VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

GRAFICKÝ SIMULÁTOR SUPERSKALÁRNÍCH PROCESORŮ S WEBOVÝM ROZHRANÍM

WEB BASED SIMULATOR OF SUPERSCALAR PROCESSORS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. MICHAL MAJER**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. JIŘÍ JAROŠ, Ph.D.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

155079

Ústav:	Ústav počítačových systémů (UPSY)
Student:	**Majer Michal, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Kybernetická bezpečnost
Název:	**Grafický simulátor superskalárních procesorů s webovým rozhraním**
Kategorie:	Počítačová architektura
Akademický rok:	2023/24

Zadání:

1. Seznamte se s architekturou současných superskalárních procesorů a tvorbou webových aplikací.
2. Prostudujte současné grafické simulátory těchto procesorů, zaměřte se především na RISC-V simulátor Jana Vávry a Jakuba Horkého.
3. Navrhněte postup pro rozšíření tohoto simulátoru o webové rozhraní a rozhraní pro příkazovou řádku.
4. Navrhněte postup pro zvýšení kvality simulace a vylepšení názornosti prezentace výpočtu procesorem.
5. Navržené řešení implementujte.
6. Navrhněte sadu demonstračních úloh, na kterých názorně předvedete fungování navrženého simulátoru.
7. Vyhodnoťte uživatelskou přívětivost a názornost navržené aplikace.
8. Diskutujte přínos vytvořeného simulátoru pro výuku hardwarových kurzů na FIT VUT v Brně.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 4 zadání.

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

Vedoucí práce:	**Jaroš Jiří, doc. Ing., Ph.D.**
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1.11.2023
Termín pro odevzdání:	17.5.2024
Datum schválení:	30.10.2023

Abstrakt

Názorná a interaktivní vizualizace superskalárního procesoru je velmi užitečnou pomůckou při studiu jeho fungování, zejména kvůli jeho složitosti. Hlavní přínos této práce je rozšíření stávajícího simulátoru procesoru instrukční sady RISC-V o nové webové uživatelské rozhraní a zkvalitnění simulace.

Vylepšeny byly téměř všechny moduly simulátoru. Velký přínos má integrace s překladačem jazyka C. Simulátor byl rozšířen o HTTP a CLI rozhraní. Mimo jiné byly také odstraněny chyby v implementaci, vylepšen sběr statistik a doplněna instrukční sada. K implementaci webové aplikace byla využita knihovna React.

Výsledkem práce je funkční a otestovaná aplikace, která je připravena k použití v praxi a bude mít pozitivní přínos pro vzdělávání.

Abstract

A clear and interactive visualization of the superscalar processor is a valuable tool for studying its operation, particularly due to its complexity. The main contribution of this work is the extension of the existing RISC-V instruction set simulator with a new web-based user interface and improvements of the simulation quality.

Nearly all modules of the simulator have been enhanced. Among other things, errors in the implementation have been resolved, statistics collection has been improved, and the instruction set has been expanded. The integration with the C language compiler is of great benefit. The simulator has been expanded to include HTTP and CLI interfaces. The React library has been utilized for implementing the web application.

The result of the work is a functional and tested application, ready for practical use and with a positive impact on education.

Klíčová slova

Simulátor, RISC-V, Superskalární procesor, Webová aplikace, Nasazení aplikací, React, Uživatelské rozhraní, API

Keywords

Simulator, RISC-V, Superscalar processor, Web application, Application deployment, React, User interface, API

Citace

MAJER, Michal. *Grafický simulátor superskalárních procesorů s webovým rozhraním.* Brno, 2024. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Jiří Jaroš, Ph.D.

Grafický simulátor superskalárních procesorů s webovým rozhraním

Prohlášení

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

.....
Michal Majer
13. května 2024

Poděkování

Rád bych vyjádřil poděkování svému vedoucímu práce, docentu Jiřímu Jarošovi, za jeho vedení a odbornost během celého procesu tvorby této práce. Jeho vedení pro mě bylo přínosné a klíčové pro dosažení úspěchu. Také děkuji všem svým blízkým za jejich velikou podporu.

Obsah

Seznam obrázků

Kapitola 1

Úvod

V současném světě hrají superskalární procesory klíčovou roli, tato technologie je základem téměř všech moderních výkonných čipů. Je důležité, aby programátoři chápali jejich fungování a důsledky, zejména pak dopad na výkon programů a algoritmů, které navrhují. Složité mechanismy jako spekulativní vykonávání instrukcí, vyrovnávací paměť či predikce skoků mohou ale být pro studenty obtížně pochopitelné.

Proto je klíčové hledat způsoby, jak usnadnit výuku a poskytnout studentům nástroje, které jim pomohou lépe porozumět konceptům superskalárních procesorů. Vytvoření interaktivních nástrojů a zlepšení jejich dostupnosti mohou být klíčem k tomu, aby se tato komplexní témata stala pro studenty přístupnějšími a snáze pochopitelnými.

V minulých letech na Fakultě informačních technologií VUT v Brně vznikl výukový simulátor superskalárního procesoru architektury RISC-V [8, 19] pro potřeby předmětu Architektury výpočetních systémů. Simulátor v současné době slouží jako interaktivní a názorná ukázka probíraných principů. V simulátoru je možné nastavit libovolný program, krokovat jeho vykonání po jednotlivých taktech a prohlížet aktuální stav procesoru.

Cílem této práce je navázat na předchozí vývoj a vylepšit simulátor v několika směrech. Nové řešení bude implementováno jako dva nezávislé programy: webová aplikace a simulační server. Jedním z hlavních cílů je vylepšit rozhraní simulátoru, což by mělo pozvednout jeho užitečnost a přilákat více uživatelů. Web je moderním způsobem vývoje a distribuce aplikací. Hlavní přínos by měl spočívat v dramaticky lepší přístupnosti aplikace, jelikož již nebude nutné provádět její instalaci. Webové technologie také umožní implementovat vizuálně bohatější a názornější prezentaci.

Druhým cílem je zvýšení kvality simulace. V tomto ohledu bude pozornost věnována především novým funkcím, doplnění instrukční sady a opravě chyb. Proběhne také rozsáhlá refaktorizace kódu současného řešení, jejíž účelem bude umožnění následné implementace nových funkcí.

V první části práce se v kapitolách 2 a 3 zaměřuji na principy, na kterých jsou moderní superskalární procesory založeny. Uvedeny jsou především ty principy, které jsou v simulátoru demonstrovány, například spekulativní zpracování instrukcí nebo předvídání skoků. Kapitola 4 krátce uvádí základy instrukční sady RISC-V. V kapitole 5 jsou prozkoumány metody tvorby webových aplikací a jejich nasazení do provozu. Je zmíněna knihovna React, která bude v projektu využita. Následně v kapitole 6 analyzuji dosavadní stav simulátoru. Výstupem analýzy je výčet zlepšení stávajících mechanismů a návrh zcela nových funkcí. Výsledný plán je rozveden v kapitole 7. Nejdůležitějšími z vylepšení budou aplikační rozhraní pro simulátor a integrace s překladačem jazyka C.

V závěrečné části práce je detailně prezentováno provedení navrženého řešení. Každá ze dvou částí řešení je popsána samostatně. Kapitola 8 se věnuje implementaci rozšíření simulátoru a soustředí se především na simulační API a další nové funkce. Kapitola 9 se podrobněji rozebírá realizace uživatelského rozhraní webové aplikace.

Kapitola 10 uvádí postupy testování výsledné aplikace včetně výsledků uživatelského testování a úprav implementovaných v reakci na zpětnou vazbu.

Kapitola 2

Skalární procesor

Jádro procesoru představuje centrální prvek CPU, který provádí výpočet popsaný instrukcemi určité *instrukční sady*. Hlavní motivací při jeho hardwarové implementaci je rychlost výpočtu. Protiváhami tohoto úsilí je především cena a spotřeba výsledného čipu. V této kapitole popíšu různé techniky využívané v efektivních implementacích jader procesorů.

Instrukční sada tvoří *programátorův model CPU*. Jedná se o abstraktní reprezentaci funkcionality a chování procesoru s cílem usnadnění vyvíjení kódu. Příklady instrukční sady jsou RISC-V, nebo Intel 64.

Instrukce je základním prvkem výpočtu. Příkladem instrukce může být sečtení dvou čísel, nebo načtení z paměti. Z abstraktního pohledu se procesor nachází v určitém stavu a vykonáním každé instrukce se dostává do následujícího stavu. Tento stav je představován hodnotami v registrech. Stav procesoru a stav hlavní paměti společně tvoří stav výpočtu.

Instrukční sada pouze *popisuje prostředí a zdroje*, které program může využít. Procesor může výpočty docílit libovolným způsobem, pokud se výpočet navenek jeví v souladu s instrukční sadou. Této volnosti využívá hardware, který může zpracovávat více instrukcí současně, a to i *mimo původní pořadí* programu, případně i *spekulativně*. [13]

Podívejme se nejdříve na skalární procesor. Skalární procesor paralelně zpracovává instrukce lineární linkou o pěti stupních.

2.1 Složky výpočetního jádra

Každá instrukce zpracovávaná skalárním procesorem prochází *řetězenou linkou*. V každém taktu procesoru projde instrukce jednou fází. To znamená, že v ideálním případě procesor opustí v každém taktu jedna instrukce. Linka je dělena do následujících fází [9]:

1. *Fetch* – Načtení instrukce,
2. *Decode* – Dekódování instrukce, čtení registrů,
3. *Execute* – Vykonání samotného výpočtu,
4. *Memory Access* – Přístup k paměti,
5. *Write Back* – Zápis do registrů (propsání stavu procesoru).

Tato stádia jsou vhodně zvolena, protože jejich provedení je *nezávislé*. Jejich nezávislost dovoluje vydělit pro každou fázi speciální hardware a jejich výpočet *paralelizovat*. [14]

Ne každá instrukce potřebuje k výpočtu všechna stádia. Například uložení do hlavní paměti nevyžaduje poslední fázi zápisu do registru. V takovém případě se v dané fázi neprovede žádný výpočet.

2.1.1 Fáze výpočtu

Následuje detailnější popis fází klasické pětistupňové linky.

Instruction Fetch (IF)

Prvním krokem zpracování instrukce je její načtení z paměti. Z paměťového modulu je načtena instrukce na adrese uložené v registru `PC`. Implementace mají pro tento účel dedikovanou vyrovnávací paměť a jednotku přednačítání, které zajišťují, že tato fáze proběhne v jednom cyklu. [9]

Registr `PC` je ovlivňován skokovými instrukcemi. Jejich výsledky jsou ale známy až v pozdějších fázích linky. Proto jednotka fetch v taktech, kdy není známa adresa následující instrukce, vkládá do linky prázdné instrukce (`nop`, nebo také *bubbles*).

Jedním způsobem, jak na výpočet skoku nemuset čekat, je skok *předpovědět*. Tento koncept bude rozveden v sekci 3.4.

Instruction Decode (ID)

Během této fáze jsou binární instrukce dekódovány – převedeny na interní reprezentaci CPU. Tato fáze také zahrnuje identifikaci operačního kódu a načtení příslušných registrů z registrového pole. Hodnoty operandů a operační kód jsou linkou předány další fázi.

Ve fázi decode je také často implementována výkonnostní optimalizace pro skokové instrukce. Decode je rozšířen o další hardware, který dokáže vypočítat cíl a podmínku skoku. Předsunutím výpočtu skoku z fáze execute o fázi dříve se sníží pokuta o 1 takt.

Execute (EX)

Ve fázi execute dochází k provedení výpočtu instrukce. Hardware zahrnuje ALU, posuvný registr, násobičku a děličku. Výpočet probíhá nad daty načtenými ve fázi ID.

Aritmetické a logické instrukce vykonávají své operace, paměťové instrukce zde vypočítají adresu pro přístup do paměti a skokové instrukce počítají podmínku a adresu skoku.

Délka fáze execute se může mezi instrukcemi významně lišit. Tabulka 2.1 ukazuje několik instrukcí implementace Intel Ice Lake instrukční sady x86.

Instrukce	Latence (cykly)	Popis
MOV r, r	1	Kopie hodnoty mezi registry
CMP r, r	1	Porovnání hodnot, nastavení příznaků
ADD (32b)	1	Součet dvou hodnot v registrech (šířka 32 bitů)
MUL (32b)	4	Násobení dvou hodnot v registrech (šířka 32 bitů)
DIV (32b)	12	Dělení dvou hodnot v registrech (šířka 32 bitů)
FMUL	4	Násobení dvou hodnot v registrech (float 32 bitů)
FSIN	60-120	Výpočet funkce $sin(x)$

Tabulka 2.1: Vybrané instrukce a jejich latence v cyklech architektury Intel Ice Lake. Předpokládá se, že neproběhnou přístupy do paměti nevzniknou výjimky. [7]

Memory Access (MA)

Fáze „Memory Access“ v CPU zahrnuje přístup do paměti pro čtení nebo zápis dat. Využívá k němu adresu, která byla dříve vypočtena ve fázi execute. Instrukce, které nepřistupují k paměti v této fázi neprovádí žádnou operaci.

Pokud proces pracuje v režimu s *virtuálním adresovým prostorem*, je nutné virtuální adresu přeložit na fyzickou adresu. K překladu slouží hardwarové jednoty PT walker a Translation Lookaside Buffer (TLB).

Délka přístupu do hlavní paměti (DRAM) může trvat přibližně 100 nanosekund, tedy 100 taktů při frekvenci 1 GHz [9]. Pokud by všechny přístupy do paměti byly vyřízeny v hlavní paměti, čas přístupu do paměti by dominoval výpočtu. Za účelem zkrácení doby přístupu procesor k paměti přistupuje skrze systém *cache*. Cache kopíruje část hodnot z hlavní paměti do paměti s významně kratší dobou přístupu, čímž zajišťuje, že fáze MA může být provedena v rámci cyklu. Pokud hodnota v cache není, je nutné celou linku zastavit a počkat na načtení z hlavní paměti. Detailnější popis vyrovnávací paměti je uveden v sekci 2.2.

Zápisy do paměti jsou nejdříve provedeny pouze do dočasného registru, aby mohly být v případě výjimky anulovány.

Write Back (WB)

V této fázi dochází k zápisu výsledku instrukce do registrů. K registrům v rámci cyklu přistupuje i hardware fáze decode. Tento strukturní konflikt musí být vyřešen.

Jelikož je WB poslední fází, jsou zde řešeny výjimky, které jsou sem propagovány z předešlých fází. Výjimka je obsloužena ve dvou krocích: (1) linka je vypláchnuta a (2) PC je nastaven na adresu procedury, která výjimku ošetřuje.

2.1.2 Řetězená linka

Řetězení je formou časového paralelismu. Více instrukcí postupuje jednotlivými stupni linky najednou, čímž zvyšují vytížení a propustnost procesoru. Jednotlivé stupně jsou spojeny za sebou, mezi každými dvěma stupni se nacházejí registry pro předání dat. Rozdělením zpracování každého z n kroků do speciální hardwarové stanice je možné v procesoru vytvořit řetězenou linku (*pipeline*) hloubky n. Maximální možné zrychlení linky odpovídá hloubce linky. Ideálně v každém taktu procesor opouští jedna instrukce a $IPC = 1$ (IPC – instruction per clock).

Pro dosažení maximálního zrychlení je nutné zajistit nepřetržitý přísun nezávislého kódu ke zpracování a stejnou dobu výpočtu každého stupně. Pokud výpočty stupňů nejsou stejně dlouhé, je nutné jako čas cyklu zvolit nejdelší z nich. Náběh a doběh linky a zpoždění registrů mezi jednotlivými stupni se při výpočtu zrychlení zanedbávají.

Kromě výše uvedených komplikací je v reálném kódu propustnost snižována zastavováním linky (*stall*). Důvodem zastavování linky jsou *konflikty*. [14]

V jádře se nachází řídící logika (control unit), která signály ovládá linku. Jednoduchá řídící logika může být implementována jako konečné stavové řízení. [14]

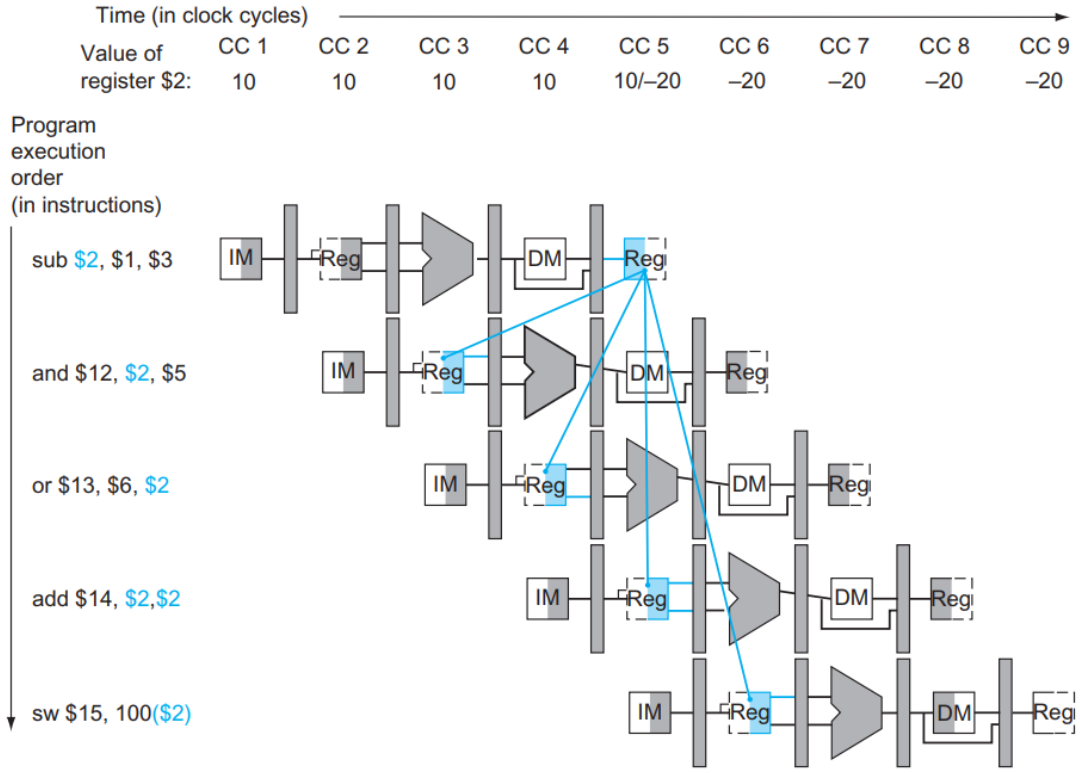
Konflikty

Mezi dvojicí instrukcí dochází k datovému konfliktu, když jedna instrukce provádí výpočet nad daty generovanými druhou instrukcí. Tuto situaci označujeme *read after write* (RAW).

K předání dat dochází prostřednictvím registrů nebo hlavní paměti. Tato závislost může při paralelizaci ve zřetězené lince způsobit chybný výpočet. Stát se tak může, pokud druhá instrukce začne výpočet dříve, než ho první instrukce zpřístupní.

Řešením konfliktu je pozdržení výpočtu dokud výsledek instrukce není k dispozici. Obrázek 2.1 ilustruje situaci, při které se musí linka zastavit pro zachování správnosti výpočtu. Zvýrazněné závislosti mezi registry ukazují, že hodnota `$2` čtená instrukcí `and` není výsledkem předchozí instrukce. Konflikty mohou být částečně kompenzovány vhodným návrhem programu, respektive překladačem, který kód přeuspořádá tak, aby maximálně vykryl čekání na konflikty užitečnou prací.

Pokročilejším řešením je předávání dat mezi fázemi linky speciálními cestami, takzvanými zkratkami. Zkratky mohou eliminovat potřebu zastavovat linku, čímž zvýší její propustnost. [9]



Obrázek 2.1: Sekvence instrukcí vykonávaných ve skalární lince. [14]

2.2 Vyrovnávací paměť

Paměť, která je dostatečně rychlá pro současné procesory, je zároveň velmi drahá. Řešením tohoto problému je *hierarchie pamětí* – víceúrovňová struktura, kde úrovně blíž procesoru mají menší kapacitu, ale větší rychlost. Výsledkem je vyvážený stav mezi výkonem a cenou. [9]

Hierarchie pamětí funguje dobře, protože přístupy do paměti nebývají zcela náhodné, ale řídí se *principem lokality*. Prvním typem lokality je časová lokalita. Ta tvrdí, že k paměťovým místům, ke kterým bylo přistoupeno nedávno, bude pravděpodobně v blízké době přistoupeno znovu.

Druhým typem lokality je prostorová lokalita. Ta předvídá, že k fyzicky blízkým paměťovým místům se přistupuje blízko v čase.

Pokud programy tyto principy při práci s pamětí dodržují, mají tendenci vykazovat lepší výkon. [13]

Hierarchie pamětí na nejrychlejší úrovni využívá registry. Na další úrovni je vyrovnávací paměť (cache), poslední úroveň tvoří hlavní paměť. Modely s více úrovněmi cache jsou běžné, k popsání základních principů se ale budu věnovat modelu s jednou úrovní cache. Programátor s pamětí pracuje jako s celkem, hierarchie pamětí se projevuje pouze rychlostí výpočtu.

Cache obsahuje části hlavní paměti (bloky) se kterými procesor momentálně pracuje. Nové bloky jsou alokovány při čtení nebo zápisu na paměťové místo, které se v cache momentálně nenachází. Záznam v cache obsahuje informaci o původní lokaci bloku v paměťovém prostoru, aby později mohl být vrácen do hlavní paměti. Bloky jsou v paměti zarovnané na násobek své velikosti.

Nejčastěji používaná strategie ukládání bloků je *asociativní cache*. Úložiště pro bloky je v této variantě rozděleno do skupin o m blocích. Každý blok hlavní paměti je částí své adresy (indexem) mapován na právě jednu skupinu.

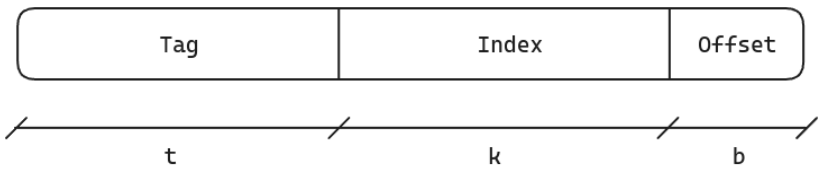
Zajímavé jsou krajní případy. Pokud $m = 1$, potom se cache nazývá přímo mapovaná. Pokud m odpovídá kapacitě cache, potom se cache nazývá plně asociativní.

Pokud ve skupině není pro nový blok místo, je nutné jeden blok vybrat, přemístit ho zpět do hlavní paměti a tím místo uvolnit. Nejznámější strategie výběru bloku ve skupině (victim) jsou:

- náhodný výběr,
- FIFO (výběr nejstaršího bloku),
- LRU (nejdéle nepoužitý blok).

Efektivita cache se vyjadřuje četností nalezení požadovaného bloku (hit rate) v procentech.

Hledání bloku ve skupině je prováděno paralelním porovnáním jiné části adresy (tagu). Použití částí adresy je naznačeno na obrázku 2.2. Prvních `t` bitů je použito jako *tag* k vyhledání v rámci skupiny. Následujících `k` bitů adresuje skupinu a posledních `b` bitů adresuje obsah bloku.



Obrázek 2.2: Adresa paměti a její části.

Kapitola 3

Superskalární procesor

Délka výpočtu je určena třemi hlavními faktory: počtem instrukcí, frekvencí hodinového signálu a počtem instrukcí provedených za hodinový signál (*Instructions Per Clock – IPC*). Architektura superskalárních procesorů zvyšuje výkon zvýšením IPC, typicky až nad hodnotu 1 – jinými slovy, mohou vykonat 2 a více instrukcí ve stejný čas. [13]

Superskalární procesory rozšiřují řetězení na úrovni instrukcí ze skalárních řetězených procesorů. K časovému paralelismu přidávají *prostorový paralelismus*, který spočívá v rozšíření linky na m instrukcí v každém stupni a duplikací potřebných hardwarových jednotek. Výsledkem je, že superskalární procesory mohou vydat k výpočtu více instrukcí v jednom taktu. Cenou za zrychlení je zvýšení složitosti obvodu a tím nižší dosažitelný kmitočet, vyšší spotřeba energie a větší plocha čipu.

Existuje velké množství technik sloužících ke snižování doby zastavení linky. V této kapitole je blíže rozvedeno dynamické plánování, provádění kódu spekulativně a mimo pořadí. Tyto koncepty bývají vysvětlovány a implementovány zároveň, pokusím se je ale vysvětlit izolovaně. Zmíněné metody jsou pouze výběrem z možných způsobů implementace superskalárních procesorů.

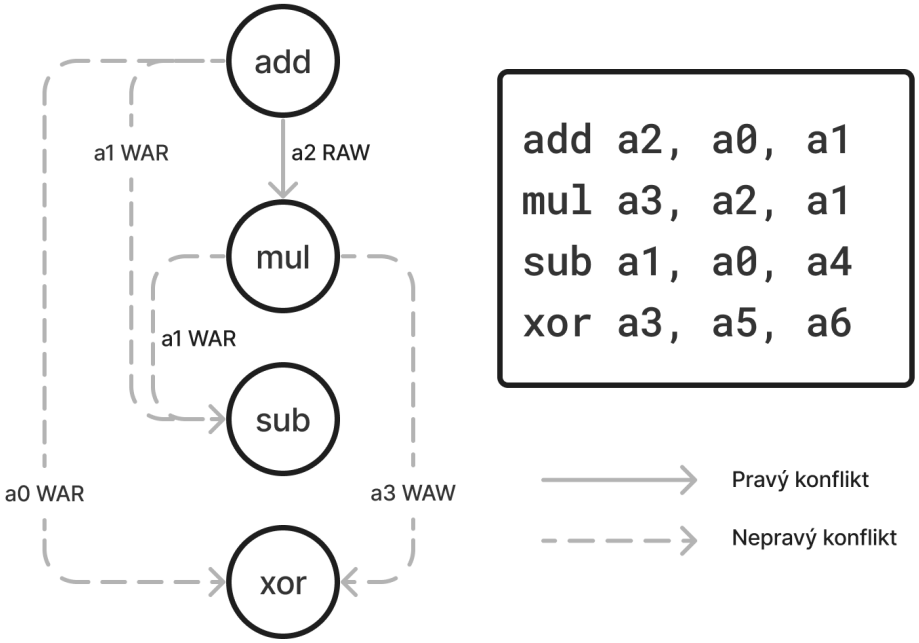
3.1 Konflikty

Kapitola o skalárních procesorech (viz sekce 2.1.2) definovala konflikt jako datovou závislost mezi dvěma instrukcemi. Pro superskalární procesory je užitečné tento problém rozvést blíže.

