VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

VIZUALIZACE ČINNOSTI VYROVNÁVACÍCH PAMĚTÍ PROCESORU

VISUALIZATION OF CPU CACHE SYSTEM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **DANIEL PEŘINA**
AUTHOR

VEDOUCÍ PRÁCE **VOJTĚCH MRÁZEK, Ing. Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Peřina Daniel**
Program: Informační technologie
Název: **Vizualizace činnosti vyrovnávacích pamětí procesoru**
Visualization of CPU Cache System
Kategorie: Uživatelská rozhraní

Zadání:

1. Seznamte se s technikami ukládání dat ve vyrovnávací paměti a s možnostmi vizualizace činnosti procesoru.
2. Zpracujte rešerši na uvedená témata.
3. Navrhněte grafický simulátor procesoru a různých typů vyrovnávací paměti, dbejte na jeho konfigurovatelnost a názornost.
4. Simulátor implementujte.
5. Pro simulovaný procesor vytvořte sadu programů, které budou demonstrovat různé vlastnosti (zejména negativní) použitých typů vyrovnávacích pamětí.
6. Vyhodnoťte vlastnosti simulátoru, zejména jeho schopnost ukázat chování pamětí.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Mrázek Vojtěch, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 29. října 2021

Abstrakt

Cílem této práce je navrhnout a implementovat simulátor činnosti vyrovnávací paměti procesoru. Protože v dnešních počítačích je řádový rozdíl mezi výkonem procesoru a hlavní paměti je nutné používat vyrovnávací paměti jako mezivrstvu. Simulátor má vizuálně demonstrovat vliv těchto pamětí na různých algoritmech a problémy, které se mohou vyskytnout při jejím nevhodném využití. Pro jednoduché použití je simulátor implementován jako webová aplikace pomocí frameworku Vue.js. Aplikace umožňuje zadat assembler kód a ten poté vykonat na různých typech vyrovnávací paměti. Aplikace vizualizuje tok dat mezi vyrovnávací a hlavní pamětí. Pro simulátor byly také vytvořeny ukázkové programy, které předvádějí různé vlastnosti a problémy vyrovnávacích pamětí. Díky této aplikaci je možné názorně předvést význam těchto pamětí.

Abstract

The goal of this work is to design and implement CPU cache simulator. In today's computers there is a difference of orders of magnitude between performance of CPUs and the main memory and thus it is necessary to use caches as an interlayer. The simulator will demonstrate effect of caches on different algorithms and problems that can occur if they are used inappropriately. For ease of use the simulator is implemented as a web application using framework Vue.js. User can enter assembly code and then execute it on several different types of caches. The application visualizes data flow between main memory and cache. Several sample programs were also created, which demonstrate various properties and problems of caches. With this application it is possible to clearly show significance of cache memory.

Klíčová slova

vyrovnávací paměť, rychlá vyrovnávací paměť, vyrovnávací paměť procesoru, paměťová hierarchie, simulátor, simulátor vyrovnávacích pamětí, vizualizace, vizualizace vyrovnávacích pamětí, assembler, webová aplikace, javascriptový frontend framework, Vue.js

Keywords

cache, cache memory, CPU cache, memory hierarchy, simulator, cache simulator, visualization, cache visualization, assembler, web appliacation, JavaScript front-end framework, Vue.js

Citace

PEŘINA, Daniel. *Vizualizace činnosti vyrovnávacích pamětí procesoru.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Vojtěch Mrázek, Ing. Ph.D.

Vizualizace činnosti vyrovnávacích pamětí procesoru

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doktora Vojtěcha Mrázka. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Daniel Peřina
5. května 2022

Poděkování

Rád bych poděkoval Ing. Vojtěchu Mrázkovi, Ph.D. za odborné vedení této práce, cenné poznámky a vstřícnost při konzultacích, které mi pomohly tuto práci zkompletovat.

Obsah

1 Úvod **2**

2 Vyrovnávací paměti v procesorech **3**
2.1 Vyrovnávací paměť s přímým mapováním 5
2.2 Vyrovnávací paměť se skupinovým asociativním mapováním 7
2.3 Vyrovnávací paměť s plně asociativním mapováním 8
2.4 Výběr oběti u vyrovnávacích pamětí s asociativním mapováním 8
2.5 Udržování koherence dat 9

3 Vizualizace činnosti procesoru a vyrovnávacích pamětí **11**
3.1 Existující simulátory činnosti procesoru a vyrovnávacích pamětí 11
3.2 Prostředky pro vizualizaci činnosti vyrovnávacích pamětí procesoru 16

4 Návrh simulátoru činnosti vyrovnávacích pamětí **21**
4.1 Návrh grafického uživatelského rozhraní aplikace 21
4.2 Návrh modelů vyrovnávací paměti 23
4.3 Návrh logického členění implementace aplikace 25
4.4 Návrh assembleru 27

5 Implementace simulátoru činnosti vyrovnávacích pamětí **29**
5.1 Komponenta překladač kódu 30
5.2 Komponenta paměť 32
5.3 Komponenta procesor 38
5.4 Pomocné komponenty 40
5.5 Testování procesoru a překladače 43

6 Ukázkové programy pro simulátor činnosti vyrovnávacích pamětí **44**
6.1 Výchozí ukázkový program 45
6.2 Zpracování prvků matice 4x4 46
6.3 Průchod lineárním spojovým seznamem 47
6.4 Opakované cykly 48

7 Závěr **49**

Literatura **50**

A Ukázkové programy pro simulátor činnosti vyrovnávacích pamětí **52**

B Obsah přiloženého paměťového média **57**

Kapitola 1

Úvod

Od začátku osmdesátých let výkon procesorů počítačů s vývojem technologií rapidně rostl. Procesory dosahovaly čím dál vyšší rychlosti čili byly schopné za jednotku času vykonat čím dál více instrukcí. Oproti tomu výkon pamětí, tedy doba nutná pro přístup k požadovaným datům a doba potřebná pro přenos těchto dat do procesoru, nerostl tak rychle. V dnešní době je několika řádový rozdíl mezi výkonem procesoru a pamětí používanou jako hlavní paměť počítače. Je nežádoucí, aby procesor při každém přístupu do paměti musel dlouho čekat než z ní dostane požadované data. Proto se využívají takzvané rychlé vyrovnávací paměti, které jsou sice dražší, ale výrazně rychlejší než hlavní paměť, a slouží tak jako mezivrstva propojující procesor a hlavní paměť.

Vyrovnávací paměti mají různé možnosti vnitřní organizace, jenž mají odlišné vlastnosti. Tyto různé druhy vyrovnávacích pamětí jsou popsány v kapitole 2 společně s algoritmy, které těmto pamětím pomáhají vykonávat svoji funkci. Pro správné pochopení funkce vyrovnávací paměti a rozdílů mezi jednotlivými způsoby organizace paměti jsou užitečné grafické simulátory činnosti vyrovnávacích pamětí procesoru. Na internetu je volně dostupných několik takovýchto simulátorů, které jsou představeny v kapitole 3.

Ačkoliv tyto simulátory detailně popisují samotnou funkci vyrovnávací paměti, neumožňují demonstrovat vliv vyrovnávací paměti na reálných algoritmech a programech. Chybí grafický simulátor, který by vizualizoval činnost vyrovnávací paměti na libovolném programu. A právě proto je cílem této práce navrhnout a implementovat grafický simulátor činnosti vyrovnávacích pamětí procesoru, který bude umožňovat zadat libovolný program zapsaný v programovacím jazyku assembler a následně tento program simulovat na různých typech vyrovnávací paměti. Simulátor bude implementován po vzoru jiných simulátorů jako webová aplikace za použití javascriptových frontend frameworků, které poskytují prostředky pro snadnější vývoj webových aplikací[1]. Návrh grafického uživatelského rozhraní aplikace, vnitřní logiky simulátoru a také použitého assembleru je popsán v kapitole 4.

Pro implementaci byl zvolen javascriptový frontend framework *Vue.js*. Samotná implementace simulátoru je popsána v kapitole 5, v níž jsou shrnuty rozhodnutí provedená během vývoje aplikace a také zobrazeny důležité nebo zajímavé sekce kódu. Nakonec byla pro tento simulátor vytvořena sada ukázkových programů, které demonstrují různé vlastnosti a problémy vyrovnávacích pamětí. Tyto programy jsou okomentovány v kapitole 6 a jejich zdrojové kódy jsou umístěny v příloze A.

[1] https://mrazekv-students.github.io/bp22_cpu_perina/

Kapitola 2

Vyrovnávací paměti v procesorech

V této kapitole je shrnuta obecná organizace paměti v moderních počítačích a blíže charakterizovány rychlé vyrovnávací paměti, které představují mezivrstvu mezi hlavní pamětí počítače a samotným procesorem. Informace obsaženy v této kapitole byly čerpány z literatury [18], [19] a [10].

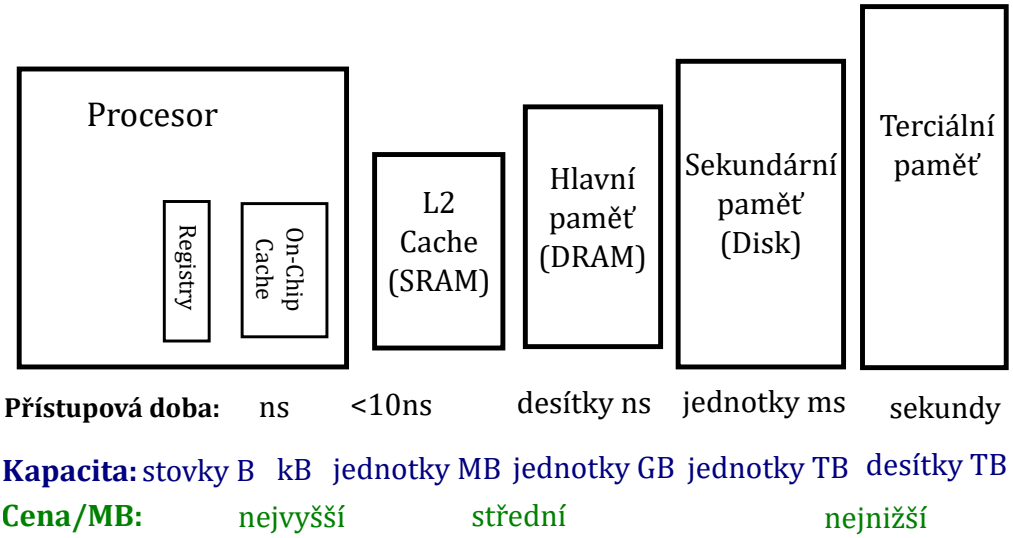
Existuje několik druhů pamětí, které mají různé vlastnosti jako je celková kapacita, přístupová doba, přenosová rychlost, poměr cena/bit a další. Obvykle platí, že čím větší má paměť kapacitu, tím menší je poměr cena/bit, ale také je delší přístupová doba. Velkokapacitní paměti jsou tudíž pomalé, zatímco rychlé paměti nemohou být z cenových důvodů příliš velké. V dnešních počítačích je řádový rozdíl mezi rychlostí procesoru a hlavní pamětí DRAM[1] [5]. Je nežádoucí, aby při každém přístupu do paměti procesor mnoho taktů čekal, než se načtou nebo uloží data. Proto se využívá takzvaná **paměťová hierarchie**.

Paměťová hierarchie je rozdělení celkové paměti počítače do několika vrstev, kde každá vrstva využívá jiné technologie. Vrstvy paměti jsou organizovány tak, že čím blíže je daná vrstva k procesoru, tím menší je její přístupová doba, ale kvůli horšímu poměru cena/bit je její celková kapacita menší. Data se pak při použití kopírují na nižší vrstvy, aby k nim bylo v budoucnu možné přistoupit rychleji. Paměťová hierarchie se obecně dělí na pět základních úrovní, jak je vyobrazeno na obrázku 2.1:

- **Registry a vyrovnávací paměť v procesoru** – Tyto paměti jsou umístěny přímo na čipu procesoru. Mají nejkratší přístupovou dobu, obvykle v řádu jednotek nanosekund, ale jejich celková kapacita je malá, pouze v řádu stovek až tisíců bytů.
- **Vyrovnávací paměť SRAM**[2] – Tato paměť je umístěna nejblíže k čipu procesoru. Obvykle je dále rozdělena na několik úrovní (značených L1, L2, L3) lišící se mimo jiné velikostí, rychlostí a sdílením mezi dvojicí jader či všemi jádry. Její přístupová doba se pohybuje v jednotkách až desítkách nanosekund a její kapacita je v rozmezí jednotek až desítek megabytů.
- **Hlavní paměť DRAM** – Tato paměť představuje hlavní operační paměť počítače. V dnešní době mají hlavní paměti kapacitu až desítky gigabytů a přístupovou dobu ve vyšších desítkách nanosekund. Společně s registry a vyrovnávací pamětí procesoru a vyrovnávací pamětí SRAM tvoří hlavní paměť DRAM primární paměť.

[1] DRAM - Dynamic Random Access Memory (dynamická paměť)
[2] SRAM - Static Random Access Memory (statická paměť)

- **Sekundární paměť** – Tato paměť slouží jako hlavní uložiště počítače. V dnešní době se pro sekundární paměť používají nejčastěji technologie HDD[3] a SSD[4]. Jejich přístupové doby bývají v řádu jednotek milisekund a kapacita dosahuje až jednotek terabytů.
- **Terciální paměť** – Tato úroveň paměti na rozdíl od předchozích úrovní nebývá přímou součástí počítače. Obvykle se jedná o externí paměti například ve formě magnetických pásek. V dnešní době lze jako terciální paměť zařadit také vzdálená datová úložiště na cloudu. Tyto paměti mají největší kapacitu a také největší přístupovou dobu, která může dosahovat až mnoha sekund.



Obrázek 2.1: Obecné schéma organizace celkové paměti počítače do paměťové hierarchie [18]. Na obrázku jsou zaznamenány přibližné přístupové doby, kapacity a poměr cena/megabyte jednotlivých úrovní paměťové hierarchie.

Správné fungování hierarchie paměti závisí na dodržení základních předpokladů, které se nazývají **principy lokality**. Existují dva principy lokality: časová lokalita a prostorová lokalita.

- **Časová lokalita** uvádí, že pokud je přistoupeno k určitému paměťovému bloku, existuje velká šance, že tento blok bude v dohledné době využit znovu. Tudíž má smysl tento paměťový blok uložit do vyrovnávací paměti blíže k procesoru.
- **Prostorová lokalita** uvádí, že pokud je přistoupeno k určitému paměťovému bloku, existuje velká šance, že v dohledné době se bude přistupovat i k jeho sousedům v paměťovém prostoru. Tudíž má smysl do vyrovnávací paměti přesunout větší část paměti, než ta, ke které se zrovna přistupuje.

Dodržování těchto principů musí být zajištěno jak technickým provedení paměťové hierarchie a správou paměti na úrovni operačního systému, tak správnými programátorskými praktikami. Při nedodržení těchto principů nebude hierarchie paměti plnit svou funkci, nebo

[3] HDD - Hard Disk Drive (pevný disk)
[4] SSD - Solid-State Drive (polovodičový disk)

bude ještě více zpomalovat přístup do paměti. Jednoduchý příklad nedodržení principů lokality na úrovni aplikačního programátora je průchod maticí. Pokud je matice v paměti uložena po řádcích, ale program ji prochází po sloupcích, tak při přístupu ke každému členu matice je nutné načíst nový paměťový blok z hlavní paměti do vyrovnávací paměti. Rychlost průchodu je tedy stejná, jako kdyby se každý prvek matice četl přímo z hlavní paměti, a navíc je ještě zpomalena o režii hierarchie paměti.

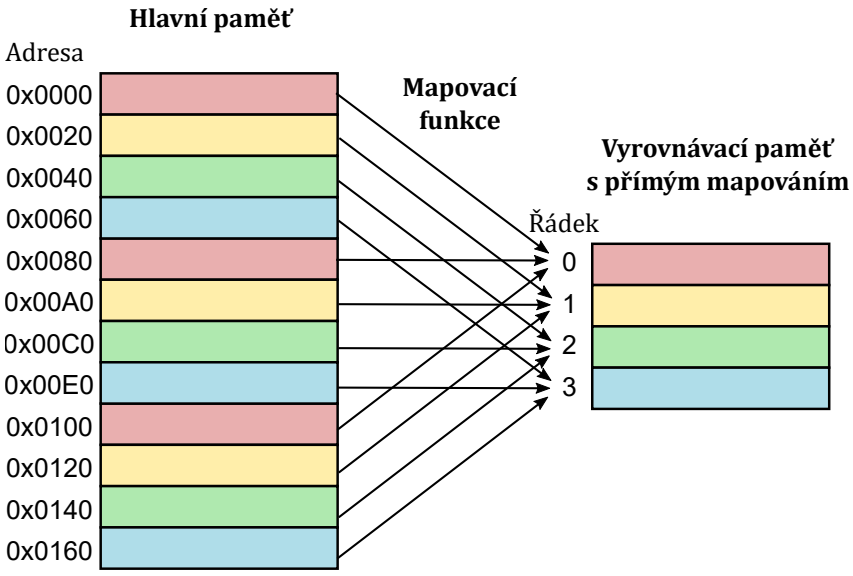
Dále se text bude zabývat pouze vyrovnávací pamětí SRAM. Vyrovnávací paměť SRAM (nebo též rychlá vyrovnávací paměť) je mezivrstva mezi hlavní pamětí a vnitřní pamětí procesoru. Je organizována do bloků o konstantní velikosti, která je obvykle stejná, jako je velikost bloku dat v hlavní paměti. Data je tak možné přesouvat blokovým přenosem. V hlavní paměti je však podstatně více paměťových bloků, než je kapacita vyrovnávací paměti, a tudíž ne všechny bloky hlavní paměti mohou být současně uloženy ve vyrovnávací paměti. Vyrovnávací paměť musí obsahovat mechanismy, které umožňují z hlavní paměti nahrát požadované bloky a nepotřebné bloky z ní odstraňovat.

Při přístupu k datům v paměti se současně s přístupem do hlavní paměti prohledá vyrovnávací paměť. Pokud se požadovaný datový blok ve vyrovnávací paměti nachází, nastává *cache hit* a řadič paměti zastaví přístup do hlavní paměti. Pokud datový blok není nalezen, nastává *cache miss* a je nutné čekat na data z hlavní paměti. Poměr mezi cache hit a cache miss se nazývá **pravděpodobnost nalezení bloku** p_{hit} (*hit rate*) a představuje základní údaj o účinnosti vyrovnávací paměti. V praxi se organizace a správa vyrovnávací paměti navrhuje tak, aby se pravděpodobnost nalezení bloku pohybovala v rozmezí 95–99%.

Z časového hlediska je podstatná **přístupová doba**. V případě, že se daný datový blok ve vyrovnávací paměti nenachází, se k přístupové době navíc přičítá **ztrátová doba** (miss penalty), což je doba potřebná k přesunutí datového bloku z hlavní paměti do vyrovnávací paměti a případně i uvolnění místa ve vyrovnávací paměti.

2.1 Vyrovnávací paměť s přímým mapováním

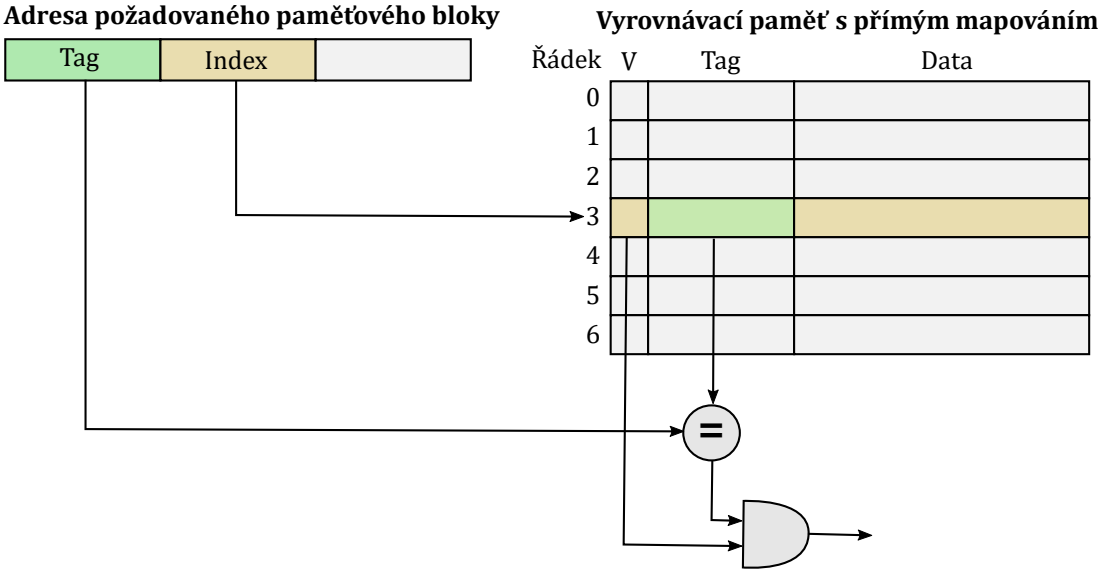
Vyrovnávací paměť s přímým mapováním je nejjednodušší a také obvykle nejméně efektivní způsob organizace vyrovnávací paměti. Je tvořena 2^n paměťovými rámci, které mají n-bitové adresy. Hlavní paměť je pak tvořena 2^m paměťovými rámci s m-bitovými adresami, kde obvykle platí, že $m > n$ (hlavní paměť má větší kapacitu než paměť vyrovnávací). Paměťové bloky v hlavní paměti je tak třeba mapovat na paměťové bloky vyrovnávací paměti pomocí takzvané **mapovací funkce** M. Tato funkce vypočte z adresy rámce v hlavní paměti jeho adresu ve vyrovnávací paměti, obvykle odstraněním nejvyšších několika bitů z adresy. Takto je adresa ve vyrovnávací paměti dána nejnižšími n bity z adresy v hlavní paměti, což ve výsledku znamená, že rámce jsou ve vyrovnávací paměti rovnoměrně rozloženy. Na obrázku 2.2 je vyobrazena vyrovnávací paměť obsahující čtyři bloky, z nichž každý má 2-bitovou adresu. Bloky hlavní paměti jsou do vyrovnávací paměti mapovány postupně, a tudíž každé čtyři bloku mohou být do vyrovnávací paměti načteny současně. Tímto způsobem je při sekvenční čtení paměti minimalizován počet konfliktů a je tedy dodržen princip prostorové lokality.



