy jednotlivých inštrukcií, o ktorých bude viac v sekcií 6.4.5. Inštrukcie do pipeline sú vkladané podľa toho, aká hodnota je v registri PC. O jeho inkrementáciu sa stará metóda *incrementPC*, ktorá prestavuje PC register aj pri skokoch. Nakoniec, keď sú spracované všetky inštrukcie, vytvorené stavy sú predané kontajneru *HomePage*.

6.4.5 Pipeline, Instructions, UserInterface

Objekt *Pipeline* obsahuje informácie o jednotlivých fázach a aké inštrukcie sa v nich nachádzajú. Metoda *init* slúži na inicializáciu pipeline, je používaná hlavne po vykonaní skokových inštrukcií. Metóda *pushInstructionIn* vkladá nové inštrukcie do pipeline a zároveň posúva ostatné inštrukcie do ďalších fáz.

Objekt *Instructions* slúži na združenie všetkých podporovaných inštrukcií. Každá z týchto inštrukcií samostatne implementuje daných 6 metód, ktoré sú v tomto objekte uvedené. V sekcií 6.4.4 bola spomenutá metóda *nextStep*, ktorá volá práve tieto metódy na základe toho, v akej fáze sa daná inštrukcia nachádza. Metóda *checkParams* slúži na kontrolu operandov, ako bolo spomenuté v sekcií 6.4.3.

Nakoniec objekt *UserInterface*, ktorý obsahuje napríklad textový popis pre každý stav vizualizácie, čo ktorá inštrukcia vykonala. Obsahuje taktiež informácie o tom, ktorý register bol v danom stave zmenený, aby sa pri tejto zmene vykonala animácia.

6.5 Zobrazovanie stavov vizualizácie na časovej osi

Vo vizualizácií sa nachádza prezentačný komponent, ktorý funguje na princípe časovej osi. Zobrazuje všetky stavy, ktoré boli vygenerované v objekte *Cpu* zo sekcie 6.4.4. Na základe

týchto zobrazených stavov je možné sa vo vizualizácií pohybovať. Tento komponent sa stará nie len o prechody medzi stavmi, ale zobrazuje aj textový opis jednotlivých prechodov, ktorý je čerpaný z objektu *UserInterface* zo sekcie 6.4.5.

Jednotlivé stavy je možné prehliadať, či už krok po kroku, alebo v rôznom poradí. Tieto všetky stavy sú uložené v objekte *HomePage*. Aktívnym stavom je ten, ktorý sa práve nachádza aj v redux store objekte. Takže pre zmenu zobrazovaného stavu stačí len prestaviť objekt store. Jeho zmenou sa upravia aj react komponenty, ktoré práve z tohto objektu dáta čerpajú.

6.6 Problémy pri skokoch a detekcia hazardov

Niektoré operandy pri skokových inštrukciách môžu byt buď konštanta alebo register. Pri konštante je možné skontrolovať jej cieľ už pri kontrole operandov, či je skok platný a daný riadok existuje. Pri použití registra je to komplikovanejšie, samotný cieľ sa dá zistiť, len pred samotným skokom, resp. pri vytváraní toho stavu. Ak by register obsahoval neplatnú skokovú adresu, vizualizácia ukončí vytváranie ďalších stavov. Posledným vytvoreným stavom sa tak stane stav, kde neplatný skok nastal.

Ďalším z možných problémov, ktorý môže nastať je, že sa program zacyklí, napríklad skokom na samého seba. Tento problém je riešený počítadlom skokov na jednotlivé riadky. Ak toto počítadlo presiahne nejakú, predom definovanú hodnotu, vizualizácia sa nedokončí a upozorní na tento fakt užívateľa chybovou správou.

Dátové hazardy, ktoré môžu nastať, síce nie sú nijak riešené, ale sú detekované. Medzi detekované typy hazardov patrí RAW[6] a taktiež riadiace hazardy pri skokových inštrukciách. Príklad RAW hazardu bol spomenutý v sekcií 2.3.2.

Detekcia hazardov prebieha tým, že ak inštrukcia bude počas svojho výpočtu meniť hodnotu dátového registra, zvýši hodnotu v objekte registra pod kľúčom *lock*. Tento objekt bol už ukázaný v kóde 6.1. Takže, ak iná inštrukcia bude chcieť pracovať s hodnotou daného registra, jednoduchým porovnaním zistí, či register už je alebo nie je používaný.