V případě, kdy dochází k přeuspořádání pořadí vykonání instrukcí, je nutné uvažovat i *nepravé* datové konflikty. Pořadí pravých datových konfliktů (RAW) musí být respektováno, protože na rozdíl od nepravých nesou význam výpočtu. Jinými slovy nelze prohodit pořadí vykonání instrukcí s pravým datovým konfliktem.

Nepravé konflikty můžeme charakterizovat jako konflikty jmen. Vznikají znovupoužitím jména paměťového místa v důsledku konečného počtu registrů, nebo vícenásobným vykonáním stejné instrukce. Důležité je, že nepravou závislost lze na rozdíl od pravé závislosti odstranit bez ovlivnění správnosti výpočtu, protože mezi instrukcemi nejsou vyměňována žádná data [13]. Nepravé závislosti lze řešit přejmenováním při tvorbě programu (programátorem nebo překladačem), nebo za běhu řídícím hardwarem procesoru. Algoritmy Scoreboarding a Tomasulo, které jsou při řešení konfliktů využívány, uvedu v sekci 3.2.

Obrázek 3.1 vizualizuje pravé i nepravé konflikty mezi registry. Nepravé konflikty jsou zobrazeny přerušovanou čarou, pravé konflikty plnou čarou.



Obrázek 3.1: Program a jeho datové závislosti zobrazené jako graf.

3.1.1 Řídící konflikty

Skokové instrukce manipulují programový čítač (PC). Důsledkem je, že pořadí vykonání instrukcí je známo až při samotném výpočtu. Adresa následující instrukce není při zřetězeném zpracování známa po prvním stupni linky (procesor nezná ani typ zpracovávané instrukce, dokud není dekódována), procesor tedy typicky předpokládá, že instrukce skoková není a začíná zpracovávat následující instrukci.

Dokud není vypočítána adresa následující instrukce, může být počítána špatná větev programu, proto je žádoucí tento výpočet urychlit. Techniky pro eliminaci pokut skokových instrukcí zahrnují predikci skoků a předsazení výpočtu podmínky v lince. Pokud dojde ke zpracovávání nesprávné instrukce, musí být z linky odstraněna.

3.1.2 Strukturní konflikty

Ke strukturnímu konfliktu dochází, pokud vykonání dvou instrukcí vyžaduje stejný prostředek. Prostředkem je myšlen hardwarový modul, například funkční jednotka, nebo zápisová brána.

Tento typ konfliktu se řeší serializací zpracování, neboli čekáním. Dopady konfliktů se zmenšují znásobením hardwaru, například přidáním více aritmetických jednotek (ALU).

3.2 Zpracování instrukcí mimo pořadí

Většina procesorů se zpracováním mimo pořadí jsou zároveň superskalární, ale nemusí být nutně. Během výpočtu musí být zachována platnost programovacího modelu, který říká, že projevy instrukcí musí být aplikovány v pořadí. Zpracování instrukcí mimo pořadí dovoluje provádět práci na jiných instrukcích, než ta, která je právě v pomyslné lince na řadě, aniž by model porušila.

Instrukce nemohou být vykonávány v libovolném pořadí, algoritmus musí instrukci označit jako připravenou k vykonání. Analýza závislostí spočívá v detekci datových konfliktů registrů tak, jak je popsána v předešlé sekci 3.1. Komplikace nastává u paměťových operací – konflikty RAW, WAW a WAR nelze odhalit analýzou závislostí registrů, protože konflikty vznikají v hlavní paměti a mohou být ověřeny až jsou vypočítány adresy.

Aby byla dodržena sémantika programu a zároveň bylo umožněno vykonávat paměťové instrukce mimo pořadí, je v procesorech zaveden koncept *relaxované paměťové konzistence* [9]. Čtení a zápisy se mohou přeuspořádat, pokud není narušena správnost programu. Pokud je z nějakého důvodu nutné vynutit pořadí paměťových operací, je nutné vložit explicitní bariéry. Paměťové instrukce za bariérou se začnou vykonávat až jakmile jsou všechny paměťové instrukce před bariérou dokončeny.

Linka procesoru se dělí na dvě části. Front-end superskalárního procesoru odpovídá stupňům Instruction Fetch (IF) a Instruction Decode (ID). Back-end odpovídá stupňům Execute (EX), Memory Access (MA) a Write Back (WB). Front-end pracuje v pořadí programu. Back-end pracuje mimo pořadí programu (OOO – out of order), instrukce back-end opouští opět v původním pořadí. V rámci back-endu se mohou libovolně promísit přístupy do paměti a vykonávání všech instrukcí v okně. Stále ale musí být respektovány datové závislosti.

V cyklu zpracování instrukce přibývá fáze potvrzení instrukce (*instruction commit*). Vydání proběhne jakmile je instrukce na řadě a výsledek je vypočítán. Výsledkem vydání je propsání do vnějšího stavu procesoru.

Procesor identifikuje nezávislé instrukce a vykoná je paralelně. Tím snižuje počet zastavení linky a zvyšuje IPC.

Skokové instrukce představují asi 20% instrukcí programu. Z toho vyplývá, že okno instrukcí, o kterých víme, že budou vykonány, je příliš malé. Proto bývá zpracování mimo pořadí nejčastěji spojeno se spekulativním vykonáváním.

3.2.1 Reorder Buffer

Pokud jsou instrukce vykonávány mimo pořadí, musí být v hardwaru udržena informace o původním pořadí. Za tímto účelem back-end obsahuje *Reorder Buffer* (ROB). Jedná se o cyklický buffer s typickou kapacitou 100-200 položek [9]. Instrukce, jejich výsledky a související příznaky jsou zde uchovávány v programovém pořadí.

Do fáze commit vstupují instrukce na čele ROB. Jakmile instrukce opustí ROB, přestává být spekulativní. Výsledek instrukce je propsán do architekturních registrů a instrukce je považována za potvrzenou.

Zaplněním ROB vzniká strukturní konflikt a předchozí fáze linky se musí pozastavit.

3.2.2 Algoritmus Tomasulo

Dva nejznámější algoritmy pro dynamické plánování instrukcí jsou *ScoreBoarding* a *Tomasulo*. ScoreBoarding má velká omezení[1], proto blíže rozvedu pouze algoritmus Tomasulo.

Hlavním hardwarovým prvkem algoritmu je *rezervační stanice* (RS), buffer pro ukládání operandů. RS může být centrální, nebo individuální pro každý druh instrukcí. Položka RS má následující pole:

- *busy bit* – příznak, zda je položka obsazena a validní,

[1] ScoreBoarding je omezen na plánování v rámci *basic bloku* instrukcí, WAW a WAR konflikty řeší čekáním.

- *operace* – druh operace (například sčítání),
- *operandy* – trojice (hodnota, tag, valid),
  - *hodnota* – kopie hodnoty operandu,
  - *tag* – ukazatel na registr operandu,
  - *valid* – příznak, zda je pole hodnota validní,
- *destinace* – ukazatel na registr, do kterého má být výsledek zapsán.

Tato struktura řeší konflikty RAW – instrukce je poslána do funkční jednotky až v moment, kdy jsou všechny operandy připraveny.

Konflikty WAR a WAW jsou vyřešeny *přejmenováním registrů*. Podstata těchto falešných konfliktů není datová, jedná se o *konflikt jmen*. Výsledek každé instrukce se zapíše do nového, unikátně pojmenovaného registru. Dekódované instrukce místo původních jmen operandů použijí jejich nejaktuálnější přejmenování. S novými jmény registrů v kódu zůstanou pouze pravé RAW konflikty.

K ilustraci přejmenování poslouží obrázek 3.2. Všimněte si, že vstupní registry používají nejaktuálnější přejmenování a výstupní registry vytvoří nové přejmenování.

```
1       add x2, x1, x1
2       sub x2, x2, x3
3       mul x4, x5, x2
4       shr x5, x1, x4
```

(a) Původní jména operandů.

```
1       add t0, x1, x1
2       sub t1, t0, x3
3       mul t2, x5, t1
4       shr t3, x1, t2
```

(b) Jména operandů po přejmenování.

Obrázek 3.2: Přejmenování registrů.

Je nutné v hardware udržovat informaci o posledním přejmenování architekturních registrů, a to ze dvou důvodů: (1) přejmenování operandů nových instrukcí a (2) propsání výsledků při propouštění instrukcí. Implementace přejmenování vyžaduje dva prvky. Prvním je tabulka RAT (*Register Alias Table*). RAT implementuje mapování jmen architekturních registrů na jejich nejaktuálnější přejmenování.

Druhým prvkem je úložiště spočtených výsledků. Zde jsou možné dvě implementace. Ve variantě přejmenování v ROB položky ROB obsahují spočtenou hodnotu instrukce. Výsledek je ve fázi commit propsán do architekturního registru.

Druhou variantou je přejmenování v RRF (*Rename Register File*). Zde se výsledky ukládají do velkého pole registrů. Pole může být spojeno s polem architekturních registrů, v takovém případě propsání do architekturního registru proběhne přepsáním ukazatelů do pole.

Komunikace výsledků probíhá přes sdílenou sběrnici. Funkční jednotky na sběrnici posílají výsledky; pole registrů, ROB a RS naslouchají a aktualizují své hodnoty.

3.2.3 Load/Store jednotka

Podpora vykonávání paměťových instrukcí mimo pořadí vyžaduje speciální hardware. Instrukce load a store musí být udržovány v programovém pořadí, v tabulkách Load Buffer a Store Buffer.

Položka v Load Bufferu obsahuje vypočtenou adresu. Adresa může být vypočtena mimo pořadí. Load může načíst z paměti, pokud jsou všechny adresy předchozích instrukcí store

vypočteny a nepřekrývají se s loadem. Pokud je nalezen store se stejnou adresou a položka store bufferu obsahuje zapisovanou hodnotu, může proběhnout optimalizace předání hodnoty, čímž se ušetří jedno čtení z paměti.

Instrukce store může být vykonána, pokud je na čele ROB.

3.3 Spekulativní zpracování instrukcí

Koncept spekulativního vykonávání jsem již zmínil v sekci 3.4 o předpovědi skoků. Při spekulativním vykonávání se spekuluje o řízení programu, datech a paměťových závislostech. Instrukce, jejíž výsledek byl předpovězen, je zpracována spolu s ostatními instrukcemi a běžným výpočtem se zkontroluje, zda byla predikce správná. [13]

Kontrola probíhá ve fázi potvrzení instrukce. Touto fází projdou pouze instrukce, které jsou jistě produktem správné předpovědi. Pokud predikce odpovídá výsledku, je možné pokračovat ve výpočtu. Pokud predikce selhala, je nutné všechny následující instrukce označit za nesprávné a smazat jejich rozpočítané výsledky. Smazat výsledky je nutné, protože mohou být produktem jiných nesprávných výsledků.

Výjimky se také musí projevit až v momentě potvrzení instrukce, protože do té doby není jisté, zda se instrukce skutečně má vykonat. Aby bylo možné spekulovat, výsledky výpočtu tedy musí být možné *navrátit*.

Hlavním předpokladem spekulativního vykonávání je ten, že předpovědi mají vysokou úspěšnost. Každá špatná predikce znamená, že se výpočetní výkon vynakládá na výpočet špatných instrukcí, nebo špatných hodnot operandů.

3.3.1 Předvídání skoků

Při spekulaci o větvení do linky vstupují a jsou zpracovávány instrukce, o kterých nemusí být jisté, jestli jsou pro výpočet nutné a že zpracovány být mají.

Superskalární procesor načítá více instrukcí v jednom taktu. To znamená, že v jednom taktu může načíst více než jednu skokovou instrukci. Fetch jednotka musí být schopna buď zastavit načítání před druhou skokovou instrukcí, nebo musí vypočítat více predikcí v rámci jednoho taktu.

Detailněji bude předvídání skoků rozvedeno v sekci 3.4.

3.3.2 Předvídání čtení z paměti

Spekulativní provádění paměťových operací sebou nese komplikaci: efekt spekulativního zápisu do paměti (nebo cache) se nesmí projevit, dokud instrukce není potvrzena (z důvodu špatné predikce, nebo výjimky). Z toho důvodu se data zapisují do Store Bufferu a do paměti se zapisují až při potvrzení instrukce.

Verzi popsanou v sekci 3.2.3 lze rozšířit spekulací. Instrukce load již nemusí čekat na všechny adresy starších instrukcí store, ale mohou spekulovat, že u žádné z instrukcí store s nedopočítanou adresou k překrytí nedojde. Nejjednodušší strategií je spekulovat, že k překrytí nedojde nikdy. Ta funguje dostatečně dobře, protože pravděpodobnost pravé závislosti je malá. Složitější prediktory nebývají v praxi používány. [12]

Tato spekulace je ověřena, když je store propouštěn. Adresa store je porovnána s položkami v Load Bufferu, v případě shody jde o špatnou spekulaci a stejně jako u spekulace se skoky se vypláchne ROB.

3.4 Předvídání skoků

Předpověď skoku má dva komponenty: předpověď podmínky skoku a předpověď cílové adresy skoku. Předpověď podmínky se nevztahuje pouze na podmíněné skokové instrukce. U nepodmíněných skoků je sice jisté, že se skok má provést, jednotka fetch ale sama o sobě nemůže identifikovat instrukci jako nepodmíněný skok – instrukce je dekódována až ve fázi decode. Z tohoto důvodu prediktory pracují s *adresou instrukce*.

Při prvním zpracování instrukce na nové adrese není známo, zda se jedná o instrukci skokovou. Proto je jedinou možností pokračovat ve zpracování sekvence instrukcí.

Při následujících načteních skokových instrukcí již o nich existuje záznam a je možné skok předpovídat.

Předpověď musí být ověřena porovnáním s výsledkem klasického výpočtu skoku. Při případném zjištění nesprávné předpovědi skoku musí být všechny následující rozpočítané instrukce z linky vypláchnuty. Po vypláchnutí linky je registr PC opraven na správný cíl skoku a procesor může pokračovat ve výpočtu. Nesprávně může být předpovězen i cíl skoku. V klasické skalární lince jsou instrukce ze špatně předpovězené větve zrušeny dříve, než jsou vykonány.

3.4.1 Předpověď podmínky skoku

V této sekci budu používat pojmy *pozitivní a negativní predikce* pro označení situace, kdy prediktor vyhodnotí, že je nebo není splněna podmínka dané skokové instrukce.

Strategie pro predikci podmínky skoku se dělí na dvě skupiny – statické a dynamické. Jejich rozdíl v tom, že dynamické strategie k predikci používají informace o chování za běhu programu, typicky historií větvení. [17]

Statické předpovědi

Nejjednodušší verzí předpovědi podmínky skoku je statická negativní predikce. V tomto případě je vždy načtena následující instrukce a není potřeba předpovídat cíl skoku. Statická pozitivní predikce může mít vyšší úspěšnost.

Jiné statické strategie mohou brát ohled na operační kód instrukce. Prediktor může například instrukce `beq` předpovídat negativně a instrukce `blt` předpovídat pozitivně. Operační kód instrukce může obsahovat příznak, kterým se prediktor může řídit. Předpověď v tomto případě učinil překladač, profilovací nástroj, nebo programátor. [17]

Další strategie mohou využít směru skoku (skok dopředu nebo dozadu), nebo vzdálenosti skoku. Směr skoku je zajímavým ukazatelem, protože v mnoha programech velkou část skoků směrem zpět tvoří smyčky, které typicky provádějí velký počet iterací. Nevýhodou je, že předpověď cílové adresy a podmínky nemůže být provedena paralelně.

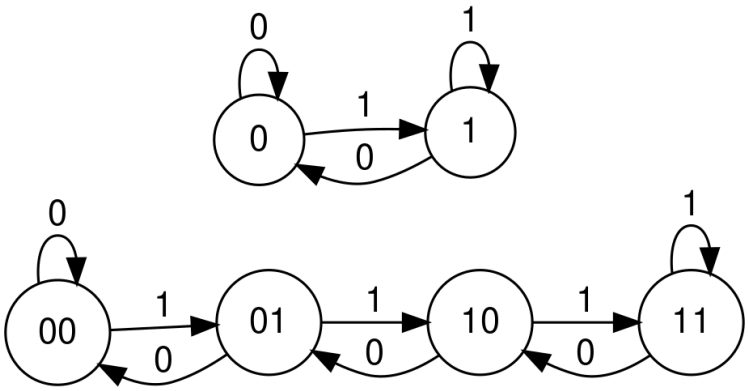
Statické strategie mají příliš malou úspěšnost pro použití v současných procesorech.

Dynamické předpovědi

Dynamická predikce skoků mění verdikt v průběhu programu. Stav prediktoru udržuje historií větvení, znalost historie se používá k predikci větvení. V ideálním případě by každá instrukce (adresa) měla vlastní stav prediktoru, z praktických důvodů se ale tabulka těchto stavů s názvem Branch History Table (BHT) adresuje částí adresy v registru PC. Pokud je tabulka vůči kódu malá, může dojít ke sdílení stavu prediktoru více instrukcemi. [14]

Dynamické prediktory se umí naučit různé vzory [9]. Nejjednodušší možností dynamické predikce je predikce na základě stavového automatu. Podle počtu stavů se prediktory označují jako 1 bitový (2 stavy) nebo 2 bitový (4 stavy), oba jsou znázorněny na obrázku 3.3. Stavy zde představují saturační čítač.

Z aktuálního stavu je potom odvozena předpověď. Pokud je aktuální stav v pravé polovině automatu, potom se skočit má, jinak se předpovídá neskočit. Přechod do nového stavu se uskuteční při zpětné vazbě prediktoru. Přechod označený „1“ znamená, že skok se doopravdy uskutečnil, přechod „0“ znamená, že ke skoku nemělo dojít.



Obrázek 3.3: Schéma 1-bitového prediktoru (nahoře) a 2-bitového prediktoru (dole).

Dynamickým prediktorům je poskytována zpětná vazba v podobě informace o úspěšnosti poslední predikce. V případě stavového automatu se stav posune po hraně značené „+“, pokud byla predikce ověřena jako správná, nebo „-“ v opačném případě. Stavy prediktorů jsou inicializovány na určitou počáteční hodnotu. První predikce mohou mít špatnou úspěšnost. Tato fáze se nazývá učící období (learning period). Po naučení vzoru se úspěšnost prediktoru ustálí. Každý druh prediktoru je schopen naučit se pouze určitou podmnožinu vzorů.

Další variantou dynamických prediktorů je *adaptivní prediktor*. K predikci větvení jsou použity dvě informace: (1) historie posledních k výsledků větvení dané instrukce a (2) záznam o chování skokové instrukce v minulých případech, kdy této instrukci předcházela stejná historie skoků. Implementace spočívá v k bitovém posuvném registru historie skoků a tabulce Pattern History Table (PHT): 2^k prediktorů pro každou instrukci. Konkrétní prediktor je adresován vektorem historie a adresou instrukce. [22]

Korelační prediktor místo lokální historie používá jedinou, globální historií.

3.4.2 Předpověď cílové adresy skoku

K predikci cílové adresy skoků se využívá cache nazývaná Branch Target Buffer (BTB). Tato tabulka se indexuje adresou skokové instrukce.

Komplikací jsou nepřímé skokové instrukce, neboli instrukce, jejichž cíl skoku není konstantní. Takové skoky jsou využívány hlavně při návratu z funkce (`ret`), nebo při práci s ukazateli na funkce. Predikce by byla nepřesná, protože funkce bývá volána z několika míst. Úspěšnost lze zlepšit zásobníkem Return Stack Buffer (RSB), na který se při vstupu do funkce adresy ukládají a při opouštění vybírají.

Kapitola 4

Architektura RISC-V

RISC-V je otevřená instrukční sada architektury RISC. Původně byla vyvinuta na UC Berkeley pro výukové účely. Instrukční sada je navržena pro maximální jednoduchost a rozšiřitelnost, s cílem mít malé požadavky na hardware implementace. [20]

Definuje registry, samotnou instrukční sadu, její kódování a rozšíření, konvenci volání (ABI), výjimky. Paměťový systém je navržen jako liitle-endian. Specifikace připouští varianty s big-endian nebo oba systémy současně.

4.1 Architektura RISC

Architektury RISC (*Reduced Instruction Set Computer*) kladou důraz na jednoduchost hardwaru, který instrukční sadu implementuje a na malou spotřebu energie. Díky důrazu na nízkou cenu, malé ploše čipu a malému příkonu se často objevuje v malých zařízeních poháněných bateriemi, například v IoT a mobilních telefonech.

RISC se vyznačuje menším počtem instrukcí. Typicky se jedná o minimální množinu instrukcí, ve které je možné popsat výpočty a práci s hardwarem. Do instrukční sady se ale dostávají i další instrukce. Zajímavostí a extrémním případem je instrukce FJCVTZS instrukční sady ARM. Jde o instrukci pro specifický převod čísla float64 na celé číslo. Tento výpočet je možné provést kombinací jiných instrukcí. Důvodem přidání této složitější instrukce do instrukční sady navzdory filozofii RISC byl výkon v důležitém případu užití – tuto konverzi často provádějí interprety JavaScriptu.

Situace s menším množstvím instrukcí se dá vylepšit makry v assembleru, takzvanými *pseudoinstrukcemi*. Pseudoinstrukce jsou lexikální náhradou za instrukce s bližším sémantickým významem. Tabulka 4.1 uvádí několik příkladů. Instrukční sadu je možné tímto způsobem virtuálně rozšířit. V současném stavu, kdy je naprostá většina strojového kódu generována překladači, ale programátorský komfort není menší instrukční sadou negativně ovlivněn.

Pseudoinstrukce	Ekvivalent z instrukční sady	Význam
mv rd, rs	addi rd, rs, 0	Kopírování hodnoty
neg rd, rs	sub rd, x0, rs	Negace celého čísla
bgt rs, rt, offset	blt rt, rs, offset	Skok, pokud rs>rt

Tabulka 4.1: Příklady pseudoinstrukcí a jejich odpovídajících reálných instrukcí.

Kódování instrukcí je optimalizováno pro jednoduché načítání a dekódování. Kódování mívá fixní délkou (např. 4 B), operační kód a operandy jsou zapsány na předvídatelných místech v kódu. Jednodušší dekódování znamená, že k jeho implementaci je zapotřebí méně hardwarových obvodů a čip má menší spotřebu.

Protipólem je architektura CISC (*Complex Instruction Set Computer*), která definuje větší množství instrukcí. Jedna instrukce může představovat i složitější operace, například aritmetickou operaci a načtení z paměti. Kódy mají typicky proměnlivou délku kódování. Častěji používané operace mají kratší kódy, což se může projevit kompaktnější reprezentací programu v paměti.

Některé implementace CISC v rámci fáze dekódování převádí instrukce na sekvenci *mikro-instrukcí*, další části linky dále operují s touto reprezentací. Mikro-instrukce více připomínají instrukční sadu RISC. Dekódování je složitější a proto má větší nároky na hardware.

4.2 Instrukční sada

Specifikace definuje základní celočíselnou sadu instrukcí ve dvou šířkách registrů, 32 bitů (`RV32I`) a 64 bitů (`RV64I`). Verze `RV64I` je analogická s `RV32I`, s tím rozdílem, že registry mají šířku 64 bitů. Procesor musí implementovat alespoň jednu z těchto dvou sad a libovolné množství rozšíření. Dále budu popisovat pouze 32-bitovou variantu.

Z pohledu programátora je stav procesoru vyjádřen 31 obecnými registry pojmenovanými `x1`-`x31` a speciálními registry `x0` a `pc`. Registr `x0` obsahuje konstantní hodnotu 0, registr `pc` obsahuje ukazatel na aktuální instrukci.

Základní instrukční sada definuje aritmetické instrukce, řídící instrukce a instrukce pro práci s pamětí. Aritmetické instrukce nevyvolávají výjimky a nekontrolují přetečení. Přetečení je možné zkontrolovat explicitní podmínkou. Kódování instrukcí dovoluje vyjádřit přímé hodnoty v rozsahu 12 bitů. Načtení 32 bitové konstanty je nutné provést kombinací instrukcí `LUI` a `ADDI`.

Skokové instrukce umožňují podmíněné a nepodmíněné relativní skoky. Skok na absolutní adresu je možný kombinací instrukcí `LUI` a `JALR`. Instrukce musí být zarovnané, proto skok na nezarovnanou adresu vyvolá výjimku. Uložení návratové adresy provádí instrukce `JAL`. Instrukce pro podmíněné skoky provádějí komparaci dvou registrů, vykonávají tedy dvě operace – *compare* a *branch*.

Reprezentace instrukce v paměti zabírá 4 bajty, je zarovnaná na 4 bajty a je zakódována v jednom ze čtyř formátů[1]. Formáty mají společné pole pro *opcode* a 5-bitové adresy operandů-registrů. Liší se ve využití zbylého prostoru, který je interpretován buď jako přímá hodnota, nebo další část *opcode*. [20]

Specifikace RISC-V je rozdělena na dvě části. Druhá část definuje privilegovaný režim, který je nutný k provozu operačního systému. Architektura poskytuje tři režimy: Machine (M), Supervisor (S) a User (U).

Speciální registry (*control and status registers* – CSR) slouží ke sběru statistik a ladění. Jejich čtení a nastavování je umožněno speciálními systémovými instrukcemi. Jejich zápis je také vyvolán jako vedlejší efekt vykonávání instrukcí. Instrukce `ECALL` slouží k žádosti o obsloužení jádrem. Obdobně jako instrukce `SYSCALL` z ISA x86, argumenty jsou definovány podle používaného ABI.

[1] Rozšíření C navíc definuje komprimované 16-bitové instrukce

4.2.1 Rozšíření instrukční sady

Důležitou vlastností RISC-V je rozšiřitelnost. Jednodušší čipy mohou implementovat pouze ta rozšíření, která ke svému účelu nutně potřebují. Případně mohou jednoduše specifikovat vlastní instrukce relevantní pro svou doménu.

Rozšíření implementovaná daným zařízením jsou specifikována základní sadou a výčtem rozšíření. Typickou sadu rozšíření vyjadřuje zkratka `RV32IMAFD`, také nazývanou `RV32G`. Významy těchto rozšíření jsou uvedeny v tabulce 4.2.

Zkratka	Popis rozšíření
M	Celočíselné násobení a dělení
A	Atomické instrukce
F	podpora čísel *single* podle standardu IEEE 754-2008[2]
D	podpora čísel *double* podle standardu IEEE 754-2008

Tabulka 4.2: Nejvýznamnější rozšíření instrukční sady RISC-V. Celý výčet je k dispozici ve specifikaci RISC-V [20].