Obrázek 2.2: Princip mapování paměťových bloků do vyrovnávací paměti [10]. Datové bloky hlavní paměti jsou do vyrovnávací paměti s přímým mapováním mapovány pomocí mapovací funkce M, kterou představují šipky. Tato funkce definuje prostření dva bity adresy v hlavní paměti jako adresu ve vyrovnávací paměti (nejnižší bity slouží pro označení odsazení v rámci bloku). Tímto způsobem mapování jsou datové bloky v hlavní paměti rovnoměrně rozprostřeny a při sekvenčním průchodu paměti je dodržen princip prostorové lokality.

Ve vyrovnávací paměti je nutné poskytnout informaci, jaký datový blok hlavní paměti je na dané adrese přítomen. K tomu slouží **adresový příznak** (tag), který je tvořen zbývajícími horními bity adresy bloku v hlavní paměti. Spojením adresového příznaku a adresy bloku ve vyrovnávací paměti se tak získá kompletní adresa datového bloku v hlavní paměti. Navíc se do vyrovnávací paměti ukládá **příznak platnosti** (valid bit) pro indikaci, zda jsou data uložená na daném bloku platná neboli zda je v daném bloku uloženo něco smysluplného. Tyto dodatečné informace, které popisují data obsažená ve vyrovnávací paměti, se nazývají metadata. Způsob, jakým se z adresy požadovaného paměťového bloku získávají z vyrovnávací paměti daná data, je znázorněn na obrázku 2.3.

Hlavní nevýhoda vyrovnávací paměti s přímým mapováním je skutečnost, že dva datové bloky, které se mapují na stejnou adresu ve vyrovnávací paměti, nemohou být v paměti uloženy současně. Při každém načtení jednoho z těchto bloků je druhý blok z vyrovnávací paměti vytlačen, a při dalším přístupu k němu je nutné ho znovu načíst z hlavní paměti. To výrazně zpomaluje činnost vyrovnávací paměti. Výhodou tohoto způsobu organizace vyrovnávací paměti je její jednoduchá implementace.

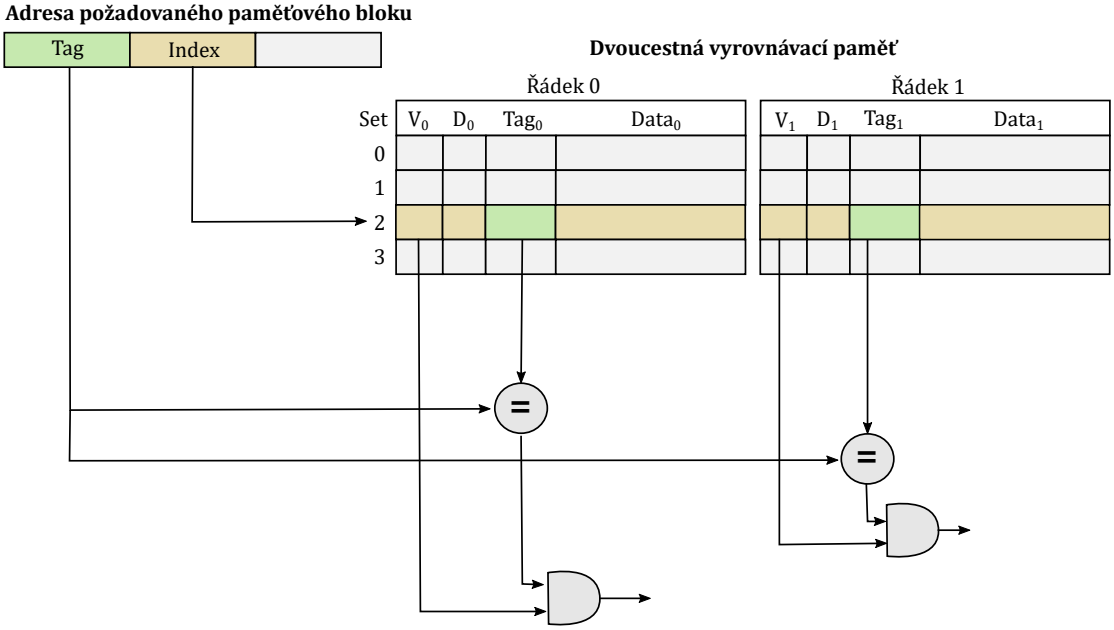


Obrázek 2.3: Struktura vyrovnávací paměti s přímým mapováním [10]. Data v paměti se vyhledávají na základě adresy ve vyrovnávací paměti (index) a adresového příznaku (tag), který blíže identifikuje původ dané položky. Pomocí příznaku platnosti (V) se určuje, zda jsou v daném bloku platná data.

2.2 Vyrovnávací paměť se skupinovým asociativním mapováním

Problém vzájemného vytlačování paměťových bloků se stejnou adresou ve vyrovnávací paměti je možné řešit zvýšením **stupně asociativity**. Vyrovnávací paměť se rozdělí do několika skupin, které mají stejný počet paměťových bloků. Každá adresa ve vyrovnávací paměti tedy označuje více bloků, což umožňuje, aby ve vyrovnávací paměti bylo uloženo několik datových bloků se stejnou adresou. Takto je možné výrazně zmenšit počet konfliktů. Jaký konkrétní datový blok je na daném bloku dané adresy vyrovnávací paměti uložen se stejně, jako u vyrovnávací paměti s přímým mapováním zjistí pomocí adresového příznaku (tag). Na obrázku 2.4 je zobrazen způsob získání požadovaných dat z vyrovnávací paměti se stupněm asociativity 2 (dvoucestná vyrovnávací paměť) na základě adresy paměťového bloku.

Stupeň asociativity rozhoduje do kolika skupin je vyrovnávací paměť rozdělena. V moderních počítačích se nejčastěji používají vyrovnávací paměti se stupněm asociativity od 4 (čtyřcestné) do 16 (šestnácticestné). Obvykle platí, že čím nižší je úroveň vyrovnávací paměti, tím menší má stupeň asociativity. Paměti s vyšším stupněm asociativity než 16 se v praxi příliš neobjevují [3]. Je nutné podotknout, že při zvýšení stupně asociativity se celková kapacita vyrovnávací paměti nezmění. Pouze je rozdělena ve více skupinách, a tudíž je kapacita jedné skupiny nižší.



Obrázek 2.4: Vyrovnávací paměť se skupinově asociativním mapováním je rozdělena do několika skupin [10]. V tomto případě je paměť rozdělena do dvou skupin neboli má stupeň asociativity 2. V paměti tudíž mohou být zároveň uloženy dva datové bloky se stejnou adresou ve vyrovnávací paměti (index). Podle adresového příznaku (tag) se pak přesně určí, který z těchto dvou bloků je požadovaný. Stejně jako u vyrovnávací paměti s přímým mapováním je i zde příznak platnosti (V), který určuje, zda jsou v daném bloku platná data.

2.3 Vyrovnávací paměť s plně asociativním mapováním

Při postupném zvyšování stupně asociativity až po maximální stupeň asociativity se vyrovnávací paměť stane plně asociativní. Paměť o velikosti n datových bloků je rozdělena na n skupin o velikosti jednoho datového bloku. Paměťové bloky z hlavní paměti je tedy možné uložit na jakoukoliv adresu vyrovnávací paměti. Adresa bloku ve vyrovnávací paměti přestává mít význam a namísto ní se paměťové bloky vyhledávají výhradně podle adresového příznaku. Tento způsob organizace vyrovnávací paměti by sice mohl být teoreticky ideální, neboť v ní nastává nejméně vzájemného vytlačování, v praxi se nicméně nepoužívá, protože při vyhledávání určitého bloku ve vyrovnávací paměti je v nejhorším případě nutné sekvenčně prohledat celou vyrovnávací paměť.

2.4 Výběr oběti u vyrovnávacích pamětí s asociativním mapováním

U vyrovnávací paměti se skupinovým asociativním mapováním a plně asociativním mapováním nastává problém, pokud jsou všechny datové bloky dané adresy ve všech skupinách obsazeny. Při načítání nového datového bloku do vyrovnávací paměti je nutné rozhodnout, která položka v paměti bude zrušena, aby bylo uvolněno místo pro novou položku. Vzniká tak problém **výběru oběti**, který se řeší různými strategiemi náhrady.

- **Least Recently Used** (LRU)[5] – Z vyrovnávací paměti se uvolní datový blok, který byl nejdéle nepoužit. Uvolní se tak položka, u které je možné předpokládat, že práce s ní již byla dokončena.
- **Most Frequently Used** (MFU)[6] – Z vyrovnávací paměti se uvolní datový blok, který byl v poslední době nejčastěji používaný. Tento, na první pohled nevhodný, algoritmus funguje na předpokladu, že práce s danou položkou byla právě dokončena a proto se do vyrovnávací paměti načítá nová položka.
- **First In, First Out** (FIFO)[7] – Z vyrovnávací paměti se uvolní datový blok, který se v paměti nachází nejdéle. Tato strategie funguje na podobném předpokladu jako LRU, že s uvolněnou položkou již byla dokončena veškerá práce, a proto se již dlouho nepoužila.
- **Náhodný výběr** (RAND) – Z vyrovnávací paměti se uvolní náhodně vybraný datový blok. Teto metoda je implementačně nejjednodušší a také dosahuje překvapivě dobrých výsledků. Zavedení nedeterminismu do funkce vyrovnávací paměti může ale být nevýhodné.

Strategie *Least Recently Used*, *Most Frequently Used* a *First In, First Out* vyžadují další obvodové doplňky na čipu vyrovnávací paměti, jako jsou registry pro udržování času posledního použití, nebo čítače četnosti použití. U těchto registrů a čítačů je pak nutné řešit možnost jejich přetečení, což vyžaduje další algoritmy, které zavčas zahájí dekrementaci vybraných čítačů. To výrazně zvyšuje složitost fyzické konstrukce vyrovnávací paměti.

2.5 Udržování koherence dat

Data často existují v hierarchii paměti v několika kopiích. Například kopie jednoho datového bloku může být uložena současně v hlavní paměti a vyrovnávací paměti. Pokud se data ve vyrovnávací paměti zápisem změní, tak data na vzdálenějších úrovní paměti ztratí platnost a nesmí se dále používat. Vzniká tak datová nekonzistence – **nekoherence dat**. Pro udržování koherence dat mezi vyrovnávací pamětí a hlavní pamětí se používají strategie udržení koherence dat:

- **Přímý zápis** (*write-through*) – Při zápisu do rychlé vyrovnávací paměti se současně zapisuje i do odpovídajícího bloku v hlavní paměti. Tato strategie je snadná na realizaci, avšak pokud je rozdíl mezi rychlostí vyrovnávací paměti a hlavní paměti vyšší, tak přímý zápis výrazně zpomaluje procesor a je tedy nevhodný.
- **Zápis s mezipamětí** (*write buffer*) – Opravné zápisy do hlavní paměti se odkládají do mezipaměti a čekají až do okamžiku uvolnění přístupu k hlavní paměti. Touto strategií nedochází ke zbytečnému čekání procesoru na uvolnění přístupu k hlavní paměti při každém zápisu do paměti.
- **Zpětný zápis vždy** – Opravné zápisy do hlavní paměti se provádí až při uvolňování daného datového bloku z vyrovnávací paměti. Tato strategie je sama o sobě nepraktická, protože se datový blok do hlavní paměti zapisuje, i když v něm nebyly provedeny žádné změny. Proto se využívá rozšíření:

[5] LRU - Least Recently Used (v poslední době nejméně používaná)
[6] MFU - Most Frequently Used (nejčastěji používaná)
[7] FIFO - First In, First Out (první dovnitř, první ven)

- **Zpětný zápis podle příznaku změny** (*write-back*, *copy-back*, *store on flag*) – Tato strategie je rozšíření strategie Zpětný zápis vždy. Zápis se do hlavní paměti provádí pouze na základě **příznaku změny** (dirty bit), který označuje, zda byla data modifikována. V praxi je tato strategie používaná nejčastěji, protože má nižší střední počet zápisů do hlavní paměti.

Kapitola 3

Vizualizace činnosti procesoru a vyrovnávacích pamětí

V této kapitole jsou popsány možnosti provedení vizualizace činnosti procesoru a vyrovnávacích pamětí. Jsou zde rozebrány již existující simulátory práce procesoru a vyrovnávacích paměti, vypsány jejich výhody a nevýhody a také vyhodnocena jejich použitelnost pro výuku činnosti vyrovnávacích pamětí. Dále jsou zde probrány možnosti implementace takovéhoto simulátoru jako webovou aplikaci. V zadání této práce bylo stanoveno, že aplikace bude pracovat kompletně na straně klienta a tudíž bude implementována pomocí webových technologií HTML, CSS a JavaScript. Jsou zde stručně popsány aktuálně populární javascriptové frontend frameworky *Angular* a *React* a podrobněji popsán framework *Vue.js*, který byl pro implementaci aplikace nakonec zvolen.

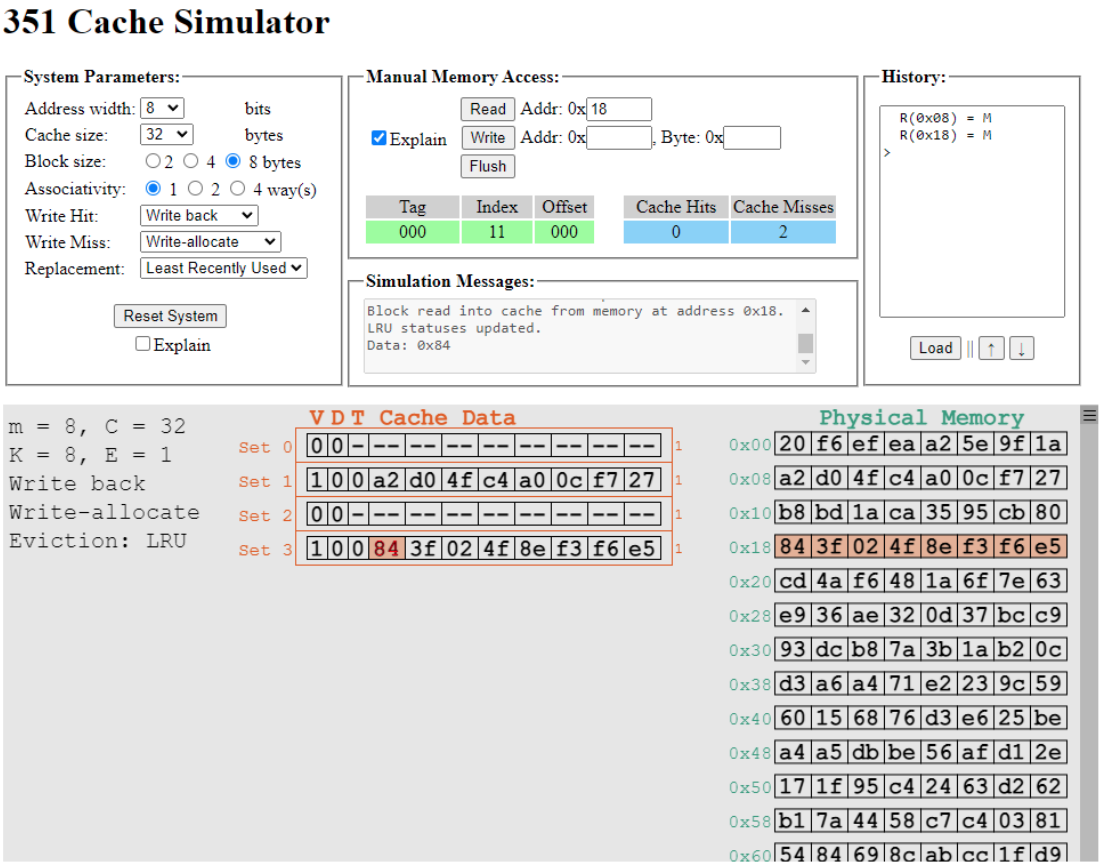
3.1 Existující simulátory činnosti procesoru a vyrovnávacích pamětí

Na internetu existuje několik volně dostupných simulátorů činnosti procesoru a vyrovnávacích pamětí. V této sekci budou blíže popsány tři takové simulátoru: *UW CSE 351 Cache Simulator* sloužící jako studijní podpora na Paul G. Allen School of Computer Science & Engineering, University of Washington [8], závěrečná práce Aryani Paramita z Nanyang Technological University *ParaCache Simulator* [14] a simulátoru RISC procesoru *emulsiV* vyvinutý Guillaume Savatonem z ESEO [17]. Tyto simulátory zde budou ve stručnosti představeny a následně bude zhodnocen rozsah jejich funkcionality, výhody a nevýhody v porovnání s ostatními simulátory a možnost jejich využití pro výuku principů činnosti vyrovnávacích pamětí.

UW CSE 351 Cache Simulator

Simulátor vyrovnávacích pamětí *UW CSE 351 Cache Simulator* [8] je relativně přímočarý simulátor činnosti vyrovnávací paměti. Slouží jako studijní podpora na Paul G. Allen School of Computer Science & Engineering, University of Washington. Umožňuje jednoduše nastavit parametry, způsob organizace a použité algoritmu vyrovnávací paměti a následně s ní přímo manipulovat. Rozhraní této aplikace je vyobrazeno na obrázku 3.1.

Simulátor umožňuje detailní nastavení parametrů vyrovnávací paměti. Je možné nastavit celkovou velikost vyrovnávací paměti i velikost jednotlivých datových bloků paměti.



Obrázek 3.1: Simulátor vyrovnávacích paměti *UW CSE 351 Cache Simulator* umožňuje nastavit parametry a způsob organizace vyrovnávací paměti (sekce System Parameters) a následovně s ní manuálně pracovat (sekce Manual Memory Access). Obsah hlavní a vyrovnávací paměti je zobrazen v dolní polovině aplikace. Při práci s pamětí jsou v ní graficky zvýrazněny položky nebo data, se kterými se právě pracuje.

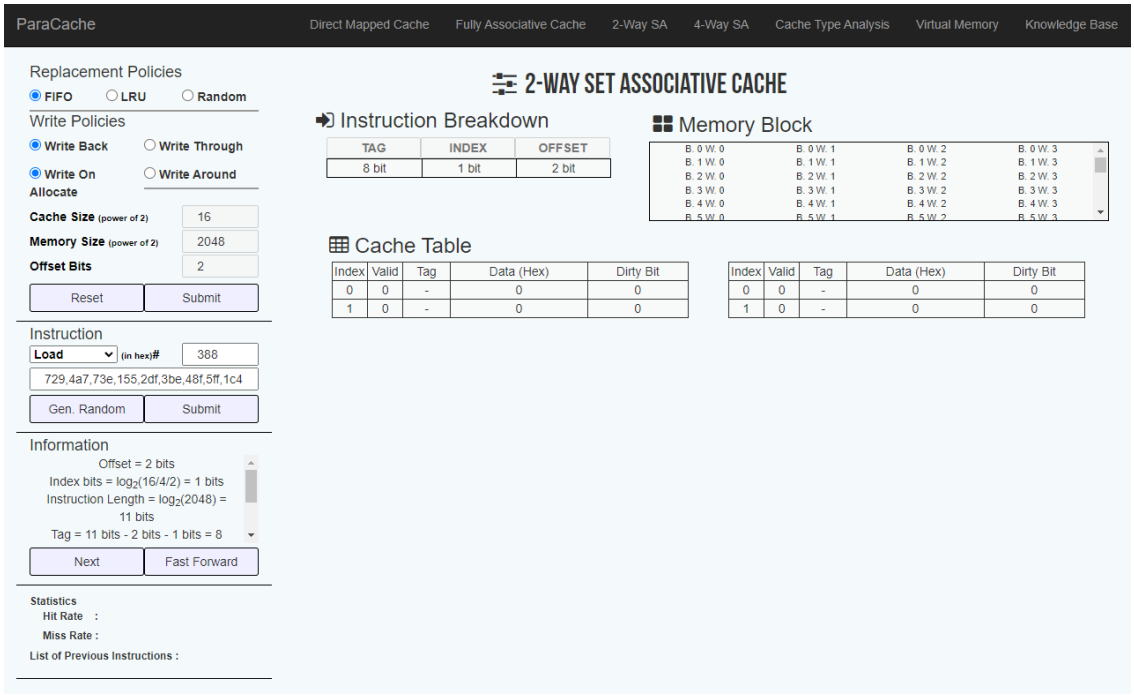
Nastavením stupně asociativity je možné simulovat vyrovnávací paměť s přímým mapováním, skupinově asociativním mapováním i plně asociativním mapováním. Také je možné specifikovat algoritmus udržování koherence dat při zápisu do paměti a způsob výběru oběti u vyrovnávací paměti s asociativním mapováním. S pamětí lze pracovat manuálně pomocí tlačítek pro čtení z paměti (Read), zápisu do paměti (Write) a vyprázdnění obsahu vyrovnávací paměti do hlavní paměti (Flush). Při zaškrtnutí zaškrtávacího políčka *Explain* se čtení nebo zápis provede krok po kroku a do konzole (označená jako *Simulation Messages*) se přitom budou vypisovat komentáře popisující jednotlivé operace. Aplikace zobrazuje kompletní obsah jak hlavní tak vyrovnávací paměti a celkový počet *cache hit* a *cache miss* v rámci dosud provedených operací. Simulátor si navíc udržuje historii provedených operací, ve které se lze volně pohybovat, díky čemuž je možné opakovaně provádět předpřipravenou sekvenci paměťových operací.