6.7 Testovanie

Pre túto prácu je použitý testovací framework Jest[7]. Hlavným z dôvodov bola veľmi jednoduchá konfigurácia a celková integrácia do aplikácie. Tá môže byť pri niektorých iných knižniciach komplikovaná a frustrujúca.

Ďalšou výhodou je veľmi jednoduchá tvorba *mock* funkcií. Mock funkcia umožňuje nahradiť nejakú existujúcu funkciu, napríklad pre žiadosť o dáta z nejakého vzdialeného servera. Táto operácia môže byť časovo náročná a preto pre testovacie účely je možné túto operáciu nahradiť falošnou. Keďže tieto dáta sú žiadané v inej funkcií a pre jej otestovanie sú tieto dáta nevyhnutné je možné použiť falošnú funkciu namiesto originálnej. Tá miesto toho aby o dáta žiadala server ich sama vytvorí a tie predá testovanej funkcií.

Ďalším užitočným nástrojom je *Snapshot* testovanie. Pri webových aplikáciach môže byť hlavne testovanie užívateľského rozhranie komplikovanejšie. Tento nástroj pracuje inak ako normálne unit testy. Namiesto vykonania nejakého kódu a porovnania jednotlivých výstupov je po prvom spustení testu výstup uložený do špeciálneho súboru. Nasledujúce spustenie testu vykoná porovnanie výstupu so špeciálnym súborom, ktorý bol pri prvom

[6] read after write

[7] https://facebook.github.io/jest/

spustení vytvorený. Test prejde, ak nastane zhoda. Ak bola vykonaná zmena v testovanom výstupe a testy z toho dôvodu neprechádzajú, je potrebné vykonať obnovenie snapshot-u.

```
test('Navigation bar snapshot', () => {
  const navigation_bar = renderer.create(
    <Nav />
  ).toJSON();
  expect(navigation_bar).toMatchSnapshot();
});
```

Výpis kódu 6.2: Snapshot test pre menu v aplikácie

Kód 6.2 vytvorí testovaciu funkciu s popisom testu ako prvý argument. Druhý argument, je funkcia, ktorá obsahuje samotný test. Funkcia *renderer.create* vytvorí z react komponenty JavaScrtip-ový objekt bez závislosti na DOM objekte. Tento objekt je serializovaný do formátu JSON. Na konci je samotné porovnanie zo *snapshotom*, ktorého ukážka je vo výpise 6.3. Tento súbor je v podstate serializovaný react komponent, ktorý bol vytvorený v teste.

```
exports[`test Navigation bar snapshot 1`] = `
<div
  className="dVZZbp">
  <a
    className="fgURVR"
    onClick={[Function]}
    style={Object {}}>
    <img
      className="jaiOvz"
      src="IMAGE_MOCK" />
    <h1
      className="jbaQLL"
      size={undefined}>
      RISC Visualization Tool
    </h1>
  </a>
</div>
`;
```

Výpis kódu 6.3: Špeciálny súbor ktorý obsahuje *snapshot* pre testovanie

Kapitola 7

Záver

V predkladanej práci sme predstavili, akým spôsobom pracuje prúdové spracovanie inštrukcií v RISC-ovej pipeline. Riešenie je vytvorené ako webová aplikácia. Rozhodli sme sa tak z dôvodu kompatibility na rôznych zariadeniach a jednoduchému prístupu bez nutnosti inštalácie. Pomocou tejto webovej aplikácie je možné vytvoriť vlastný kód v špecifickej inštrukčnej sade. Následne je tento kód spracovaný a vytvorí všetky stavy vizualizácie. Dané stavy sú vytvárané a riadené podľa pipeline. Zobrazenie týchto stavov, ako aj prechody medzi nimi je možné ovládať cez takzvanú časovú osu. Tým je zobrazený tok a poradie, v akom sa dané inštrukcie programu vykonávajú v pipeline. Okrem pipeline sú zobrazené stavy registrov a pamäte.

Práca vizualizuje len veľmi malú časť procesora, bez zamerania na konkrétne implementačné detaily, ktoré sú používané v reálnom procesore. Preto by implementácia mohla byť upravená, tak aby pracovala s rôznymi typmi procesorov. Pri rozšírení by sa zohľadnili a mohli byť použité aj ich inštrukčné sady, ktoré používajú. Tým by bolo možné vizualizovať reálny typ procesoru, čo by malo za následok, že študent alebo záujemca a o túto tematiku by mohol reálny kód spustiť a sledovať, ako je na samotnom procesore spracovávaný a vykonávaný.