RV32E je varianta základní instrukční sady, která má pouze 16 obecných registrů. Je určena pro čipy s velmi malou plochou.

4.3 Paměťový model, vlákna

RISC-V definuje 32-bitový paměťový prostor. Přístupy do paměti nemusí být zarovnané, ale nezarovnané přístupy nemusí být atomické.

RISC-V používá paměťový model *RISC-V Weak Memory Ordering* (RVWMO). V tomto modelu může jádro pozorovat paměťové instrukce jiného jádra v jiném než původním pořadí. Pro komunikaci vláken prostřednictvím sdílené paměti musí proto být zavedena synchronizace. Rozšíření A k tomuto účelu představuje instrukce pro atomické paměťové operace. Instrukce `FENCE` umožňuje realizovat paměťovou bariéru a tím vynutit pořadí paměťových operací.

4.4 Aplikační binární rozhraní

Aplikační binární rozhraní definuje konvenci volání a specifika RISC-V pro formáty ELF a DWARF. Předepisuje také velikosti a zarovnání pro datové typy jazyka C. [4]

Konvence volání označuje způsob komunikace vstupů a výstupů mezi procedurami. Při popisu se používají aliasy pro jména registrů. Tato jména odrážejí funkci registru v konvenci volání. Například registr `x2` se také nazývá `sp` (stack pointer), jelikož ukazuje na vrchol zásobníku. Pro každý registr je specifikováno, zda má volání procedury zachovat jeho hodnotu, nebo je možné ji přepsat.

RISC-V ABI preferuje předávání argumentů registry. Celočíselné argumenty se předávají registry `a0-a7`. Pro čísla s plovoucí desetinnou čárkou se používají registry `fa0-fa7`. Pokud počet registrů nestačí, další argumenty se předávají *zásobníkem*.

Kapitola 5

Webová rozhraní

Webový prohlížeč a webové technologie se v posledních letech staly uživateli očekávaným standardem pro interakci s počítačem. Pokud aplikace dokáže splnit své požadavky v prostředí prohlížeče, potom je pro vývojáře výchozí volbou. Hlavním důvodem popularity je distribuce – uživatelé mohou aplikaci najít a začít okamžitě používat. Vývoj webové aplikace je také rychlejší a levnější než vývoj na alternativních platformách.

S rostoucí popularitou, implementací nových standardů a vývojem rámcových řešení podíl webových aplikací stále roste. Mnohé desktopové programy a mobilní aplikace jsou také implementovány webovými technologiemi.

5.1 Základní koncepty a technologie

Základem obsahu na World Wide Web jsou *hypermédia*. Hypermediální dokumenty jsou spojeny navigovatelnými referencemi, takzvanými *hyperlinky*. Společně tvoří síť propojených informací, kde uživatelé mohou pohodlně přecházet mezi jednotlivými stránkami a získávat různorodý obsah, například ve formě textu nebo videa. Dokumenty mohou být interaktivní – tímto způsobem jsou implementovány složitější webové aplikace.

Dokumenty, jejich reprezentace a způsob jejich renderování jsou definovány kolekcí standardů vyvinutých WHATWG. Hlavním standardem je HTML Standard, který definuje jazyk HTML a některá API jako například `localStorage`. Standard se dále odkazuje na velké množství dalších standardů, např. HTTP, CSS, Unicode, XML a standardy obrazových formátů. [2]

Tyto standardy jsou implementovány webovými prohlížeči. Prohlížeč má roli hypermediálního klienta. Jeho prvním úkolem je komunikovat se servery v síťové architektuře *klient-server*, ve které klienti poptávají zdroje od speciálních účastníků sítě – serverů. Zdroji jsou v případě webu myšleny hypertextové dokumenty, multimédia a další soubory. Komunikace mezi uzly je rozvedena v následující sekci.

Druhým úkolem prohlížeče je tyto dokumenty zobrazovat uživateli. Uživatelská rozhraní blíže rozvedu v sekci 5.2.

5.1.1 Přenosové protokoly

Výměna dokumentů mezi serverem a klientem probíhá bezstavovým textovým protokolem *HTTP*. Jedná se o protokol typu request/response aplikační vrstvy. Využívá transportní protokol TCP, poskytuje tedy spolehlivý přenos.

Zdroje jsou na webu identifikované pomocí *Uniform Resource Identifier* (URI). V hlavičce zprávy typu request se přenáší verze protokolu, metoda, URI požadovaného zdroje, hlavičky s informacemi o klientovi a požadovaném zdroji a v některých případech i tělo zprávy s libovolnou sekvencí bajtů. Server odpovídá zprávou response, která obsahuje statusový kód, hlavičky a data určitého typu. [6]

HTTP poskytuje několik sémantických metod. Metody `GET` a `HEAD` slouží k získání dokumentů. Metodami `DELETE`, `POST` a `PUT` klient žádá server o provedení nějakého *vedlejšího efektu*, například přidání nového příspěvku na sociální síť.

Trojciferný statusový kód odpovědi indikuje, zda byl příspěvek zpracován. Konkrétní kódy mají specifické významy, dělí se do pěti skupin: informační, úspěchové, přesměrovací, chybové na straně klienta a chybové na straně serveru.

Požadavky na web se neustále vyvíjí a existuje poptávka po alternativních protokolech. Protokol HTTP se vyvíjel do verze 2 a 3. Firma Google představila vlastní protokol QUIC, který především snižuje marži šifrované komunikace a umožňuje použít jedno spojení k přenosu několika streamů (multiplexing). Vyšší efektivita přenosu se pozitivně projevuje především při používání na pomalejších mobilních sítích.

Sezení

HTTP je bezstavovým protokolem, aplikace ale pro své cíle může vyžadovat použití kontextu. Příkladem kontextu může být identita uživatele pro autorizaci požadavku nebo personalizaci. Sezení (sekvence dotazů sdílející kontext) je často implementováno pomocí *cookies*. Cookie je textový token vytvořený serverem, přenášený v hlavičce každého dotazu. Konkrétní způsob jejich využití závisí na serveru. [3]

Jedním ze schémat k ustanovení spojení je identifikátor sezení (*session ID*). Na straně serveru je tento identifikátor z hlavičky přečten a spojen s konkrétními daty v databázi sezení.

Druhým častým způsobem navázání sezení je technologie JSON Web Token (JWT). Tento token obsahuje libovolná textová data a datum expirace. Token je kryptograficky podepsán, aby byla zaručena integrita dat.

Zabezpečení

Protokol HTTP neposkytuje důvěrnost ani integritu. Pokud je aplikace vyžaduje, je potřeba navázat spojení přes protokol HTTPS. HTTPS je protokol HTTP přenášený pomocí kryptografického protokolu *Transport Layer Security* (TLS).

Protokol spočívá v ustanovení symetrického klíče sezení. Identita serveru je také ověřena u důvěřované *certifikační autority*.

Ustanovení TLS (verze 1.2) spojení přidává latenci 2 RTT (Round Trip Time). S navázáním TCP spojení a samotným HTTP dotazem se vytvoření nového spojení dostává na minimální zpoždění 4 RTT (není započítáno vyhledání v DNS). Takové zpoždění se může významně negativně projevit dlouhou čekací dobou na načtení stránky, obzvlášť na mobilních sítích. TLS verze 1.3 přináší schopnost obnovit spojení na dříve navštívenou stránku za 2 RTT díky funkcionalitě 0-RTT. [1]

Hodnoty cookies jsou přenášeny v hlavičkách HTTP, nešifrovaně. RFC 6265 [3] doporučuje, aby byly citlivé hodnoty šifrovány a podepsány, a to i v případě, že je hlavička přenášena přes HTTPS.

5.2 Skriptování

Interaktivitu uživatelských rozhraní pohání skriptovací jazyk JavaScript. Skriptům jsou přístupná API pro manipulaci DOM (Document Object Model), což je stromová reprezentace aktuálního dokumentu. Skriptování se používá k vytváření dynamických a interaktivních webových rozhraní, validaci formulářů a práci s různými API. Webová API například umožňují skriptům pracovat se souborovým systémem, nebo komunikovat pomocí HTTP.

Příklady použití rozhraní pro manipulaci stránky jsou volání jako `element.appendChild`, nebo `querySelector`. Při vývoji složitých aplikací se ale typicky tato volání používají pouze nepřímo prostřednictvím *knihoven*. Mezi nejznámější zástupce patří React, Angular, a Vue.js.

K vývoji webových aplikací se velmi často používá jazyk TypeScript. TypeScript rozšiřuje syntax JavaScriptu o typové informace, což umožňuje lepší nápovědy a kontroly ve vývojovém prostředí. Zdrojový kód je nutné před vykonáním transpilovat do JavaScriptu.

V současné době je rozvíjena technologie WASM, což je virtuální stroj založený na bajtkódu. Tato technologie umožňuje psát programy pro web v libovolném kompilovaném programovacím jazyce a slibuje vyšší výkon. Technologie je však stále v zárodku, proto ji v této práci podrobněji nepopíšu.

5.2.1 React

React je open-source[1] knihovna vyvinutá firmou Meta pro vytváření uživatelských rozhraní. Základním stavebním blokem UI je *komponent*. Každá jednotlivá stránka se skládá ze stromové hierarchie komponentů.

Komponent je uzavřený celek s definovaným rozhraním, který implementuje jeden prvek uživatelského rozhraní včetně jeho chování a vzhledu. Vývoj aplikace orientovaný na komponenty podporuje znovupoužitelnost, modularitu a testovatelnost. Komponenty mohou mít vlastní vnitřní stav a vykonávat kód v různých stádiích životního cyklu (při změně parametrů, zániku instance komponentu apod.).

React k definici komponentů používá rozšíření syntaxe JavaScriptu nazývané JSX. Díky JSX je možné vkládat fragmenty HTML přímo do skriptů. Příklad jednoduchého komponentu je uveden v příkladu 5.1. Soubory `.jsx` je nutné zkompilovat do standardního JavaScriptu.

```
1 export default function List({items}) {
2     const listItems = items.map(item =>
3         <li key={item.id}>
4             <b>{item.text}</b>
5         </li>
6     );
7     return <ul className="large-font">{listItems}</ul>;
8 }
```

Výpis 5.1: Příklad komponentu definovaného v `.jsx` souboru. Komponent renderuje array předanou v parametrech jako list v HTML. Komponent definuje jak logiku, tak i vzhled. S fragmenty HTML je možné pracovat jako s hodnotami.

Knihovna je velmi populární, díky tomu lze v projektech využít velkého množství dalších knihoven (například pro práci s globálním stavem), nebo využít celé předpřipravené komponenty. Nevýhodou je nižší výkon aplikací oproti řešení v čistém JavaScriptu a velikost knihovny, kterou je nutné stáhnout při návštěvě stránky (130 kB kódu).

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

V současné době je doporučováno React používat prostřednictvím jiného rámcového řešení, jakým je například *Next.js*.

5.2.2 Next.js

Next.js[2] je *fullstackovým frameworkem*. Znamená to, že výsledná aplikace zastává funkci serveru i klienta. Tento framework doplňuje React do uceleného řešení pro webové aplikace.

Strom stránek je definovaný strukturou souborů a složek (file-system based router). Jedná se o častý způsob definice struktury webové aplikace. Každá stránka je definovaná kořenovým komponentem.

Stránky jsou alespoň částečně staticky renderované na serveru. Pokud stránka obsahuje dynamický obsah, je dodatečně renderovaný na straně klienta. Tato kombinace renderování na stranách serveru i klienta se nazývá hybridní přístup k renderování. Pouze první stránka je stažena ze serveru – následující navigace mezi stránkami jsou vykonány JavaScriptem.

Framework se také stará o cachování, přednačítání zdrojů a další optimalizace s cílem zvýšit výkon aplikací. Významně také zjednodušuje nasazení aplikace do provozu prostřednictvím serverů firmy Vercel, autora frameworku. [18]

5.3 Uživatelská rozhraní

Uživatelská rozhraní (User Interface – UI) se převážně zaměřují na aspekt *použitelnosti* – efektivitu a spokojenost, s jakou je uživatel schopen dosáhnout svých cílů. Celková příjemnost produktu ale může záviset na více faktorech, než pouze použitelnost. Pokud se návrh aplikace zaměří pouze na použitelnost, celkový dojem z aplikace nemusí být optimální. [10]

Použitelnost je ovlivněna mnoha faktory, například intuitivností, spolehlivostí, nebo úsilím nutným k používání aplikace. Estetická příjemnost, uspořádanost a čitelnost rozhraní jsou také důležitými faktory pro její pozitivní vnímání. [16]

5.3.1 Použitelnost

Použitelnost je vlastnost systému vyjadřující jeho jednoduchost pro používání i naučení.

Jedním ze způsobů evaluace použitelnosti je kognitivní analýza (*Cognitive walkthrough*). Tato metoda se zaměřuje na nejdůležitější úkoly v aplikaci a jednotlivé kroky, ze kterých se úkol skládá. Analýzu provádí vývojář z perspektivy nového uživatele a posuzuje jeho schopnost dosáhnout svých cílů při použití aplikace. Metodu je možné začít využívat již v raných fázích vývoje, jakmile je k dispozici prototyp. Výstupem analýzy je seznam prvků rozhraní, které mohou být pro nové uživatele představovat překážky. [15]

Postup kognitivní analýzy je jednoduchý. Postupně jsou analyzovány jednotlivé úkoly z předem definovaného seznamu. Jeden z účastníků provádí vybraný úkol a zastavuje se při každém kroku. Ostatní účastníci debatují o potenciálu uživatele úspěšně krok provést. K hodnocení jim pomáhají předem připravené otázky.

5.3.2 Dostupnost

Dostupnost se v kontextu webu zaměřuje na podporu široké škály možností interakce s aplikacemi. Důležitou skupinou jsou zde invalidní uživatelé a uživatelé mobilních zařízení. Stan-

[2] https://nextjs.org/

dardizační organizace World Wide Web Consortium (W3C) vydává směrnice *Web Content Accessibility Guidelines*[3] určené pro vývoj dostupných aplikací.

Velká část podpory dostupnosti spočívá v anotaci obsahu tak, aby byl lépe strojově zpracovatelný. Příkladem může být význam vstupních polí, správné použití sémantických HTML značek, nebo textové alternativy k obrazovým datům. Prezentace by se měla adaptovat na různá rozlišení a orientaci obrazovky. Měla by mít dostatečný kontrast textu a pozadí. Důležitá je také možnost ovládat celou aplikaci pomocí klávesnice.

Accessible Rich Internet Applications (ARIA) je skupina atributů, kterými lze sémanticky anotovat HTML značky. Například atribut `role` u značky vyjadřuje jeho roli v rozhraní a používá se v situacích, ve kterých nelze použít vhodnou sémantickou značku. Role elementu může být strukturní (`tooltip`, `note`), widget (`searchbox`, `slider`) a další.

Kvalitní knihovny s prvky uživatelského rozhraní jsou navrženy v souladu se standardy a mohou zajistit lepší dostupnost bez nutnosti investovat značné množství zdrojů do vyvinutí vlastního řešení.

5.3.3 Měření uživatelského zážitku

Měření uživatelského zážitku ve webových aplikacích je klíčovým prvkem moderního návrhu a vývoje aplikací. Pro vytvoření dobrého rozhraní je nezbytné porozumět tomu, jak uživatelé vnímají a využívají webové aplikace. Uživatelský zážitek (UX – *User Experience*) zahrnuje vizuální dojem, snadnost navigace, efektivitu úkonů a celkovou přívětivost rozhraní. Měření těchto aspektů, ať už manuální, nebo automatické, pomáhá vývojářům identifikovat problémy a zlepšovat užitečnost aplikace.

Kromě designu je důležitý i výkon aplikace, který se projevuje délkou čekání. Uživatelé vnímají vizuální odezvu jako okamžitou, pokud se odehraje do 30 ms. Vnímaná kvalita významně klesá, pokud je odezva pomalejší než 100 ms. [11]

Kvalitativní metody

Vnímání produktu jeho uživatelem je velmi subjektivní. Pokud chceme získat povědomí, je nutné provést průzkum.

Poskytnutí formuláře nebo dotazníku pro zpětnou vazbu je nejjednodušším způsobem sběru dat od uživatelů. Má ale zásadní nevýhodu – uživatel musí věnovat vlastní čas a úsilí k poskytnutí zpětné vazby. Důsledkem je, že jsou hlášeny pouze problémy, které si uživatel uvědomuje, dokáže popsat a jsou pro něj dostatečně důležité. Navíc uživateli musí záležet na tom, aby byl nedostatek opraven.

Uživatelská přívětivost je také měřitelná analýzou sezení. Metoda *Real User Monitoring* (RUM) instrumentuje aplikaci a tvoří záznam akcí v čase. Poznatky o nedostatcích lze získat porovnáním akcí úspěšných a neúspěšných uživatelů.

Kvantitativní metody

Stav aplikace z pohledu výkonu lze měřit značně jednodušeji. *Web Vitals* je soubor metrik vyjadřujících kvalitu uživatelského zážitku převážně týkajících se rychlosti načítání aplikace a odezvě po interakci. Metriky odrážejí architekturu aplikace i infrastrukturu, na které je aplikace nasazena. Následuje popis několika metrik.

[3] https://www.w3.org/WAI/standards-guidelines/wcag/

Largest Contentful Paint (LCP) měří dobu od navigace na stránku do zobrazení největšího prvku UI. Tento čas zahrnuje dobu sestavení spojení se serverem. Za dobrý výsledek se považuje hodnota menší než 2,5 s.

Cumulative Layout Shift (CLS) měří míru neočekávaných posunů prvků rozhraní. Tyto posuny jsou způsobeny postupným načítáním obsahu, typicky obrázků, reklam a jiného dynamického obsahu. Velká nestabilita rozvržení stránky způsobuje frustraci uživatele a stává se, že vyvolá nechtěnou akci.

Interaction to Next Paint (INP) měří responzivitu aplikace jako prodlevu mezi interakcí a prezentací další vyrenderované obrazovky. Za přijatelné zpoždění se považuje 200 ms. Některé akce přirozeně trvají delší dobu, v takovém případě je dobré dát vizuální zpětnou vazbu, aby uživatel nenabyl dojmu, že aplikace přestala odpovídat.

Další měřené metriky zahrnují *Time to First Byte* (TTFB), nebo *First Contentful Paint* (FCP).

Vyhledávače tyto metriky využívají k odhadu kvality webové stránky a její upřednostnění ve výsledcích, což může být důležité pro dosažení cílů organizace. Výsledky měření mají určitou distribuci hodnot podle výkonu zařízení a kvality internetového spojení se serverem. Je doporučeno pracovat se 75. percentilem hodnot metrik. Nekvalitní připojení i slabější hardware je možné v prohlížeči emulovat. [21]

5.4 Architektura systému

V architektuře webových aplikací se klíčově uplatňuje model klient-server, v rámci kterého klient (webový prohlížeč) a server komunikují přes standardizované protokoly. Dva základní přístupy ke komunikaci jsou hypermediální API a datová API.

Hypermediální API je rozhraní, které poskytuje hypermédia. Typicky se jedná o dokumenty nebo části dokumentu v jazyce HTML, obrázky a videa ve formátech přímo podporovaných prohlížeči. Tyto odpovědi jsou přímo zobrazovány klientem. Jedná se o původní přístup poskytování dat v kontextu webových aplikací. Příkladem jsou statické HTTP servery, nebo servery renderující HTML při dotazu (*Server-Side Rendering*).

Oproti tomu datové API poskytuje strukturovaná data, která nejsou určena k přímému zobrazení. Prvotní stažení stránky stále proběhne v podobě HTML, součástí stránky je ale JavaScriptový kód, který dál pracuje s datovým API. Příchozí data jsou zpracovávána do nového stavu DOM, který prohlížeč prezentuje uživateli.

Populárními zástupci knihovnen, kterými lze implementovat jak hypermediální, tak datová API jsou například Django, Express, nebo Ruby on Rails.

5.4.1 Globální stav aplikace

Často řešeným problémem aplikací je komunikace mezi moduly. V JavaScriptových aplikacích se tento problém často řeší globálním stavem. Globální stav je množina dat *dostupná ze všech částí aplikace*. Typicky jsou sdílena data o sezení, konfigurace aplikace a uživatelská data.

Dříve zmíněná knihovna React podporuje práci s globálním stavem ve formě `Context API`. *Kontext* vytvořený v určitém komponentu je zpřístupněn všem jeho potomkům ve stromu. Knihovna *Redux* je dalším způsobem jak v Reactové aplikaci implementovat správu globálního stavu. Stav je zde reprezentován jediným objektem, který je možné libovolně číst. Změny globálního stavu jsou možné výhradně skrze *akce* – čisté funkce transformující daný stav do nového stavu. Redux vybízí k centralizaci logiky aplikace v definici stavu a jeho

akcích. Nabízí také další zajímavé funkce jako perzistenci stavu napříč sezeními, nebo ladící rozšíření do prohlížeče s možností přehrání změn stavu.

5.5 Vývojové praktiky

Při vývoji webových aplikací je výhodné používat moderní vývojové praktiky, které zajišťují efektivní správu kódu, plynulý vývoj, komunikaci a nasazení aplikace do provozu. Cílem je co největší část repetitivní práce *automatizovat*.

Continuous Integration (CI) je koncept automatizace akcí jako například sestavení projektu, testování a nasazení. Automatizace především snižuje pravděpodobnost lidské chyby při nasazení. Součástí procesu je generování hlášení o výsledcích akcí. CI je úzce spjatý se *systémem správy verzí*.

Git[4], nejpopulárnější systém pro správu verzí kódu, hraje klíčovou roli v dokumentování a uchování historie změn v kódu. Hlavním konceptem gitu je *commit*, záznam o stavu kódu. Každý commit je uzlem v grafu, kde hrany vyjadřují následnost commitů. Události v repozitáři mohou být spouštěčem pro automatizované testování a nasazení. [5]

Současným trendem je široce využívat knihoven ke zrychlení a usnadnění vývoje. S tím napomáhá manažer balíčků, který poskytuje jednoduchou správu závislostí projektu. Populární knihovny zároveň bývají robustní, dobře navržené a mají otevřené zdrojové kódy (open source).

5.5.1 Nasazení

Proces nasazování do provozu představuje důležitý okamžik v životním cyklu webové aplikace. Automatizované postupy umožňují rychlé a konzistentní nasazení nových verzí na produkční nebo předprodukční server. To nejen minimalizuje riziko lidských chyb, ale také šetří manuální úkony.

Častým problémem při nasazování aplikace na server bývají její požadavky na prostředí – verze operačního systému, nainstalované programy, proměnné prostředí a další. Systém *Docker*[5] tento problém řeší tak, že celé prostředí popíše předpisem pro jeho vytvoření. [5] Při spuštění *kontejneru* virtualizační vrstvou se tento předpis vykoná v izolovaném prostředí. Výsledkem je předvídatelné nasazení převážně abstrahované od konkrétního systému. Změna serveru na kterém aplikace běží nepředstavuje žádné nebo pouze minimální změny v kódu a konfiguraci.

5.5.2 Testování

Robustní testovací řešení významně usnadňuje vývoj nových funkcí aplikace, stejně jako změny v existující funkcionalitě.

Existují různé druhy testování softwaru. Každé se zaměřuje na určitou vrstvu softwarového produktu:

- Testování na úrovni softwarových modulů (Unit Testing),
- Testování interakce modulů (Integration Testing),
- Holistické testování (End-to-end testing).

[4] https://git-scm.com/
[5] https://www.docker.com/

Testování by mělo být automatizované. Ponechání procházejících testů v projektu zabraňuje regresím. Nevhodný návrh testů může mít za důsledek jejich přílišnou závislost na vnitřní implementaci. Při změnách v implementaci takové testy vyžadují častou opravu, což ztěžuje údržbu.

Webové rozhraní lze testovat mnoha způsoby a má svá specifika. Testy mohou ověřovat, zda se na stránce vyskytuje daný text a provést gramatickou kontrolu. Testy pro dostupnost mohou kontrolovat kontrast textu a anotaci HTML značek. Můžeme aplikovat kvantitativní metody (sekce 5.3.3) a sbírat metriky od skutečných uživatelů.

Testování funkcí webových uživatelských rozhraní je často prováděno manuálně. Rozhraní bývají složitá a jejich automatizace je velmi pracná na implementaci a údržbu. GUI by mělo být testováno ve více prohlížečích.

Kapitola 6

RISC-V simulátor Jana Vávry a Jakuba Horkého

Tato kapitola se bude zabývat analýzou simulátoru, který byl výsledkem diplomových prací, na které navazuji [8, 19]. Poskytne přehled o fungování systému, zaměří se na jednotlivé moduly a funkce simulátoru, pro které budou v následující kapitole navržena zlepšení.

V současném stavu má simulátor podobu desktopové aplikace s grafickým uživatelským rozhraním. Celý systém je implementován v programovacím jazyce *Java*[1] s použitím knihovny JavaFX[2]. Aplikace nemá rozhraní příkazové řádky.

6.1 Architektura systému

Kód simulátoru je organizovaný do modulů. Implementace využívá objektově orientovaného přístupu k programování. Třídy simulace se dělí na datové (modely) a behaviorální (jednotlivé simulované bloky).

Centrální třída simulace `BlockScheduleTask` si udržuje seznam referencí na všechny simulované bloky procesoru. V případě kroku simulace vpřed nebo zpět je posluchačům v definovaném pořadí zaslaná odpovídající zpráva. Ke komunikaci je použit návrhový vzor *Observer*.

V programu existuje globální instance stavu simulace. Komponenty uživatelského rozhraní obdrží reference na potřebné objekty systémem založeném na návrhovém vzoru vkládání závislostí (Dependency Injection).

Každý komponent uživatelského rozhraní má své třídy *view* a *controler*, které definují vzhled a chování podle archtektury MVVM.

6.2 Interpretace instrukcí

Každá instrukce z instrukční sady je popsána několika atributy. Výpis 6.1 uvádí popis jedné z instrukcí. Definuje jméno instrukce, počet operandů a třídu instrukce (aritmetická, paměťová, skoková). Součástí popisu je také výraz, který definuje vztah zdrojových a cílových operandů.

[1] https://www.java.com/en/
[2] https://openjfx.io/

```
1   {
2     "name": "add",
3     "instructionType": "kArithmetic",
4     "inputDataType": "kInt",
5     "outputDataType": "kInt",
6     "instructionSyntax": "add rd rs1 rs2",
7     "interpretableAs": "rd=rs1+rs2;"
8   }
```

Výpis 6.1: Popis instrukce `add`. Položka `interpretableAs` obsahuje matematický popis instrukce.

Tento výraz je ve fázi execute interpretován objektem představujícím příslušnou funkční jednotku. Jedná se o interpret s precedenční analýzou operátorů. Implementace obsahuje precedenční tabulku a operátor přiřazení řeší zvlášť.