Tato aplikace umožňuje simulovat mnoho různých způsobů organizace vyrovnávací paměti. Díky možnosti provádění operací čtení a zápis do paměti krok po kroku a detailních komentářů k jednotlivým krokům je jednoduché pochopit princip činnosti vyrovnávacích pamětí. Simulace avšak může být prováděna pouze ručně. Simulátor neobsahuje žádný

procesor a není tedy možné testovat chování vyrovnávací paměti na reálných algoritmech a programech. Vizualizace pohybu dat mezi hlavní a vyrovnávací paměti také není příliš názorná. Aplikace je tak vhodná spíše pro individuální použití než pro vysvětlení funkce vyrovnávacích pamětí na přednáškách.

ParaCache Simulator

Projekt *ParaCache Simulator* je kolekce simulátorů činnosti vyrovnávacích pamětí [14]. Je to závěrečná práce Aryani Paramita z Nanyang Technological University implementující simulátory vyrovnávací paměti s přímým mapováním, dvoucestným skupinovým mapováním, čtyřcestným skupinovým mapováním a plně asociativním mapování. Také je v tomto projektu zahrnut analyzátor výkonu až čtyř různých druhů vyrovnávací paměti a simulátor virtuální paměti. Jednotlivé simulátory vyrovnávací paměti mají stejné rozhraní, a tudíž dále bude rozebrán pouze simulátor vyrovnávací paměti s dvoucestným skupinovým mapováním, který je zobrazen na obrázku 3.2.



Obrázek 3.2: Simulátor vyrovnávací paměti s dvoucestným skupinovým asociativním mapováním z projektu *ParaCache Simulator*. V simulátoru lze nastavit parametry vyrovnávací paměti a následně simulovat provádění seznamu instrukcí pro čtení z paměti nebo zápisu do paměti. Simulátor zobrazuje obsah hlavní a vyrovnávací paměti a navíc zobrazuje statistiky přístupu do vyrovnávací paměti.

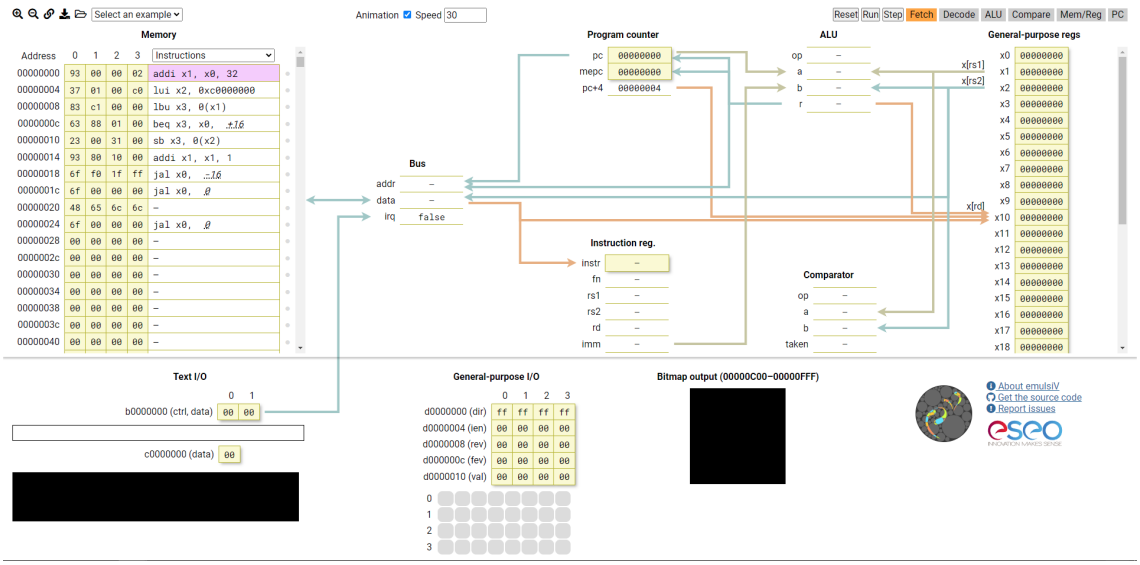
V simulátoru je možné nastavit velikost hlavní a vyrovnávací paměti a také specifikovat algoritmus zápisu do paměti a způsob výběru oběti. S pamětí lze pracovat pomocí instrukcí pro čtení z paměti (Load) a zápisu do paměti (Store) na zadanou adresou. Také je možné vytvořit řetězec adres, ze kterých se bude postupně číst nebo zapisovat. Vykonání instrukce je možné provést naráz nebo krok po kroku, přičemž s každým krokem je vypsán komentář popisující zrovna prováděnou operaci. Při provádění instrukce krok po kroku se barevně vyznačují pole v paměti, s nimiž se zrovna pracuje, a při vyhodnocení obsahu vyrovnávací

paměti jsou zobrazena logická hradla, pomocí nichž se kontroluje adresový příznak a příznak platnosti.

V této kolekci simulátorů je možné simulovat čtyři základní organizace vyrovnávací paměti v různých specifických nastaveních. Při provádění instrukcí pro čtení z paměti a zápisu do paměti jsou jednotlivé operace v paměti barevně zvýrazněny a slovně popsány což umožňuje rychlé pochopení funkce vyrovnávací paměti. Nicméně jsou zde podobné problémy jako v předchozím simulátoru. Není možné automaticky provést zadaný seznam instrukcí, všechny operace je nutné provést ručně a navíc se při kombinací instrukcí pro zápis a čtení musí mezi nimi manuálně přepínat. Program neobsahuje procesor, a tudíž není možné simulovat chování vyrovnávací paměti na reálných programech. Vizualizace některých kroků při zápisu nebo čtení může být nepřehledná a bez slovního popisu by byla matoucí. Navíc je aplikace poměrně pomalá. Tudíž má tato webová aplikace podobné možnosti použití jako aplikace popsaná v předchozí sekci, a to především pro individuální použití.

Simulátor procesoru emulsiV

Simulátor *emulsiV* je vizuální simulátor procesoru Virgule vyvinutý Guillaume Savatonem z univerzity ESEO [17]. Procesor Virgule je 32-bitový RISC procesor implementující minimální podmnožinu instrukční sady RISC-V. Simulátor i procesor slouží pro výuku počítačové architektury a návrhu počítačových systému pro začátečníky na univerzitě ESEO. Aplikace graficky znázorňuje strukturu procesoru, představuje jazyky a nástroje pro tvorbu nizkoúrovňových programů a vizualizuje způsob, jakým se každá instrukce procesoru zpracovává a vykonává [16]. Rozhraní aplikace je zobrazeno na obrázku 3.3.



Obrázek 3.3: Simulátor *emulsiV* je kompletní simulátor minimálního RISC procesoru Virgule. Umožňuje zadat assembler kód a následně vizualizovat zpracování jednotlivých instrukcí na všech komponentách procesoru. Simulátor také umožňuje zpracovávat textový a obecný vstup/výstup. Aplikace obsahuje několik předpřipravených ukázkových programů.

Tato aplikace umožňuje poměrně komplexní simulaci práce procesoru postaveného na architektuře RISC-V. Jsou v ní vizualizovány jednotlivé prvky procesoru a data, které se v nich aktuálně vyskytují: datová sběrnice (Bus), čítač instrukcí (Program counter), instrukční registr (Instruction reg.), aritmeticko-logická jednotka (ALU), komparátor (Comparator)

a sada 32 obecně účelných registrů (General-purpose regs). Procesor dokáže také pracovat s periferiemi textový vstup/výstup, obecný vstup/výstup a bitmapový výstup. Jednotlivé prvky procesoru jsou propojeny šipkami, které představují datové cesty mezi prvky. Do aplikace je možné zapsat assembler program, načíst kód z hexadecimálního souboru nebo vybrat jeden z předpřipravených ukázkových programů. Tento program pak lze celý spustit, nebo případně manuálně krokovat po jednotlivých instrukcích. Pokud jsou v simulátoru zapnuté animace tak je proces vykonávání instrukcí a tok dat mezi jednotlivými prvky procesoru graficky vizualizován. Díky tomu je možné detailně zobrazit zpracování instrukcí a pohyb konkrétních dat v celém procesoru.

Aplikace je pouze simulátorem procesoru a neobsahuje hlavní ani vyrovnávací paměť, nicméně do této práce byla zařazena právě z důvodu vizualizace funkce procesoru. Procesoru je možné zadat libovolný program (program je omezen pouze velikostí paměti programu), jehož vykonávání je následně detailně graficky vizualizováno. Je zobrazen způsob načtení a dekódování instrukce z paměti programu, pohyb dat z instrukčního registru do aritmeticko-logické jednotky nebo obecně účelných registrů a práce se skokovými instrukcemi. Jednotlivé prvky procesoru mají vizualizovány své datové cesty a při toku dat je daná datové cesta zvýrazněna. Samotný přesun dat je také animován. Aplikace je v tomto ohledu ideální pro výuku na přednáškách a je tak dobrým zdrojem inspirace pro možnost vizualizace činnosti vyrovnávacích pamětí procesoru.

Shrnutí dostupných simulátorů vyrovnávací paměti

Dostupné simulátory vyrovnávacích pamětí umožňují se seznámit s funkcí vyrovnávací paměti. Čtení z paměti a zápis do paměti je v nich detailně popsán krok za krokem a většinou je i názorně vizualizován. Nicméně oba simulátory umožňují buď dlouho trvající provedení jedné instrukce po jednotlivých operacích, nebo okamžité provedení celé instrukce. Není dostupný mód, ve kterém by se instrukce prováděli rychle za sebou, ale pořád by se zobrazovali animace pohybu dat. Absence procesoru u obou simulátorů představuje velkou nevýhodu. Manuální práce s pamětí nemůže efektivně simulovat chování reálných programů a algoritmů, ve kterých se právě projevují rozdíly mezi jednotlivými organizacemi vyrovnávacích pamětí. V tomto případě je názorný simulátor RISC procesoru *emulsiV*, přestože neobsahuje žádnou vnější paměť mimo registry. Tento simulátor názorně vizualizuje vykonávání assembler instrukcí na všech komponentách procesoru. Také detailně zobrazuje tok dat mezi jednotlivými komponentami.

Chybí simulátor, který by kombinoval prvky jak simulátoru vyrovnávací paměti, tak simulátoru procesoru s animovanou vizualizací neboli grafický simulátor vyrovnávací paměti s procesorem. Tímto simulátorem by bylo možné spustit libovolný assembler program a na něm simulovat chování jednotlivých vyrovnávacích pamětí. Simulátor by jednoduše a rychle vizualizoval tok dat mezi procesorem, vyrovnávací pamětí a hlavní pamětí. Jednotlivé operace čtení a zápisu do vyrovnávací paměti by nemuseli být nijak okomentované, simulátor by sloužil spíš pro prezentaci celkové funkce a vlivu vyrovnávací paměti na program, než na vysvětlení principu funkce vyrovnávacích pamětí. Na to dostačují právě výše zmíněné simulátory. Také není třeba simulovat komponenty procesoru. Simulace bude zaměřena na vyrovnávací paměť a tak procesor bude co nejvíce minimální a abstrahovaný. Cílem této práce je právě vytvořit takovýto simulátor, který by sloužil především pro pochopení funkce a praktického použití vyrovnávacích pamětí.

3.2 Prostředky pro vizualizaci činnosti vyrovnávacích pamětí procesoru

Grafický simulátor činnosti vyrovnávacích pamětí procesoru je možné napsat jako desktopovou aplikaci v libovolném jazyce jako je například Qt, C#, Java, aj. Od aplikace je ale vyžadováno, aby měla univerzální – platformě nezávislé – řešení a byla jednoduše spustitelná, což implementace simulátoru jako desktopová aplikace nesplňuje. Implementace simulátoru jako webovou aplikace, podobně jako jsou výše zmíněné simulátory, je nezávislé na operačním systému počítače a také nevyžaduje instalaci nebo stáhnutí aplikace. Vytvoření webové aplikace znamená použití technologií HTML, CSS a JavaScript. Protože je vytvářený simulátor již rozsáhlejší program, bylo by využití pouze samotného javascriptu příliš nepraktické, a tak bude pro vývoj aplikace použit takzvaný **javascriptový frontend framework**.

Javascriptové frontend frameworky jsou systémy poskytující obecné nástroje pro vývoj frontend části (také zvané jako prezentační části) webových aplikací. Tyto systémy umožňují jednodušeji programovat škálovatelné a udržitelné webové aplikace tím, že zajišťují obecně vyžadované funkce jako je například aktualizace DOM[1] webové stránky při změně vnitřního stavu aplikace, práce s cookies nebo komunikaci se serverovým API. Definují vlastní způsob organizace zdrojových souborů webové aplikace, který pomáhá s udržováním přehlednosti ve větších projektech. Navíc frameworky často rozšiřují funkcionalitu standardních webových technologií HTML, CSS a JavaScript o nové syntaktické elementy. Existuje mnoho různých javascriptových frontend frameworků, které se liší uplatněnými paradigmaty i rozsahem, od jednoduchých knihoven funkcí po rozsáhlé systémy zásadně měnící způsob programování webových aplikací. Z těch byly blíže prostudovány tři v dnešní době nejpopulárnější frameworky: *React*, *Angular* a *Vue.js*, z nichž byl nakonec pro implementaci aplikace zvolen framework *Vue.js*. V následujícím textu jsou stručně popsány základní vlastnosti, výhody a nevýhody těchto frameworků.

React

React je javascriptová knihovna vhodná pro tvorbu uživatelských rozhraní jednostránkových (single-paged) a mobilních aplikací. Na rozdíl od jiných kompletních frameworků se jádro Reactu soustředí pouze na jednu specifickou oblast, a to na prezentační vrstvu. Z hlediska klasické architektury MVC (Model-View-Controller) se tedy jedná především o část *view*, pro funkcionalitu dalších částí architektury MVC je nutné použít další knihovny nebo balíčky. V současné době je React nejvíce používaným javascriptovým frontend frameworkem [12].

Základním prvkem Reactu jsou takzvané **komponenty** (components), které představují zapouzdřené znovupoužitelné HTML elementy, z nichž se následně skládá celkové uživatelské rozhraní aplikace. Komponenty mají **vlastnosti** (props) umožňující zvenčí specifikovat funkcionalitu komponenty, a vnitřní **stav** (state). Komponenty mají jednosměrně datové mapování z nadřazené komponenty na podřízenou právě pomocí vlastností. Pro komunikaci v opačném směru nebo mezi sourozeneckými komponentami se používá zpracování událostí podřízených komponent pomocí metod poskytnutými nadřazenými komponentami. Deklarativní způsob definice těchto komponent umožňuje jednoduše vytvářet interaktivní uživatelské rozhraní s předvídatelným chováním, které je navíc jednodušší testovat [15].

[1] DOM – Document Object Model (objektový model dokumentu)

```
class HelloMessage extends React.Component {
    render() {
      return <div>Hello {this.props.name}</div>;
    }
  }

root.render(<HelloMessage name="Taylor" />);
```

Obrázek 3.4: Příklad minimální React komponenty obsahující jednoduchý HTML blok [15]. Komponenta má jednu vlastnost (props) `name`. V metodě `render()` lze vidět rozšíření jazyka JavaScript JSX, pomocí něhož je možné definovat HTML kód přímo v javascriptovém kódu. Komponenta se pro uživatele zobrazí zavoláním metody `root.render(html)`, obsahující jako parametr HTML element reprezentující tuto komponenty.

Na obrázku 3.4 je zobrazen syntax definice jednoduché komponenty s vlastností. Komponenta definuje metodu `render()`, ve které je deklarovaná vizuální struktura komponenty. React využívá syntaktické rozšíření jazyka JavaScript zvané JSX (JavaScript Syntax Extension), které umožňuje v javascriptovém kódu deklarovat strukturu komponenty podobným způsobem jako v HTML. Bez tohoto rozšíření by se vizuální struktura komponenty musela definovat pomocí metod DOM (Document Object Model), což je podstatně náročnější.

React využívá virtuální DOM, což je paměťová struktura, která reprezentuje skutečnou strukturu dokumentu. V této struktuře je uložena hierarchie komponent tvořících výslednou webovou aplikaci. Při změně vnitřního stavu aplikace React porovná tuto strukturu se skutečným DOM stránky, vypočítá rozdíl mezi nimi a následně se pomocí tohoto rozdílu překreslí pouze ty elementy, ve kterých nastala změna a ne celou stránku. Díky tomu se komplexnější webové aplikace překreslují výrazně rychleji.

React je velmi flexibilní knihovna, která nevynucuje žádné složité konvence nebo souborovou strukturu. Často se používá v kombinaci s dalšími knihovnami s různými architekturami a přístupem k webové prezentaci. React se v těchto kombinacích používá vždy pro vykreslování uživatelského rozhraní. Nicméně kvůli této vysoké flexibilitě a velkému množství balíčků třetích stran implementující podobnou funkcionalitu neexistuje žádný ustálený vývojový postup. Programátor tak musí věnovat značný stav studiu těchto balíčků, aby zvolil ten vhodný, a případná pozdější změna bývá těžko proveditelná.

Angular

Angular je kompletní javascriptový frontend framework pro tvorbu webových aplikací, který staví na komponentové architektuře se službami a základech objektově orientovaného programování. Je to součást takzvaného *MEAN stacku* pro vývoj dynamických webových aplikací, skládající se z nástrojů MongoDB, Express.js, Angular a Node.js. Využívá rozšíření jazyka JavaScript o statické typování zvané **TypeScript**. Angular je po frameworku React jedním z nejrozšířenějších javascriptových frontend frameworků [13].

Základním stavebním prvkem Angularu je, podobně jako u Reactu, komponenta (component). Komponenta je samostatný prvek, který představu znovupoužitelný HTML element, z nichž se následně skládají složitější prvky uživatelského rozhraní. Komponenta

je definovaná typescriptovou třídou rozšířenou **direktivou** (dekorátorem) `@Component()`, která specifikuje HTML šablony komponenty, CSS selektor definující způsob použití komponenty v HTML kódu a seznam CSS pravidel specifikující vizuální vzhled komponenty. Příklad jednoduché Angular komponenty je zobrazen na obrázku 3.5. HTML šablona komponenty může být zapsána přímo v direktivě třídy komponenty (jako inline šablona), nebo může být umístěna ve vlastním souboru, na který se poté direktiva třídy odkazuje. Tato šablona má plný přístup k vlastnostem třídy, k níž je přiřazena, a může je použít jako obsah nebo hodnotu vlastnosti HTML prvku. Šablony jsou navíc rozšířeny o speciální direktivy, jako je `*ngIf` pro podmíněné vykreslení nebo `*ngFor` pro cyklické vykreslení prvku [2].

```
import { Component } from '@angular/core';
@Component({
    selector: 'hello-world',
    template: `
        <h2>Hello World</h2>
        <p>This is my first component!</p> `
})
export class HelloWorldComponent {
    // The code in this class drives the
    // component's behavior.
}
```

Obrázek 3.5: Minimální Angular komponenta obsahující nadpis a text [2]. Komponenta je tvořena třídou, obsahující její vnitřní logiku a data, rozšířenou dekorátorem `Component`. Dekorátor specifikuje HTML šablonu komponenty a její selektor neboli její HTML název. Veškeré proměnné a metody definované ve třídě jsou volně dostupné v její HTML šabloně.

Pro samotnou logiku aplikace se využívají **služby** (services), které poskytují komponentě určitou funkcionalitu jako například komunikaci se serverovým API. Komponenta vyžadující danou službu ji získá pomocí systému **Dependency Injection** poskytnutým Angularem. Dependency Injection je návrhový vzor, který umožňuje deklarovat závislosti třídy komponenty, bez toho aniž by bylo nutné se starat o jejich instanciaci. O tu se postarají vnitřní mechanismu frameworku. Pomocí tohoto návrhového vzoru je možné psát flexibilnější a lépe testovatelné komponenty. Požadovaná služba se komponentě přiřadí pomocí direktivy `@Injectable()`.

Angular je komplexní framework poskytující nástroje pro vývoj multiplatformních aplikací, celkové i jednotkové testování komponent a vytváření efektivních komplexních animací. Oproti frameworku React je méně flexibilní, ale zato zase poskytuje určitý předepsaný styl programování a má ustálený vývojový postup. Nicméně pro menší projekty může být Angular se svým rozsahem až příliš těžkopádným řešením.

Vue.js

Vue.js je javascriptový frontend framework pro tvorbu uživatelských rozhraní webových aplikací. Využívá standardní webové technologie HTML, CSS a JavaScript, které rozšiřuje pouze o základní direktivy. Je postaven na komponentové architektuře, která umožňuje

deklarativně programovat jednotlivé prvky uživatelského rozhraní. Vue.js je po Reactu a Angularu poslední ze tří nejvíce rozšířených javascriptových frontend frameworků [7].

Základní prvek Vue.js je, stejně jako v předchozích frameworcích, komponenta. Vue.js prosazuje vytváření komponent jako **jedno-souborové komponenty** (Single-File Component), kde šablona, vnitřní logika a stylové předpisy dané komponenty jsou uloženy v jednom souboru s příponou `*.vue`. Vnitřní logika komponenty se skládá z vlastností (props), vnitřního stavu (data) a funkcí. Jsou podporovány dva styly programování komponent: **Options API** a **Composition API**.

- Při použití Options API je vnitřní logika komponenty zapsána jako vlastnosti objektu, který reprezentuje celou komponentu. Tento způsob zápisu má blízko k objektově orientovanému programování a je tedy často používaný při třídním návrhu aplikace.
- Při použití Composition API je vnitřní logika zapsána pomocí importovaných API funkcí. Tento způsob zápisu je oproti Options API flexibilnější a umožňuje lépe znovu využít již existující vnitřní logiku.