Ďalšou variantou, ktorá by mohla obohatiť vizualizáciu je pridanie všeobecnej elektrotechnickej schémy, ktorá by zobrazovala pipeline. Boli by zobrazované jednotlivé spoje a komunikácia, ktorá medzi jednotlivými fázami prebieha. Keďže pipeline pracuje aj z pamäťou a registrami, bolo by možné sa zamerať bližšie aj na túto časť.

Literatúra

[1] *MIPS-I Assembly Language Instruction Set*. [Online], [cit. 2017-03-05].
URL http://www.ece.umd.edu/~manoj/759M/MIPSALM.html

[2] ALMAER, D.: *What can we learn from how jQuery symbiotically pushed the Web Platform forward?* [Online], [cit. 2017-03-11].
URL https://medium.com/ben-and-dion/what-can-we-learn-from-how-jquery-symbiotically-pushed-the-web-platform-forward-ce6b20cd4e98

[3] ANTONIO, C.: *Pro React*. Berkeley, CA: Apress, 2015, ISBN 1484212614.

[4] ARUN, T.: *RISC-V: Berkeley Hardware for Your Berkeley Software(Distribution*. [Online], [cit. 2017-02-05].
URL
https://www.bsdcan.org/2016/schedule/attachments/385_riscv_bsdcan16.pdf

[5] BERNERS, T. L.: *The World Wide Web: Past, Present and Future*. [Online], [cit. 2017-03-05].
URL https://www.w3.org/People/Berners-Lee/1996/ppf.html

[6] CHEN, G. N., Crystal; SHIMANO, K.: *RISC Architecture*. [Online], [cit. 2017-02-03].
URL https:
//cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

[7] DANDAMUDI, S. P.: *Guide for RISC processors: for programmers and engineers*. New York: Springer, 2004, ISBN 0-387-21017-2.

[8] DIEFENDORFF, K. a. E. S. (editor): *The PowerPC user instruction set architecture*, Micro, IEE, 1994, ISSN 0272-1732.

[9] ESPONDA, M.; Rojas, R.: *The RISC Concept - A Survey of Implementations*. Berlin University, [cit. 2017-02-04].
URL http://www.inf.fu-berlin.de/lehre/WS94/RA/RISC-9.html

[10] JÁNOŠÍKOVÁ, L.: *Prúdové spracovanie inštrukcií, architektúra Intel Haswell*. [Online], [cit. 2017-02-04].
URL http://frdsa.utc.sk/~janosik/Kniha/Prudove_sprac.html

[11] KLAUZINSKI, P. a. J. M.: *Mastering JavaScript Single Page Application Development*. Birmingham: Packt Publishing, 2016.

[12] MROZEK, J.: *Začínáme s AngularJS*. [Online], [cit. 2017-03-11].
URL https://www.zdrojak.cz/clanky/zaciname-s-angularjs/

[13] PATTERSON, D.; SÉQUIN, C. H.: *RISC I: A Reduced Instruction Set VLSI Computer*. [Online], [cit. 2017-02-03].
URL http://isaid.rice.edu/HennessyPatterson/RISC1.pdf

[14] SHEA, D.: *A Brief History of Web Design*. [Online], [cit. 2017-03-05].
URL https://youtu.be/G0uvjoeogwE

[15] TAKADA, M.: *Modern web applications: an overview*. [Online], [cit. 2017-03-11].
URL http://singlepageappbook.com/goal.html

[16] VUEJS: *Comparison with Other Frameworks*. [Online], [cit. 2017-03-15].
URL https://vuejs.org/v2/guide/comparison.html#Directives-vs-Components

[17] VUEJS: *Introduction*. [Online], [cit. 2017-03-15].
URL https://vuejs.org/v2/guide/index.html

[18] Webpack: *Motivation*. [Online], [cit. 2017-04-21].
URL http://webpack.github.io/docs/motivation.html

[19] WILLIAMS, C.: *Power9: Google gives Intel a chip-flip migraine, IBM tries to lures big biz*. [Online], 2016, [cit. 2017-04-13].
URL http://www.theregister.co.uk/2016/04/07/open_power_summit_power9/

[20] WONG, C.: *CISC: Complex Instruction Set Computing*. Stanford University, [cit. 2017-02-03].
URL http://slideplayer.com/slide/5905161/