Každý druh operace má svůj vlastní interpret (`CodeLoadStoreInterpreter`, `CodeBranchInterpreter`, `CodeArithmeticInterpreter`). Důvodem jsou odlišné požadavky na popis a vykonání těchto instrukcí.

Tento přístup má své výhody i nevýhody. Nespornou výhodou je možnost definovat nové instrukce pouhou úpravou konfiguračních souborů. Jako nevýhody vnímám nižší výkon simulace a vyšší složitost kódu.

Interpret a jeho jazyk není dostatečně silný pro vyjádření některých instrukcí, případně pseudoinstrukcí s implicitními argumenty. Některé informace o instrukcích jako například cíl skoku, nebo cílový registr jsou vyjádřeny implicitně. Implementace nepopisuje všechny instrukce základní instrukční sady, zato ale implementuje část rozšíření M pro násobení a rozšíření F pro čísla s plovoucí desetinnou čárkou.

Syntax assembleru je nestandardní, není kompatibilní s assemblerem generovaným rozšířenými překladači jako GCC. Důsledkem je, že programy vytvořené překladačem nebo nalezené na internetu musí být před spuštěním upraveny. Nejsou podporovány žádné direktivy pro definování globálních dat.

6.3 Konfigurace simulace

V uživatelském rozhraní simulátoru lze konfigurovat velikosti bufferů, chování cache a prediktoru skoků.

Lze také konfigurovat množství a latence jednotlivých funkčních jednotek. Konfigurace ALU spočívá v definování povolených operací jazyka pro popis instrukcí zmíněného v sekci 6.2. Chybí zde konfigurace latence konkrétních operací. Například operace dělení typicky trvá výrazně více taktů, než sčítání.

Simulace má více konfiguračních možností v podobě souborů JSON, které definují instrukce (viz ukázka 6.1 v sekci 6.2) a registry. Tuto konfiguraci není nutné poskytovat uživatelům, protože pro instrukční sadu RISC-V jsou instrukce a registry přesně definovány.

6.4 Zpětná simulace

Značná část složitosti celého systému spočívá v možnosti krokovat simulaci dopředu i zpět v čase. Tato funkcionalita je dosažena tím způsobem, že každý funkční blok implementuje krok v čase o takt dopředu (`simulate`) a inverzní operaci (`simulateBackwards`).

Každá změna stavu musí být reverzibilní, stav procesoru musí držet celou svou historií a proto s délkou simulace velikost stavu značně roste. Simulace například udržuje seznam všech vydaných instrukcí a historii všech paměťových transakcí. Dostatečně dlouhá simulace musí nutně vést k vyčerpání zdrojů.

6.5 Reprezentace registrů

Registry jsou organizovány do skupin podle datového typu (integer a float). V těchto skupinách jsou i zobrazované v uživatelském rozhraní, spekulativní registry zobrazené nejsou.

Číselné hodnoty jako stavy registrů a mezivýpočty při interpretaci jsou reprezentovány výhradně v datovém typu `double`. Tento datový typ neodpovídá chápání registrů v architektuře RISC-V, která s registry pracuje jako s bitovým polem.

Používání `double` pro reprezentaci registrů má několik implikací:

- grafické rozhraní nemá dostatek informaci pro interpretaci obsahu registru a proto ho musí zobrazit jako desetinné číslo, a to i v případech, kdy je význam obsahu jiný (celé číslo, pravdivostní hodnota, ukazatel),
- interpretace celočíselných instrukcí musí provádět přetypování před a po výpočtu,
- přesnost simulace je nedostatečná – tento nedostatek je nejzřetelnější u výsledků bitových operací.

6.6 GUI

Hlavní okno simulátoru (na obrázku 6.1) poskytuje vhled do stavu celého procesoru a prvky k ovládání simulace. Kód, registry a buffery jsou reprezentovány tabulkami. Horní lišta umožňuje spustit celou simulaci, nebo ji postupně krokovat. Čáry mezi jednotkami procesoru znázorňují datové cesty.

Celé znázornění procesoru se nevejde do jednoho okna, k prohlédnutí některých částí je nutné okno posunout posuvníkem. Pro pohled do cache a na statistiky jsou vyhrazena speciální okna přístupná z pravé lišty. V horní části obrazovky se nacházejí tlačítka pro přechod do editoru kódu a nastavení parametrů simulace.

6.6.1 Statistiky

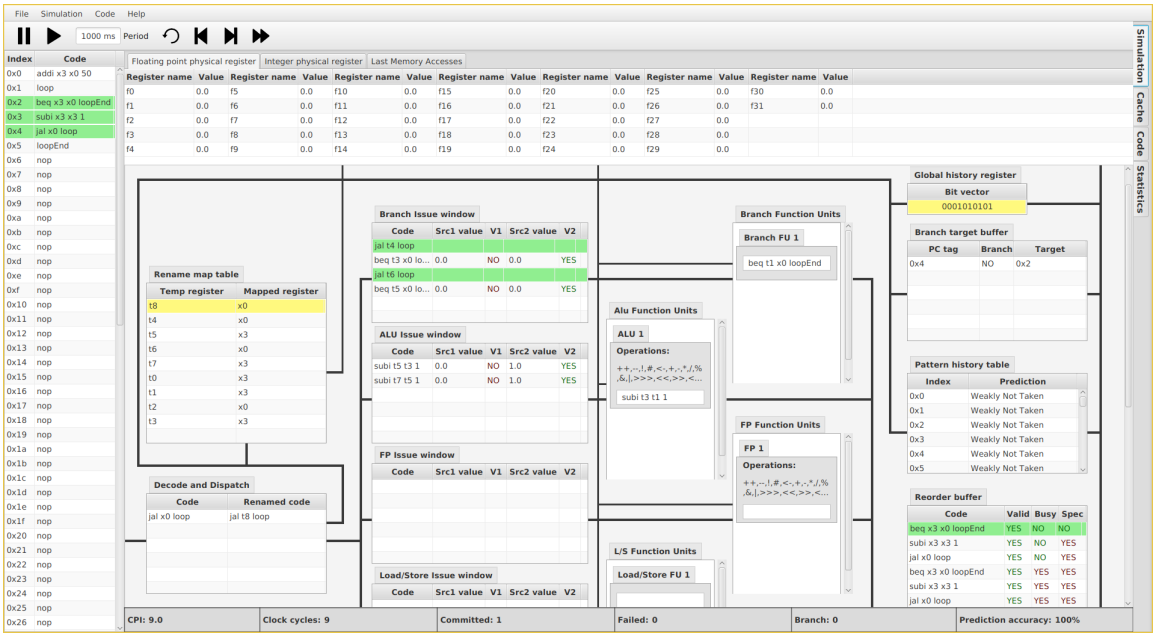
Vybrané statistiky jsou zobrazeny v dolní části hlavního simulačního okna (obrázek 6.1). Detailnější pohled poskytuje záložka *Statistics*, kterou je možné vidět na obrázku 6.2. Zde se nachází více číselných statistik a dva grafy vývoje statistik v čase. Tabulka ve spodní části obrazovky se rozšíří o druhý řádek se statistikami.

Chybí mi zde více statistik, například instrukční mix (poměr typů instrukcí v kódu), nebo vytíženost funkčních jednotek. Prezentace statistik také není příliš názorná.

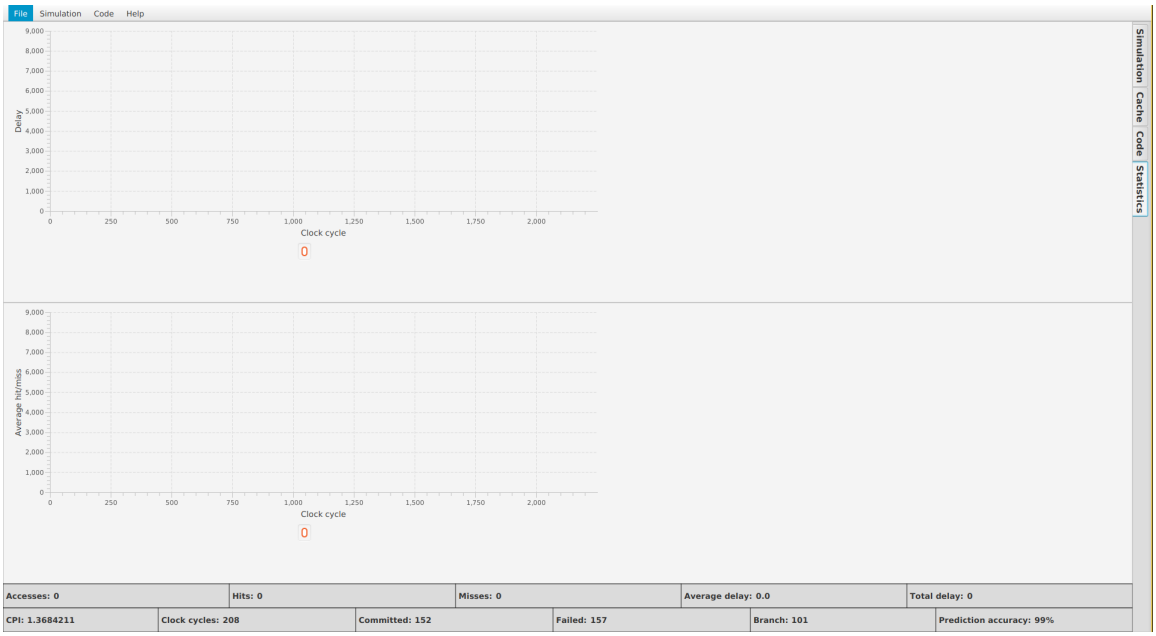
6.7 Editor kódu a kompilátor

Aplikace umožňuje vytvářet vlastní kódy v assembleru RISC-V, nebo v jazyce C, který je do assembleru následně přeložen. Kód lze následně přenést do simulátoru.

Editor (na obrázku 6.3) je praktický. Je možné nahrát jeden z mnoha předpřipravených příkladů kódu, což snižuje úsilí nutné k vyzkoušení simulátoru. Zajímavou vlastností

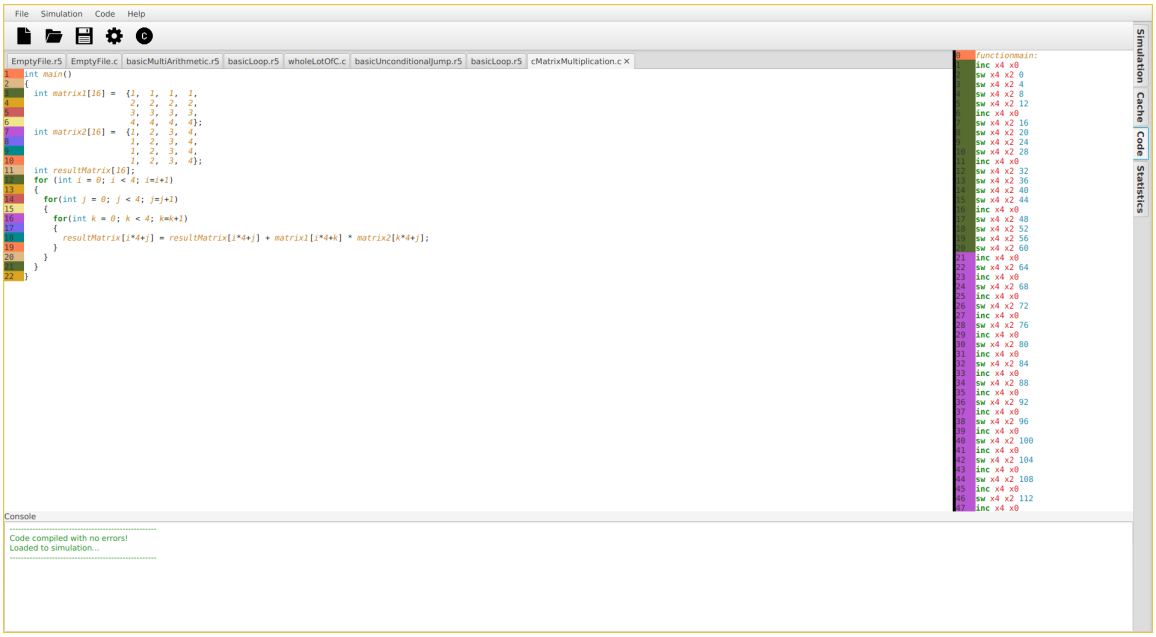


Obrázek 6.1: Hlavní okno původní aplikace v průběhu simulace. [8, 19]



Obrázek 6.2: Okno pro zobrazení statistik simulace. [8, 19]

je vizualizace vztahu mezi kódem v jazyce C a assemblerem v podobě zvýraznění řádků programů.



Obrázek 6.3: Editor kódu zabudovaný v původní aplikaci.

Mít překlad plně pod svou kontrolou je velkou výhodou, zejména pokud simulátor nepodporuje všechny instrukce a direktivy, které běžný překladač může vygenerovat, nebo pokud se syntax odchyluje od standardu. Udržovat vlastní překladač jako součást simulátoru ale považuji za velkou zátěž. Vyladit všechny chyby je příliš těžké, zpětná vazba kompilátoru v podobě chybových hlášení jsou obecná. Absence podpory celého standardu C znamená, že kód stažený z internetu často nemusí fungovat. Uživatel navíc nezná limity překladače, což znesnadňuje práci.

Jediným způsobem jak definovat data je jako jednorozměrné pole v lokální proměnné ve funkci přímo v kódu. Taková definice dat se navíc přeloží jako série instrukcí `store` uvnitř těla funkce, což funkci učiní méně čitelnou a zanese šum do běhových statistik programu.

6.8 Testování

Testování je v projektu na skvělé úrovni. Dobře pokrývá jednotlivé moduly i celkové chování simulátoru, čímž napomáhá snadnějšímu pochopení kódu a ulehčuje refaktorizaci a implementaci nové funkčnosti.

Projekt používá testovací framework JUnit.

Jakub Horký zmiňuje [8], že implementací nových funkcí musel velkou část testů opravovat. Z mých experimentů mám podobné zkušenosti, zejména se jedná o holistické testy, které vynucují přesný stav simulace v každém kroku.

Kapitola 7

Návrh rozšíření simulátoru

V této kapitole se budu zabývat návrhem rozšíření simulátoru na základě analýzy z předchozí kapitoly. Navrhnu změnu architektury na serverovou s webovým klientem. Vylepšení se zaměří na dva aspekty, 1) uživatelské rozhraní a 2) přesnost simulace.

Musí být zachována dobrá rozšiřitelnost a modularita simulátoru. Implementace by měla v případě budoucího rozšíření povolit nenáročné rozšíření o další část instrukční sady, nebo funkční blok jako například vektorová jednotka.

Při návrhu uživatelského rozhraní jsem se inspiroval původním rozhraním simulátoru Jana Vávry a Jakuba Horkého. Programátorská rozšíření stavěla na objektové struktuře původního návrhu.

7.1 Architektura systému

Desktopovou aplikaci s grafickým uživatelským rozhraním přetvořím na HTTP server s bezstavovým aplikačním rozhraním využívající HTTP a rozhraním pro lokální práci v příkazové řádce.

HTTP API bude využívat nová klientská webová aplikace. Ta bude implementovat pouze zobrazení stavu procesoru, editování kódu a konfiguraci simulace. Vytvořením samostatně stojící prezentační vrstvy získává projekt možnost tyto dva systémy nezávisle nasazovat, udržovat a případně znovu implementovat.

Rozhraní příkazové řádky bude sloužit k automatizovanému spouštění simulací.

7.1.1 Webové technologie

Poskytování uživatelských rozhraní pomocí webového prohlížeče má značné výhody. Uživatel nemusí plnit složité požadavky na hardwarovou a softwarovou výbavu nutnou k instalaci a spuštění softwaru. Většina uživatelů již má počítač s moderním webovým prohlížečem a internetové připojení.

Vzhledem k původní implementaci v jazyce Java se nabízí možnost implementovat renderování webového uživatelského rozhraní jako modul. Java k takovým účelům dokonce poskytuje rámcová řešení. Zachování monolitické povahy simulátoru by bylo výhodou, aplikace ale kvůli požadavkům na názornost a interaktivitu bude vyžadovat značné množství logiky na straně klienta.

Ekosystém pro návrh a implementaci front-endových aplikací pomocí JavaScriptu je značně vyspělý. Z těchto důvodů jsem se rozhodl klientskou aplikaci vyvinout s knihovnou

React a frameworkem NextJS. Při vývoji budu moci využít existující ekosystém pokročilých nástrojů a mnoha knihoven.

7.1.2 Aplikační rozhraní

Simulátor a webová aplikace budou komunikovat pomocí protokolu HTTP aplikačním rozhraním založeným na formátu JSON.

Stav navracený ze simulátoru by měl být *normalizovaný*. Simulátor jakožto objektově orientovaný program pracuje s grafem navzájem se odkazujících objektů. Serializace takového grafu do hloubky způsobí zacyklení. Navíc, nenormalizovaná data znamenají redundanci v podobě kopií. Normalizace datových objektů proběhne nahrazením referencí za identifikátory v době serializace.

Možným rizikem je latence serveru při interaktivní simulaci. Při vývoji a hodnocení budu tuto metriku sledovat a v případě nevyhovujících parametrů implementuji opatření.

7.2 Jednotlivá vylepšení simulátoru

Kromě dále zmíněných konkrétních vylepšení obecně zvýším kvality kódu pomocí refaktorování a dokumentace. Cílem je mít dobře čitelný, výkonnější a udržitelnější kód.

7.2.1 Reprezentace hodnot registrů

V sekci 6.5 jsem uvedl omezení současného systému, který hodnoty registrů reprezentuje v datovém typu `double`.

Navrhuji stav registru reprezentovat jako bitové pole. Interpretace hodnoty bude záviset na právě prováděné instrukci. Tato reprezentace dovolí přesnou simulaci všech instrukcí, včetně bitových operací.

Bitové pole bude široké 64 bitů, aby bylo připravené pro případné přidání 64 bitové instrukční sady. V případě implementace vektorových registrů bude nutná malá úprava

Spolu s bitovým polem bude v objektu registru uložena metainformace o významu obsahu, tedy o datovém typu registru. Zdrojem této informace bude popis instrukce, která hodnotu vytvořila. Tato informace bude soužit pouze k účelům zobrazování hodnoty v GUI a při ladění, při simulaci nebude mít význam.

7.2.2 Interpretace instrukcí

Změny v interpretaci instrukcí úzce souvisí se změnou reprezentace hodnot registrů. Plánuji podporovat celou instrukční sadu RISC-V včetně pseudoinstrukcí. Tento cíl vyžaduje jisté další úpravy.

Navrhuji sjednotit interprety do jedné implementace, která bude dostatečná pro všechny případy užití.

Precedenční interpret nahradím interpretem výrazů v postfixové notaci. Jeho implementace je jednodušší, výrazy nepotřebují závorky a jeho vyjadřovací síla je dostatečná. Navíc nebude potřeba zvláštního parsování pro operaci přiřazení.

Výpočet některých instrukcí pracuje s hodnotami, které nepatří mezi operandy dané instrukce. Typickým příkladem je použití hodnoty PC při výpočtu adresy skoku. Dalším příkladem je instrukce `jal`, která načítá hodnotu PC do registru. Další kategorií jsou pseudoinstrukce, které často mají implicitní argumenty. Jako příklad uvedu pseudoinstrukci `ret`, která odpovídá instrukci `jalr x0, x1, 0`.

Funkční popis takových instrukcí vyřeším zavedením konceptu proměnných do interpretace. Popis instrukce bude obsahovat všechny argumenty a jejich datové typy. Tyto datové typy budou řídit interpretaci bitů uložených v daných registrech. Příklad navrhovaného popisu instrukce naleznete na obrázku 7.1.

```
{
 "name": "add",
 "instructionType": "kArithmetic",
 "arguments": [
   {
     "name": "rd",
     "type": "kInt",
     "writeBack": true
   },
   {
     "name": "rs1",
     "type": "kInt"
   },
   {
     "name": "rs2",
     "type": "kInt"
   }
 ],
 "interpretableAs": "\\rs1 \\rs2 + \\rd ="
},
```

Výpis 7.1: Nový popis instrukce `add` detailně popisuje argumenty a jejich datové typy.

7.2.3 Rozhraní simulátoru, reprezentace stavu

Bude odstraněno grafické uživatelské rozhraní, aplikace bude komunikovat bezstavově schématem požadavek/odpověď. Požadavek bude moct být podán prostřednictvím příkazové řádky (CLI), nebo HTTP dotazu.

Hlavním požadavkem bude dotaz na stav simulace v určitém kroku pro určitou konfiguraci procesoru. Výstupem bude stav procesoru, statistiky o běhu a ladící výstupy.

Další možné požadavky na simulátor budou:

- překlad programu z jazyka C do assembly,
- kontrola správnosti programu v assembly,
- kontrola správnosti konfigurace CPU pro simulaci.

Simulace spouštěná z příkazové řádky přijme jako argument konfiguraci procesoru, včetně simulovaného programu. Výstupem budou především statistiky o dokončeném běhu. Rozhraní příkazové řádky je určeno primárně pro hromadné vyhodnocování, nebude umožňovat interaktivní simulování, ani překlad programů v jazyce C.

HTTP API bude očekávat `POST` dotazy s parametry předávanými v těle zprávy v jazyce JSON. Odpovědi budou také v jazyce JSON. Jazyk JSON jsem zvolil kvůli dobré podpoře jeho zpracování ve webových aplikacích. Stav procesoru může být objemný, proto plánuji podporovat kódování ZIP, které významně sníží množství dat přenášené po síti.

Dalším problémem je serializace stavu procesoru. Stav má strukturu obecného grafu objektů s mnoha cykly, které JSON není schopen nativně vyjádřit. Řešením bude stav normalizovat, převést ho z obecného grafu do stromu a chybějící hrany vyjádřit implicitně identifikátory.

V současné podobě aplikace nemůže existovat více instancí procesoru. Stav procesoru bude muset být přesunut do rodičovského objektu, který bude obsahovat reference na všechny své komponenty.

7.2.4 Zpětná simulace

Aby bylo možné realizovat zpětnou simulaci bezstavovými dotazy podle současné implementace, musel by být stav procesoru přenášen spolu s dotazem. Na tento stav by se pak aplikoval krok dopředu nebo zpět. Logika zpětné simulace je zároveň velmi složitá, výrazně zvětšuje velikost stavu a obsahuje těžko odhalitelné chyby.

Navrhuji zcela jiný přístup. Simulace bude probíhat *pouze* ve směru dopředu v čase. Dotaz bude obsahovat počáteční konfiguraci (kód a vlastnosti procesoru) a požadovaný takt t. Simulátor provede t kroků z času 0 do času t a stav simulace vrátí.

Interaktivní simulace funguje na principu dotazu na stav $t-1$ nebo $t+1$. Jinými slovy zpětná simulace může být realizována dopřednou simulací z výchozího stavu. Dotaz tedy nemusí přenášet stav simulace, ale pouze počáteční konfiguraci. To je výhodné z pohledu množství přenášených dat po síti.

Pro lepší ilustraci uvedu konkrétní příklad. Předpokládejme situaci, kdy uživatel provozuje interaktivní simulaci a nachází se na 20. taktu. Pokud uživatel požádá o krok zpět (stav v taktu 19), spustí se simulace z výchozího stavu (stav v taktu 0) a provede se 19 kroků simulace vpřed.

Předpokladem je, že je simulace deterministická. Rizikem tohoto přístupu je latence při interaktivní simulaci – délka výpočtu následujícího stavu bude růst s aktuální pozicí simulace v čase, protože celá simulace musí proběhnout znovu. V případě, že odpověď serveru nebude spolehlivě dosahovat interaktivní rychlosti, je možné implementovat cache a stavy spekulativně předpočítávat.

7.2.5 Přesnost simulace

Mým cílem je implementovat všechny instrukce základní instrukční sady RV32I a rozšíření M a F. Výjimkou budou instrukce pro komunikaci s jádrem, protože není implementovaný privilegovaný režim ani přepínání kontextu. Implementuji i velkou část pseudoinstrukcí.

Každá z instrukcí bude přesně odpovídat specifikaci. Instrukce programu ale budou i nadále vnitřně reprezentovány polem objektů, nebudou zavedeny a čteny z paměti.

Skokové a paměťové operace pracují s adresami návěstí. V současné implementaci ale hodnota návěstí představuje index do pole instrukcí. Návěstím je nutné přiřadit reálné adresy, aby hodnoty odpovídaly očekávání překladače a aby mohly návěstí ukazovat na staticky alokovaná pole a konstanty (více v sekci 7.2.10).

7.2.6 Konfigurace simulace

Stávající konfigurace je převážně vyhovující. Hlavní změnou konfigurace jádra bude zjednodušení konfigurace ALU. Místo seznamu operací si uživatel bude moci vybrat jednu nebo více z pěti funkcionalit: sčítací operace, bitové operace, násobení, dělení a speciální funkce.

V GUI přidám detailní popisy ke každé možnosti konfigurace, aby bylo zřejmé jaký efekt má na simulaci. Přibude i validace vstupů s dobrou zpětnou vazbou pro uživatele.

Novou funkcí bude definice paměťových míst přímo v konfiguraci. Je užitečné sledovat běh algoritmů na určitých datech, například některá chování cache se projeví až při práci s větším množstvím (kilobajty) dat Navrhuji přidat možnost pohodlně definovat větší da-

tové vstupy položkami v aplikačním dotazu (tedy jinou cestou, než direktivami v kódu). Paměťové místo bude definované svým názvem, datovým typem, požadavkem na zarovnání v paměti a seznamem číselných hodnot.

7.2.7 Kompilace programů v jazyce C

Překladače jsou jedním z nejkomplexnějších problémů oblasti informatiky. Proto by bylo vhodné použít hotové řešení.

Vybral jsem si překladač *GCC*[1], jeden z nejpoužívanějších překladačů pro jazyk C. Poskytuje užitečná chybová hlášení, je dobře otestovaný, rychlý a implementuje veškeré potřebné funkce jazyka C.

GCC má mnoho funkcí a obsáhlou dokumentaci. Plánuji uživatelům zpřístupnit ovládání optimalizací. Bude také možné si vyzkoušet efekt direktiv jako `#pragma unroll`.

Tento překladač podporuje *cross-compilation*, kompilaci pro jinou architekturu, než ta, na kterém je překladač spouštěn.

Překladač bude front-endové aplikaci zpřístupněn jako součást simulačního API. Serverová aplikace bude binární program GCC spouštět přes shell se zvolenými přepínači. Řetězec s programem bude poslán na jeho standardní vstup, z výstupu bude přečten assembler pro RISC-V. Chybová hlášení budou předána zpět uživateli webového editoru.