Příklad jednoduché komponenty je vyobrazen na obrázku 3.6.

```
<template>
  <button @click="count++">{{ count }}</button>
</template>
<script>
export default {
  data() {
    return {
      count: 0
}}}
</script>
<style>
</style>
```

Obrázek 3.6: Příklad jednoduché Vue.js komponenty obsahující tlačítko počítající počet jeho stisknutí [22]. Komponenta je tvořena ze tří bloků: `template` obsahující HTML šablonu, `script` s vnitřní logikou a daty komponenty a `style` obsahující CSS předpisy komponenty. Komponenta je zapsaná ve stylu Options API. Vnitřní stav této komponenty je zde tvořen pouze daty (`data`), neobsahuje vlastnosti (`props`) ani metody.

Framework Vue.js kombinuje vlastnosti frameworků React a Angular a zároveň se snaží být jednoduchý a nenáročný. Z Reactu přebírá způsob zápisu šablony, vnitřní logiky a stylových předpisů komponenty v jednom souboru a komunikaci mezi komponentami pomocí objektů reprezentující vlastnosti (props) a vnitřní stav (state). Navíc stejně jako React využívá i Vue.js Virtuální DOM pro rychlé a výkonné překreslování pouze těch elementů stránky, ve kterých nastala změna. Z Angularu naopak je převzat kombinace zápisu HTML a JavaScriptu v podobě direktiv jako je například direktiva `v-bind` pro navázání vlastnosti elementu na proměnnou a `v-if` pro podmíněné vykreslení. Na rozdíl od obou těchto

frameworků používá Vue.js standardní JavaScript bez žádných syntaktických rozšíření (ale navíc je podporován také TypeScript). Co se týče komplexnosti, představuje Vue.js střední cestu mezi velmi flexibilním Reactem vyžadujícím doplňující moduly a naopak příliš komplexním Angularem.

Právě kvůli této vyváženosti mezi flexibilitou a komplexností byl framework Vue.js nakonec zvolen pro implementaci webové aplikace simulátoru činnosti vyrovnávacích pamětí procesoru. Vue.js nevyžaduje znalost žádných, pro tento framework specifických, technologií a umožňuje relativně volný způsob implementace. Na druhou stranu je to plnohodnotný framework a není tudíž nutné vyhledávat moduly nebo balíčky implementující základní funkcionalitu. Vue.js je tak ideální pro středně velké jednostránkové webové aplikace.

Kapitola 4

Návrh simulátoru činnosti vyrovnávacích pamětí

V této kapitole je popsán proces návrhu grafického uživatelského rozhraní a logického rozdělení webové aplikace simulátor činnosti vyrovnávacích pamětí procesoru (*Cache Simulator*). Jsou zde uvedeny grafické modely, které byly vypracovány v rámci návrhu aplikace a podle nichž byla aplikace následně implementována.

Cílem této aplikace je simulovat a graficky vizualizovat činnost vyrovnávacích pamětí při vykonávání programu. V aplikaci bude editor kódu, do kterého bude možné zadat program v assembleru. Tento program bude možné spustit a následně ovládat jeho vykonávání jako pozastavení programu nebo krokování programu po jednotlivých instrukcích. V aplikaci bude možné specifikovat typ vyrovnávací paměti, která bude pro simulaci použita. Bude vizualizován obsah hlavní a vyrovnávací paměti a také stav procesoru a jeho registrů. Při vykonávání zadaného programu bude vizualizován tok dat mezi hlavní pamětí, vyrovnávací pamětí a procesorem. Aplikace by měla být implementovaná tak, aby bylo možné přidat další typy vyrovnávacích pamětí a případně ji rozšířit o další nastavení vyrovnávacích pamětí (algoritmus výběru oběti, způsob zápisu do paměti, atd.).

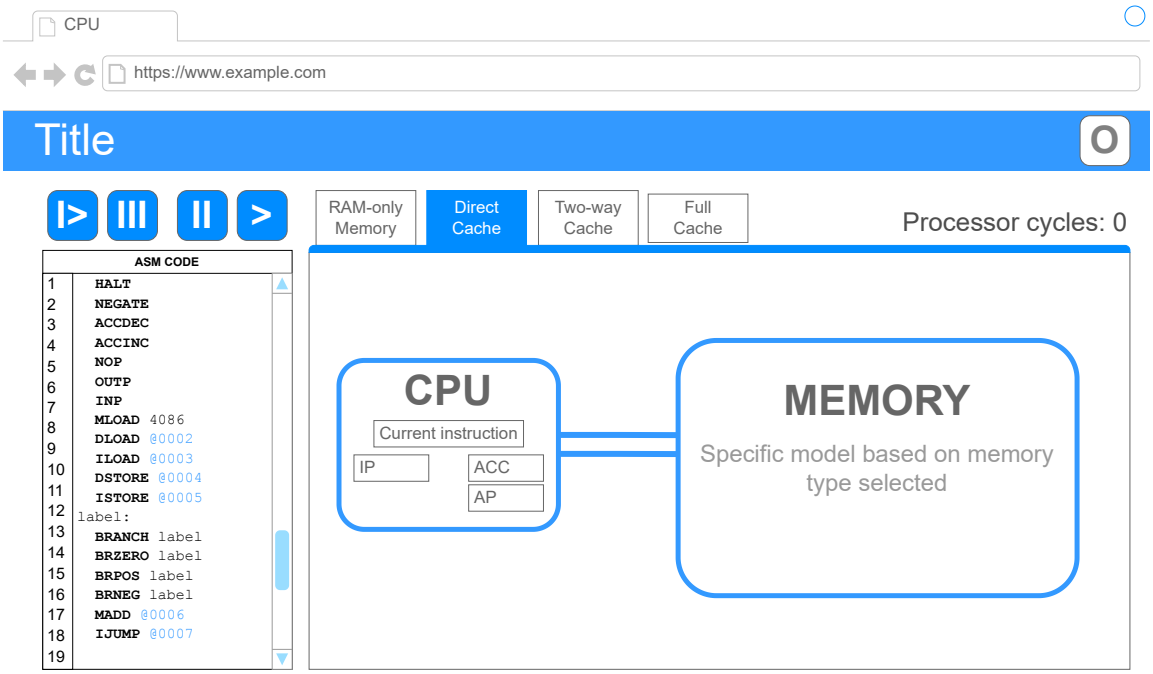
Výsledná aplikace má sloužit jako podpora výuky funkce vyrovnávacích pamětí a hierarchie paměti v rámci předmětu Návrh počítačových systémů (INP) na Fakultě informatiky Vysokého učení technického v Brně. Cílová uživatelská skupina jsou tak vysokoškolští studenti v oboru informačních technologií, kteří se s danou látkou ještě nesetkali. Aplikace by měla jasně a přehledně vizualizovat tok dat mezi hlavní a vyrovnávací pamětí a procesorem a umožnit studentům experimentovat s různými vstupními programy a organizacemi vyrovnávací paměti. Studenti by z aplikace měli zjistit jak významný je vliv vyrovnávací paměti a proč je důležité při programování brát v potaz organizaci paměti. Tudíž by výsledná aplikace měla co nejvíce abstrahovat od prvků nesouvisejících s vyrovnávací pamětí.

4.1 Návrh grafického uživatelského rozhraní aplikace

Jako první v rámci návrhu aplikaci byl vypracován celkový návrh grafického uživatelského rozhraní. Grafické uživatelské rozhraní (dále GUI[1]) bylo vypracováno právě první, aby pomohlo s definicí možností použití aplikace a díky tomu i požadavky na logickou strukturu aplikace. Návrh byl inspirován již existujícími simulátory vyrovnávacích pamětí popsaných v kapitole 3.

[1] GUI - Graphical User Interface

Vytvořený drátěný model (wiremodel) GUI je vyobrazen na obrázku 4.1. Základní cíl GUI této aplikace je umožnit uživateli zadat assembler program (program zapsaný v jazyku symbolických instrukcí), jednoduše ovládat provádění tohoto programu a zároveň přehledně vizualizovat tok dat mezi procesorem a pamětí. GUI tak bude rozděleno do dvou hlavních částí: **ovládací část** a **modelová část**.



Obrázek 4.1: Drátěný model grafického uživatelského rozhraní aplikace. Aplikace je rozdělena do dvou částí: ovládací část, obsahující editor kódu a tlačítka pro ovládání provádění programu, a modelovou část s výběrem modelu vyrovnávací paměti. V modelové části je vizualizován procesor s registry a model paměti podle vybraného modelu. Model paměti je zde abstrahován na obecný model. Záhlaví stránky obsahuje název aplikace a tlačítko pro zobrazení nastavení simulátoru.

V ovládací části se bude vyskytovat editor kódu, do kterého bude možné zadat program zapsaný v assembleru. Editor bude podporovat jednoduché zvýrazňování kódu; zvýrazněné budou jména instrukcí, parametry instrukcí označující adresu v paměti a také komentáře. Při běhu programu se bude označovat řádek, na němž se nachází instrukce, která bude provedena jako další. Automatické doplňování kódu nebude v editoru podporováno. Nad editorem kódu budou tlačítka pro ovládání provádění zadaného programu s následujícími funkcemi (zapsané v pořadí zleva doprava):

- Tlačítko **START** zahájí vykonávání programu zadanému v editoru kódu, nebo znovuspustí pozastavený program. Tlačítko bude aktivní před samotným spuštěním programu a při pozastavení prováděného programu. Po jeho stisknutí se začne vykonávat celý program, a to dokud nebude ukončen dosáhnutí konce programu nebo stisknutím tlačítka STOP, anebo pozastaven instrukcí `HALT` nebo stisknutím tlačítka PAUSE.

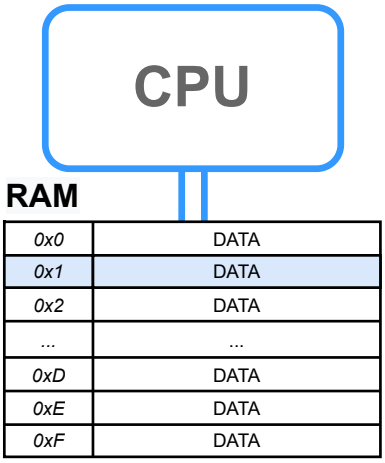
- Tlačítko **STOP** ukončí vykonávání aktuální programu. Tlačítko se stane aktivní, jakmile bude zahájeno vykonání programu. Při stisknutí se ukončí provádění programu a aplikace se uvede do stavu, v jakém byla před zahájením programu.
- Tlačítko **PAUSE** pozastaví vykonávání aktuálního programu. Tlačítko bude aktivní pouze při aktivním provádění programu. Při stisknutí se dokončí právě rozpracovaná instrukce a program dále čeká. V tomto stavu bude možné začít krokovat program.
- Tlačítko **STEP** bude sloužit pro manuální krokování instrukcí pozastaveného programu. Tlačítko bude aktivní pouze při pozastavení provádění programu tlačítkem PAUSE nebo instrukcí `HALT`. Po stisknutí se vykoná jedna následující instrukce, která bude v editoru kódu vyznačena, a program bude dále čekat.

Modelová část bude tvořena lištou karet, která bude obsahovat čtyři karty reprezentující modely vyrovnávacích pamětí navrhnuté v sekci 4.2. Paměťový model bude možné změnit pouze, pokud nebude spuštěn program. Na pravé straně od karet modelů pamětí bude umístěn čítač cyklů procesoru, který bude sloužit pro porovnání efektivnosti jednotlivých typů vyrovnávací paměti na daném algoritmu. Samotný model se bude skládat z modelu procesoru (CPU) a modelu vyrovnávací paměti (MEMORY) umístěných vedle sebe. Model procesoru bude zobrazovat aktuálně vykonávanou instrukci a dále hodnoty registrů instrukční ukazatel (IP), akumulátor (ACC) a adresová ukazatel (AP). Konkrétní struktura modelu paměti bude záviset na zvoleném typu paměti v liště karet, které jsou blíže popsané v sekci 4.2. Oba tyto modely budou propojeny rourou, pomocí níž bude možné vizualizovat tok dat mezi procesorem a pamětí a naopak.

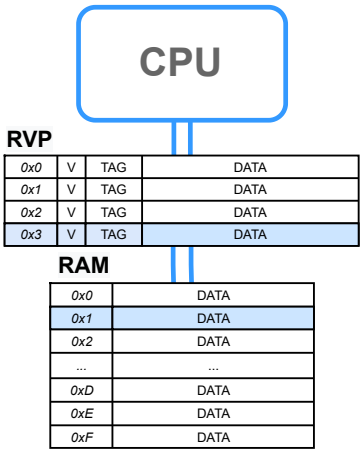
Ve vrchní části aplikace se bude nacházet záhlaví obsahující název aplikace s logem, jméno autora (v návrhu se nevyskytuje) a tlačítko pro zobrazení nastavení vybraných parametrů simulátoru. Nastavení simulátoru bude implementováno jako modální okno obsahující textové vstupu pro jednotlivé parametry. Dále bude aplikace obsahovat dynamická vyskakovací upozornění v pravém spodním rohu aplikace pro zobrazení chybových hlášení simulátoru.

4.2 Návrh modelů vyrovnávací paměti

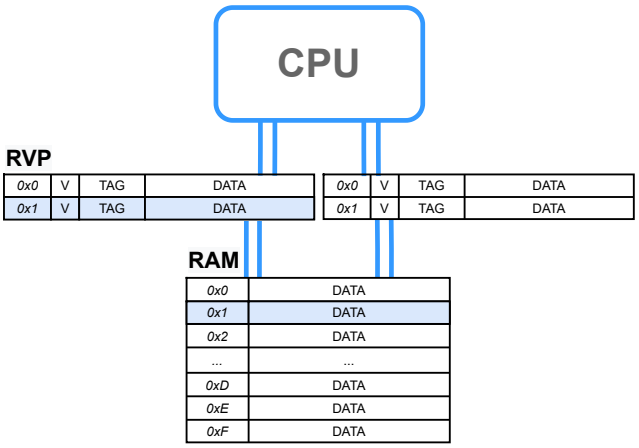
Následně byly vypracovány grafické modely vyrovnávacích pamětí. Ze studia technik vyrovnávacích pamětí provedeným v kapitole 2 byly identifikovány tři základní způsoby organizace vyrovnávací paměti: vyrovnávací paměť s přímým mapováním (sekce 2.1), vyrovnávací paměť se skupinovým asociativním mapováním (sekce 2.2) a vyrovnávací paměť s plně asociativním mapováním (sekce 2.3). Navíc byla mezi tyto modely zařazena i paměť bez vyrovnávací paměti z důvodu automatického testování a také pro možnost porovnání s ostatními modely. Vytvořené modely pamětí jsou zobrazeny na obrázku 4.2.



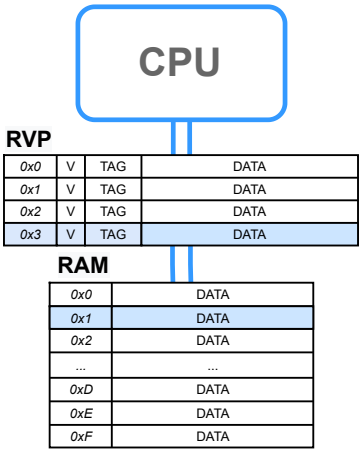
(a) Paměť bez vyrovnávací paměti



(b) Vyrovnávací paměť s přímým mapováním.



(c) Vyrovnávací paměť se skupinovým asociativním mapováním.



(d) Vyrovnávací paměť s plně asociativním mapováním.

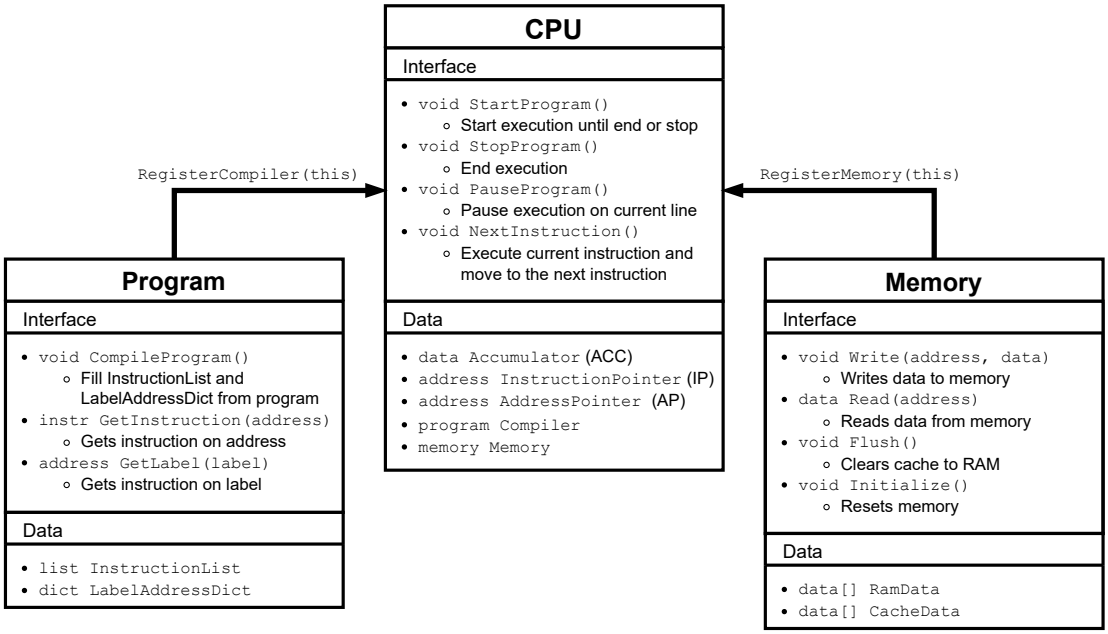
Obrázek 4.2: Na základě studia vyrovnávacích pamětí procesoru byly vytvořeny tři modely vyrovnávacích pamětí a jeden model bez vyrovnávací paměti. Model je složen z procesoru (CPU), hlavní paměti (RAM) a případně vyrovnávací paměti (RVP). Všechny části modelu jsou propojeny rourami, které slouží pro znázornění toku dat.

Struktura všech čtyř modelů bude podobná a bude se skládat z procesoru (CPU), hlavní paměti (RAM) a vyrovnávací paměti (RVP). V návrhu jsou jednotlivé modely umístěné pod sebou, nicméně v aplikaci budou z důvodu zobrazení větší kapacity pamětí umístěny vedle sebe. Model procesoru je zde zjednodušen, jeho skutečná struktura je popsána v sekci 4.1. Oba modely pamětí budou vizualizovány pomocí jednoduché tabulky, kde řádky budou reprezentovat jednotlivé paměťové bloky. Každý řádek bude označen adresou paměťového bloku a bude obsahovat data uložená v daném paměťovém bloku (DATA) a případně i další metadata. Model hlavní paměti bude obsahovat pouze sloupce pro adresu a data daného paměťového bloku. Model vyrovnávací paměti bude navíc obsahovat i sloupce obsahující příznak platnosti tohoto bloku (V) a adresový příznak dat uložených na daném řádku (TAG). V aplikaci budou mít všechny sloupce pro lepší srozumitelnost záhlaví označující jednotlivé sloupce. Pokud velikost paměti bude příliš velká a model se nevleze do modelové části aplikace, tak se na straně tabulky zobrazí posuvník a bude možné se v tabulce

posouvat. Všechny tyto komponenty budou propojeny rourami, které budou vizualizovat tok dat mezi procesorem a pamětí a mezi jednotlivými úrovněmi paměti. Aby bylo zřejmé, se kterými daty se manipuluje, bude jak hlavní paměť tak vyrovnávací paměť umožňovat zvýraznění řádku (datovém bloku), s nímž se právě manipuluje. Toto zvýraznění bude provedeno formou barevného pulsu a je na modelu reprezentováno obarveným řádkem.

4.3 Návrh logického členění implementace aplikace

Jako poslední byl proveden návrh logického členění vnitřní implementace aplikace. Z předchozích modelů vyplynulo, že aplikace bude rozdělena na tři základní moduly: **překladač**, který se bude starat o překlad zadaného assembler programu, **procesor** ovládající vykonávání a průběh programu a **paměť**, která bude poskytovat metody pro přístup do paměti a vnitřně implementovat funkci vyrovnávací paměti. Na základě tohoto rozdělení byl vytvořen logický model zobrazený na obrázku 4.3.



Obrázek 4.3: Logický model aplikace je rozdělen do tří oddělených modulů: překladač (Program), procesor (CPU) a paměť (Memory). Pro poskytnutí metod překladače a paměti modulu procesoru, který je centrálním modulem simulátoru, se používají události procesoru `RegisterCompiler` a `RegisterMemory`. Při inicializaci modulu překladače nebo paměti se vyvolá tato událost a v ní se jako parametr předá objekt obsahující odkazy na metody implementující funkcionalitu daného modulu.

Modul překladače (na obrázku značený jako Program) se bude starat o správné přeložení assembleru zadanému v editoru kódu. Metodou `void CompileProgram()` se provede lexikální a syntaktická kontrola a program se přeloží do vnitřní reprezentace programu. Přeložený programu bude vnitřně reprezentován jako seznam instrukcí `InstructionList`. Zároveň se v přeloženém programu vyhledají všechny návěští a spolu se svou pozicí se uloží do asociativního pole `LabelAddressDict`, kde jako klíč bude sloužit jméno návěští a pozice návěští v kódu bude hodnota. Dále bude tento modul poskytovat metodu `instruction GetInstruction(address)` pro získání instrukce na základě její adresy a metodu `address`

`GetLabel(label)` pro získání pozice daného návěští. Takto bude modul překladače sloužit i jako paměť programu, díky čemuž se procesor nebude muset nijak starat o uložení programu. Při inicializaci překladače se pomocí události procesoru `RegisterCompiler` předá procesoru objekt obsahující odkazy na právě zmíněné metody.

Každý model paměti popsaný v sekci 4.2 bude reprezentován vlastním modulem Memory. Tyto moduly budou mít stejné rozhraní, ke kterému bude procesor přistupovat přes objekt předaný jako parametr události `RegisterMemory`, ale implementace jednotlivých metod bude pro každý druh paměti jiná. Data uchovaná v paměti budou uloženy v poli `RamData` v případě hlavní paměti a v poli `CacheData` v případě vyrovnávací paměti. Každý modul paměti bude implementovat tyto čtyři metody:

- Metodu `void Write(address, data)` pro zápis dat na zadanou adresu v paměti. V rámci této metody bude implementováno chování vyrovnávací paměti při zápisu dat.
- Metodu `void Read(address)` pro čtení dat z dané adresy v paměti. V rámci této metody bude implementováno chování vyrovnávací paměti při čtení dat.
- Metodu `void Flush()` pro hromadný zápis všech dat ve vyrovnávací paměti do hlavní paměti. V případě paměti bez vyrovnávací paměti bude mít tato metoda prázdné tělo.
- Metodu `void Initialize()` pro inicializaci paměti do počátečního stavu. Tato metoda nastaví hodnoty všech položek polí `RamData` a `CacheData` na nulové hodnoty a inicializuje pomocné proměnné na jejich výchozí hodnoty.

Modul procesor (CPU) bude centrální modul aplikace, který bude řídit chod celého programu a pracovat s ostatními moduly. Budou zde ošetřeny události registrace modulů překladače a paměti `RegisterCompiler` a `RegisterMemory`, ve kterých bude předaný objekt uložen do odpovídající proměnné modulu. Jakákoliv práce s překladačem nebo pamětí bude prováděna právě prostřednictvím těchto objektů. Modul bude poskytovat metody pro ovládání programu, které budou přímo navázané na ovládací tlačítka popsané v sekci 4.1:

- Metoda `void StartProgram()` započne nebo obnoví provádění programu. Bude napojena na tlačítko START.
- Metoda `void StopProgram()` ukončí provádění programu. Bude napojena na tlačítko STOP.
- Metoda `void PauseProgram()` pozastaví vykonávání programu na aktuální instrukci. Bude napojena na tlačítko PAUSE.
- Metoda `void NextInstruction()` vykoná následující instrukci označenou instrukčním ukazatelem (`InstructionPointer`). Bude napojena na tlačítko STEP.

Dále budou v tomto modulu implementovány metody vykonávající funkce jednotlivých instrukcí assembleru, popsaných v následující sekci. Procesor implementovaný tímto modulem bude střádačového typu a tak bude obsahovat pouze tři registry: registr instrukční ukazatel (`InstructionPointer`) označující následující instrukci, registr akumulátor (`Accumulator`) pomocí něhož se bude pracovat s daty a registr adresový ukazatel (`AddressPointer`), v němž bude uložena adresa paměti, do níž se právě přistupuje.

Aplikace je navrhnuta s ohledem na možné rozšíření. Architektura modulů je navržena tak, že je možné kdykoliv přidat další druh paměti bez nutnosti provádění úprav v modulu

procesoru nebo překladače. Podobně to je i u modulu překladače, takže je možné simulátor předělat na jiný programovací jazyk, přičemž je nutné v modulu procesoru upravit pouze metody implementující jednotlivé instrukce. Modely paměti jsou navrženy s nastavitelnou kapacitou hlavní a vyrovnávací paměti a natvrdo danými algoritmy zápisu do paměti a výběru oběti. Jsou ale připraveny na možnost specifikace algoritmů zápisu do paměti a výběru oběti pomocí vlastností (props) komponenty. Takto by simulátor mohl být rozšířen na univerzální simulátor činnosti vyrovnávacích pamětí procesoru s podobným rozsahem přizpůsobení parametrů vyrovnávací paměti, jako mají simulátory popsané v sekci 3.1.

4.4 Návrh assembleru

Pro potřeby simulátoru bude použit vlastní assembler převzatý z literatury [21]. Tento assembler je postaven pro minimální RISC procesor obsahující jeden datový registr střádač (dále značen jako ACC), instrukční ukazatel (IP) a paměť. Názvy a význam instrukcí assembleru je vypsán v tabulce 4.1. Názvy instrukcí nerozlišují velká a malá písmena.

Instrukční sada bude rozšířena o návěští, pomocí nichž bude možné ve skokových instrukcích odkazovat na specifické místo v kódu bez nutnosti definice konkrétní adresy v programu. Návěští se budou zapisovat jako řetězec rozlišující velká a malá písmena končící znakem dvojtečka (":"). Návěští nesmí být shodné s názvem jakékoliv instrukce. Do instrukční sady bude přidána jedna pomocná vnitřní instrukce, a to instrukce `END` označující konec programu. Tuto instrukci nebude implementovaný překladač rozeznávat jako validní instrukci, pouze ji po úspěšném dokončení překladu programu přidá na konec seznamu obsahující vnitřní reprezentaci programu. Pro rozlišení parametru označující hodnotu a parametru označující adresu v paměti se adresy v paměti budou zapisovat se znakem zavináč ("@") před samotnou adresou. Do kódu bude možné psát řádkové komentáře, které začínají znakem středník (";").

Instrukce	Parametr	Popis
HALT	Žádný	Zastaví provádění programu na této pozici
NEGATE	Žádný	Vytvoří dvojkový doplněk (negace) hodnoty v ACC
ACCDEC	Žádný	Sníží hodnotu v ACC o jedna
ACCINC	Žádný	Zvýší hodnotu v ACC o jedna
NOP	Žádný	Prázdná operace
OUTP	Žádný	Původně zapsání hodnoty v ACC na port se zadanou adresou, protože simulátor nemá vstup-výstupní zařízení tak instrukce nemá žádný efekt
INP	Žádný	Původně načtení hodnoty na portu zadaným adresou do ACC, protože simulátor nemá vstup-výstupní zařízení tak instrukce nahraje do ACC náhodnou hodnotu v rozmezí $\langle 0, 1023)$
MLOAD	Hodnota	Nahraje do ACC zadanou hodnotu
DLOAD	Adresa	Nahraje do ACC hodnotu uloženou v paměti na zadané adrese
ILOAD	Adresa	Nahraje do ACC hodnotu uloženou v paměti na adrese, která je určena obsahem paměti na zadané adrese
DSTORE	Adresa	Uloží hodnoty v ACC na zadanou adresu v paměti
ISTORE	Adresa	Uloží hodnotu v ACC na adresu v pamětí, která je určena obsahem paměti na zadané adrese
BRANCH	Návěští	Změní hodnotu v IP na pozici daným návěštím
BRZERO	Návěští	Změní hodnotu v IP na pozici daným návěštím pokud se hodnota ACC rovná nule
BRPOS	Návěští	Změní hodnotu v IP na pozici daným návěštím pokud je hodnota ACC větší než nula
BRNEG	Návěští	Změní hodnotu v IP na pozici daným návěštím pokud je hodnota ACC menší než nula
MADD	Adresa	Přičte k hodnotě v ACC hodnotu v paměti na zadané adrese
IJUMP	Adresa	Nepřímý skok na adresu uloženou v paměti na zadané adrese
FLUSH	Žádný	Vyprázdní obsah vyrovnávací paměti do hlavní paměti a vynuluje čítač cyklů procesoru (dodatečná instrukce)

Tabulka 4.1: Sada instrukcí assembleru použitého jako programovací jazyk v simulátoru vyrovnávacích pamětí [21]. Instrukce mohou mít maximálně jeden parametr, podle kterého lze instrukce rozdělit do čtyř skupin: instrukce bez parametru, instrukce s adresovým parametrem, instrukce s parametrem návěští a instrukce s hodnotovým parametrem. Pomocí této strohé instrukční sady je možné implementovat libovolný algoritmus.

Kapitola 5

Implementace simulátoru činnosti vyrovnávacích pamětí

V této kapitole je popsán proces implementace aplikace simulátor činnosti vyrovnávacích pamětí. Aplikace byla implementována na základě návrhu popsaným v kapitole 4, v němž byla celková architektura aplikace rozdělena do tří hlavních modulů. Je zde popsána vnitřní implementace těchto modulů, jejich pomocné komponenty, způsob komunikace s okolí a také jsou zde vyobrazeny důležité nebo problematické sekce kódu. Jsou zde popsány významné implementační rozhodnutí a jejich dopad na výslednou funkcionalitu aplikace, případně na implementaci dalších prvků aplikace. Dále jsou zde popsány další relevantní komponenty, které zajišťují vedlejší funkce nebo tvoří grafické uživatelské rozhraní. Nakonec je zde uveden způsob testování správné funkce překladače a procesoru assembleru.

Aplikace má být podle návrhu vytvořena jako webová aplikace bez backend serveru, a tak byl pro implementaci zvolen javascriptový frontend framework **Vue.js** verze 3.2.20. Vue.js poskytuje nadstavbu nad standardními webovými prostředky HTML, CSS a JavaScript nebo TypeScript, která umožňuje deklarativní a komponentově orientované programování uživatelských rozhraní [22]. Blíže je framework Vue.js popsán v sekci 3.2. Pro testování byl použit javascriptový testovací framework **Jest** poskytující jednoduchý přístup k jednotkovému a celkovému testování javascriptových programů. Jako systém správy verzí byla použita webová služba GitHub, v níž byly vytvořeny pracovní postupy pro průběžnou integraci (CI[1]) a průběžné nasazení (CD[2]). Výsledná aplikace je nasazena na webhostingu GitHub Pages.

Aplikace byla vyvíjena inkrementálně ve třech fázích: implementace základní funkcionality, stylování grafického uživatelského rozhraní (GUI) a implementace rozšíření. Nejdříve byly vytvořeny komponenty implementující všechny vyžadované funkce překladače, procesoru a modelu paměti bez vyrovnávací paměti s pouze rudimentárním GUI. Na těchto základních komponentách byla vytvořena sada jednotkových testů pro překladač a procesor. Následně byly doplněny zbývající typy vyrovnávacích paměti a otestována jejich funkce. Po potvrzení správné funkcionality všech komponent se začalo pracovat na stylu GUI a animacích sloužící pro vizualizaci funkce vyrovnávací paměti. Nakonec byly implementovány dodatečná rozšíření jako čítač cyklů procesoru nebo okno pro nastavení vlastností simulátoru.

[1] CI – Continuous Integration
[2] CD – Continuous Deployment

5.1 Komponenta překladač kódu

Jako první byl implementován překladač assembleru. Implementace překladače je rozdělena do dvou souborů: `Compiler.js`, který obsahuje samotný překladač zadaného assembleru do vnitřní reprezentace programu, a `CodeEditor.vue` obsahující editor kódu a metody implementující rozhraní překladače tak, jak je popsané v sekci 4.3. Pro jednotné definování názvů instrukcí assembleru je použita třída `Instruction` uložená ve stejnojmenném javascript souboru, která funguje jako výčet (enum) názvů instrukcí. Tento způsob definice výčtu v JavaScriptu byl převzat z [9] a je znázorněn na obrázku 5.1.

```
export default class Instruction {
    static HALT = new Instruction("HALT");
    static NEGATE = new Instruction("NEGATE");
    static ACCDEC = new Instruction("ACCDEC");
    static ACCINC = new Instruction("ACCINC");
    // ...

    constructor (name) {
        this.name = name
    }
}
```

Obrázek 5.1: Javascriptová třída `Instruction` sloužící jako výčet (enum) instrukcí. Jednotlivé hodnoty výčtu jsou uloženy jako statické (třídní) vlastnosti této třídy. K dané hodnotě výčtu lze přistoupit následujícím způsobem: `Instruction.XXX.name`.

Překladač kódu

Modul překladač (`Compiler.js`) exportuje jedinou funkci `startCompilation`, která jako parametry přijímá vstupní assembler a výstupní seznam instrukcí ve vnitřní reprezentaci programu `instructionList`. Tato funkce započne překlad programu. Poněvadž je celý vstupní assembler předáván ve formě textového řetězce je nejprve nutné jej rozdělit na jednotlivé lexémy (slova). To je provedeno dvojím rozdělením vstupního kódu; nejdřív na řádky a poté na jednotlivá slova. Rozdělení kódu na řádky umožňuje u každé instrukce zaznamenat její umístění ve zdrojovém kódu. Následně se jednotlivé lexémy postupně, jeden po druhém, procházejí a na základě jejich hodnoty se provede další zpracování. Překladač nerozlišuje velká a malá písmena.

Instrukce použitého assembleru lze rozdělit do 4 skupin na základě jejich parametrů. Instrukce v rámci jedné skupiny lze v překladači zpracovávat stejným způsobem:

- Instrukce bez parametru jsou zpracovány `processParameterlessInstruction()`.
- Instrukce s parametrem adresa jsou zpracovány `processAddressInstrution()`.
- Instrukce s parametrem hodnota jsou zpracovány `processValueInstruction()`.
- Instrukce s parametrem návěštím jsou zpracovány `processLabelInstruction()`.

Tyto funkce provedou kontrolu formátu a hodnoty případného parametru a následně vloží instrukci s jejím parametrem a číslem řádku, na němž se nachází, na konec seznamu instrukcí ve vnitřní reprezentaci programu `instructionList`. Parametr typu adresa je hexadecimální hodnota začínající znakem zavináč ("@"), hodnotový parametr je desítkové číslo a parametr návěští je řetězec označující dané návěští. Pokud formát parametru neodpovídá dané instrukci nebo má neplatnou hodnotu tak se vyvolá výjimka s chybovým hlášením.

Dva druhy lexémů jsou zpracovány odlišně, a to komentáře a označení návěští. Pokud lexém začíná znakem středník (";"), tak je zbytek řádku považován za komentář a je tedy nepotřebné ho zohledňovat ve vnitřní reprezentaci programu. Zpracování zbytku řádku je tak přeskočeno. Pokud je lexém označením návěští (formát zápis návěští popsán v sekci 4.4), tak je ve výstupním seznamu instrukcí reprezentován jako vnitřní instrukce `LABEL`, jejíž parametr obsahuje název tohoto návěští. Jméno návěští nesmí být shodné s názvem jakékoliv instrukce assembleru.

Editor kódu

Vue.js komponenta **editor kódu** (`CodeEditor.vue`) zajišťuje celkově tři funkce: grafický editor kódu, překladač assembleru a také paměť programu. Pro grafický editor kódu byl využit balíček *Vue Prism Editor*[3]. Tento balíček implementuje minimální editor kódu podporující zvýrazňování syntaxe, formátování kódu a číslování řádků. Balíček byl zvolen jednak kvůli své malé velikosti a jednoduchosti editoru, ale také protože to je jeden z mála editorů kódu aktualizovaných pro Vue.js verzi 3. Syntaktické zvýrazňování kódu je zajištěno knihovnou *Prism*, která poskytuje pravidla pro syntaktické zvýrazňování mnoha dostupných programovacích jazyků a také prostředky pro definici vlastních pravidel a stylů syntaktického zvýrazňování[4]. Zvýrazňovací pravidla jsou zapsaná pomocí regulérních výrazů, které vyjadřují daný token (lexém). Pro použitý assembler byly vytvořeny vlastní pravidla, umístěná v souboru `bp22Highlighting.js`, a styly nacházející se v souboru `bp22Style.css`.

Po dokončení vytváření objektu komponenty editor kódu je v těle metody `created()`, která reprezentuje právě tuto událost, vyvolána událost `RegisterCompiler` rodičovské komponenty a jako její parametr je předán objekt obsahující odkazy na metody implementující rozhraní překladače: Metoda `CompileProgram()` započne překlad zadaného assembler programu pomocí funkce poskytnuté modulem `Compiler.js`. Po úspěšném dokončení překladu se ještě provede jeden průchod seznamem instrukcí `instructionList` a do asociativního pole `labelDict` se zaznamenají všechny návěští a jejich pozice ve vnitřní reprezentaci kódu. Metoda `GetInstruction(address)` slouží pro získání objektu reprezentující instrukci podle dané adresy a metoda `GetLabel(label)` pro získání pozice ve vnitřní reprezentaci kódu označené daným návěštím. Pokud daná adresa instrukce nebo návěští neexistuje, tak se vyvolá výjimka.

Použitý balíček pro editor kódu neobsahuje prostředky pro zvýrazňování řádků, jenž se využívá při vykonávání programu pro označení aktuální pozice v kódu. Tudíž byly do rozhraní překladače navíc přidány dvě další metody implementující tuto funkcionalitu. Pomocná metoda `GetNextLine(address)` slouží pro získání čísla řádku instrukce na dané adrese. Pokud tato adresa instrukce neexistuje, tak se vrací hodnota -1, která znamená neoznačovat žádný řádek. Samotné zvýraznění daného řádku zdrojového kódu obstarává metoda `HighlightLine(lineNumber)`. Tato metoda nejdřív zruší případné předchozí zvýraznění řádku, který je zaznamenán v proměnné `highlightedLine`, a následně zvýrazní řádek

[3] https://github.com/koca/vue-prism-editor/tree/feature/next
[4] https://prismjs.com/

daným parametrem metody. Zvýraznění je implementováno výběrem DOM uzlu reprezentující daný řádek v editoru kódu, k němuž je následně přiřazena CSS třída `highlight-line` zajišťující samotné zvýraznění. Tento způsob zvýraznění byl převzat ze zdrojového kódu balíčku *VueTut* [20] a je znázorněný na obrázku 5.2.

```
HighlightLine(newLine) {
  // Odstranit původní zvýraznění
  if (oldLine >= 0 ) {
    var oldEl = $el.querySelector(`.prism-
editor__line-number:nth-child(${oldLine+1})`);
    if (oldEl == null) return;
    oldEl.classList.remove('highlight-line');
  }
  oldLine = newLine;
  // Nové zvýraznění
  if (newLine >= 0) {
    var newEl = $el.querySelector(`.prism-
editor__line-number:nth-child(${newLine+1})`);
    if (newEl == null) return;
    newEl.classList.add('highlight-line');
  }
}
```

Obrázek 5.2: Metoda zajišťující označení čísla řádku v editoru kódu obsahující instrukci, která se provede další. Každé číslo řádku představuje v HTML struktuře stránky samostatný element. Díky tomu můžeme specifikovat n-tý řádek jako $(n+1)$-týho sourozence CSS třídy `prism-editor__line-number`. Pak už stačí tomuto elementu přidat nebo odebrat třídu `highlight-line` zajišťující samotné grafické zvýraznění.

Zvýraznění řádku bylo z důvodu vnitřní struktury editoru kódu implementováno pouze jako barevné označení čísla řádku, na kterém se daná instrukce nachází. Pole pro kód je implementováno jako textové pole (`text area`) obsahující jednotlivé tokeny, které reprezentující lexémy, a tudíž je mnohem obtížnější vybrat pomocí dotazu v DOM struktuře konkrétní element. Kvůli tomu není možné přesně zvýraznit danou instrukci v případě, kdy je na jednom řádku zapsáno více instrukcí, ale pouze celý řádek.

5.2 Komponenta paměť

Následně byly vytvořeny všechny čtyři typy pamětí. Jako první byla implementována paměť bez vyrovnávací paměti (*Ram Only Memory*), která je ze všech navrhnutých pamětí nejjednodušší a na níž bude také provedeno testování procesoru, a až poté byly implementovány paměti s vyrovnávací pamětí. Implementace paměti je rozdělena do dvou adresářů: v adresáři `memory` se nachází komponenty jednotlivých typů paměti obsahující veškerou vnitřní logiku paměti a v adresáři `model` jsou komponenty implementující grafickou reprezentaci

jednotlivých prvků paměti. Paměti s vyrovnávací pamětí navíc využívají třídu `CacheBlock` reprezentující jeden paměťový blok ve vyrovnávací paměti a třídu `MemoryUtils` v níž jsou implementovány společné funkce. Tímto způsobem hlavní komponenta daného typu paměti kompletně zapouzdřuje grafické rozhraní paměti a zároveň přenechává vizualizaci a formátování jiným komponentám. Obsahem každého datového bloku paměti je pole čtyř číselných hodnot, k nimž lze ve vyrovnávací paměti přistupovat individuálně, ale v hlavní paměti se s nimi zachází jako s celkem.

Vizuální modely prvků paměti

Jednotlivé prvky grafického uživatelského rozhraní paměti jsou implementovány vlastními komponentami. Tyto komponenty zajišťují vizualizaci daného prvku paměti a animace práce s pamětí, ale neobsahují žádnou vnitřní logiku, která by ovlivňovala chování paměti. Aplikace využívá čtyři takovéto komponenty: **model procesoru**, **model vyrovnávací paměti**, **model hlavní paměti** a **konektor**.

Model procesoru, nacházející se v souboru `ProcessorModel.vue`, je jednoduchá komponenta, která vizuálně modeluje procesor a obsah jeho registrů. Model zobrazuje následující registry: instrukční ukazatel (IP), akumulátor (ACC) a adresový ukazatel (AP) a navíc je v něm také zobrazen název aktuálně prováděné instrukce. Registry jsou reprezentovány komponentami `RegisterLabel.vue` obsahující popisek registru a jeho obsah. Registr adresový ukazatel je formátován jako hexadecimální číslice.

Model vyrovnávací paměti (`CacheModel.vue`) je implementován jako prostá tabulka. Každý řádek tabulky reprezentuje jeden paměťový blok vyrovnávací paměti a sloupce představují v následujícím pořadí: adresu bloku vyrovnací paměti, příznak platnosti, adresový příznak obsahu a nakonec samotný obsah daného bloku vyrovnávací paměti. Model používá metody `FormatAddressHex(address, tagLenght)` a `FormatAddressBin(address, tagLength)` pro formátování adresy a adresového příznaku jako hexadecimální a binární číslo. Parametr `tagLength` je použit pro určení celkového počtu bitů adresového příznaku, protože jej v případě vyššího stupně asociativity vyrovnávací paměti nelze nijak odvodit z kapacity paměti. V modelu je pro názornost zobrazena jak hexadecimální, tak binární hodnota adresy a adresového příznaku. Pokud má příznak platnosti nepravdivou hodnotu, tak je barevně zvýrazněn. To je provedeno metodou `HighlightInvalid(valid)`, která v případě negativní hodnoty přiřadí buňce tabulky s příznakem platnosti CSS třídu implementující toto zvýraznění. Hodnotu buňky příznaku platnosti lze dvojklikem změnit na opačnou, vyvoláním události `SwitchValidBit` s parametrem indexu řádku vyrovnávací paměti.