Překladač generuje jiný kód, než jaký v současnosti překladač očekává. Změny, které musím implementovat zahrnují:

- změnu syntaxe (viz sekce 7.2.8),
- přidání podpory pro aliasy registrů,
- implementaci všech instrukcí, které překladač může generovat,
- alokaci zásobníku volání (viz sekce 7.2.10)
- zastavení simulace při vyprázdnění zásobníku volání,
- filtrování ladících symbolů a nepoužitých direktiv.

7.2.8 Syntax programů v asembleru RISC-V

Protože plánuji používat klasický překladač, budu muset simulátor upravit tak, aby přijímal tradiční assembler RISC-V. To znamená především:

- přidání oddělovačů mezi parametry (čárky a závorky),
- parsování direktiv (například `.word`),
- implicitní paramtery (pro pseudoinstrukce).

Ladící výstupy budou realizovány komentáři, takže neovlivní syntax assembleru.

[1] https://gcc.gnu.org/

7.2.9 Sběr statistik o běhu

Sběr statistik bude detailnější. Zaměřím se jak na globální statistiky, tak i statistiky jednotlivých instrukcí.

Konkrétní návrhy nových statistik:

- statický a dynamický instrukční mix,
- vytíženost každé funkční jednotky,
- počty provedení každé instrukce,
- nové charakteristiky (FLOPS, aritmetická intenzita).

Se statistikami tečně souvisí sběr ladících zpráv. Při vytváření programu bude možné k libovolné instrukci přidat komentář ve speciálním formátu se zprávou. Při potvrzení instrukce s ladící zprávou bude zpráva přidána do logu. Zpráva umožní vložení aktuální hodnoty registrů. Bude také zaznamenána časová známka zprávy, která bude prezentována vedle zprávy v uživatelském rozhranní.

7.2.10 Zavádění dat do paměti procesu

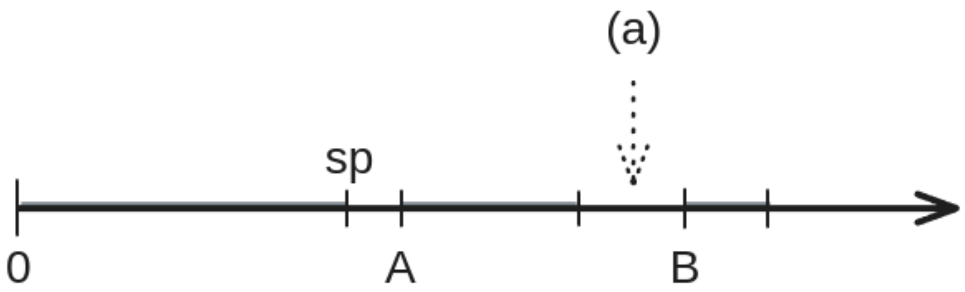
Data definovaná v konfiguraci i v assembleru direktivami[2] musí být staticky alokována v hlavní paměti procesoru. Alokace proběhne podle požadavků na datový typ a *zarovnání*.

Program v jazyce C bude přeložen do direktiv. Tímto způsobem bude možné alokovat globální proměnné, včetně struktur a řetězců definovaných v jazyce C.

Kód s pamětí následně bude pracovat pomocí návěstí (labels), která do této doby byla používaná pouze pro adresy skoků. Hodnoty návěstí budou představovat ukazatele na tato pole.

ABI používané překladačem počítá s alokovaným zásobníkem volání. Proto inicializace paměti musí vyhradit místo pro zásobník a zapsat ukazatel do registru `x2` (neboli `sp`).

Příklad rozložení zásobníku volání a dvou polí se zarovnáním lze vidět na obrázku 7.1. Počáteční adresy jsou vyhrazeny pro zásobník volání, poté jsou alokována jednotlivá pole. Prázdné místo mezi bloky (zvýrazněno jako *(a)* může vznikat požadavkem na zarovnání počátku pole v paměti.



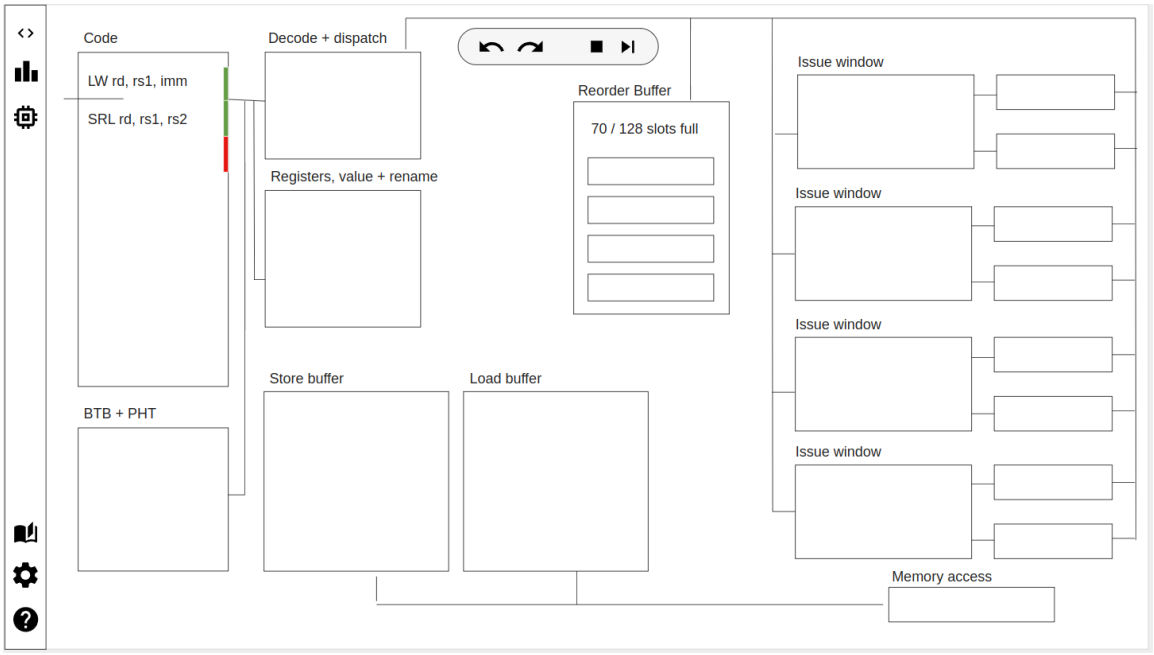
Obrázek 7.1: Schéma rozložení dat v paměti procesoru.

[2] Jako příklad, `A: .word 1,2,3,4` definuje pole `A` o čtyřech 32 bitových prvcích.

7.3 Návrh webové aplikace

Webová aplikace bude víceoknová (*Multi Page Application*) s rozhraním v anglickém jazyce. Vícejazyčná podpora vyžaduje mnoho práce, i za použití internacionalizačních knihoven jako `i18next`. Cílová skupina aplikace je globální a angličtina dovolí projekt používat v mezinárodním prostředí. Studenti informatiky často angličtinu ovládají na dobré úrovni. Pro studium vysokoškolského kurzu AVS je angličtina předpokladem.

Obrázek 7.2 schématicky naznačuje možný vzhled hlavního simulačního okna. Myšlenka současného rozhraní se nemění – jednotlivé bloky procesoru jsou reprezentovány odpovídajícími komponenty, související bloky jsou spojeny čarami, které představují datové cesty procesoru.



Obrázek 7.2: Schéma grafické reprezentace stavu simulace.

Bloky simulace i jednotlivé instrukce by měly mít možnost detailnějšího náhledu. Ten by kromě detailních dat měl poskytnout odkaz na dokumentaci dané instrukce nebo bloku.

Další část, která se příliš nezmění, bude ovládání simulace. V horní části obrazovky budou k dispozici tlačítka pro krokování a dokončení simulace. Pro větší komfort a dostupnost bude možné simulaci ovládat klávesnicí.

Nebude nutné uchovávat žádné uživatelské informace, nebude vyžadována ani autentizace přihlášením. Tím odpadá nutnost jakékoliv centrální databáze a řešení souladu s regulacemi. Konfigurace aplikace bude uložena v lokálním úložišti prohlížeče.

7.3.1 Editor kódu

Záložka pro editování kódu bude umožňovat vytváření a upravování kódu v jazyce C a v assembleru RISC-V.

Cílem je při psaní kódu poskytnout dobrou zpětnou vazbu. Editor bude využívat API simulátoru pro překlad a kontrolu kódu z jazyka C do assembleru. Případné chyby budou

v textovém poli znázorněny. Vztah částí kódu původního a přeloženého programu bude graficky vizualizován.

K dispozici bude několik příkladů kódů, jak pro jazyk C tak Assembler. Příklady umožní novým uživatelům rychle začít se simulátorem experimentovat.

Vytvoření dat pro simulaci bude probíhat na jiné stránce, ale uživateli by měly být poskytnuty potřebné informace v samotném editoru.

7.3.2 Výukové materiály

Cílovými uživateli aplikace budou studenti informatiky, především kurzu AVS. Aplikace nebude předpokládat znalost instrukční sady jako prerekvizitu, proto bude poskytovat základní informace o instrukční sadě RISC-V v podobě vysvětlujícího textu a tabulek. Popis může používat odborné termíny a může se opřít o obecné znalosti z oblasti programování a jazyka C.

7.3.3 Konfigurační stránka

Konfigurace obsahuje mnoho položek, proto její vytvoření může zabrat čas a bránit od plynulého používání aplikace. Aplikace by proto měla poskytnout rozumný výchozí profil, který by byl vyhovující pro širokou škálu experimentů.

Z vlastní zkušenosti při experimentech s existující konfigurací simulátoru vím, že může být obtížné představit si pod názvem konfiguračního pole jeho efekt na simulaci. Budu klást důraz na kvalitu jmen a popisů jednotlivých polí formulářů.

Všechny vytvořené konfigurace budou uloženy lokálně v úložišti prohlížeče a budou perzistovány napříč sezeními. Tímto návrhem se vyhnu nutnosti spravovat uživatele v databázi. Alternativním přístupem by mohlo být použití autentizace a uložení dat aplikace externí službou. Lokální úložiště je však dostatečnou a nekomplikovanou variantou.

Definice dat

Pro definici simulačních dat (paměti) bude vyhrazena zvláštní sekce konfigurace. Uživatel bude moci definovat libovolný počet pojmenovaných paměťových lokací, na které se tímto jménem bude odkazovat v kódu.

Počet prvků paměťového místa bude konfigurovatelný. Hodnoty jednotlivých prvků budou inicializovány jedním ze tří způsobů:

1. kopírováním konstanty,
2. náhodnými daty,
3. daty ze souboru CSV.

7.3.4 Prezentace statistik

V průběhu simulace bude možné nahlédnout na stránku se statistikami. Zde budou prezentovány běhové statistiky popsané v sekci 7.2.9. Stránka poskytne přesná tabulková data i jejich vizualizaci, například teplotní mapou a grafy.

Statistiky týkající se konkrétních bloků budou také přístupné z jejich detailního pohledu ve „vyskakovacím okně“.

7.4 Případy užití a kritéria příjmutí

Je plánováno aplikaci využít pro výuku předmětu AVS na FIT VUT v Brně. Aplikace bude proto mít potenciálně stovky uživatelů. Studenty kurzu AVS bych řadil mezi zkušené uživatele webových aplikací, nicméně bude klíčové klást důraz na intuitivnost, spolehlivost, dostupnost a výkon.

Spolehlivost aplikace ověřím dobrým automatizovaným testováním na úrovni modulů i celku. Uživatelské rozhraní budu testovat převážně manuálně, ve dvou fázích. Nejdříve budu testovat aplikaci v průběhu vývoje. Ověřím funkčnost na několika populárních internetových prohlížečích.

Později aplikaci nechám vyzkoušet několika studentům informatiky. Při testování budu sledovat jejich práci a sbírat zpětnou vazbu. Zpětnou vazbu využiji ke zlepšení aplikace. Dalším zdrojem zpětné vazby je vedoucí mé práce, pan docent Jaroš, se kterým vývoj aplikace pravidelně konzultuji.

Z pohledu implementace je cílem co nejvyšší bezúdržbovost s dostatečnou dokumentací pro případné drobné opravy. Kód také může někdo v budoucnu (stejně jako já nyní) rozšiřovat. Klíčová bude především kvalita podpůrných dokumentů a řádně okomentovaný kód s výstižnými jmény.

Zvážen bude i výkon aplikace. Cílem je interaktivní simulace, proto by server měl na dotazy odpovídat do 100 ms. Zároveň by mělo být možné obsluhovat stovky uživatelů současně. Splnění tohoto cíle ověřím zátěžovými testy.

Program bude pravděpodobně promítán na plátně během výuky. Proto budu brát ohled na kontrast, možnost libovolného zvětšení prezentace simulace a na její správné škálování.

Kapitola 8

Implementace rozšíření simulátoru

Tato kapitola se zaměřuje na konkrétní postup implementace úprav a rozšíření simulátoru, které byly popsány v sekci 7.2. Podrobněji bude popsána hlavní simulační smyčka, zpracování assembleru a implementace serverového rozhraní.

8.1 Refaktorizace

Refaktorizace kódu byla prvním implementačním úkolem. Probíhala zároveň s analýzou a návrhem a sloužila jako příprava k pozdějšímu přidávání nových funkcí. Úpravy v této fázi byly menšího rozsahu, především se jednalo o lokální změny v rámci funkcí a přidávání komentářů. V druhé fázi přišly na řadu větší celky, včetně jejich rozhraní. Důvodem k zásahu bylo buď zvýšení čitelnosti, nové požadavky na modul, nebo potřeba změnit reprezentaci pro snadnou serializaci (a následné zobrazení na webu).

Určit hranici mezi refaktorováním a novou implementací není jednoduché. Většina modulů se změnila zásadním způsobem, včetně způsobu interakce mezi moduly. Příloha A uvádí tabulku s přehledem jednotlivých modulů aplikace a můj přínos.

Třída `SimCodeModel` představující instrukci v systému obsáhla stavové informace z ROB, nově také nese informace o vyvolané výjimce a lépe strukturované informace o skoku a své pozici v kódu.

Paměťový modul (hlavní paměť a cache) zaznamenal významné změny, ale princip fungování zůstal stejný. Tyto bloky nyní pracují v *transakčním režimu*. Funkční bloky, které požadují data z paměti generují nový objekt představující transakci. Při registraci správa paměti tento objekt vyplní údajem o délce vybavení transakce. Transakce umožňují lehkou konfiguraci doby vybavení z paměti, jsou kompatibilní s vypláchnutím linky a obsahují metadata, která jsou zajímavá při interaktivní simulaci.

Není možné zmínit každé vylepšení. Jako příklad menších úprav uvedu změnu vyjádření prediktorů skoků. Původní návrh využíval dědičnosti tříd. Implementace byla zredukována na jedinou třídu `BitPredictor`, kterou je možné parametrizovat. Výsledkem bylo zjednodušení kódu (4 původní třídy byly sloučeny do jedné univerzální) a lepší kompatibilita se serializací do JSONu.

8.1.1 Registry

Problémem jak z výkonnostního tak z návrhového hlediska byla práce s registry. Instrukce držely reference na registry jako řetězce s jejich názvy (např. `„x5"`, `„tg24"`). Při *každém* přístupu k hodnotě registru bylo nutné provést lineární vyhledávání nad celým registrovým

polem, tedy nad stovkami registrů a pro každý prvek muselo proběhnout porovnání s hledaným řetězcem. Metadata registru byla navíc udržována v jiné datové struktuře, takže bylo nutné provést druhé vyhledání. Také bylo nutné tyto struktury udržovat synchronizované.

Reference v podobě řetězců jsem nahradil opravdovými *referencemi* na objekt. Nyní přístup k registru odpovídá následování ukazatele, což je mnohem efektivnější operace. Registr je vyhledáván pouze jednou při inicializaci programu a jednou při přejmenování instrukce.

Nový návrh objektu `RegisterModel` obsahuje svůj datový typ a stav alokace. Hodnota již není reprezentována datovým typem `double`, ale novým objektem třídy `RegisterDataContainer`, která dokáže reprezentovat jak celá čísla, tak i čísla ve formátu plovoucí desetinné čárky. Objekt zapouzdřuje pole 64 bitů v podobě proměnné typu `long`, ke kterému nabízí jak přímý přístup, tak interpretovaný přístup.

Registr také obsahuje všechny informace nutné k přejmenování. Všechny registry mají počet referencí, architekturní registry využívají seznam přejmenování a spekulativní registry obsahují odkaz na odpovídající architekturní registr.

Díky detailním metadatům je možné registry v interaktivní simulaci detailně zkoumat.

Další souvislostí je, že díky odkazovatelnosti na objekty `RegisterDataContainer` nemusí být hodnoty kopírovány mezi buffery (například v blocích `issue` a `load buffer`).

8.2 Simulace

Významnou změnou je nový způsob zpětné simulace navržené v sekci 7.2.4. Bloky nyní neobsahují logiku pro zpětnou simulaci ani datové struktury, které byly pro zpětný chod nutné. Kód se stal významně jednodušším, protože operace nyní nemusí být reverzibilní.

Původní implementace simulační logiky používala návrhové vzory jako *Singleton* a *Dependency Injection*, které komplikovaly vytváření instancí procesoru. V aplikaci navíc nemohlo souběžně existovat více instancí procesoru, což znemožnilo implementaci obsluhy dotazů na server. Tento přístup také komplikoval testování procesoru jako celku.

Zapouzdřil jsem proto všechny bloky procesoru a jejich inicializaci do třídy `Cpu`. Rozhraní pro vykonání simulace se stalo významně jednodušším. Příklad provedení simulace je uveden v kódu 8.1. Po volání `execute` je objekt `cpu` ve stavu na konci simulace. Tento stav je možné zkoumat v rámci testu, nebo prezentovat uživateli.

```
1 SimulationConfig config = SimulationConfig.getDefaultConfiguration();
2 config.code = "addi x8, x8, 5";
3 Cpu cpu = new Cpu(config);
4 cpu.execute(false);
```

Výpis 8.1: Příklad spuštění simulace s výchozí konfigurací a vlastním kódem.

Funkce `execute` je funkcí pro pohodlí při testování. Pro potřeby interaktivní simulace existuje metoda `simulateState(targetTick)`, která ve smyčce volá `step`. Detailnější popis rozhraní třídy Cpu je uveden v sekci 8.2.1.

Simulace je deterministická. Determinismus je v místech, kde se pracuje s náhodou, docílen inicializací pseudonáhodného generátoru do stejného výchozího stavu.

8.2.1 Inicializace a Hlavní smyčka simulace

Inicializace objektu procesoru sestává z následujících kroků:

1. validace konfigurace,

2. načtení popisu instrukcí a registrů,
3. inicializace podpůrných objektů (statistiky a manažery),
4. parsování assembleru,
5. inicializace paměti (data, zásobník),
6. vytvoření bloků procesoru a jejich vzájemných referencí,
7. inicializace hodnot registrů,
8. nastavení PC na počátek programu.

Konfigurace procesoru probíhá při fázi inicializace. Jedná se o velikosti pamětí, zpoždění bloků, prováděný program a data. Kompletní výčet se nachází v příloze D.

Dalším krokem je získání popisů instrukcí a nové kopie registrového pole. Tato data jsou načtena ze souborů při startu aplikace a cachována v paměti.

Inicializace paměti na začátku paměťového prostoru vyhrazuje konfigurovatelné množství paměti pro zásobník volání. Do vyšších adres se poté kopírují data definovaných objektů (polí, konstant a struktur). Alokace jsou zarovnané podle požadavků.

Vstupním bodem programu může být podle konfigurace libovolné návěstí nebo adresa. Na požadovanou hodnotu je nastaven registr PC bloku Fetch. Samotná simulace spočívá v opakovaném volání funkce `step`, která představuje jeden takt procesoru. Simulace probíhá buď do konce programu pro CLI, nebo do požadovaného taktu v případě interaktivní simulace.

Jeden takt je simulován sekvenčním voláním všech bloků procesoru. Pořadí bloků bylo oproti původnímu řešení změněno. Důvodem byly změny ve vydávání instrukcí do funkčních jednotek a změna paměťového systému. V případě funkčních jednotek byla simulace kroku rozdělena do dvou funkcí volaných v různý čas v rámci taktu, aby bylo možné nasimulovat dokončení jedné instrukce a načtení nové v rámci jednoho taktu.

Ve funkci `simStatus` probíhá při každém cyklu kontrola, jestli má simulace pokračovat, nebo se ukončit. Simulace může být ukončena potvrzením výjimky, limitní podmínkou proti zacyklení, nebo řádným ukončením programu.

Simulátor podporuje dva režimy vykonávání programů. V prvním případě je simulace ukončena jakmile je linka prázdná a PC je za poslední instrukcí programu. Tento režim je vhodný pro jednoduché ukázky bez zásobníku volání.

Druhým režimem je režim se *zásobníkem volání*. Registr `sp` je inicializován na adresu vrcholu zásobníku a registr `ra` je inicializován na speciální adresu. Jakmile ROB potvrdí skok na tuto adresu, vysílá signál a simulace je ukončena. Tento mechanismus zastupuje operační systém a runtime, který by typicky vstupní funkci obalil kódem pro ukončení procesu.

8.2.2 Bloky procesoru

Každý blok je inicializován konstruktorem s relevantními parametry konfigurace a referencemi na ostatní bloky.

Bloky mezi sebou komunikují přímými referencemi. Atributy jsou z velké části privátní, kód ale obsahuje velké množství metod, které tyto atributy přímo nastavují. Tím je narušeno zapouzdření. Byla snaha zredukovat počet referencí, aby se zjednodušil model komunikace a bylo jednodušší o systému uvažovat.

Rozhraní mezi řídící třídou `Cpu` a každým blokem je funkce `simulate(int cycle)`. Výjimku tvoří funkční jednotky, jejichž logika je rozdělena do dvou fází – dokončení a začátek vykonávání instrukce. Rozdělení řeší problém opuštění a vstupu instrukcí do funkční jednotky během stejného taktu. V rámci budoucí práce navrhuji prozkoumat rozdělení simulace více bloků do několika fází, což by mohlo zjednodušit logiku.

Číslo cyklu je nyní předáváno při volání simulate. Nové rozhraní preferuji před původním řešením, kde si každý blok udržoval vlastní čítač.

Funkční jednotky zůstaly neřetězené. Implementace zřetězení by mohla být předmětem budoucí práce, protože zřetězené linky odpovídají reálným implementacím funkčních jednotek.

8.2.3 Běhové statistiky

Stav simulace obsahuje centrální objekt průběžně shromažďující statistiky. Jednotlivé statistiky se vztahují buď k celému procesoru, nebo ke konkrétní instrukci. Celý výčet statistik je k dispozici v příloze B.

Na základě statistických dat je možné vyhodnotit úspěšnost spekulativního počítání. Je k dispozici počet vypláchnutí linky a úspěšnost predikce skoků. Kromě absolutních čísel jsou spočítány i některé metriky (IPC, FLOPS, cache hit rate).

Velká část statistik sleduje paměťový systém. Sledují se počty přístupů do paměti, počet zásahů do cache (*cache hits*, viz sekce 2.2) a množství čtených a zapsaných dat. Přístupy a úspěšnost vyhledání v cache se evidují i pro jednotlivé instrukce.

8.3 Instrukce a jejich interpretace

Objektový návrh instrukcí založený na dědičnosti a lineárním vyhledávání v seznamu instrukcí jsem nahradil *kompozicí*. Instance instrukce v procesoru se přímo odkazuje na instrukci v programu, která se zase přímo odkazuje na obecný popis konkrétní instrukce.

Původní popis instrukce uvedený v sekci 6.2 jsem rozšířil podle návrhu v sekci 7.2.2, aby byl více explicitní a dokázal vyjádřit význam všech implementovaných instrukcí. Implicitní argumenty instrukcí jako `call` a `ret` jsou nyní podporovány.

Vykonání výrazu instrukce probíhá v nové třídě `Expression`, která implementuje jednoduchý interpret založený na zásobníku. Několik ukázek je uvedeno v tabulce 8.1. Proměnná značená zpětným lomítkem přidá na zásobník hodnotu, operand vybere operandy ze zásobníku a výsledek vloží zpět na zásobník. Instrukce `jal` ukazuje možnost použití `pc` při interpretaci. Dvojtečka rozděluje podvýrazy u paměťových a skokových instrukcí. U paměťových operací je prvním podvýrazem akce (load/store), druhým podvýrazem je adresa. U skokových operací je prvním podvýrazem adresa skoku a druhým podvýrazem je podmínka skoku.

Výstup z výrazu je dvojí. Prvním výstupem je hodnota, která po provedení interpretace zůstane v zásobníku. Tohoto mechanismu využívají výrazy pro výpočet adresy skoku nebo podmínky skoku. Druhým výstupem je přiřazení do proměnné ve výrazu. Binární operátor `=` ve výrazu má vedlejší efekt, který hodnotu propíše i do registru.

Výjimky jsou generovány při vykonání kódu (přístup na nepovolenou adresu, dělení nulou). Existence výjimky je kontrolována při potvrzení instrukce, v souladu s návrhem procesoru (viz sekce 2.1.1).

Instrukce	Výraz	Význam
`add rd, rs1, rs2`	`\rs1 \rs2 + \rd =`	Sčítání
`fmin.s rd, rs1, rs2`	`\rs1 \rs2 \rs1 \rs2 > pick \rd =`	Přiřazení menšího z operandů do `rd`
`lb rd, imm(rs1)`	`load:8:\rs1 \imm +`	Načtení 8 bitů z adresy `rs1+imm`
`fsgnj.s rd, rs1, rs2`	`\rs1 bits 0x7fffffff & \rs2 bits 0x80000000 & | float \rd =`	Spojení hodnoty v plovoucí čárce `rs1` a znaménko z `rs2`
`jal rd, imm`	`\pc 4 + \rd = \imm \pc +:true`	Nepodmíněný skok na `imm` a zapsání návratové adresy do `rd`

Tabulka 8.1: Vybrané instrukce a výrazy, které je popisují.

8.3.1 Zpracování programu

Při implementaci jsem se snažil pokrýt všechny případy, které překladač jazyka C běžně generuje. Cílem bylo minimalizovat šanci, že překladač vygeneruje kód nekompatibilní se simulátorem.

Pro zpracování kódu jsem zvolil dvou-průchodové řešení. V prvním průchodu jsou zpracovávány instrukce a paměťové definice (direktivy). Text programu je rozdělen na jednotky jazyka (*tokeny* jako například symbol, komentář, nebo nový řádek). Ve smyčce jsou poté postupně tokeny zpracovávány podle gramatiky a jednotlivé instrukce ukládány. Objekty instrukcí jsou zde referencemi provázány s objekty popisujícími jejich chování a s objekty registrů.