Model vyrovnávací paměti umožňuje pomocí barevného pulsu pozadí řádku zvýraznit, se kterým paměťovým blokem se právě pracuje. To je zajištěno metodou `HighlightRow(id)`, která je poskytnuta rodičovské komponentě pomocí události `RegisterCache` vyvolané při inicializaci komponenty model vyrovnávací paměti. Kód zajišťující barevný puls je zobrazen na obrázku 5.3. Pro případ, kdy je vyvoláno zvýraznění řádku, zatímco předchozí zvýraznění stále probíhá je použita metoda `ResetHighlight()`, která předchozí zvýraznění předčasně ukončí. Zvýraznění a následné postupné vyblednutí je provedeno pomocí struktury `rowStyle`, ve které je specifikováno pozadí zvýrazněné buňky. Tato struktura je přiřazena k atributu `style` všem buňkám v rámci specifikovaného řádku tabulky. Styl je přiřazen k jednotlivým buňkám řádku tabulky a ne k celému řádku z důvodu zachování původního pozadí tabulky, protože JavaScript neposkytuje nástroje pro plynulý přechod barvy mezi dvěma hodnotami. Takto si pozadí řádku udržuje svou barvu pozadí, které buňka překrývá jinou barvou, jíž se postupně mění průhlednost.

```
HighlightRow(id, fadeTime) {
  ResetIntervals(); ResetHighlight();
  highlightedRow = Math.floor(id/4);
  var i = fadeTime; // Animace MAX -> MIN

  // Opakující se interval zajišťující animaci
  fadeOut.interval = setInterval(() => {
    var fadeHex = (Math.floor((i/fadeTime)*255))
      .toString(16).padStart(2, '0');
    style.background = highlightColor+fadeHex;
    i -= 10;
  }, 10);
  // Časovač konce animace
  fadeOut.timeout = setTimeout(() => {
    clearInterval(fadeOut.interval);
    fadeOut.interval = null;
    fadeOut.timeout = null;
  }, fadeTime);
}
```

Obrázek 5.3: Metoda implementující barevné zvýraznění řádku v tabulce paměti. Toto zvýraznění pomocí barevného pulsu slouží pro vizuální označení paměťového bloku, se kterým se právě pracuje. Metoda inicializuje asynchronní interval a časovač. V intervalu je každých deset milisekund upravena hodnota alfa kanálu barvy pozadí řádku. Ta je přiřazena do proměnné `style`, která uchovává styl aplikovaný na řádek. Časovač slouží pro ukončení intervalu po uběhnutí doby trvání animace. Pro vyvarování neočekávaných stavů se před spuštěním intervalu a časovače provede metoda `ResetIntervals()`, jenž zajistí ukončení intervalu a časovače z předchozího volání metody.

Model hlavní paměti (`RamModel.vue`) je velmi podobný modelu vyrovnávací paměti. Je tvořen prostou tabulkou, ve které řádky představují jednotlivé paměťové bloky hlavní paměti a sloupce obsahují adresu a obsah daného bloku. Adresa paměťového bloku je formátována na hexadecimální a binární hodnotu stejně jako adresa bloku ve vyrovnávací paměti. Stejně tak i tento model umožňuje barevným pulsem pozadí řádku zvýraznit, se kterým paměťovým blokem se pracuje. To je provedeno metodou `HighlightRow(id)`, která je zpřístupněna v parametru události `RegisterRam` vyvolané při vytvoření této komponenty. Implementace této funkce je stejná jako na obrázku 5.3.

Model konektoru (`Connector.vue`) slouží pro vizuální propojení modelu procesoru, vyrovnávací paměti a hlavní paměti. Je to vlastně jednoduchá horizontální roura, která posunem horizontální lišty umístěné v ní vizualizuje pohyb dat mezi jednotlivými prvky paměti. Pomocí události `RegisterConnector` poskytuje tato komponenta dvě asynchronní metody: `FromCpuToMemory()` a `FromMemoryToCpu()`. Obě tyto metody započnou animaci pohybu dat, liší se pouze, jakým směrem se lišta pohybuje. Pro správné a srozumitelné

fungování aplikace, kdy se musí čekat na dokončení této animace, je nutné používat tyto metody s klíčovým slovem `await`.

Hlavní logika animace je obsažena v metodě `AnimateBar`. Metoda je rozdělena na dvě části: v první se provede posunutí horizontální lišty a v druhé tato lišta postupně vybledne. Obě tyto části fungují na podobném principu jako zvýraznění řádku tabulky hlavní nebo vyrovnávací paměti. Vytvoří se asynchronní interval, který za každý časový úsek upraví objekt reprezentující styl horizontální lišty, a časovač, jenž za určitou dobu tento interval ukončí. V případě animace posunutí se čeká, dokud se nedokončí neboli dokud nedoběhne k ní přiřazený časovač, v případě vyblednutí se hned pokračuje dál bez čekání. Kód zajišťující animaci horizontálního posuvníku je zobrazen na obrázku 5.4, kód pro vyblednutí je podobný kódu, co je znázorněn na obrázku 5.3. I zde je metoda `ResetIntervals()` pro ukončení spuštěných časovačů v případě, že je animace konektoru vyvolána před dokončení předchozí animace.

```
async AnimateBar(fillTime, fadeTime) {
  var i = 10; // Animace MIN -> MAX
  // Opakující se interval zajišťující animaci
  fillBar.interval = setInterval(() => {
    barStyle.width = ((i/fillTime)*100)+'%';
    i += 10;
  }, 10);
  // Časovač konce animace
  var promise = new Promise((resolve) => {
    fillBar.timeout = setTimeout(() => {
      clearInterval(fillBar.interval);
      fillBar.timeout = fillBar.interval = null;
      resolve();  // Slib dokončen
    }, fillTime);
  });
  await promise;  // Čeká se na dokončení slibu
  // Vyblednutí lišty
  // ...
}
```

Obrázek 5.4: Část metody animace konektoru, která provádí posunutí horizontální lišty. Metoda vytvoří asynchronní interval, v němž se každých deset milisekund zvětší šířka horizontální lišty. Při animaci posunu je potřeba aby celý simulátor čekal na její dokončení. K tomu je použita třída `Promise`, která pomáhá s implementací asynchronního chování. Vytvoří se instance třídy `Promise` v níž je umístěn časovač posunu lišty. Jakmile v časovači uběhne zadaný čas tak se přeruší interval a metodou `resolve()` se ukončí čekání na slib.

Paměť bez vyrovnávací paměti

Paměť bez vyrovnávací paměti je oproti ostatním typům paměti specifická právě absencí vyrovnávací paměti, a tudíž je její vnitřní implementace odlišná od ostatních typů. Je obsažena v souboru `RamOnlyMemory.vue` a využívá komponenty model procesoru, model hlavní paměti a konektor popsaných v předchozím textu pro vizuální reprezentaci. Při vytvoření komponenty se pomocí události `RegisterMemory` registruje u rodičovské komponenty objekt obsahující odkazy na metody `Write()`, `Read()`, `Flush()` a `Initialize()`. Metoda `Flush()` nemá s absencí vyrovnávací paměti význam a tudíž je její tělo prázdné. Metoda je zachována jenom kvůli udržení jednotného způsobu práce s ostatními typy paměti.

Obsah paměti je reprezentován jako dvourozměrné pole čísel, kde první rozměr je dán velikostí paměti a druhý rozměr je vždy roven čtyřem. Metody pro čtení z paměti (`Read(address)`) a zápis do paměti (`Write(address, data)`) jsou pak pouze jednoduchým přístupem na daný index v poli, při němž se spustí metody pro zvýraznění daných dat a animaci konektoru. Obě metody jsou asynchronní a je nutné je používat s klíčovým slovem `await`, jinak může nastat datová nekonzistence. Tato nekonzistence nastává právě kvůli animacím konektoru, které zdržují samotný přístup do paměti, zatímco procesor pokračuje se zpracováním další instrukce. Při přístupu na adresu mimo rozsah paměti se vyvolá výjimka `RangeError`.

Paměti s vyrovnávací pamětí

Zbývající tři typy pamětí s vyrovnávací pamětí, vyrovnávací paměť s přímým mapováním (umístěná v souboru `DirectCacheMemory.vue`), s dvoucestným skupinovým mapováním (`TwoWayCacheMemory.vue`) a s plně asociativním mapováním (`FullCacheMemory.vue`), mají podobnou strukturu, a tudíž budou popsány dohromady. Všechny tři plně využívají vytvořených vizuálních modelů pro reprezentaci procesoru, vyrovnávací paměti, hlavní paměti a cest mezi nimi. Paměti poskytují své rozhraní stejným způsobem jako paměť bez vyrovnávací paměti, a to pomocí události `RegisterMemory`. Hlavní paměť je reprezentována také jako dvourozměrné pole čísel a vyrovnávací paměť jako jednorozměrné, v případě paměti s dvoucestným mapováním dvourozměrné, pole objektů třídy `CacheBlock`. Třída `CacheBlock` je pomocná třída reprezentující jeden paměťový blok ve vyrovnávací paměti a obsahuje samotná data, adresový příznak, příznak platnosti a pomocný příznak pro první použití tohoto paměťového bloku.

Protože při čtení a zápisu do paměti se ve všech třech typech pamětí opakovaně vykonávají stejné posloupnosti operací, byla vytvořena pomocná třída `MemoryUtils.js`. Tato třída slouží jako mezivrstva pro práci s pamětí. V konstruktoru je nutné této třídě předat objekty reprezentující veškeré komponenty potřebné pro práci s vyrovnávací pamětí a animacemi.

Třída poskytuje následující asynchronní metody:

- Metoda `readFromRam(cacheAddress, ramAddress, tag, path?)` pro přesun bloku z hlavní paměti do vyrovnávací paměti.
- Metoda `writeToRam(cacheAddress, ramAddress, path?)` pro přesun bloku z vyrovnávací paměti do hlavní paměti.
- Metoda `readFromCache(cacheAddress, path?)` pro čtení dat z vyrovnávací paměti.
- Metoda `writeToCache(cacheAddress, tag, data, path?)` pro zápis dat do vyrovnávací paměti.

Všechny tyto metody zajišťují grafické zvýraznění použitého datového bloku, spuštění animace konektoru, aktualizaci či navrácení požadovaných dat a také inkrementaci čítače cyklů procesoru o hodnotu přístupu do paměti. Parametr `path` je volitelný parametr použitý u paměti s dvoucestným skupinovým mapováním, který specifikuje, se kterou skupinou vyrovnávací paměti se právě pracuje. Všechny výše uvedené metody mají proto dvě implementace: jednu pro skupinově asociativní paměti a druhou pro paměť s přímým mapováním a plně asociativním mapováním. Navíc třída `MemoryUtils` poskytuje pomocnou metodu `getRamAddressFromCache()` pro výpočet adresy paměťového bloku v hlavní paměti z adresy vyrovnávací paměti a adresového příznaku.

Čtení i zápis do paměti má ve všech třech typech paměti s vyrovnávací pamětí podobné logické schéma, které je popsané v pseudokódu na obrázku 5.5. Nejdříve se z parametru adresy paměťového bloku vypočítá adresa ve vyrovnávací paměti a adresový příznak. Následně se zkontroluje obsah vyrovnávací paměti na dané adrese. Pokud adresový příznak na dané adrese vyrovnávací paměti odpovídá vypočtenému adresovému příznaku, tak se požadovaná data nachází ve vyrovnávací paměti a je možné je přímo číst nebo přepsat. Jinak data nejsou uložena ve vyrovnávací paměti a musí se do ní nejprve načíst. U modelu vyrovnávací paměti s dvoucestným skupinovým mapováním a plně asociativním mapováním se musí vybrat paměťový blok, do kterého budou data načtena, neboli je potřeba vybrat oběť pro nová data. Jako algoritmus výběru oběti je použit náhodný výběr oběti popsaný v sekci 2.4. Pro zlepšení výkonu vyrovnávací paměti byl algoritmus modifikován tak, že dosud nepoužité paměťové bloky jsou vybrány přednostně. To omezuje situaci, kdy se kvůli náhodě opakovaně vybírá ten samý paměťový blok jako oběť, přestože ve vyrovnávací paměti existují ještě nepoužité paměťové bloky. Jestliže je vybraný paměťový blok již obsazen jinými daty je nutné zkontrolovat hodnotu příznaku platnosti. Pokud je jeho hodnota kladná, tak jsou data ve vyrovnávací paměti koherentní s daty v hlavní paměti a paměťový blok se může přímo přepsat novým paměťovým blokem, jinak je potřeba tyto data nejprve uložit do hlavní paměti a až poté je přepsat novým paměťovým blokem.

Dále všechny modely paměti s vyrovnávací pamětí implementují metodu `Flush()` pro přesunutí všech dat z vyrovnávací paměti do hlavní paměti a metodu `Initialize()` pro inicializaci polí reprezentující hlavní a vyrovnávací paměť a vytvoření nové instance třídy `MemoryUtils`. Také implementují metodu `SwitchValidBit(row)` vyvolané událostí dvojklik z modelu vyrovnávací paměti, která přepne hodnotu příznaku platnosti daného paměťového bloku ve vyrovnávací paměti.

```
function CacheReadWrite(address, data) {
  if(!isValid(address)) throw Error();
  // Vypočítat adresu ve VP a adresový příznak
  var cAdr = CalcCacheAdr(address);
  var cTag = CalcCacheTag(address);

  // Zkontrolovat obsah VP
  if (isInCache(cAdr, cTag)) {
    WriteReadToCache(cAdr, cTag, data); return;
  }
  // Blok není ve VP
  // Výběr oběti
  if (!isDirectCache()) nAdr = VictimRand(cAdr);
  else nAdr = cAdr;
  // Vybraný blok je koherentní, můžeme přepsat
  if (cache[nAdr].valid) {
    LoadFromRam(address, nAdr, cTag);
    WriteReadToCache(nAdr, cTag, data);
  }
  // Vybraný blok není koherentní, musíme uložit
  else {
    tmpAdr=GetRamAddress(cache[nAdr].tag, nAdr);
    StoreToRam(tmpAdr, nAdr);
    LoadFromRam(address, nAdr, cTag);
    WriteReadToCache(nAdr, cTag, data);
  }
}
```

Obrázek 5.5: Pseudokód algoritmu pro čtení nebo zápis do paměti s vyrovnávací pamětí. Všechny tři implementované typy vyrovnávacích pamětí mají svou vnitřní implementaci metod pro čtení a zápis do paměti podle tohoto schématu. Liší se v především v algoritmu kontroly obsahu vyrovnávací paměti (metoda `isInCache()`), výběru oběti (metoda `VictimRand()`) a také způsoby výpočtu adresy ve vyrovnávací paměti a adresového příznaku.

5.3 Komponenta procesor

Po vytvoření komponenty paměti bez vyrovnávací paměti byl implementován procesor. Komponenta procesor (`TheProcessor.vue`) představuje centrální komponentu aplikace, v níž je umístěno celkové rozložení aplikace a metody implementující funkce ovládacích

tlačítek. Samotný procesor, implementující chování jednotlivých funkcí assembleru, je obsažen ve třídě `Cpu` (`Cpu.js`).

Procesor assembleru

Ve třídě `Cpu` je implementováno chování jednotlivých instrukcí assembleru. V konstruktoru jsou této třídě předány objekty nutné pro vykonávání instrukcí: objekt s rozhraním použitého typu paměti, registr akumulátor a registr adresový ukazatel, a navíc také čítač cyklů procesoru. Třída má jedinou veřejnou metodu `execute(instruction)`, která za pomocí výčtu instrukcí `Instruction` identifikuje instrukci a tuto konkrétní instrukci následně vykoná. Každá assembler instrukce je implementovaná vlastní metodou vykonávající funkci assembler instrukce tak jak je popsané v sekci 4.4. Pokud procesor narazí na neznámou instrukci, tak se vyvolá výjimka.

Metody implementující instrukce assembleru vrací objekt, který specifikuje další činnost procesoru. Pro jednotnost jsou všechny možné následující činnosti procesoru po vykonání instrukce definovány ve výčtové třídě `ExecutionResult`, jenž je implementována stejným způsobem jako třída `Instruction` popsaná v sekci 5.1. Tato třída definuje pět různých akcí, které mohou nastat po vykonání instrukce:

- `NextInstruction` – Po dokončení aktuální instrukce se provede instrukce na následující pozici v seznamu instrukcí. Většina instrukcí assembleru pokračuje právě touto činností.
- `HaltExecution` – Po dokončení aktuální instrukce se pozastaví provádění programu, jako kdyby bylo zmáčknuto tlačítko PAUSE. Touto činností pokračuje pouze instrukce `HALT`.
- `EndExecution` – Po dokončení aktuální instrukce se ukončí provádění programu, jako kdyby bylo zmáčknuto tlačítko STOP. Touto činností pokračuje pouze pomocná instrukce `END` pro ukončení vykonávání programu.
- `MoveToLabel` – Po dokončení aktuální instrukce se pokračuje instrukcí označené daným návěští. Návěští je specifikováno jako další vlastnost návratového objektu. Touto činností pokračují skokové instrukce a podmíněné instrukce, pokud je podmínka splněna.
- `MoveToAddress` – Po dokončení aktuální instrukce se pokračuje instrukcí na dané adrese v seznamu instrukcí. Adresa je specifikována jako další vlastnost návratového objektu. Touto činností pokračuje pouze instrukce `IJUMP`.

Komponenta procesoru

Komponenta procesoru (`TheProcessor.vue`) obsahuje výše popsané komponenty pro editor kódu, kontejner umožňující výběr použitého typu paměti a čtyři tlačítka pro ovládání programu. Tyto tlačítka mění stav simulátoru mezi čtyřmi stavy: *not started*, *started*, *halted* a *ended*. Aktuální stav simulátoru je uložen v proměnné `currentState`. V této komponentě se udržují objekty reprezentující překladač (`compiler`) a paměť (`memory`) získané z registračních událostí `RegisterCompiler` a `RegisterMemory` a instance třídy procesoru `Cpu`. Jsou zde také proměnné představující registry procesoru: registr instrukční ukazatel (`instructionPointer`), registr akumulátor (`accumulator`) a registr adresový ukazatel

(`addressPointer`). Akumulátor a adresový ukazatel jsou reprezentovány jako objekty s jedinou vlastnosti `value` obsahující jejich hodnotu. To je z důvodu rozdílu způsobu předávání proměnných hodnotového datového typu, jako je právě typ `Number`, a referenční datových typů jako parametry metod a funkcí. V javascriptu neexistuje klíčové slovo `ref` pro předávání hodnotových datových typů odkazem. Tudíž jsou tyto dva registry, které mohou být modifikovány na různých místech programu, reprezentovány jako objekt, jehož vlastnost obsahuje právě hodnotu registru.

Komponenta procesoru implementuje čtyři metody pro ovládání chodu simulovaného programu:

- Asynchronní metoda `StartProgram()` slouží pro spuštění nebo znovuspuštění programu zadaného v editoru kódu. Pokud je simulátor ve stavu *not started* nebo *ended* tak se provede překlad programu, inicializuje se obsahu registrů a paměti a vytvoří se nová instance procesoru `Cpu`. Následně se změní stav simulátoru na *started* a začnou se v cyklu vykonávat jednotlivé instrukce programu, dokud je stav simulátoru *started*. Po každém provedení instrukce se nějakou dobu čeká, aby byl čas na vnímání simulace.

- Metoda `StopProgram()` ukončí provádění aktuálního programu. Metoda změní stav simulátoru na *not started* a provede inicializaci registrů a paměti pro resetování simulátoru.

- Metoda `PauseProgram()` pozastaví automatické vykonávání programu. Metoda změní stav simulátoru na *halted* a simulátor po dokončení rozpracované instrukce dále čeká.

- Asynchronní metoda `ExecuteInstruction` slouží pro vykonání jedné instrukce. Používá se jak při krokování programu tlačítkem STEP tak i pro automatické vykonávání programu v cyklu v metodě `StartProgram`. Metoda získá podle hodnoty registru instrukční ukazatel metodou překladače `GetInstruction` následující instrukci. Tato instrukce se procesorem vykoná a podle hodnoty návratového objektu se určí nová hodnota instrukčního ukazatele nebo se změní stav simulátoru. Na konci se pomocí metod překladače `GetNextLine` a `HighlightLine` v editoru kódu zvýrazní následující instrukce.

Tato komponenta také zachytává výjimky vyvolané v komponentách překladače, paměti a samotném procesoru. Jsou zde dva body zachytávání výjimek: pro výjimky vyvolané během překladu programu v metodě `StartProgram` a pro výjimky vyvolané během vykonávání instrukce v metodě `ExecuteInstruction`. Po zachycení výjimky se zobrazí upozornění s chybovou zprávou. Pro zobrazení chybových upozornění se využívá balíček *Vue.js notifications*[5].

5.4 Pomocné komponenty

Během vývoje aplikace bylo také vytvořeno množství pomocných komponent, které sice nesouvisí přímo s komponentami navrženými v kapitole 4, ale poskytují jim další funkcionalitu. V této sekci budou popsány dvě hlavní pomocné komponenty: **lišta karet** a **okno nastavení simulátoru**.