Při zpracování instrukcí je prováděno mnoho kontrol, například na počty a typy argumentů. Kód korektně pracuje s pseudoinstrukcemi, implicitními argumenty a speciální syntaxí se závorkami pro paměťové instrukce.

Kód 8.2 ukazuje příklad paměťových definic v programu. Celkem jsou podporovány direktivy `.byte`, `.hword`, `.word`, `.align`, `.ascii`, `.asciiz`, `.string`, `.skip` a `.zero`.

```
1  x:
2    .word 5 # celociselna promrnna x
3
4    .align 4
5  arr:
6    .zero 64 # 64 bajtu pameti zarovnanych na hranici 2^4 = 16 bajtu
7
8  hello:
9    .asciiz "Hello World" # retezec zakonceny nulovym bajtem
```

Výpis 8.2: Příklady dat definovaných v assembleru. Na takto definovanou paměť se v programu odkazuje návěstími (například `arr`).

Po prvním průchodu nejsou hodnoty všech operandů definované, protože operand může referovat na ještě nezpracované návěstí. Druhý průchod doplní vynechané hodnoty a zpracování programu je tím dokončeno.

Komplikací při doplňování hodnot je podpora aritmetických výrazů v argumentech instrukcí (například `lla x4, arr+64`). Tato funkce je implementována z toho důvodu, že překladač takové výrazy často generuje.

Proto mezi prvním a druhým průchodem probíhá alokace paměti. Po alokaci jsou známy všechny hodnoty návěstí a je možné vypočítat finální hodnoty argumentů instrukcí. Skokové instrukce používají ke skoku relativní hodnoty, proto je někdy nutné z absolutní hodnoty návěstí odečíst pozici instrukce. Výrazy se vyhodnocují jednoduchým vyhodnocovacím programem, který musí mít k dispozici hodnoty návěstí.

8.4 Aplikační rozhraní

Aplikace je spustitelná z příkazové řádky. Formátem vstupní i výstupní komunikace je JSON. První argument slouží k výběru příkazu. Příkaz `cli` spustí aplikaci v režimu příkazové řádky, příkaz `server` v režimu HTTP serveru.

Oba příkazy mají zabudovanou nápovědu, kterou lze vyvolat argumentem `help`. Argumenty jsou také uvedeny v příloze C.

Log je tištěn na standardní výstup. Zprávy se týkají obsluhy HTTP dotazů (viz sekce 7.1.2) a chybových stavů. Zprávy jsou strukturované a obsahují časovou známku, úroveň důležitosti, zdroj a samotnou zprávu.

8.4.1 Simulační parametry

Simulátor přijímá tři vstupy: konfiguraci procesoru, program a data. Vstupy jsou tímto způsobem rozděleny, aby je bylo možné volně kombinovat. Například je jednoduché měřit skriptem výkon konfigurace pro různé programy a různé velikosti dat.

Konfigurace procesoru a data jsou očekávána ve formátu `JSON`, konkrétně ve formátu specifikovaném v příloze D. Kód je libovolný textový soubor s validním programem podporované podmnožiny assembleru RISC-V. Validitu kódu je možné zkontrolovat ve webové aplikaci, nebo pomocí API serveru.

8.4.2 Serializace stavu

Zajímavým problémem byl návrh serializace stavu procesoru. JSON reprezentuje data jako stromovou strukturu, ale stav procesoru je obecný graf – obsahuje vzájemné reference mezi bloky.

První iterace designu používala přístup, který graf objektů kódovala následujícím způsobem:

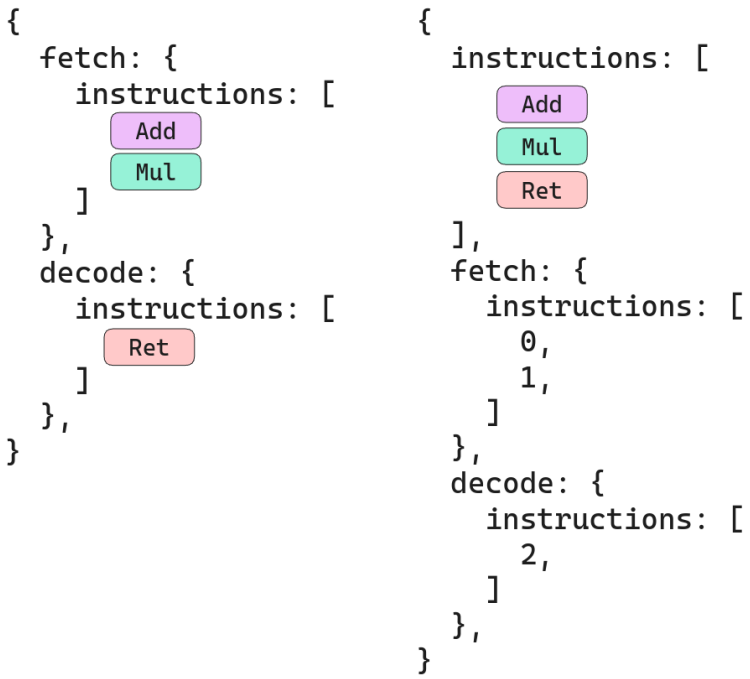
- První výskyt objektu je serializován a je mu přiřazeno ID
- Všechny další výskyty objektu jsou nahrazeny objektem vyjadřujícím referenci pomocí ID

Díky této notaci je možné jednoduchým programem zrekonstruovat původní graf. Nevýhodou je, že bez této rekonstrukce je výstup méně čitelný.

Finální řešení používá populární knihovnu Jackson pro jazyk Java a vlastní řešení pro správu instancí. Jackson dovoluje u konkrétních tříd deklarovat, aby se jejich instance serializovala jako ID, podobně jako v prvním řešení.

Druhým dílem řešení byla správa instancí třídami, které nazývám `Managers`. Manager je zodpovědný za vytváření všech instancí dané konkrétní třídy (instrukce, registry) a sledování těchto instancí. V moment, kdy má proběhnout serializace stavu procesoru, se serializují i managery. Výsledkem je, že jsou všechny instance serializovány společně.

Vzniká tak *normalizovaný* strom, se kterým se jednoduše pracuje ve webové aplikaci. Porovnání stavu a normalizovaného stavu se nachází na obrázku 8.1.



Obrázek 8.1: Zjednodušený stav procesoru (vlevo) a jeho normalizovaná verze (vpravo).

8.4.3 HTTP Server

Server je implementovaný za pomocí knihovny *Undertow*[1]. Tato knihovna poskytuje vcelku jednoduché rozhraní pro paralelní obsluhu HTTP požadavků a definici zdrojů (*endpointů*). Obsluha požadavku je přesunuta do pracovního vlákna, takže obsluha je neblokující.

Všechny endpointy využívají obecnou třídu pro zpracování těla POST požadavku ve formátu JSON, vyvolání daného kódu pro obsluhu a odeslání odpovědi.

HTTP server má konfigurovatelnou adresu (host a port), dobu po které dojde k vypršení požadavku a přepínač pro použití komprese zip v odpovědi.

Každý endpoint je definován svou cestou (např. `/compile`), obsluhovací metodou a datovým typem vstupu a výstupu. Obsluhovací metoda kontroluje přítomnost povinných parametrů a vykoná aplikační logiku. Metoda by měla korektně pracovat s očekávanými chybami a vrátit chybovou hodnotu, která v daném kontextu dává smysl.

Neobsloužené výjimky jsou zachyceny serverem a vyřešeny generickou chybovou hláškou, aby nedošlo k pádu aplikace.

8.4.4 Rozhraní příkazové řádky

Rozhraní příkazové řádky nabízenou funkcionalitou odpovídá HTTP serveru. Místo předávání argumentů v těle POST požadavku se konfigurace specifikuje soubory. Všechny parametry rozhraní jsou uvedeny v příloze C.

[1] https://undertow.io/

Přepínač	Význam
`-xc`	Překlad jazyka C (nelze odvodit z přípony souboru)
`-g`	Mapování řádků jazyka C do assembleru
`-march=rv32imfd`	Definice architektury a rozšíření M, F, D
`-mabi=ilp32d`	Generování funkcí s ABI předávající argumenty registry
`-o /dev/stdout`	Výstup na standardní výstup
`-S`	Výstup ve formě assembleru
`-fcf-protection=none`	Vypnutí bezpečnostních ochran skokových instrukcí
`-fno-stack-protector`	Vypnutí ochrany proti přetečení bufferů
`-fno-asynchronous-unwind-tables`	Vypnutí generování dat pro obsluhu výjimek
`-mno-explicit-relocs`	Vypnutí relokace symbolických adres (operátory `%hi()` a `%lo()`)
`-ffunction-sections`	Vytvoření zvláštní sekce pro každou funkci
`-fdata-sections`	Vytvoření zvláštní sekce pro každý datový objekt
`-fno-dwarf2-cfi-asm`	Snížení šumu v generovaném kódu
`-finhibit-size-directive`	Snížení šumu v generovaném kódu
`-mstrict-align`	Zabránění generování nezarovnaných paměťových přístupů
`-nostdlib`	Zákaz využití standardní knihovny jazyka C
`-fdiagnostics-format=json`	Výstup chyb ve formátu JSON
`-fPIE`	Generování pozičně nezávislého kódu (position-independent code)
`-fno-plt`	Zákaz generování nepřímých skoků pomocí PLT (Procedure Linkage Table)

Tabulka 8.2: Jednotlivé přepínače pro překlad. K těmto přepínačům je ještě přidán přepínač pro žádanou úroveň optimalizace. Většina přepínačů pomáhá generovat čitelnější kód vhodnější ke zpracování simulátorem.

8.4.5 Integrace s GCC

Pro účely překladu programů definovaných v jazyce C do jazyka RISC-V assembleru HTTP server poskytuje endpoint `/compile`. Samotný překlad programu je uskutečněn třídou `GccCaller`, která poskytuje rozhraní pro volání kompilátoru GCC[2].

Proces překladu spočívá ve spuštění nového procesu, který vykoná program GCC s určitými přepínači (*flags*) a uživatelem zadaným kódem. Standardní vstup, standardní výstup a standardní chybový výstup procesu jsou přesměrovány do paměti objektu `GccCaller`. Návratová hodnota procesu vypovídá o úspěchu překladu.

Cesta programu GCC je konfigurovatelná při spuštění serveru. Dále je konfigurovatelná úroveň optimalizace programu. Povoleny jsou přepínače `-O0`, `-O2`, `-O3` a `-Os`. Výčet všech použitých přepínačů se nachází v tabulce 8.2.

[2] https://gcc.gnu.org/

Kompilace každého objektu do zvláštní sekce znemožní optimalizace využívající relativní polohu kódu a dat, což je žádoucí vzhledem k tomu, že data budou v simulátoru alokována na jinou než předpokládanou pozici.

Existence `memcmp` a `memset` je do GCC vestavěna a nelze zabránit generování jejich volání. Rovněž se mi nepodařilo zakázat volání jiných vestavěných funkcí jako například `__builtin_popcount(n)`. Endpoint tedy programy s těmito voláními zpracovává a nehlásí chybu, i když s takovým programem simulátor není schopen pracovat.

Výstup v podobě assembleru obsahuje velké množství informací, které jsou pro simulátor nadbytečné a navíc snižují čitelnost kódu. Proto výstup překladače prochází filtrem, který odstraní nedůležité direktivy, návěstí a data. Výstupem je tak podstatně čitelnější kód, obsahující pouze relevantní údaje. Příklad celé transformace je uveden v 8.2.

```c
int f(int x) {
    return 2 * x;
}
```

(a) Vstupní program v jazyce C.

```
  .file "<stdin>"
  .option pic
  .attribute arch, "rv32i2p1_m2p0_f2p2_d2p2_zicsr2p0"
  .attribute unaligned_access, 0
  .attribute stack_align, 16
  .text
  .section .text.f,"ax",@progbits
  .align 2
  .globl f
  .type f, @function
f:
  slli a0,a0,1
  ret
  .ident "GCC: (Ubuntu 12.3.0-1ubuntu1~22.04) 12.3.0"
  .section .note.GNU-stack,"",@progbits
```

(b) Assembler před filtrovacím krokem.

```
f:
  slli a0,a0,1
  ret
```

(c) Kód po filtrování.

Obrázek 8.2: Kroky zpracování programu jazyka C.

Filtr nejdříve rozdělí text na sekce a smaže ty, které nepopisují kód ani data. Text je dále rozdělen na jednotlivé řádky a řádek assembleru je spojen s odpovídajícím řádkem programu v C[3]. Dalším krokem je přečtení instrukcí a sběr používaných návěstí. Na základě této informace jsou smazány všechny direktivy, na které neodkazuje žádné používané návěstí.

Výsledkem je čistý program s dodatečnou informací o vztahu mezi řádky původního programu a přeloženého assembleru. Mapování je využito v editoru kódu webové aplikace.

[3]Tento krok souvisí s vizualizací vysvětlenou v kapitole 9.3.2.

Kapitola 9

Implementace webové aplikace

Požadavky na aplikaci ze sekce 7.3 lze splnit mnoha způsoby. Vzhledem k rozsahu aplikace budu volit z rámcových řešení pro webové aplikace. Faktory při výběru frameworku byly popularita, kvalita dokumentace, osobní zkušenost a dispozice knihoven.

Pro implementaci jsem zvolil JavaScriptový framework Next.js[1]. Jedná se o v současnosti nejrozšířenější způsob psaní aplikací s knihovnou React. Jeho hlavními znaky jsou úzké propojení se serverovou stranou a hybridní přístup k renderování stránek.

Next.js aplikace je možné psát buď v jazyce JavaScript nebo TypeScript. Kvůli rozsahu aplikace jsem zvolil TypeScript s předpokladem, že typové informace usnadní implementaci.

Příloha G obsahuje více obrázků webové aplikace.

9.1 Logika aplikace

Z pohledu klientské aplikace interaktivní simulace spočívá v následujících krocích:

1. kontaktovat simulační server s požadovanou konfigurací,
2. získat a uložit odpověď (nový stav procesoru),
3. zpřístupnit nový stav všem relevantním komponentům,
4. spustit renderování komponentů (prezentace).

Tento postup se opakuje při každém kroku simulace. Pro implementaci práce se stavem jsem zvolil knihovnu pro správu globálního stavu *Redux*[2].

Redux poskytuje rámcové řešení pro popis stavu, přechodů mezi stavy a selektory. Redux konceptualizuje globální stav jako objekt, který mezi stavy přechází funkcemi zvanými *reducers*. Jedná se o čisté funkce, knihovna se zde inspirovala funkcionálním programováním.

Přístup k části stavu je zprostředkován *selektory*. Selektor je funkce, která ze stavu vybírá jeho podmnožinu. Díky použití selektorů a přechodů je možné určit datové závislosti mezi komponentami a globálním stavem a znovu renderovat pouze ty komponenty, jejichž data byla změněna.

Knihovna mi dovolila velmi jednoduše implementovat perzistenci stavu. Díky tomu je konfigurace zachována mezi sezeními. Důležité je při každé změně schématu globálního

[1] https://nextjs.org/
[2] https://redux.js.org/

stavu definovat *migraci*, aby aplikace nepracovala se starým, neočekávaným formátem dat. Tato technika je běžně používaná v databázích.

Další přidanou hodnotou je vývojové rozšíření prohlížeče, které umožňuje prohlížet všechny přechody mezi stavy na časové ose. Detailní náhled stavu umožňuje zobrazit změny z posledního přechodu (*diff*), nebo aplikaci do zvoleného stavu přenést. Tyto funkce jsou umožněny transakční/funkcionální architekturou. Možnost detailně prohlížet stav aplikace mi několikrát usnadnilo řešení problémů.

Globální stav se organizuje do celků s názvem *slice*. Jeden slice se má týkat stavu jedné funkcionality. V aplikaci jsou 4 slicy:

- `isaSlice` – ukládá konfigurace procesoru a aktivní konfiguraci,
- `cpustateSlice` – ukládá aktuální stav interaktivní simulace,
- `compilerSlice` – obsahuje stav editoru kódu,
- `simConfigSlice` – obsahuje aktuální simulovanou konfiguraci a kód.

Jedním z problémů na který jsem narazil při implementaci byla dvojí implementace kontrola oborů hodnot parametrů. Jedna implementace se nacházela v serverové části a druhá ve webové části kódu. Udržení implementací v synchronizovaném stavu probíhalo manuálně, což je práce náchylná k chybám.

9.2 Komunikace se serverem

Výsledkem překladu projektu ve frameworku Next.js je serverová aplikace a staticky vygenerované zdroje (HTML, CSS, JS). Prvním požadavkem na server se stáhne statický základ stránky, poté na straně klienta proběhne inicializace Reactu (hydratace) a UI se stává interaktivním. Veškeré další požadavky na server včetně navigace probíhají přes webové `fetch API`.

Webový server kromě stránek, stylů a skriptů nabízí dva statické zdroje: příklady kódů a strukturovaný popis instrukcí RISC-V. Oba objekty jsou poskytovány přes HTTP API ve formátu JSON.

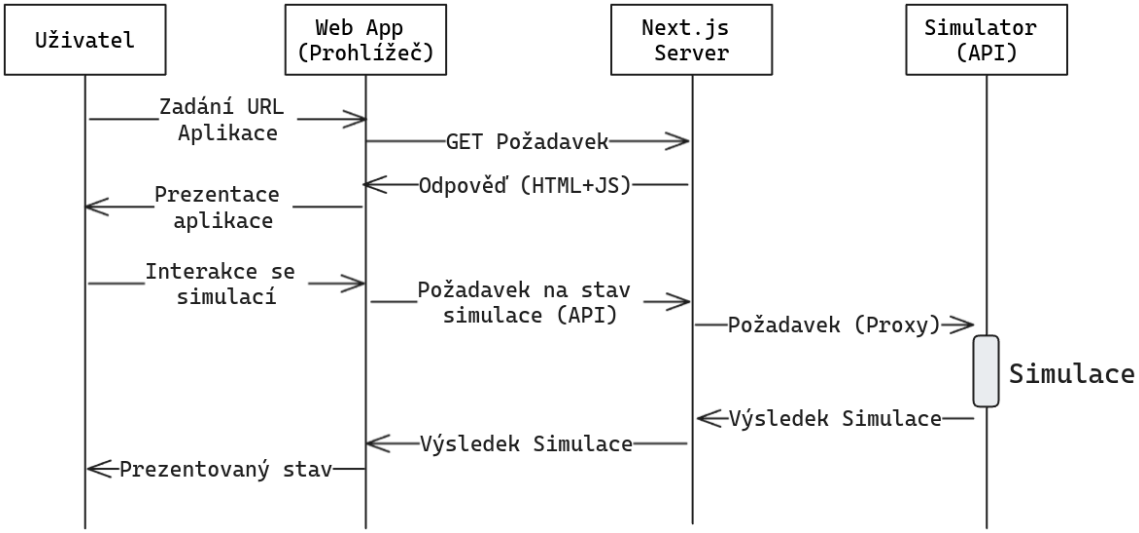
První verze řešení využívala přímé volání simulačního serveru, které muselo být nasazeno na adrese dostupné klientovi. Finální řešení přístup klienta ke službám simulačního serveru zprostředkovává přes webový server, který pracuje jako *proxy*. Při nasazení je tedy zapotřebí konfigurovat jedinou adresu, na které klient získá i webové rozhraní i simulační služby. Celá komunikace je nastíněna na obrázku 9.1.

Odpověď simulačního serveru je uložena do globálního stavu, čímž spustí renderování nového stavu simulace. Redux umožňuje narušit funkcionální povahu přechodů mezi stavy a při přechodu provést HTTP dotaz.

Framework Next.js ve výchozím nastavení poskytuje mnoho optimalizací. Mezi nejvýznamější patří caching a prefetching zdrojů. Výsledkem prefetchingu odkazů zobrazených na stránce ve výsledku znamená okamžitou navigaci mezi stránkami.

9.3 Stránky

Aplikace se skládá z osmi hlavních pohledů. V následujících podsekcích rozvedu některé z jednotlivých stránek, jejich účel a způsob implementace.



Obrázek 9.1: Sekvenční diagram komunikace prohlížeče a serveru.

Stránka nastavení poskytuje pouze tři funkce – smazaní lokálního úložiště, export dat a výběr mezi světlou a tmavou barevnou paletou.

9.3.1 Simulační okno

Simulační okno je jádrem celé aplikace. Je zároveň nejsložitější částí aplikace. Okno obsahuje schéma procesoru a dva samostatné prvky: horní ovládací menu a pravou lištu.

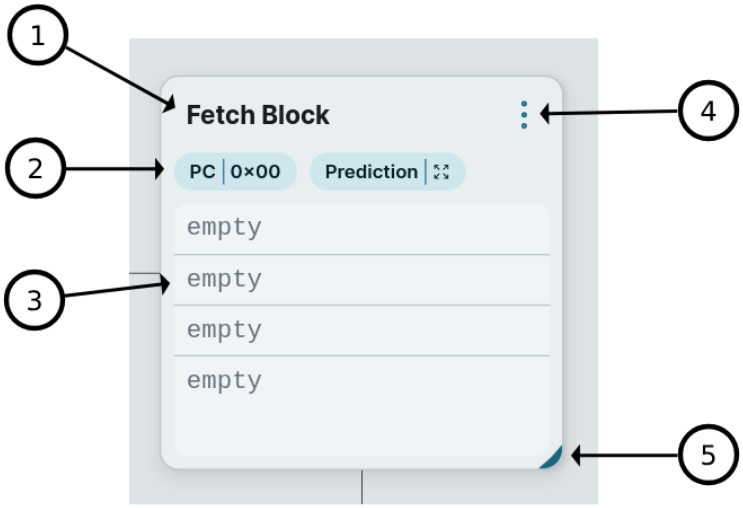
Prostřední část okna je věnována schématu procesoru. Vedle sebe zde leží rozmístěny jednotlivé obdélníkové bloky, které odpovídají blokům v simulátoru. Mezi bloky vedou spoje, které naznačují jejich souvislost.

Všechny bloky sdílejí ovládací prvky. Blok jednotky fetch je uveden na obrázku 9.2. V horní části vedle názvu bloku (1) se nachází tlačítko (4) vyvolávající vyskakovací okno s detailem bloku. Pod ním je několik nejdůležitějších informací o aktuálním stavu (2). Potažením pravého dolního rohu (5) se dá změnit velikost bloku. Rozmístění bloků je k této změně responzivní. Zbytek plochy je specifický pro každý druh bloku. Nejčastěji obsahuje seznam instrukcí (3), které se v bloku momentálně nacházejí.

Celý pohled na procesor lze posouvat a přibližovat. Technicky je této možnosti docíleno sledováním pohybu myši nad oknem a přepočítáváním odsazení všech bloků od počátku souřadného systému.

Simulaci je možno ovládat jak myší, tak i klávesnicí. Pro ovládání myší je v horní části obrazovky menu s tlačítky umožňující krok v simulaci dopředu nebo zpět, skok na začátek nebo konec simulace. Stejné možnosti jsou k dispozici klávesovými zkratkami. Druhé menu v horní části obrazovky nabízí automatické krokování simulace. Prodleva mezi kroky je nastavitelná v textovém poli.

Schématický pohled na stav simulace poskytuje celkový přehled, nemůže ale zobrazit všechny detaily, které jsou k dispozici. K tomu slouží vyskakovací náhledy. Kliknutím na blok nebo instrukci se objeví okno, které v tabulkové formě poskytuje dostupné informace. Pro instrukce se jedná o časové známky důležitých aktivit (fetch, execute), hodnoty parametrů a příznaky (například zda je instrukce spekulativní, či zda jsou všechny parametry připraveny k vykonání). Detaily bloků jsou specifické pro každý blok. Například detail bloku



Obrázek 9.2: Reprezentace bloku Fetch.

hlavní paměti (na obrázku 9.3) ukazuje všechny ukazatele v programu, jejich adresy a větší znázornění celé paměti.

Po najetí na instrukci nebo registr se zvýrazní všechny jejich výskyty v ostatních blocích. Tato funkce je velmi užitečná pro zorientování ve stavu simulace. Implementace je rozvedena v sekci 9.4.2, kde je popsán i proces její optimalizace. Po najetí myší na parametr instrukce se také objeví bublina s jeho hodnotou. U registru se navíc objeví informace o jeho přejmenování.

Lišta na pravé straně zobrazuje vybrané statistiky a log. Lišta má dva stavy: výchozí a expandovaný. V expandovaném stavu vzniká místo pro více statistik a textu zpráv z logu. Ke zobrazení jsem vybral statistiky, které považuji za nejrelevantnější při simulování: počet taktů, počet vydaných instrukcí, IPC a úspěšnost predikce skoků. Kompletní statistiky jsou zobrazeny na zvláštní stránce (viz sekce 9.3.4). Každá zpráva v logu má svou časovou známku (takt, kdy zpráva vznikla). Kliknutím na číslo zprávy se simulace do tohoto taktu přenese.

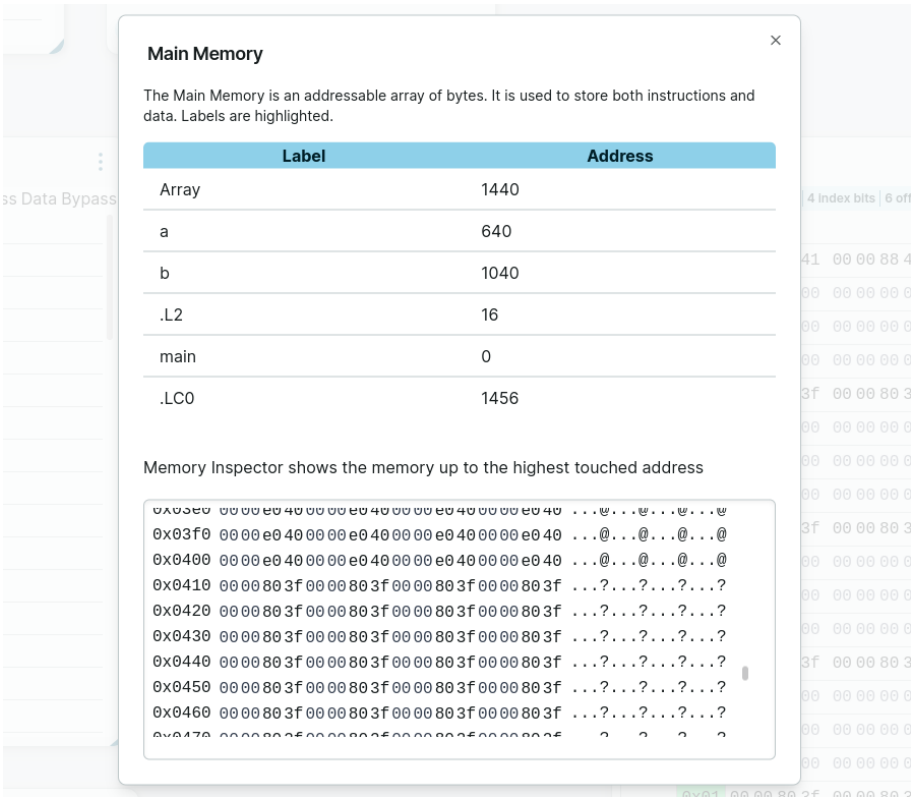
9.3.2 Editor kódu

Editor (na obrázku 9.4) nabízí lištu s akcemi a dvě textová pole. V levém sloupci lze ovládat kompilaci. Prostřední okno slouží k psaní kódu v jazyce C, pravé okno zobrazí výsledek překladu.