[5] https://github.com/kyvg/vue3-notification

Lišta karet

Komponenta lišta karet (`TheTabContainer.vue`) slouží pro výběr modelu paměti, na němž bude provedena simulace. Figuruje jako mezivrstva mezi procesorem a konkrétní použitou pamětí, kvůli čemuž je v ní nutné přeposílat událost registrace paměti `RegisterMemory` a *props* předávané komponentě paměti. HTML šablona této komponenty je zobrazena na obrázku 5.6.

```
<div class="tab-button-row">
  <tab-button :displayValue="'RAM-Only Memory'"
    :active="activeTab['RamOnlyMemory']"
    :function="ChangeTab('RamOnlyMemory')"/>
  <tab-button :displayValue="'Direct Cache'"
    :active="activeTab['DirectCacheMemory']"
    :function="ChangeTab('DirectCacheMemory')"/>
  <tab-button :displayValue="'Two-Way Cache'"
    :active="activeTab['TwoWayCacheMemory']"
    :function="ChangeTab('TwoWayCacheMemory')"/>
  <tab-button :displayValue="'Full Cache'"
    :active="activeTab['FullCacheMemory']"
    :function="ChangeTab('FullCacheMemory')"/>
  <cycle-counter :cycles="cycleCounter.value"/>
</div>
<div class="tab-container">
  <component :is="currentTab"
    @RegisterMemory="RegisterMemory"
    :instruction="inst" :instructionPointer="ip"
    :accumulator="acc" :addressPointer="ap"/>
</div>
```

Obrázek 5.6: HTML šablona lišty karet je rozdělena na dvě části. V první části jsou umístěné jednotlivé karty. Při kliknutí na kartu se vyvolá metoda `ChangeTab("Název")`, která nastaví proměnnou `currentTab` na název vybraného typu paměti. V druhé části je umístěn speciální element `component`, který slouží jako zástupce pro obecnou Vue.js komponentu. Pomocí direktivy `:is` je podle názvu specifikována skutečná komponenta, jež se na daném místě vykreslí.

Komponenta se skládá ze dvou částí. V první části se nachází karty (`TabButton.vue`) sloužící pro přepínání aktuálního modelu paměti. Karta reprezentující určitý model paměti zde musí být explicitně vypsaná, aby šlo k danému modelu paměti přistoupit. Také je zde z důvodu rozložení aplikace umístěn čítač cyklů procesoru, přestože nemá s pamětí ani touto komponentou nic společného. Při kliknutí na nějakou kartu se zavolá metoda `ChangeCurrentTab` jenž vloží název zvoleného modelu paměti do proměnné `currentTab`.

V druhé části se nachází pro Vue.js specifický element `component`, který představuje obecnou komponentu. Pomocí direktivy `:is`, do níž je přiřazena právě proměnná `currentTab`, je podle jména specifikovaná konkrétní komponenta, tedy v tomto případě konkrétní model paměti, která se má na tomto místě vykreslit.

Při spuštění programu se karty v liště karet stávají neaktivní a nelze na ně kliknout, a tudíž není možné během vykonávání programu změnit použitý typ paměti. Pro zvýraznění aktuálně vybrané záložky je použito asociativní pole `activeTab`, v němž je ke každému typu paměti přiřazena hodnota značící, zda je tato záložka aktivní. Pokud je záložka aktuálně vybraná tak je na tuto komponentu aplikován dodatečný stylový předpis označující, že je právě aktivní.

Okno nastavení simulátoru

Komponenta nastavení simulátoru (`TheSettings.vue`) umožňuje nastavení některých vlastností implementovaného simulátoru. Tlačítko pro otevření nastavení simulátoru je umístěno v záhlaví aplikace. V nastavení je možné specifikovat dobu trvání animací prováděných při běhu simulace v milisekundách, ceny přístupu do vyrovnávací a hlavní paměti a také velikost vyrovnávací a hlavní paměti. Nastavení je implementováno jako modální okno a bylo převzato z oficiálních příkladů komponent Vue.js [11].

Nastavení využívá vlastní komponentu pro číselný vstup (`NumberInput.vue`), která byla převzata z [23] a jejíž šablona je zobrazena na obrázku 5.7. Tato komponenta využívá direktivu `v-model` usnadňující napojení hodnoty vstupu a události pro změnu hodnoty. Takto není nutné ručně ošetřovat událost pro změnu hodnoty vstupu, framework Vue.js to zajistí automaticky.

```
<div class="number-input">
  <span>{{ label }}</span>
  <input :value="modelValue"
    @input="$emit('update:modelValue',
      Number($event.target.value))"
  />
</div>
```

Obrázek 5.7: HTML šablona číselného vstupu je tvořena elementem `input`. Tento element má definovanou vlastnost `:value`, která je navázaná na vlastnost této komponenty, a událost `@input` vyvolanou při změně hodnoty v elementu. Při ní se vyvolá událost `update:modelValue` rodičovské komponenty v níž je předána nová hodnota. Díky této struktuře stačí komponentě číselný vstup definovat pouze direktivu `v-model:"var"`. Aktualizaci hodnoty této proměnné při vstupu zajistí automaticky systém Vue.js.

Globální hodnoty

Simulátor využívá několik globálních proměnných, které jsou použity napříč aplikací. Tyto proměnné jsou definovány v souboru `main.js` ve struktuře `app.config.globalProperties`, kterou Vue.js používá jako uložiště globálních dat. Jsou zde definovány doby trvání animací při simulaci práce vyrovnávací paměti, ceny přístupů do paměti a velikost vyrovnávací

a hlavní paměti. Jsou zde vlastně umístěny hodnoty, které lze nastavit v nastavení simulátoru. Také je zde uložena hodnota čítače cyklů instrukcí. Všechny tyto číselné hodnoty jsou zabaleny v objektu, aby je bylo možné z jakéhokoliv místa aplikace aktualizovat. Pokud by byly uložené přímo jako číselné hodnoty (hodnotový datový typ), tak by změna v jedné komponentě aplikace nebyla zohledněna v ostatních komponentách.

5.5 Testování procesoru a překladače

Pro otestování správné funkcionality překladače assembleru a procesoru byla vytvořena sada jednotkových testů pro překladač a procesor. Pro testování byl použit javascriptový testovací framework **Jest**, který poskytuje prostředky pro jednotkové a celkové testování javascriptového kódu. Provedení testů bylo zařazeno do github workflows, tak aby bylo při průběžné integraci a průběžném nasazení automaticky spuštěno. Celkem bylo vytvořeno 21 testů pro překladač assembleru, z nichž dvanáct testuje správný překlad všech druhů a kombinací instrukcí a komentářů a devět testuje chybové stavy, které mohou při překladu nastat. Pro procesor bylo napsáno dohromady 51 testů. Pět těchto testů kontroluje možné chybové stavy a zbylých 46 testuje správnou funkci všech instrukcí procesoru. Testy procesoru předpokládají, že testovaný kód byl přeložen překladačem a je syntakticky správný.

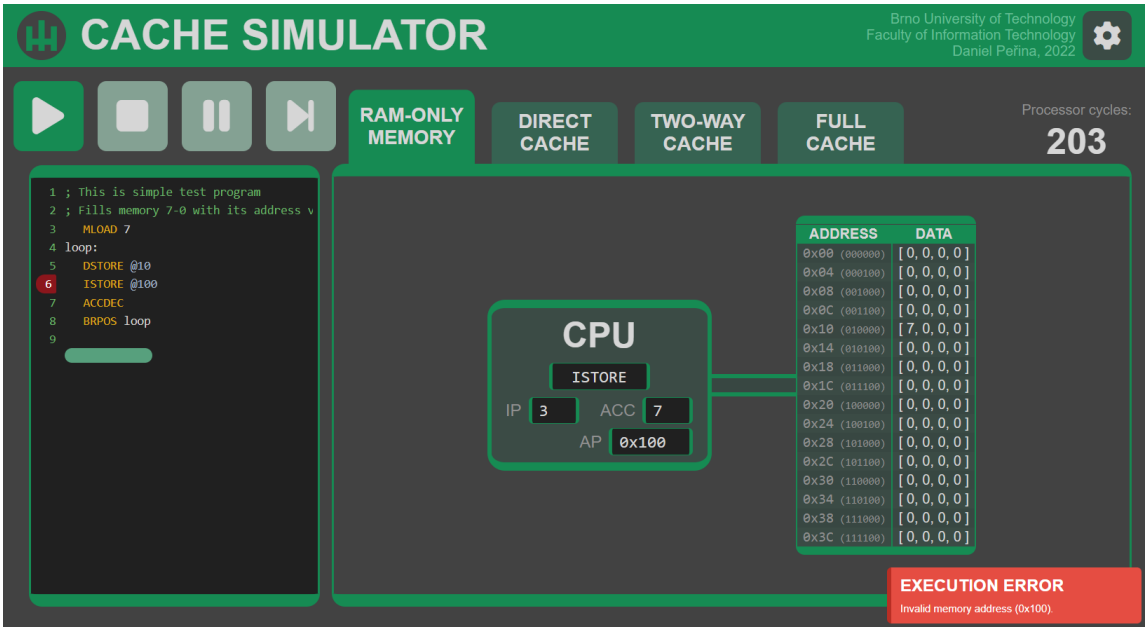
Testy pro překladač assembleru lze rozdělit do několika skupin podle typu instrukce. Jsou zde umístěny testy pro instrukce bez parametru, instrukce s adresovým parametrem, instrukce parametrem návěštím a instrukce s hodnotovým parametrem. Tyto testy jsou provedeny vždy jako test na jednu instrukci daného typu a poté pro několik instrukcí tohoto typu. Dále je proveden test pro několik instrukcí různých typů zapsané jak na jednom řádku, tak na více řádcích. Je zde i test pro kód s komentáři nebo prázdný kód bez jediné instrukce. Také jsou otestovány chybové stavy, jako je neznámá instrukce, chybějící parametr instrukce nebo chybný formát parametru.

Procesor je otestován po jednotlivých instrukcích za použití modulu paměti bez vyrovnávací paměti. U testů instrukcí se testuje jak návratový objekt instrukce určující další chování procesoru, tak i případně hodnota registru akumulátor nebo obsah dat na dané adrese v paměti. Každá instrukce je ošetřena jedním testem, vyjma podmíněných instrukcí, pro které jsou provedeny testy na všechny možné výsledné stavy. Také jsou zde otestovány možné chybové stavy, jako je provádění neznámé instrukce nebo čtení nebo zápis z neplatné adresy v paměti.

Kapitola 6

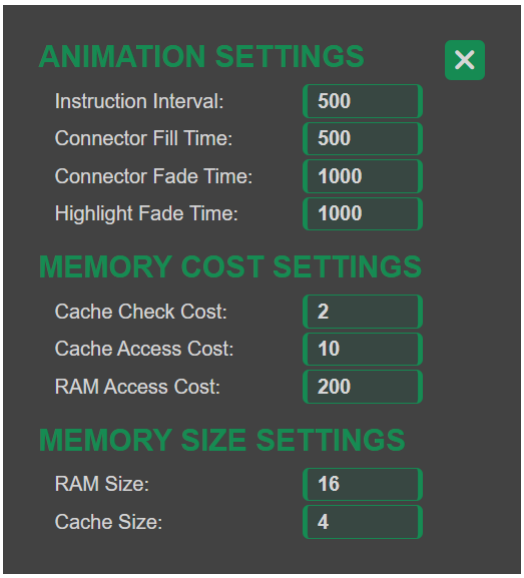
Ukázkové programy pro simulátor činnosti vyrovnávacích pamětí

V této kapitole budou popsány navrhnuté ukázkové programy pro vytvořený simulátor činnosti vyrovnávacích pamětí procesoru. Programy jsou napsané v assembleru popsaném v sekci 4.4. Slouží pro demonstraci různých vlastností (a to zejména těch negativních) implementovaných typů vyrovnávacích pamětí a také problémy, které se mohou vyskytnout, pokud nejsou algoritmy navrhnuty s ohledem na použitou organizaci vyrovnávací paměti počítače. Tyto paměťové problémy jsou zde shrnuty a případně jsou zde zdůrazněny další zajímavé vlastnosti implementovaných programů. Zdrojové kódy těchto programů jsou uloženy v příloze A.



Obrázek 6.1: Výsledné grafické uživatelské rozhraní vytvořeného simulátoru. V editoru kódu je zadán výchozí assembler program, který je upravený na řádku 6. Aplikace je ve stavu, kdy při vykonávání zadaného programu nastala chyba, která je zobrazena jako chybové hlášení na pravém spodním rohu stránky. Instrukce, na níž se program zastavil, je v editoru kódu označena.

Na obrázku 6.1 je zobrazeno grafické uživatelské rozhraní výsledné aplikace, jež je vytvořeno podle návrhu popsaného v sekci 4.1. Simulátor obsahuje editor kódu pro simulovaný assembler program, v němž je na obrázku zvýrazněna následující instrukce. Tlačítka nad editorem kódu umožňují spustit a ovládat zadaný program. Pomocí karet v modelové části aplikace je možné přepínat mezi jednotlivými typy paměti. V případě, že nastane chyba při překladu nebo vykonávání programu, tak se v pravém spodním rohu obrazovky zobrazí chybové hlášení, tak je vidět na obrázku. V pravém horním rohu je tlačítko pro zobrazení nastavení. Obsah nastavení je zobrazen na obrázku 6.2.



Obrázek 6.2: Modální okno obsahující nastavení vytvořeného simulátoru. Je zde možné nastavit dobu trvání následujících animací v milisekundách: prodlení mezi jednotlivými instrukcemi, posun v konektoru, vyblednutí konektoru a vyblednutí zvýraznění použitého bloku v paměti. Může se zde upravit cena kontroly vyrovnávací paměti, přístupu do vyrovnávací paměti a přístupu do hlavní paměti v počtu cyklů procesoru. Také je zde možné specifikovat velikosti hlavní a vyrovnávací paměti v počtu paměťových bloků. Nastavení zobrazené na obrázku je výchozí nastavení aplikace.

6.1 Výchozí ukázkový program

Ve vytvořeném simulátoru je zabudovaný jednoduchý výchozí program sloužící jako příklad způsobu zápisu assembleru a také pro rychlé předvedení činnosti simulátoru. Zdrojový kód tohoto programu je zobrazen v příloze A. Program pomocí jednoho cyklu sestupně naplní hlavní paměť od adresy `0x7` po adresu `0x0` hodnotami rovnající se adrese dané paměťové buňky.

Tento program slouží pouze jako ukázkový program pro demonstraci zápisu assembleru simulátoru a nereprezentuje žádný významný algoritmus nebo problém, přesto se i zde projevuje základní nedostatek vyrovnávací paměti s přímým mapováním. Data sloužící současně jako ukazatel do paměti i jako hodnota, jsou ve výchozím nastavení simulátoru mapována na první paměťový blok vyrovnávací paměti. Na stejný blok je mapována i druhá polovina dat zapisovaných do paměti (adresy 0x0 a 0x3). Při zápisu těchto dat tak nastávají opakované konfliktů mezi paměťovým blokem obsahující hodnotu ukazatele do paměti a blokem

obsahující data, které se navzájem vytlačují. Dochází tak k mnoha zbytečným přístupům do hlavní paměti, což zpomaluje celý program. Tento problém je úplně vyřešen zvýšením stupně asociativity vyrovnávací paměti neboli použitím modelu vyrovnávací paměti s dvoucestným mapováním nebo plně asociativním mapováním. Při porovnání trvání algoritmu počtem cyklů procesoru při použití paměti bez vyrovnávací paměti a s vyrovnávací pamětí je na první pohled zřejmé jak významný vliv mají vyrovnávací paměti. I na takto jednoduchém programu a procesoru je program s použitím vyrovnávací paměti vykonán až čtyřikrát rychleji než když vyrovnávací paměť chybí.

6.2 Zpracování prvků matice 4x4

Zpracování všech prvků matice je typický algoritmus, na němž se demonstruje význam programování s ohledem na vnitřní organizaci paměti. Obsah matice je v paměti uložen sekvenčně a to buď po řádcích, nebo po sloupcích. Stejně tak může být algoritmus průchodu přes všechny prvky matice implementován buď po řádcích, nebo po sloupcích. Problém nastává, když způsob uložení matice v paměti a metoda průchodu obsahu matice nejsou stejné, například když je matice v paměti uložena po řádcích, ale zpracování prvků matice je implementováno po sloupcích [1]. V příloze A jsou zobrazeny zdrojové kódy programů do simulátoru implementující zpracování prvků matice, která je v paměti uložena po řádcích, jak metodou po řádcích tak po sloupcích. Dimenze matice byly zvoleny jako 4×4 z důvodu lepší vizualizace obsahu matice v hlavní paměti, poněvadž jeden paměťový blok v simulátoru obsahuje dohromady čtyři hodnoty.

Průchod matice po sloupcích	
Použitý typ vyrovnávací paměti	**Cena programu**
Bez vyrovnávací paměti	25 676 cyklů
Paměť s přímým mapováním	4 bloky: 5 800 cyklů 2 bloky: 11 400 cyklů
Paměť s dvoucestným mapováním	6 297 cyklů
Paměť s plně asociativním mapováním	6 300 cyklů

Průchod matice po řádcích	
Použitý typ vyrovnávací paměti	**Cena programu**
Bez vyrovnávací paměti	22 005 cyklů
Paměť s přímým mapováním	4 bloky: 5 513 cyklů 2 bloky: 8 713 cyklů
Paměť s dvoucestným mapováním	2 981 cyklů
Paměť s plně asociativním mapováním	3 422 cyklů

Tabulka 6.1: Tabulky cen programu *Průchod matice po sloupcích* a *Průchod matice po řádcích* při použití různých typů vyrovnávací paměti reprezentované v počtu cyklů procesoru nutných k vykonání programu. Matice je v paměti uložena po řádcích. Na první pohled lze vidět, že průchod matice po řádcích je výrazně efektivnější a to zejména při použití vyrovnávací paměti s dvoucestným mapováním a plně asociativním mapováním.

V tabulce 6.1 jsou zapsány konečné doby trvání obou těchto programů v cyklech procesoru na všech typech paměti po tom, co byly spuštěny ve vytvořeném simulátoru ve výchozím nastavení. Už na první pohled lze vidět podstatný rozdíl v době trvání programu. Vytvořený algoritmus využívá dohromady pět paměťových bloků v hlavní paměti:

čtyři obsahují data matice, kde v každém bloku je uložen celý řádek, a jeden blok slouží pro uchování řídících hodnot. Při zpracování prvků matice po sloupcích se se všemi pěti bloky pracuje průběžně, proto je nutné je všechny neustále udržovat ve vyrovnávací paměti. A protože má vyrovnávací paměť ve výchozím nastavení kapacitu pouze čtyř paměťových bloků, tak opakovaně dochází ke konfliktu řídícího bloku a jednoho datového bloku. Nejvíce to je zřetelné při použití vyrovnávacích pamětí s dvoucestným mapováním a plně asociativním mapováním, kde je použit náhodný výběr oběti a tak se často vybírá nevhodná oběť. Naopak při zpracování prvků matice po řádcích se vždy aktivně pracuje pouze se dvěma paměťovými bloky: blokem s řídícími hodnotami a jedním datovým blokem, které obsahuje řádek matice, jenž se právě zpracovává. Díky tomu při použití vyrovnávacích pamětí se dvoucestným mapováním a plně asociativním mapováním nastane pouze jeden konflikt, takže je menší šance, že se vybere nevhodná oběť. Rozdíl ceny programu při použití paměti bez vyrovnávací paměti je dán menší složitostí algoritmu zpracování prvků matice po řádcích, poněvadž se v něm paměť prochází sekvenčně a nemusí se v ní na rozdíl od algoritmu zpracování prvků po sloupcích skákat.

Z výsledků simulace lze vypozorovat, že při použití vyrovnávací paměti s přímým mapováním o kapacitě čtyř paměťových bloků je malý rozdíl mezi algoritmy zpracování prvků matice po sloupcích nebo po řádcích. To je kvůli celkové velikosti matice – čtyř paměťových bloků – což je rovno kapacitě vyrovnávací paměti. Datové konflikty tak nastávají pouze při přístupu k paměťovému bloku s řídícími hodnotami a jedním blokem s daty matice, které se mapují na stejnou adresu ve vyrovnávací paměti. U obou algoritmů tak nastane stejný počet konfliktů, rozdíl je pouze v časovém rozprostření těchto konfliktů: při zpracování prvků po sloupcích jsou rovnoměrně rozprostřené po celém průběhu algoritmu, naopak při zpracování prvků po řádcích nastanou všechny konflikty při zpracování řádku matice, který je mapován na stejnou adresu vyrovnávací paměti jako řídící data. Proto byl algoritmus simulován také na vyrovnávací paměti s přímým mapováním o kapacitě dvou paměťových bloků, kde už je možné pozorovat rozdíl v době trvání obou algoritmů, poněvadž ve vyrovnávací paměti mohou být naráz uloženy pouze dva paměťové bloky s daty matice.

Na reálných procesorech by tyto dva algoritmy vykazovaly ještě větší rozdíly v době jejich provádění. Implementovaný procesor obsahuje pouze jeden datový registr a veškeré aritmetické instrukce a instrukce přístupu do paměti pracují právě s tímto registrem. Proto je nutné řídící hodnoty také uchovávat v paměti. Paměťový blok, v němž jsou tyto hodnoty uložené, je velmi často používán, což vede k dalším paměťovým konfliktům. V procesoru, který by měl více registrů, by tyto hodnoty mohli být uloženy v těchto registrech a výkon algoritmu zpracování prvků matice by tak závisel pouze na organizaci paměti.

6.3 Průchod lineárním spojovým seznamem

Průchod lineárním spojovým seznamem je další z algoritmů, jehož výkon může být značně ovlivněn organizací vyrovnávací paměti. Lineární spojový seznam je datová struktura skládající se z prvků stejného datového typu seřazených za sebou. Každý prvek seznamu obsahuje své data a dále ukazatel (odkaz) na následující, případně i předcházející, prvek v seznamu. Tyto prvky nemusí být v hlavní paměti uloženy za sebou, naopak mohou být uloženy na různých místech v paměti a i na přeskáčku. Takovýto seznam slouží jako dynamická obdoba klasického pole, která má flexibilní počet prvků a do níž je možné vkládat prvky na libovolnou pozici v seznamu bez nutnosti realokace paměti, jak je nutné v normálním poli. Zdrojový kód algoritmu průchodu lineárním spojovým seznamem je zapsán v příloze A.