Textová pole jsou implementována za pomocí knihovny CodeMirror[3]. Tato knihovna poskytuje dobré uživatelské rozhraní pro editaci kódu, například běžné zkratky, zvýraznění syntaxe, zobrazení varování, očíslování řádků a další. Důležitá byla i možnost vytvořit vlastní rozšíření.

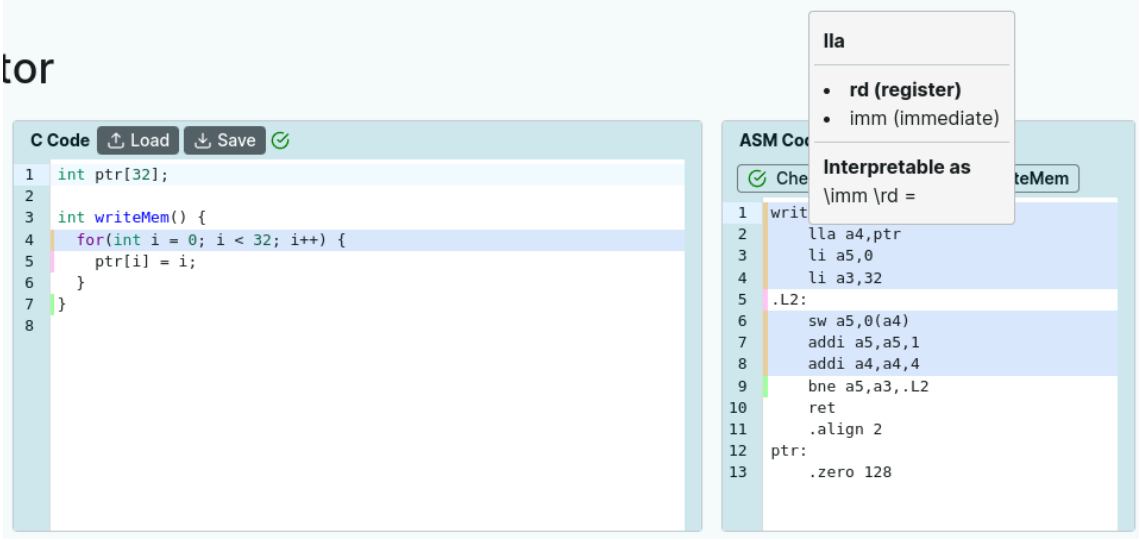
Souvislost částí kódu je naznačena barevným páskem na levé straně textového pole. Najetím myší na skupinu řádků se vztah vyznačí výrazněji. Při najetí myší na instrukci (obrázek 9.5) se také objeví vyskakovací okno s informací o argumentech instrukce.

[3] https://codemirror.net/5/

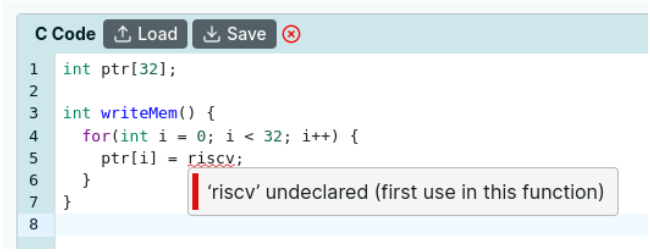


Obrázek 9.3: Detail bloku hlavní paměti.

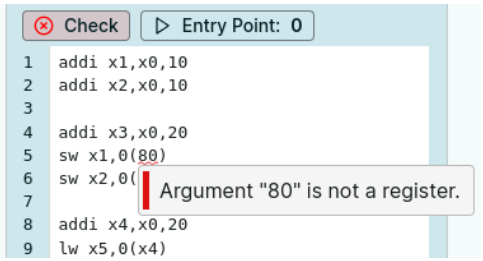
Obrázek 9.4: Editor kódu s kompilátorem.



Obrázek 9.5: Vzhled editoru při najetí myší na instrukci.



(a) Chyba v programu jazyka C. Hlášení jsou předávána z kompilátoru GCC.



(b) Chyba v Assembleru RISC-V. Hlášení pochází z vlastního kompilátoru assembleru (viz sekce 8.3.1).

Obrázek 9.6: Zobrazení chyb v editoru.

O vyznačení chyb v textovém poli se stará knihovna CodeMirror, chyby je ale nutné z výstupu simulátoru převést do správného formátu. Chybná část kódu je podtržena, po najetí na oblast myší je zobrazena chybová zpráva (viz obrázek 9.6). Zpráva o chybách nebo úspěchu překladu se také objevuje v rohu obrazovky.

V neposlední řadě je zde možné zvolit vstupní bod programu. Vstupním bodem může být první instrukce, nebo libovolné návěstí.

9.3.3 Konfigurace paměti a procesoru

Konfigurace paměti (simulačních dat) a procesoru jsou formulářové stránky. Obsahují velké množství vstupních textových polí s názvem, vysvětlivkami a validací. Formuláře kopírují strukturu konfigurace simulátoru.

Logika formulářů je implementována pomocí knihovny *React Hook Form*. Schéma formuláře je definováno objekty validační knihovny *Zod*.

Vstupní textové pole a přepínač (*radio input*) jsou znovupoužitelné komponenty.

Konfigurace procesoru je rozdělena do záložek, které všechny možnosti organizují do kategorií. V horní části obrazovky je možné přepínat mezi uloženými konfiguracemi, nebo

založit konfiguraci novou. Zvláštním prvkem formuláře je „podformulář“ pro definice funkčních jednotek.

Konfigurace paměti umožňuje přidávat, upravovat a mazat datové objekty. Datový objekt má jméno, zarovnání, datový typ a samotná data. Data je možné definovat několika způsoby:

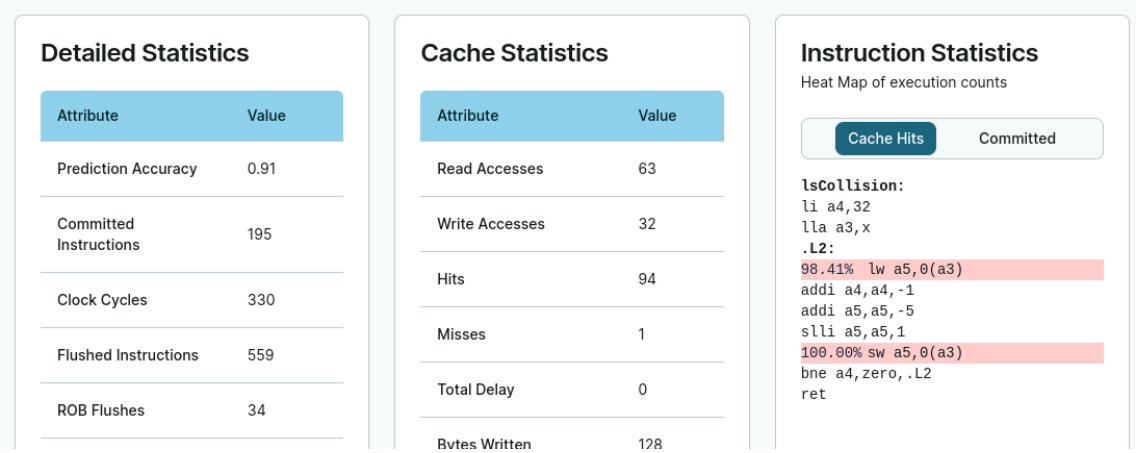
1. explicitním výčtem prvků,
2. opakováním konstanty (po vzoru `memset`),
3. rozsahem náhodných prvků (např. 32 čísel v rozsahu 0-15).

Formulář nemá plnou vyjadřovací schopnost – jsou konfigurace paměti, které lze definovat pouze ručně psanou definicí ve formátu JSON. Takové definice ale lze do webové aplikace importovat a pracovat s nimi.

9.3.4 Statistiky

Jak bylo zmíněno v sekci o hlavním simulačním okně (9.3.1), nejdůležitější statistiky jsou zobrazeny v boční liště při simulaci. Detailnější pohled na sbírané statistiky je k dispozici ve speciálním okně.

Stránka je organizovaná do karet podle kategorií statistik. Tři karty jsou ukázány na obrázku 9.7. Modul na pravé straně ukazuje statistiky ke konkrétní instrukci v podobě *teplotní mapy*. Lze v něm přepnout mezi pohledem na úspěšnost nalezení dat v cache a podílem vydaných instrukcí v průběhu programu.



Obrázek 9.7: Výřez obrazovky statistik poskytovaných na vyhrazené stránce.

9.3.5 Edukativní stránky

Součástí aplikace je krátký informativní úvod do architektury RISC-V. Tento text vysvětluje základní informace o registrech a konvenci volání, což může být užitečné pro někoho, kdo má zkušenosti s assemblerem, ale ne konkrétně s RISC-V. Pro hlubší porozumění jsou poskytnuty odkazy na detailnější materiály.

Jiná edukativní stránka se věnuje přehledu architektury procesoru. Každému důležitému komponentu je věnována krátká sekce.

Poslední stránkou je nápověda. Ta uvádí zkratky, kterými je možné simulaci ovládat. Obsahuje také tipy pro psaní kódu pro simulátor, včetně jeho specifik a odchylek.

9.4 Design a použitelnost

Mnoho komponentů (například formuláře, nebo karty objevující se po přejetí myší a vyskakovací okna) je implementovaných pomocí knihoven předpřipravených komponentů. Tyto komponenty zajišťují dobrou úroveň dostupnosti a správné chování napříč zařízeními. Zároveň umožňují detailní úpravu podle požadavků programátora.

Komponenty jsou stylovány kombinací CSS a stylovací knihovny *TailwindCSS*[4], která poskytuje obecné stylovací třídy. Na stránkách jsou využity sémantické značky HTML (například `<section>`, `<nav>`, které přesněji vyjadřují význam částí stránky.

Navrhnout vhodnou a funkční barevnou paletu je těžký designový úkol, který byl přenechán nástroji *Material Theme Builder*[5]. Algoritmus od společnosti Google, který se používá v operačním systému Android, generuje vizuálně zajímavou paletu splňující požadavky na kontrast. Paleta má také variantu pro tmavý režim rozhraní. Obrázek 9.9 ukazuje porovnání světlé a tmavé palety rozhraní.

Zvláštní důraz byl kladen na možnost ovládat celou aplikaci klávesnicí. Motivací bylo dát všem uživatelům tuto možnost, neméně důležité ale bylo udělat maximum pro dobrou dostupnost aplikace. Rozhraní simulace obsahuje velké množství interaktivních prvků (dlouhé tabulky instrukcí), což může učinit navigaci na konkrétní prvek zdlouhavé. Tento nedostatek není zcela vyřešen, prázdná pole jsou ale při navigaci ignorována.

Zobrazení aplikace je optimalizováno pro všechny velikosti obrazovky. Obrázek 9.8 jako příklad ukazuje rozložení stránky kompilátoru na mobilním zařízení (obrázek 9.5 ukazuje stejnou část rozhraní na široké obrazovce stolního počítače). Prvky mají užší okraje a komponenty jsou uspořádány vertikálně. Aplikaci je možné na mobilním zařízení úspěšně používat, omezení malé dotykové obrazovky ale snižuje pohodlí.

9.4.1 Komponenty rozhraní, React

Rozhraní je členěno do hierarchie komponentů. Většina komponentů je definovaných ve zvláštním souboru, ale některé příliš specifické komponenty se nachází v souboru společně s místem použití.

Při návrhu uživatelského rozhraní pomocí React komponentů byly dodržovány zásady jediné odpovědnosti (SRP), což pomohlo udržet každou komponentu co nejvíce oddělenou a specializovanou na svou úlohu. Každá komponenta byla navržena s definovaným účelem, rozhraním a funkcionalitou, což usnadňuje správu kódu a jeho budoucí rozšíření. Komponenty jsou ve struktuře projektu organizovány podle stránek nebo účelu, ke kterému náleží.

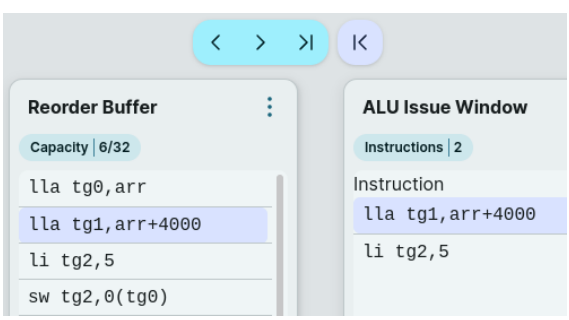
Dalším důležitým principem byla znovupoužitelnost. Některé komponenty jsou využívány v různých částech aplikace a v různých kontextech. Příkladem je komponent pro zobrazení programu, který je využitý jak v simulačním tak statistickém okně.

9.4.2 Optimalizace renderování

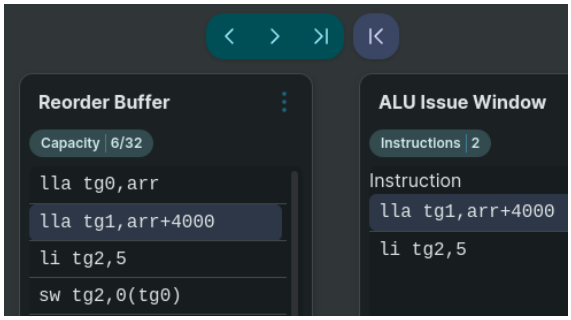
Důležitou součástí rozhraní simulace je interaktivní zvýraznění všech výskytů instrukce nebo registru napříč bloky. První řešení informaci o novém „aktivním“ prvku zapisovalo do

[4] https://tailwindcss.com/
[5] https://m3.material.io/theme-builder

Obrázek 9.8: Pohled kompilátoru v rozložení pro mobilní telefony.



(a) Simulátor ve světlém režimu rozhraní.



(b) Simulátor v tmavém režimu rozhraní.

Obrázek 9.9: Světlá a tmavá varianta palety barev.

globálního stavu. Styl zvýraznění byl vypočítáván JavaScriptem, takže změna stavu spustila renderování *celého* náhledu. Tento přístup nebyl optimální a negativně se projevoval na responzivitě rozhraní. Přejetím kurzoru napříč oknem se mohly vykonat i desítky událostí způsobujících renderování.

Řešením bylo přesunutí datové závislosti z Reactu do CSS. Jádro řešení zůstává stejné – JavaScript stále naslouchá události najetí myši na instrukci a změny se stále zapisují do globálního stavu. Důležitým rozdílem je, že změna stavu vyvolává změnu v dynamicky generovaném kaskádovém stylu místo změny v závislostech Reactu. Vyhodnocení CSS pravidel probíhá v kódu prohlížeče, který je mnohem výkonnější. Stylování probíhá pravidlem, které vybírá elementy na základě hodnoty jeho atributu (ID). Po implementaci této optimalizace probíhá zvýrazňování okamžitě.

Druhou významnou optimalizací bylo využití virtuálních rolovacích seznamů v blocích, které zobrazují velké množství prvků. Virtuální seznam využívá předpokladu, že většina prvků seznamu není zobrazených ve stejný okamžik. Renderovány jsou jen ty prvky, které mají být v daný okamžik viditelné. Jestli má prvek být viditelný je vypočteno na základě velikosti prvků v seznamu a velikosti rolovacího okna.

Implementace optimalizace se projevila významným snížením doby renderování, obzvlášť v náhledu do hlavní paměti procesoru, který má velké množství elementů. Při pokusu jsem naměřil pokles v počtu HTML elementů na stránce z 11700 na 6500.

Kapitola 10

Testování, dokumentace a nasazení

V této kapitole budou popsány okolní činnosti související s vývojem aplikace. Zejména se zaměří na testování simulátoru a webové aplikace.

Celý projekt obsahuje asi 33 000 řádků kódu (bez komentářů a prázdných řádků). Velikost projektu je uvedena pro kontext množství komentářů: asi 6 000 řádků (15 000 včetně hlaviček souborů). Při refaktorizaci simulátoru byl razantně zvýšen počet komentářů, jak v hlavičkách funkcí a tříd, tak uvnitř kódu.

Během vývoje byl využit systém pro správu verzí `git`. Kód byl před každým sloučením do hlavní větve prověřen vedoucím práce.

V projektu byl použit styl kódu z příručky *Handbook SC FIT* (příručky pro skupinu SC@FIT). Použití jasně definovaných pravidel pro úpravu kódu zajišťuje konzistenci a přehlednost kódu, která usnadňuje orientaci a úpravy pro současné i budoucí členy týmu.

10.1 Testování simulátoru

Během implementace simulátoru (sekce 8.2) byly intenzivně využívány jak existující testy, tak nově vytvořené testy, které byly postupně doplňovány. Testy zvyšovaly důvěru ve správnost úprav i nových funkcí simulátoru. Projekt v současném stavu obsahuje více než 400 testů. Pokrytí řádků kódu testy v simulátoru je 83%. Pokrytí v blocích simulátoru je 94%.

Jednotkové testy (unit tests) testují izolované moduly. Izolované testování jednotlivých tříd a systémů simulátoru bylo stěžejním pro úspěšnou implementaci. Důkladněji testované byly moduly Cache, prediktory a překlad kódu. Kód byl navrhován tak, aby byl deterministický a problémy reprodukovatelné, což zjednodušuje testování.

Systém byl jako celek testován z mnoha aspektů. Každá instrukce má svůj test kontrolující její správné chování. Tento typ testů typicky kontroluje stav na konci simulace. Testovací skript navíc kontroluje, že všechny přiložené ukázky kódu proběhnou na simulátoru bez chyby.

Jedna sada testů simulátoru je velmi detailní a stav kontroluje po každém taktu. Detailní testy jsou zákonitě více spjaté s implementací a vyžadují větší množství údržby.

Je také testována funkčnost několika složitějších programů jako například třídění pole algoritmem *quicksort*, práce s lineárním seznamem a polymorfismus (*dynamic dispatch*).

10.1.1 Výkonnostní testování

Pro testování výkonu byl simulátor profilován v režimu server. V rámci testování také vznikl jednoduchý benchmark implementovaný pomocí Java Microbenchmark Harness (JMH).

Režim	Počet uživatelů	Latence (ms)		Propustnost (transakce/s)
		Medián	90. percentil	
Přímý	30	70,66	118	25,96
	100	680	1248,9	53,61
Docker	30	77	283	24,49
	100	1135	2031,9	42,07

Tabulka 10.1: Naměřené hodnoty latence pro 4 uvedené případy.

Nejdůležitějším závěrem z testování výkonu je následující: v režimu serveru asi 60% doby obsloužení požadavku zabírá práce s formátem JSON. Tento formát je ze své povahy nepříznivý pro výkon. Důsledkem dominance komunikační marže je, že výkonnostní zisky z optimalizace simulace se přestávají vyplácet. Změna komunikačního protokolu je směr, který je zajímavý v další práci prozkoumat.

Proběhlo také zátěžové testování nástrojem Apache JMeter™[1]. Charakteristika testu je následující:

- dvě velikosti testu: 30 a 100 uživatelů,
- dva běhové režimy: přímý a prostřednictvím Dockeru,
- každý uživatel interaktivně simuluje 40 kroků simulace jednoho ze dvou programů,
- náběh 4 s, 1 s pauzy mezi každým požadavkem uživatele (think time),
- použití gzip,

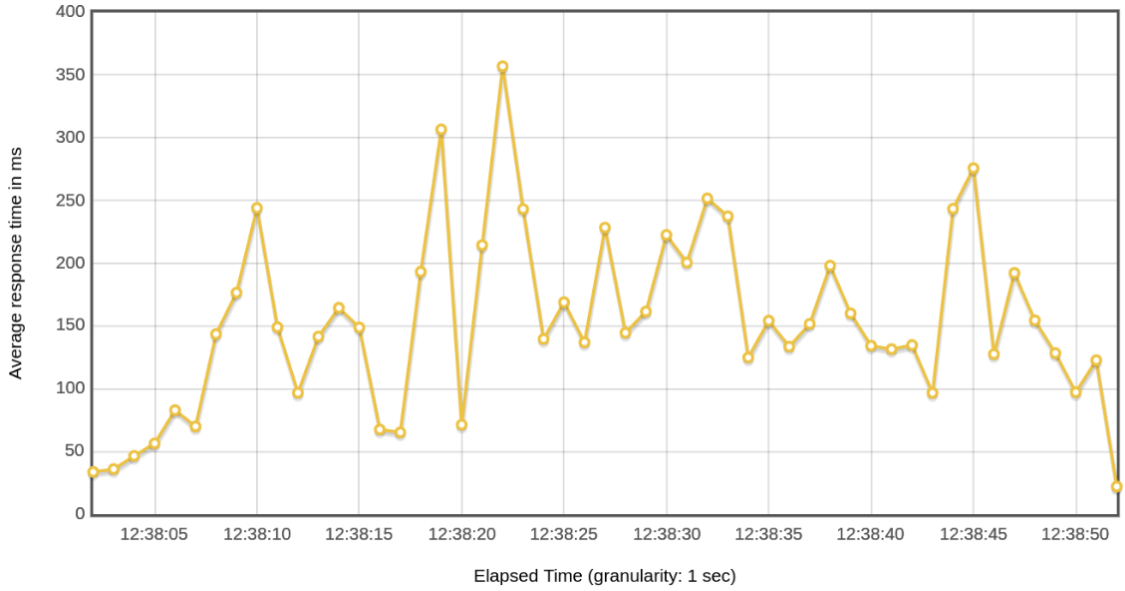
Použitím komprese gzip se zvýšila propustnost na lokálním serveru o 40%. Tabulka 10.1 uvádí naměřená data. Veškeré měření proběhlo lokálně na notebooku s procesorem `Intel i5 8300h` a 16 GB DDR4 RAM. Závěrem je, že server dobře zvládá menší množství současných uživatelů, bez ohledu na běhový režim, i když Docker má znatelný dopad na výkon aplikace. Větší množství uživatelů výrazně negativně ovlivňuje latenci takovým způsobem, že se výrazně zhorší komfort užívání aplikace. Během testu nedošlo ani k pádu aplikace ani k selhání dotazu. V reálném provozu bude latence pravděpodobně vyšší v důsledku delší cesty paketů internetem. Větší množství uživatelů je možné řešit provozem na silnějším hardwaru, nebo rozdělením zátěže mezi několik serverů.

Na obrázku 10.1 jsou uvedeny dva grafy latence odpovědí serveru. U obou testů lze pozorovat, že latence se drží v určitých mezích a nerostou neomezeně. Zvýšená zátěž nikdy během testování nezpůsobila odmítnutí požadavku.

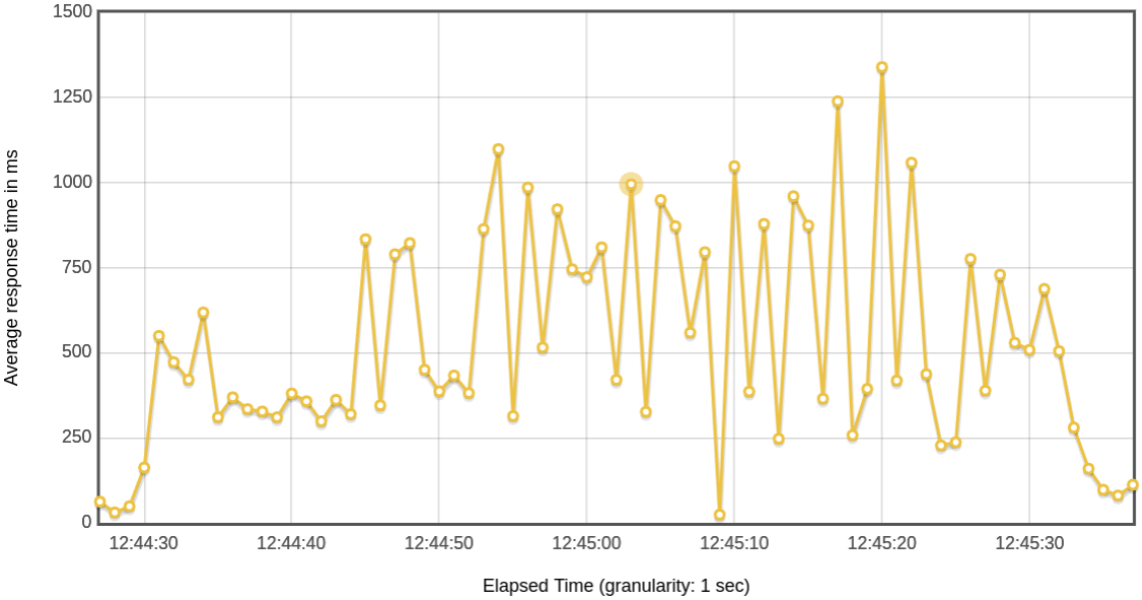
10.2 Testování webu

Webová část projektu byla testována především manuálně. Uživatelská rozhraní jsou proměnlivá a automatizace jejich testování je časově náročné. Vývoj probíhal v prohlížeči Google Chrome. V rámci testování byl také proveden manuální test funkcionality v prohlížeči Mozilla Firefox. Takto byly pokryty dva nejpopulárnější prohlížeče a tím i software většiny uživatelů.

[1] https://jmeter.apache.org/



(a) Průběh latence odpovědi od serveru v čase pro 30 současných uživatelů.



(b) Průběh latence odpovědi od serveru v čase pro 100 současných uživatelů.

Obrázek 10.1: Grafy latence serveru při interaktivní simulaci.

Testování proběhlo na zařízeních různé výkonnosti a formátu. Vývojové nástroje poskytují možnost uměle omezit výkon zařízení a rychlost připojení pro zjednodušení testování aplikace v širokém spektru situací.

Moduly webové aplikace byly v porovnání se simulátorem testovány mnohem méně. Testování zde bylo zaměřeno pouze na malé izolované funkce.