Právě protože prvky lineárního spojového seznamu na sebe odkazují a na rozdíl od prvků obyčejného pole nemusí být v paměti uložené přímo za sebou, mohou nastat při práci s tímto seznamem výkonnostní problémy způsobené právě vyrovnávací pamětí. Jednotlivé prvky mohou být uloženy na velmi rozdílných adresách, které mohou navíc být mapovány na stejnou adresu ve vyrovnávací paměti. Ve vyšších programovacích jazycích je alokace paměti zajištěna operačním systémem a programátor tak nemůže specifikovat místo uložení daného prvku. Větší spojový seznam je kvůli tomu rozprostřen napříč výrazně větším počtem paměťových bloků, než by bylo pole o stejném počtu prvků. To má samozřejmě za následek větší množství paměťových konfliktů. Další problém může také způsobit i větší velikost jednoho prvku seznamu, který kromě samotných dat obsahuje i odkazy na vedlejší prvky. Do jednoho paměťového bloku se tak vejde méně samotných dat, kvůli čemuž seznam zabírá více paměťových bloků, což má zase za následek větší množství paměťových konfliktů [6].

Proto je při práci s lineárními datovými strukturami vždy nutné pečlivě zhodnotit, kterou strukturu využít. Lineární spojové seznamy sice nabízí množství výhodných a pohodlných vlastností, ale jsou paměťově náročnější než obyčejné pole. Navíc ve vyšších programovacích jazycích je správa paměti zajištěna operačním systémem a tak nelze specifikovat místo uložení jednotlivých prvků seznamu. To může u velkých seznamů vytvářet výkonnostní problémy, které závisí na aktuálním využití paměti a nelze je předvídat.

6.4 Opakované cykly

Dalším algoritmem nevyužívající pozitivních vlastností vyrovnávacích pamětí je zpracování dat pole ve více cyklech [4]. Pokud je veškeré zpracování dat provedeno v rámci jednoho cyklu, je dodržen princip časové lokality, a tak stačí paměťový blok s potřebnými daty načíst z hlavní paměti do vyrovnávací pouze jednou. Naproti tomu pokud jsou data zpracovávána ve více cyklech, tak může být potřeba daný datový blok z hlavní paměti načítat vícekrát. Není tak dodržena časová lokalita. Algoritmus je potom pomalejší nejen kvůli režii dalšího cyklu, ale především kvůli podstatně většímu množství přístupu do hlavní paměti. Tento problém je nejvýraznější pokud je zpracovávané pole větší než je kapacita vyrovnávací paměti nebo v případě že je mezi cykly, zajišťující zpracování, jiný kód, který využívá jinou část paměti. Při zpracování malého množství dat více cykly, které jsou navíc hned za sebou, je tento problém zanedbatelný a algoritmus je potom pomalejší pouze kvůli režii dodatečných cyklů. Zdrojové kódy do simulátoru implementující zpracování dat v jednom a dvou cyklech jsou vyobrazeny v příloze A. Kódy jsou vytvořeny tak, aby bylo možné jednoduše změnit velikost zpracovávaného pole a tak otestovat rozdíl mezi zpracováním malého a velkého množství dat.

Kvůli těmto problémům je nutné mít na paměti, že zpracování velkých dat více průchody může způsobit velké zpomalení programu. Při použití vyrovnávací paměti s přímým mapováním není rozdíl mezi zpracováním jedním a více cykly příliš vysoký, ale vyrovnávacích pamětí s větším stupněm asociativity může být doba zpracování dat pomocí více cyklů až několika násobně vyšší. Je proto žádoucí, aby algoritmus zpracovávající data pole provedl veškeré operace během jednoho průchodu, nebo aby bylo v případně nutnosti více průchodu použito asynchronní zpracování dat a hlavní program tak nemusel být zdržován čekáním na dokončení zpracování.

Kapitola 7

Závěr

Cílem této práce bylo navrhnout a implementovat simulátor činnosti vyrovnávacích pamětí procesoru jako webovou aplikaci. Tohoto cíle bylo dosaženo a při základní znalosti funkce vyrovnávací paměti je v aplikaci možné přehledně simulovat chování různých typů vyrovnávacích pamětí a demonstrovat jejich nedostatky v různých algoritmech. Pomocí čítače cyklů procesoru umístěného v aplikaci je zřejmé jak zásadní vliv má vyrovnávací paměť na výkon počítače. Výsledná aplikace je volně dostupná na internetu[1].

V rámci této práce byly shrnuty znalosti ohledně hierarchie paměti v počítačích a dále blíže rozepsán princip činnosti vyrovnávacích paměti a různé způsobu organizace těchto pamětí. Byl proveden průzkum již existujících simulátorů činnosti vyrovnávacích pamětí procesoru a shrnuty jejich silné stránky a nedostatky, které je činí nevhodné pro simulaci vlivu vyrovnávacích pamětí na reálných algoritmech. Byly prostudovány možnosti vizualizace činnosti vyrovnávacích pamětí ve webovém prostředí pomocí takzvaných javascriptových frontend frameworků. Z prostudovaných frameworků byl nakonec pro implementaci vybrán framework *Vue.js*. Následně byl vytvořen návrh grafického uživatelského rozhraní a také logické struktury simulátoru, podle nichž byl simulátor implementován. Na závěr byla vytvořena sada ukázkových programů, na nichž je možné demonstrovat různé vlastnosti a nedostatky vyrovnávacích pamětí.

Aplikace byla implementována s ohledem na možné budoucí rozšíření. Je možné poměrně jednoduše do aplikace přidat další typ vyrovnávací paměti bez nutnosti zásahu do jiných komponent aplikace. Také assembler použitý v simulátoru je možné rozšiřovat o další instrukce pouhým rozšířením modulu překladače a přidáním metody implementující danou instrukci do modulu procesoru. V rámci budoucích rozšíření by bylo možné přidat navíc další konfiguraci vyrovnávací paměti, jako volbu algoritmu výběru oběti u vyrovnávacích pamětí s vyšším stupněm asociativity nebo strategii zápisu do vyrovnávací paměti. Také by bylo možné rozšířit vyrovnávací paměť o další vrstvy tak, aby odpovídala vyrovnávacím pamětím v dnešních počítačích, i když by toto rozšíření vyžadovalo přepracování modulu paměti. V delším časovém horizontu by mohl být simulátor rozšířen o další procesoru tak, aby bylo možné simulovat běh dvou nebo více paralelních programů a jejich vliv na činnost vyrovnávací paměti.

[1] https://mrazekv-students.github.io/bp22_cpu_perina/

Literatura

[1] Agarwal, A. *Computer Organization: Locality and Cache friendly code* [online]. GeeksforGeeks, 2021 [cit. 28.04.2022]. Dostupné z: https://www.geeksforgeeks.org/computer-organization-locality-and-cache-friendly-code/.

[2] *Angular: What is Angular?* [online]. Google, 2022 [cit. 14.04.2022]. Dostupné z: https://angular.io/guide/what-is-angular.

[3] Bikker, J. *INFOMOV: Optimization & Vectorization: #2 – Caching* [online]. Utrecht University, 2019 [cit. 22.04.2022]. Dostupné z: http://www.cs.uu.nl/docs/vakken/mov/2019/files/OptmzdSummary%20-%20lecture2%20-%20caching.pdf.

[4] Biswas, S. *Programming for Performance: CS698L: Write CacheFriendly Code* [online]. CSE, IIT Kanpur, 2019 [cit. 29.04.2022]. Dostupné z: https://www.cse.iitk.ac.in/users/swarnendu/courses/autumn2019-cs698l/.

[5] Carvalho, C. *The Gap between Processor and Memory Speeds* [online]. Braga: Universidade do Minho - Departamento de Informática, 2002 [cit. 20.03.2022]. Dostupné z: http://gec.di.uminho.pt/Discip/MInf/ac0102/ICCA02.htm.

[6] Claesen, M. *What is a cache-friendly code?* [online]. Stack Overflow, 2021 [cit. 29.04.2022]. Dostupné z: https://stackoverflow.com/a/16699282.

[7] Glabazňa, T. *Úvod do Vue.js a první aplikace* [online]. ITnetwork.cz, 2021 [cit. 14.04.2022]. Dostupné z: https://www.itnetwork.cz/javascript/vuejs/uvod-do-vuejs-a-prvni-aplikace.

[8] Hsia, J. *351 Cache Simulator* [online]. Paul G. Allen School of Computer Science & Engineering, University of Washington, 2018 [cit. 22.04.2022]. Dostupné z: https://courses.cs.washington.edu/courses/cse351/cachesim/.

[9] Kamani, S. *Using Enums (Enumerations) In Javascript* [online]. 2021 [cit. 25.04.2022]. Dostupné z: https://www.sohamkamani.com/javascript/enums/.

[10] Matthews, S. J., Newhall, T. a Webb, K. C. Storage and the Memory Hierarchy. In: Dive into Systems. *Dive into Systems: A Free, Online Textbook for Introducing Computer Systems* [online]. 1.0 RC. San Francisco, CA, USA: No Starch Press, 2022, kap. 11 [cit. 20.03.2022]. ISBN 9781718501362. Dostupné z: https://diveintosystems.org/book/index.html.

[11] *Examples: Modal* [online]. Vue.js, 2022 [cit. 25.04.2022]. Dostupné z: https://vuejs.org/examples/#modal.

[12] Máca, J. *Úvod do React* [online]. ITnetwork.cz, 2019 [cit. 14.04.2022]. Dostupné z: https://www.itnetwork.cz/javascript/react/zaklady/uvod-do-react/.

[13] Máca, J. *Úvod do Angular* [online]. ITnetwork.cz, 2020 [cit. 14.04.2022]. Dostupné z: https://www.itnetwork.cz/javascript/angular/zaklady/uvod-do-angular-frameworku.

[14] Paramita, A. *ParaCache Simulator* [online]. Nanyang Technological University, 2016 [cit. 22.04.2022]. Dostupné z: https://personal.ntu.edu.sg/smitha/ParaCache/Paracache/start.html.

[15] *React* [online]. Meta Platforms, 2022 [cit. 14.04.2022]. Dostupné z: https://reactjs.org/.

[16] Savaton, G. *About Virgule and its simulator emulsiV* [online]. École supérieure d'électronique de l'Ouest, 2020 [cit. 12.04.2022]. Dostupné z: https://guillaume-savaton-eseo.github.io/emulsiV/doc/.

[17] Savaton, G. *EmulsiV* [online]. École supérieure d'électronique de l'Ouest, 2020 [cit. 22.04.2022]. Dostupné z: https://guillaume-savaton-eseo.github.io/emulsiV/.

[18] Sekanina, L. *Návrh počítačových systémů: Paměti.* Brno: Vysoké učení technické v Brně, Fakulta informačních technologií, 2021.

[19] Sekanina, L. *Návrh počítačových systémů: Paměťová hierarchie.* Brno: Vysoké učení technické v Brně, Fakulta informačních technologií, 2021.

[20] Susek, J. *VueTut: Easily build beautiful tutorials with Vue.* [online]. 2020 [cit. 25.04.2022]. Dostupné z: https://github.com/evwt/vue-tut.

[21] Vašíček, Z. a Mrázek, V. *Návrh počítačových systémů: Cvičení 4: Implementace procesoru ve VHDL.* Brno: Vysoké učení technické v Brně, Fakulta informačních technologií, 2021.

[22] *Vue.js: Vue 3 Introduction* [online]. Evan You, 2022 [cit. 10.04.2022]. Dostupné z: https://vuejs.org/guide/introduction.html.

[23] Wilby, C. *V-model and child components* [online]. Stack Overflow, 2019 [cit. 25.04.2022]. Dostupné z: https://stackoverflow.com/a/55868431.

Příloha A

Ukázkové programy pro simulátor činnosti vyrovnávacích pamětí

V této příloze jsou umístěny zdrojové kódy ukázkových programů pro simulátor činnosti vyrovnávací pamětí procesoru popsané v kapitole 6. Programy jsou napsané v implementovaném assembleru, jehož návrh je shrnut v sekci 4.4.

Výchozí ukázkový program

```
; Jednoduchý testovací program
; Naplní paměť 0x7 až 0x0 hodnotou její
; adresy
  MLOAD 7
loop:
  DSTORE @10
  ISTORE @10
  ACCDEC
  BRPOS loop
HALT
```

Obrázek A.1: Zdrojový kód výchozího programu simulátoru. Tento program demonstruje základní způsob zápisu assembler kódu do simulátoru a také vizualizaci práce vyrovnávací paměti. V rámci něj se pomocí cyklu naplní první dva bloky hlavní paměti ($0x0 - 0x7$) hodnotami, které odpovídají adresa dané paměťové buňky.

Zpracování matice 4x4

```
; Průchod maticí 4x4
; Matice je v paměti uložena po řádcích
; Matice je procházena po sloupcích
; Při průchodu se pole matice zpracují
; Zpracované pole matici mají hodnotu 1

  ; Aktuální adresa
  MLOAD 0
  DSTORE @20
  ; Čítač sloupců
  MLOAD 4
  DSTORE @21
  ; Čítač řádků
  MLOAD 4
  DSTORE @22
  ; Základní adresa sloupce
  MLOAD -1
  DSTORE @23
  FLUSH HALT
col:
  ; Načte se základní adresa sloupce
  DLOAD @23
  ACCINC
  DSTORE @23
  ; Skok na end pokud čítač sloupců == 0
  DLOAD @21
  BRZERO end
  ; Dekrementuje se čítač sloupců
  ACCDEC
  DSTORE @21
  ; Nastaví se čítač řádků
  MLOAD 4
  DSTORE @22
  ; Nastaví aktuální adresu
  DLOAD @23
  DSTORE @20
row:
  ; Průchod
  MLOAD 1
  ISTORE @20
  ; Upraví se adresa sloupce na další
řádek
  DLOAD @20
  ACCINC ACCINC ACCINC ACCINC
  DSTORE @20
  ; Dekrementuje se čítač řádku
  DLOAD @22
  ACCDEC
  DSTORE @22
  ; Skok na col pokud čítač řádků == 0
  BRZERO col
  ; Skok na row jinak
  BRANCH row
end:
  HALT
```

Obrázek A.2: Zdrojový kód programu zpracování matice po sloupcích. Matice má rozměry 4×4 a je v paměti uložena po řádcích. Algoritmus lze rozdělit na tři celky. Nejdříve se připraví režijní data jako je čítač sloupců, čítač řádků, adresa právě zpracovávané buňky a základní adresa právě zpracovávaného sloupce. Potom následují dva cykly: cyklus průchodu sloupci matice a v něm vnořený cyklus průchodu řádky daného sloupce. Při každé iteraci se čítač sloupce nebo řádku sníží o jedna. Jakmile je čítač roven nule cyklus se ukončí. Program končí, jakmile je čítač sloupců roven nule.

```
; Průchod maticí 4x4
; Matice je v paměti uložena po řádcích
; Matice je procházena po řádcích
; Při průchodu se pole matice zpracují
; Zpracované pole matici mají hodnotu 1

  ; Aktuální adresa
  MLOAD 0
  DSTORE @20
  ; Čítač řádků
  MLOAD 4
  DSTORE @21
  ; Čítač sloupců
  MLOAD 4
  DSTORE @22
  FLUSH HALT
row:
  ; Skok na end pokud čítač řádků == 0
  DLOAD @21
  BRZERO end
  ; Dekrementuje se čítač řádků
  ACCDEC
  DSTORE @21
  ; Nastaví se čítač sloupců
  MLOAD 4
  DSTORE @22
col:
  ; Průchod
  MLOAD 1
  ISTORE @20
  ; Upraví se adresa na další buňku
  DLOAD @20
  ACCINC
  DSTORE @20
  ; Dekrementuje se čítač sloupců
  DLOAD @22
  ACCDEC
  DSTORE @22
  ; Skok na col pokud čítač sloupců == 0
  BRZERO row
  ; Skok na row jinak
  BRANCH col
end:
  HALT
```

Obrázek A.3: Zdrojový kód programu zpracování matice po řádcích. Matice má rozměry 4×4 a je v paměti uložena po řádcích. Algoritmus lze rozdělit na tři celky. Nejdříve se připraví režijní data čítač řádků, čítač sloupců a adresa právě zpracovávané buňky. Potom následují dva cykly: cyklus průchodu řádky matice a v něm vnořený cyklus pro průchod sloupců daného řádku. Při každé iteraci se čítač řádku nebo sloupce sníží o jedna. Jakmile je čítač roven nule, cyklus se ukončí. Program končí, jakmile je čítač řádků roven nule.

Průchod lineárním spojovým seznamem

```
; Průchod spojitým seznamem
; Prvek seznamu se skládá ze dvojice:
; hodnota a odkaz na další prvek

  ; Vytvořit seznam
  MLOAD 1 DSTORE @04 MLOAD 10 DSTORE @05
  MLOAD 1 DSTORE @0A MLOAD 16 DSTORE @0B
  MLOAD 1 DSTORE @10 MLOAD 22 DSTORE @11
  MLOAD 1 DSTORE @16 MLOAD 28 DSTORE @17
  MLOAD 1 DSTORE @1C MLOAD 34 DSTORE @1D
  MLOAD 1 DSTORE @22 MLOAD 40 DSTORE @23
  MLOAD 1 DSTORE @28 MLOAD 44 DSTORE @29
  MLOAD 1 DSTORE @2C MLOAD -1 DSTORE @2D

  ; Průchod seznamem
  FLUSH HALT
  ; Aktuální prvek
  MLOAD 4
  DSTORE @0
loop:
  ; Vynulovat hodnotu prvku
  MLOAD 0
  ISTORE @0
  ; Načíst odkaz na další prvek
  DLOAD @0
  ACCINC
  DSTORE @0
  ILOAD @0
  DSTORE @0
  ; Pokud odkaz je záporný tak konec
  BRNEG end
  ; Jinak pokračovat
  BRANCH loop
end:
  HALT
```

Obrázek A.4: Zdrojový kód programu pro průchod lineárním spojovým seznamem. Spojový seznam se skládá z prvků obsahující hodnotu v jedné datové buňce a adresu následujícího prvku ve vedlejší datové buňce. V první části programu se vytvoří seznam o osmi prvcích. Prvky jsou pro jednoduchost od sebe vzdáleny vždy o šest paměťových buněk, nicméně jejich pozice může být libovolná a přeházená. Jako ukazatel do paměti se používá hodnota uložená na adrese `0x0`. V cyklu se při průchodu seznamem vynuluje hodnota prvku a poté se načte adresa následujícího prvku. Záporná hodnota adresy následujícího prvku značí konec seznamu a při jejím načtení program končí.

Opakované cykly

```
; Zpracování pole pomocí jednoho cyklu
; Hodnota prvku se nejdřív nastaví na 1
; poté se inkrementuje

  ; Čítač pro pohyb v poli
  MLOAD 31
  DSTORE @3F
  FLUSH HALT
loop:
  ; Nastavit hodnotu na 1 a uložit
  MLOAD 1
  ISTORE @3F
  ; Hodnotu inkrementovat a uložit
  ILOAD @3F
  ACCINC
  ISTORE @3F
  ; Pokud čítač == 0 tak konec
  DLOAD @3F
  BRZERO end
  ; Jinak pokračovat
  ACCDEC
  DSTORE @3F
  BRANCH loop
end:
  HALT
```

```
; Zpracování pole pomocí dvou cyklů
; Hodnota prvku se nejdřív nastaví na 1
; poté se inkrementuje

  ; Čítač pro pohyb v poli
  MLOAD 31
  DSTORE @3F
  FLUSH HALT
loop_one:
  ; Nastavit hodnota na 1
  MLOAD 1
  ISTORE @3F
  ; Pokud čítač == 0 tak konec cyklu
  DLOAD @3F
  BRZERO end_one
  ; Jinak pokračovat
  ACCDEC
  DSTORE @3F
  BRANCH loop_one
end_one:
  ; Čítač pro pohyb v poli (znovu)
  MLOAD 31
  DSTORE @3F
loop_inc:
  ; Hodnotu inkrementovat
  ILOAD @3F
  ACCINC
  ISTORE @3F
  ; Pokud čítač == 0 tak konec
  DLOAD @3F
  BRZERO end_inc
  ; Jinak pokračovat
  ACCDEC
  DSTORE @3F
  BRANCH loop_inc
end_inc:
  HALT
```

Obrázek A.5: Zdrojový kódy programů pro zpracování hodnot pole pomocí jednoho nebo dvou cyklů. Hodnoty pole jsou zpracovány dvěma operacemi: nastavení hodnoty na jedna a následné inkrementování této hodnoty. Program využívá jednu pomocnou hodnotu uloženou na adrese `0x3F`, pomocí níž lze nastavit celkový počet prvků pole, které budou zpracovány. Lze tak testovat rozdíl mezi velkým a malým polem. V rámci cyklu zpracování je na danou hodnotu aplikována daná operace a následně je hodnota uložena do paměti. V algoritmu s jedním cyklem je tak mezi nastavením hodnoty na jedna a inkrementací této hodnoty uložení do paměti, přestože by nebylo nutné. To je z důvodu reprezentace netriviálního zpracování, které by nebylo možné provést pouze v rámci registrů procesoru.

Příloha B

Obsah přiloženého paměťového média

- `cache_sim_src` – Adresář obsahující zdrojové kódy vytvořeného simulátoru vyrovnávacích pamětí procesoru a zdrojové kódy použitých balíčků.
- `cache_sim_dist` – Adresář obsahující minimalizovaný simulátor připravený k nasazení.
- `thesis_src` – Adresář obsahující zdrojový kód technické zprávy.
- `BP_2022_Perina.pdf` – Technická zpráva.