Z výkonnostního hlediska byla aplikace sledována naopak více. Knihovna React (ale i web obecněji) patří k méně výkonově optimálním řešením uživatelského rozhraní, proto je nutné více dbát na optimalitu implementace. Vývojové rozšíření prohlížeče pro React poskytovalo kvalitní zpětnou vazbu, díky které byly odhaleny mnohé výkonnostní problémy (viz sekce 9.4.2 o optimalizaci renderování).

Automatizovaný nástroj pro testování přístupnosti rozhraní Google Lighthouse odhalil některé nedostatky, jako například špatné značení tlačítek a odkazů pro asistivní technologie. Odhalil i jeden případ nedostatečného kontrastu textu vůči jeho podkladu. Nástroj také navrhoval možná řešení těchto problémů.

Výsledky měření odezvy simulace v milisekundách jsou pouze orientační. Byly provedeny na stejném počítači jako výkonnostní testování. Renderování na začátku simulace (s malým počtem prvků) trvá asi 60 ms; s více zaplněnými buffery trvá okolo 80 ms.

10.2.1 Uživatelské testování

V pozdní fázi vývoje aplikace proběhlo uživatelské testování. Test proběhl v podobě online dotazníku se dvěma úkoly, které měl uživatel v aplikaci splnit. Dotazník byl rozeslán studentům a pracovníkům FIT VUT. Dotazník se ptal na úspěšnost splnění úkolů, na celkový dojem z aplikace, zpětnou vazbu v podobě ohodnocení v rozsahu 0-10 bodů a textu ve volné formě.

Ze všech účastníků průzkumu (9 osob) dokázalo splnit oba úkoly 5 osob. Úspěšnost splnění úkolu byla kontrolována otázkou, kterou bylo možné správně zodpovědět pouze po úspěšném splnění druhého úkolu.

Průměrné hodnocení výukové hodnoty aplikace přesáhlo 9 bodů na stupnici od 0 do 10, kde 10 je nejlepší možné hodnocení. Hodnocení přehlednosti rozhraní bylo horší s průměrem 8,4.

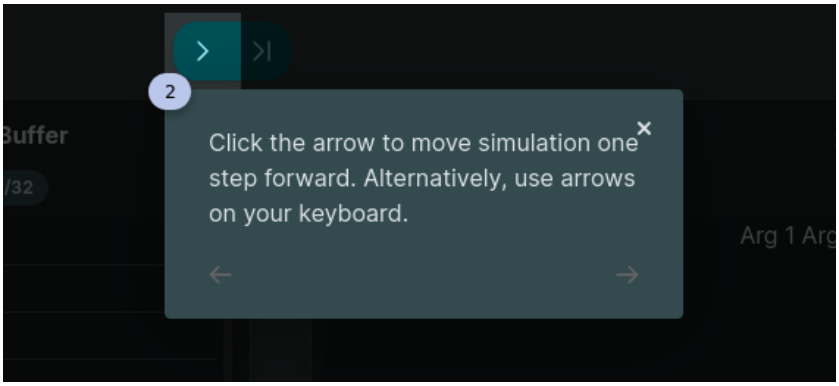
Zpětnou vazbu po zpracování vnímám obecně pozitivně. Jako nejčastější se projevily tyto problémy:

- nejasný způsob zadávání dat pro simulaci,
- nejasný způsob pohybu „pohledu“ v hlavním okně simulace.

Problém se zadáváním dat pravděpodobně souvisel s rozdělením definice kódu a definice dat do dvou záložek. Problém jsem se rozhodl vyřešit poskytnutím průvodce aplikací. Průvodce po krocích předvede načtení jednoho z příkladů a definici pole s hodnotami. Obrázek 10.2 ukazuje jeden z kroků průvodce. Relevantní část rozhraní je zvýrazněna a je zobrazen krátký vysvětlující komentář.

10.3 Nasazení

Aplikace je provozována kontejnery technologie Docker. Docker jsem zvolil z důvodů konzistence prostředí a jednoduchosti instalace. První kontejner obsahuje webový server, druhý kontejner obsahuje simulační server. Oba kontejnery ke komunikaci s vnějškem otevírají



Obrázek 10.2: Průvodce základními funkcemi simulátoru.

jeden port. Webový kontejner má síťový přístup do simulačního kontejneru, aby mohl předávat požadavky uživatele. Součástí repozitáře je skript, který oba kontejnery spustí pomocí `docker-compose`.

V případě použití Dockeru je jediným požadavkem instalace Dockeru a internetové připojení. Jako virtualizační technologie má provoz serveru v Dockeru nezanedbatelný dopad na výkon, vždy je ale možné aplikaci nainstalovat přímo.

Přímá instalace je detailně dokumentovaná v textových souborech, které jsou součástí repozitáře a také v příloze E. Je možné se inspirovat i nahlédnutím do kontejnerizačních skriptů, které obsahují přesnou sekvenci příkazů k instalaci a spuštění. Prerekvizitami jsou programy *Node.js* (runtime pro JavaScript), Java runtime, systém pro automatizaci překladu Gradle a GCC pro RISC-V (`gcc-riscv-none-elf`).

Knihovní závislosti webové aplikace jsou spravovány systémem NPM[2] (Node Package Manager). Soubory `package.json` a `package-lock.json` specifikují použité knihovny a jejich verze. Při instalaci jsou závislosti staženy z centrálního repozitáře.

Proces nasazení je *automatizovaný* službou *GitLab CI/CD*. Soubor `.gitlab-ci.yml` v kořeni projektu definuje předpisy akcí, které se mají spouštět při různých událostech v repozitáři. Pro každý commit se spouští testy. Úspěšnost testů je zobrazena v grafickém rozhraní GitLabu. Při novém *Pull Requestu* se spustí server s aplikací pro otestování a zhodnocení vedoucím.

Konfigurace simulátoru je založena na souboru s dvojicemi klíč-hodnota. Existují dvě verze této konfigurace (vývojová a produkční). Výběr probíhá při startu podle proměnných prostředí. Parametry lze přepsat argumenty příkazové řádky. Webová aplikace se konfiguruje proměnnými prostředí. Jedinou relevantní konfigurací je adresa simulačního serveru.

[2] https://www.npmjs.com/

Kapitola 11

Závěr

V této práci byly prozkoumány některé známé metody efektivní implementace superskalárních procesorů. Detailněji byla probrána instrukční sada RISC-V, kterou simulátor využívá. Také byla uvedena teorie implementace webových aplikací s přihlédnutím k návrhu uživatelských rozhraní.

Byla provedena důkladná analýza stávajícího simulátoru, zhodnocení jeho kladů i nedostatků. Na základě analýzy byla navržena mnohá zlepšení, jak z pohledu implementace simulace, tak použitelnosti jeho rozhraní.

Z mého pohledu nejdůležitější změnou je samotné přenesení simulátoru na webovou platformu. Simulátor je nyní všem dostupný přímo z prohlížeče.

Do rozhraní simulátoru se podařilo přidat režim příkazové řádky a HTTP server. Důležitou součástí simulátoru je integrace s překladačem GCC a nový parser assembleru, který umožňuje simulovat širokou škálu programů v jazyce C. Společně s množstvím předpřipravených příkladů to uživatelům umožňuje rychle experimentovat se složitějšími programy, než jaké by mohli nebo chtěli psát ručně. Simulátor také sbírá běhové statistiky a je možné definovat data, nad kterými má program pracovat.

Bylo také provedeno mnoho změn vnitřního fungování simulátoru. Projekt obsahuje množství testů, dokumentace a skriptů k instalaci, spuštění a nasazení do provozu. Příloha A uvádí tabulku s přehledem jednotlivých modulů aplikace a můj přínos.

Aplikace je v současné podobě dobře provozuschopná, o čemž svědčí kladný ohlas z uživatelského dotazníku. Na webové stránce je možné provozovat interaktivní simulaci, měnit konfiguraci, editovat kód a prohlížet statistiky o simulaci. Grafické zpracování a provázání prvků v simulátoru přináší studentům přehledný zdroj k výuce fungování moderních procesorů.

Přínos projektu pro výuku hardwaru vnímám velmi pozitivně. Doufám, že aplikace najde své využití nejen na FIT VUT v Brně, ale i v široké studentské komunitě.

Budoucí práce se může zaměřit na další rozšíření instrukční sady, například na vektorové instrukce. Dalším zajímavým směrem může být možnost programovatelného přerušení simulace v určitém bodě (*breakpoint*).

Literatura

[1] *The Transport Layer Security (TLS) Protocol Version 1.3* online. 2018. Dostupné z: https://datatracker.ietf.org/doc/html/rfc8446. Staženo: 1.5.2024.

[2] *HTML Living Standard*. 2024. Dostupné z: https://html.spec.whatwg.org.

[3] Barth, A. *HTTP State Management Mechanism*. RFC 6265. 2011. Dostupné z: https://httpwg.org/specs/rfc6265.html#top.

[4] Cheng, K. a Clarke, J. *RISC-V ABIs Specification, Document Version 1.0*. RISC-V Internationa, 2022.

[5] Dhakad, K. Adopting Continuous Integration Practices to Achieve Quality in DevOps. *International Journal of Advanced Research in Science, Communication and Technology*, 2023. Dostupné z: https://api.semanticscholar.org/CorpusID:256975037.

[6] Fielding, R.; Gettys, J. a Mogul, J. *RFC 2616, Hypertext Transfer Protocol – HTTP/1.1*. 1999. Dostupné z: http://www.rfc.net/rfc2616.html.

[7] Fog, A. *Optimization manuals: 4. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs* online. 2022. Dostupné z: https://www.agner.org/optimize. Staženo: 2.5.2024.

[8] Horký, J. *Grafický simulátor superskalárních procesorů*. Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.vut.cz/www_base/zav_prace_soubor_verejne.php?file_id=252467.

[9] Jaroš, J. *Studijní materiály předmětu AVS*. 2023.

[10] Jordan, P. Human factors for pleasure in product use. *Applied Ergonomics*, 1998, sv. 29, č. 1, s. 25–33. ISSN 0003-6870. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0003687097000227.

[11] Kaaresoja, T.; Brewster, S. a Lantz, V. Towards the Temporally Perfect Virtual Button: Touch-Feedback Simultaneity and Perceived Quality in Mobile Touchscreen Press Interactions. New York, NY, USA: Association for Computing Machinery, jun 2014, sv. 11, č. 2. ISSN 1544-3558. Dostupné z: https://doi.org/10.1145/2611387.

[12] Moshovos, A. *Memory dependence prediction*. 1998. Disertační práce. PhD thesis, University of Wisconsin-Madison. Dostupné z: https://ftp.cs.wisc.edu/sohi/theses/moshovos.pdf.

[13] Patterson, D. A. *Computer Architecture: A Quantitative Approach*. 5. vyd. Morgan Kaufmann, 2011. ISBN 978-8178672663.

[14] Patterson, D. A. a Hennessy, J. L. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. 5. vyd. Morgan Kaufmann Publishers Inc., 2013. ISBN 0124077269.

[15] Rieman, J.; Franzke, M. a Redmiles, D. Usability Evaluation with the Cognitive Walkthrough. In: *Conference Companion on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, 1995, s. 387–388. CHI '95. ISBN 0897917553. Dostupné z: https://doi.org/10.1145/223355.223735.

[16] Schenkman, B. a Jönsson, F. *Aesthetics and preferences of Web pages* online. 2000. Dostupné z: https://www.researchgate.net/publication/228802769_Aesthetics_and_preferences_of_Web_pages. Staženo: 2.5.2024.

[17] Smith, J. E. A Study of Branch Prediction Strategies. In: *Proceedings of the 8th Annual Symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society Press, 1981, s. 135–148. ISCA '81.

[18] Vercel, Inc.. *Dokumentace softwaru Next.js* online. 2024. Dostupné z: https://nextjs.org/docs. Staženo: 1.5.2024.

[19] Vávra, J. *Graphical Simulator of Superscalar Processors*. Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis-file/21991/21991.pdf.

[20] Waterman, A.; Lee, Y. a Patterson, D. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.1*. EECS Department, University of California, Berkeley, 2016.

[21] Wehner, N.; Amir, M.; Seufert, M.; Schatz, R. a Hossfeld, T. A Vital Improvement? Relating Google's Core Web Vitals to Actual Web QoE. In: *2022 14th International Conference on Quality of Multimedia Experience (QoMEX)*. 2022, s. 1–6.

[22] Yeh, T.-Y. a Patt, Y. N. Two-Level Adaptive Training Branch Prediction. In: *Proceedings of the 24th Annual International Symposium on Microarchitecture*. New York, NY, USA: Association for Computing Machinery, 1991, s. 51–61. MICRO 24. ISBN 0897914600. Dostupné z: https://doi.org/10.1145/123465.123475.

Příloha A

Přehled převzaté práce

V této příloze je uveden rozpis jednotlivých modulů výsledného softwaru a míra, do které jsem k implementaci modulu přispěl.

Modul	Původní	Refaktorizován	Nový
Rozhraní procesoru			✓
Registry a registrové pole			✓
Fáze Fetch, Decode		✓	
ROB		✓	
Předpověď skoků		✓	
Fáze Execute		✓	
Reprezentace instrukcí		✓	
Interpretace instrukcí			✓
Paměti a cache		✓	
Alokace polí v paměti			✓
Parsování a překlad kódu			✓
Nastavení simulace		✓	
Sběr statistik		✓	
Server, serializace			✓
Instrukční sada RISC-V		✓	
Testy		✓	
Webové rozhraní			✓
Rozhraní příkazové řádky			✓
Kontejnerizace			✓
Příklady programů a konfigurace		✓	

Tabulka A.1: Přehled modulů a můj přínos implementaci.

Příloha B

Statistiky sbírané při simulaci

V této příloze je uvedena tabulka jednotlivých metrik sbíraných v rámci simulace činnosti procesoru. Použité datové typy jsou z jazyka Java. Statistiky jsou konzumovány ve formátu JSON, kde tyto datové typy mají svůj ekvivalent.

Statistika	Datový typ	Popis pole
staticInstructionMix	InstructionMix	Uchovává statický instrukční mix
dynamicInstructionMix	InstructionMix	Uchovává dynamický instrukční mix
cache	CacheStatistics	Statistiky cache
fuStats	Map<String, FUStats>	Statistiky funkčních jednotek
instructionStats	List<InstructionStats>	Statistiky pro každou instrukci
committedInstructions	long	Počet vydaných instrukcí
clockCycles	long	Počet taktů výpočtu
flushedInstructions	long	Počet zahozených instrukcí z důvodu špatné predikce skoku
robFlushes	long	Počet vypláchnutí ROB
clock	int	Frekvence hodin (Hz)
correctlyPredictedBranches	long	Počet správně předpovězených skoků
conditionalBranches	long	Počet podmíněných skoků
takenBranches	long	Počet provedených skoků
mainMemoryLoadedBytes	long	Množství dat přenesených z hlavní paměti
mainMemoryStoredBytes	long	Množství dat přenesených do hlavní paměti
maxAllocatedRegisters	int	Maximální počet přidělených spekulativních registrů

Tabulka B.1: Hlavní objekt statistiky.

Statistika	Datový typ	Popis pole
intArithmetic	int	Počet celočíselných aritmetických operací
floatArithmetic	int	Počet operací s pohyblivou desetinnou čárkou
memory	int	Počet paměťových operací
branch	int	Počet instrukcí skoku
other	int	Počet ostatních instrukcí

Tabulka B.2: Objekt pro popis statického nebo dynamického instrukčního mixu.

Statistika	Datový typ	Popis pole
readAccesses	int	Počet čtení z cache
writeAccesses	int	Počet zápisů do cache
hits	int	Počet úspěšných přístupů do cache
misses	int	Počet neúspěšných přístupů do cache
totalDelay	int	Celkové zpoždění způsobené přístupy do cache
bytesWritten	int	Počet bytů zapsaných do cache
bytesRead	int	Počet přečtených bytů z cache

Tabulka B.3: Statistiky týkající se cache.

Příloha C

Argumenty simulátoru

Níže jsou uvedeny argumenty příkazové řádky pro simulátor. Argumenty jsou rozděleny do dvou skupin: argumenty pro spouštění v příkazové řádce (tabulka C.1) a argumenty pro spouštění serveru (tabulka C.2).

Spuštění programu se skládá z názvu programu, režimu (`cli` nebo `server`) a argumentů. Příklady spuštění jsou uvedeny v kódu C.1. Příklady jsou v ukázce spouštěny ze složky `Sources/simulator`. V případě jiného způsobu instalace je nutné nahradit `./scripts/run.sh` jinou cestou.

```
./scripts/run.sh cli \
  --cpu examples/cpuConfigurations/default.json \
  --program examples/asmPrograms/basicLoadStore.r5
# nebo
./scripts/run.sh server
```

Obrázek C.1: Příklad spuštění simulátoru v režimech cli a server.

Argument	Povinný	Popis
`-entry=LABEL|ADDRESS`	Ne	Adresa nebo návěští počátku programu.
`-pretty`	Ne	Formátování JSON výstupu.
`-full-state`	Ne	Plný výstup. Ve výchozím režimu je výstup zkrácený.
`-cpu=<soubor>`	Ano	Konfigurace procesoru.
`-program=<soubor>`	Ano	Program v assembleru RISC-V.
`-memory=<soubor>`	Ne	Konfigurace paměti.

Tabulka C.1: Argumenty příkazu `cli`, který spouští simulaci z příkazové řádky. Argumenty s parametrem `<soubor>` očekávají cestu k souboru.

Argument	Povinný	Výchozí hodnota	Popis
`-gcc-path=PATH`	Ne	–	Cesta k překladači GCC
`-host=ADDR`	Ne	0.0.0.0	Adresa, ke které se má server připojit
`-port=PORT`	Ne	8 000	Port, ke kterému se má server připojit
`-timeout-ms=NUMBER`	Ne	30 000	Časový limit pro požadavky v milisekundách

Tabulka C.2: Argumenty příkazu `server`, který spouští HTTP server.

Příloha D

Konfigurace procesoru

Následující tabulky popisují parametry procesoru. Tyto parametry jsou přijímány při požadavku na simulaci jak v CLI tak HTTP režimu simulátoru. Konkrétní příklady konfigurací ve formátu JSON jsou k nalezení v přiloženém kódu.

Tabulka 1: Konfigurace bufferu

Parametr	Typ	Přijatelné hodnoty	Popis
`robSize`	int	1 až 1024	Maximální počet instrukcí, které mohou být v ROB.
`commitWidth`	int	1 až 10	Počet instrukcí, které mohou být potvrzeny během jednoho cyklu - commitLimit v ROB.
`flushPenalty`	int	0 až 100	Počet cyklů hodin, které CPU potřebuje k vyprázdnění pipeline.
`fetchWidth`	int	1 až 10	Počet instrukcí, které lze načíst během jednoho cyklu.

Tabulka 2: Konfigurace predikce

Parametr	Typ	Přijatelné hodnoty	Popis
`useGlobalHistory`	boolean	true nebo false	Použití globální historie ve vektoru PHT.
`btbSize`	int	1 až 16384	Velikost bufferu cílů větve.
`phtSize`	int	1 až 16384	Velikost tabulky historie vzoru.
`predictorType`	String	"ZERO_BIT_PREDICTOR", "ONE_BIT_PREDICTOR", "TWO_BIT_PREDICTOR"	Typ prediktoru uloženého v PHT.
`predictorDefault`	int	Pro "ZERO_BIT_PREDICTOR": 0, 1. Pro "ONE_BIT_PREDICTOR": 0 až 3. Pro "TWO_BIT_PREDICTOR": 0 až 3.	Počáteční stav všech prediktorů.

Tabulka 3: Konfigurace funkčních jednotek

Parametr	Typ	Přijatelné hodnoty	Popis
`fUnits`	Pole popisu `FunctionalUnitDescription`	Not null, Not empty	Seznam jednotek FU nesmí být null a musí obsahovat alespoň jednu jednotku FU.

Tabulka 4: Konfigurace funkční jednotky FX/FP[1]

Parametr	Typ	Přijatelné hodnoty	Popis
`fuType`	Výčet (FX, FP)	FX nebo FP	Typ funkční jednotky (FX nebo FP).
`operations`	Pole `Capability`	Viz Tabulka X	Seznam schopností pro jednotky FX/FP.
`latency`	int	Nezáporná celá čísla	Latence zálohy (používá se pro typové převody).

Tabulka 5: Parametr `Capability`

Parametr	Typ	Přijatelné hodnoty	Popis
`name`	`Capability`	addition, bitwise, multiplication, division, special	Název schopností pro jednotky FX/FP.
`latency`	int	Nezáporná celá čísla	Latence pro danou schopnost.

[1] Jedná se o třídu `FunctionalUnitDescription`

Tabulka 6: Konfigurace funkčních jednotek L_S, Branch, Memory[2]

Parametr	Typ	Přijatelné hodnoty	Popis
`fuType`	Výčet (L_S, Branch, Memory)	L_S, Branch, nebo Memory	Typ funkční jednotky.
`latency`	int	Nezáporná celá čísla	Latence pro funkční jednotku.

Tabulka 7: Konfigurační parametry pro mezipaměť CPU

Parametr	Typ	Přijatelné hodnoty	Popis
`useCache`	boolean	true, false	Povolit nebo zakázat mezipaměť jedné úrovně.
`cacheLines`	int	1 až 65536	Počet cache linek.
`cacheLineSize`	int	mocnina čísla 2 mezi 1 a 512	Velikost jedné cache linky v bytech (musí být mocninou čísla 2).
`cacheAssoc`	int	1 nebo více*	Asociativita cache (1 pro přímý mapovaný).
`cacheReplacement`	String	"Random", "LRU", "FIFO"	Politika nahrazení cache.
`storeBehavior`	String	"write-back", "write-through"	Politika zápisu do cache.
`laneReplacementDelay`	int	Nezáporné celé číslo	Cykly pro nahrazení cache linky.
`cacheAccessDelay`	int	Nezáporné celé číslo	Zpoždění přístupu do cache v cyklech.

Tabulka 8: Konfigurační parametry pro paměť CPU

Parametr	Typ	Přijatelné hodnoty	Popis
`lbSize`	int	1 až 1024	Velikost zásobníku pro načítání.
`sbSize`	int	1 až 1024	Velikost zásobníku pro ukládání.
`storeLatency`	int	Nezáporné celé číslo	Latence hlavní paměti pro operace ukládání.
`loadLatency`	int	Nezáporné celé číslo	Latence hlavní paměti pro operace načítání.
`callStackSize`	int	0 až 65536	Velikost zásobníku volání v bytech.
`speculativeRegisters`	int	1 až 1024	Počet spekulativních registrů.

Tabulka 9: Konfigurační parametry pro hodiny CPU

Parametr	Typ	Přijatelné hodnoty	Popis
`coreClockFrequency`	int	kladné celé číslo	Frekvence jádra procesoru v Hz.
`cacheClockFrequency`	int	kladné celé číslo	Frekvence mezipaměti procesoru v Hz.

[2] Jedná se o třídu `FunctionalUnitDescription`

Příloha E

Pokyny pro spuštění aplikace

Tato příloha obsahuje pokyny ke spuštění kontejnerů Docker a manuální instalaci aplikace.

E.1 Spuštění kontejnerů Docker

Doporučeným a nejjednodušším způsobem provozu aplikace je pomocí připravených skriptů pro Docker (viz výpisy E.1 a E.2). Jediným požadavkem je zde Docker, `docker compose` a internetové připojení.

```
1 cd Sources && ./build_container.sh && ./run_container.sh
```

Výpis E.1: Vytvoření a spuštění celé aplikace.

```
1 ./stop_container.sh
```

Výpis E.2: Zastavení obou kontejnerů.

E.2 Manuální instalace

Pro detailní instrukce k manuální instalaci nahlédněte do souborů `Sources/Readme.md` přiložených v médiu.

Požadavky pro provoz aplikace:

- Java verze 17,
- npm verze 10.2.3,
- Node.js verze 21.2.0,
- GCC verze 12.3.0 pro RISC-V (pouze pro simulátor v režimu serveru).

Instrukce pro instalaci jsou uvedeny ve výpisu E.3, příklad spuštění ve výpisu E.4.

```
1 cd Sources/simulator
2 ./scripts/install.sh
3 ./scripts/run.sh help
```

Výpis E.3: Posloupnost příkazů k instalaci a spuštění simulátoru.

```
1 ./scripts/run.sh cli --cpu=examples/cpuConfigurations/default.json --program=examples/
      asmPrograms/basicFloatArithmetic.r5 --pretty
```

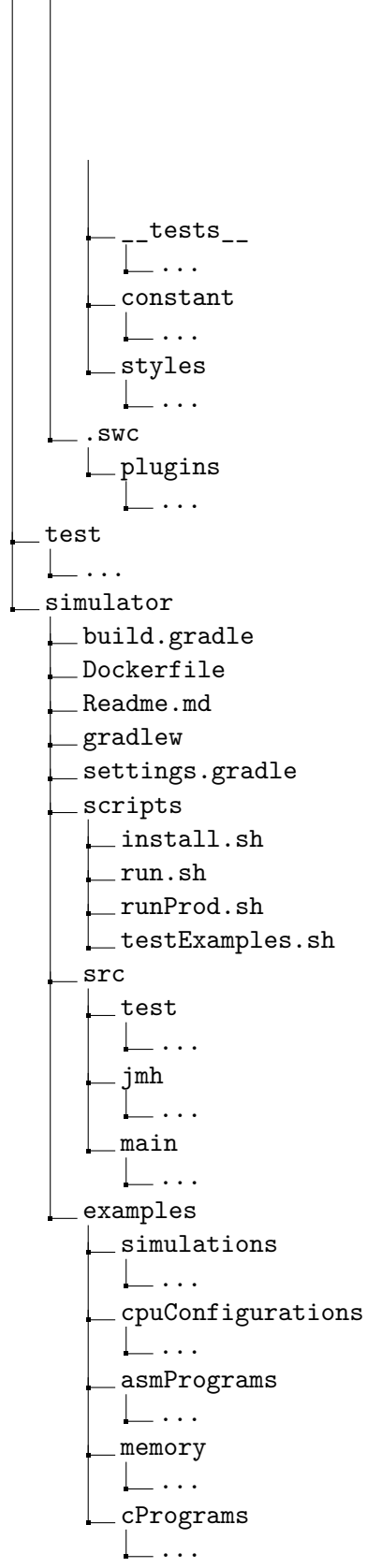
Výpis E.4: Příklad supštění simulace v režimu CLI.

Příloha F

Struktura zdrojových souborů

Soubory přiložené v médiu obsahují projekty, zdrojový kód obou aplikací a kontejnerizační skripty (složka `Sources`) a dokumentaci (`Readme.md`, `Sources/frontend/Readme.md` a `Sources/simulator/Readme.md`).

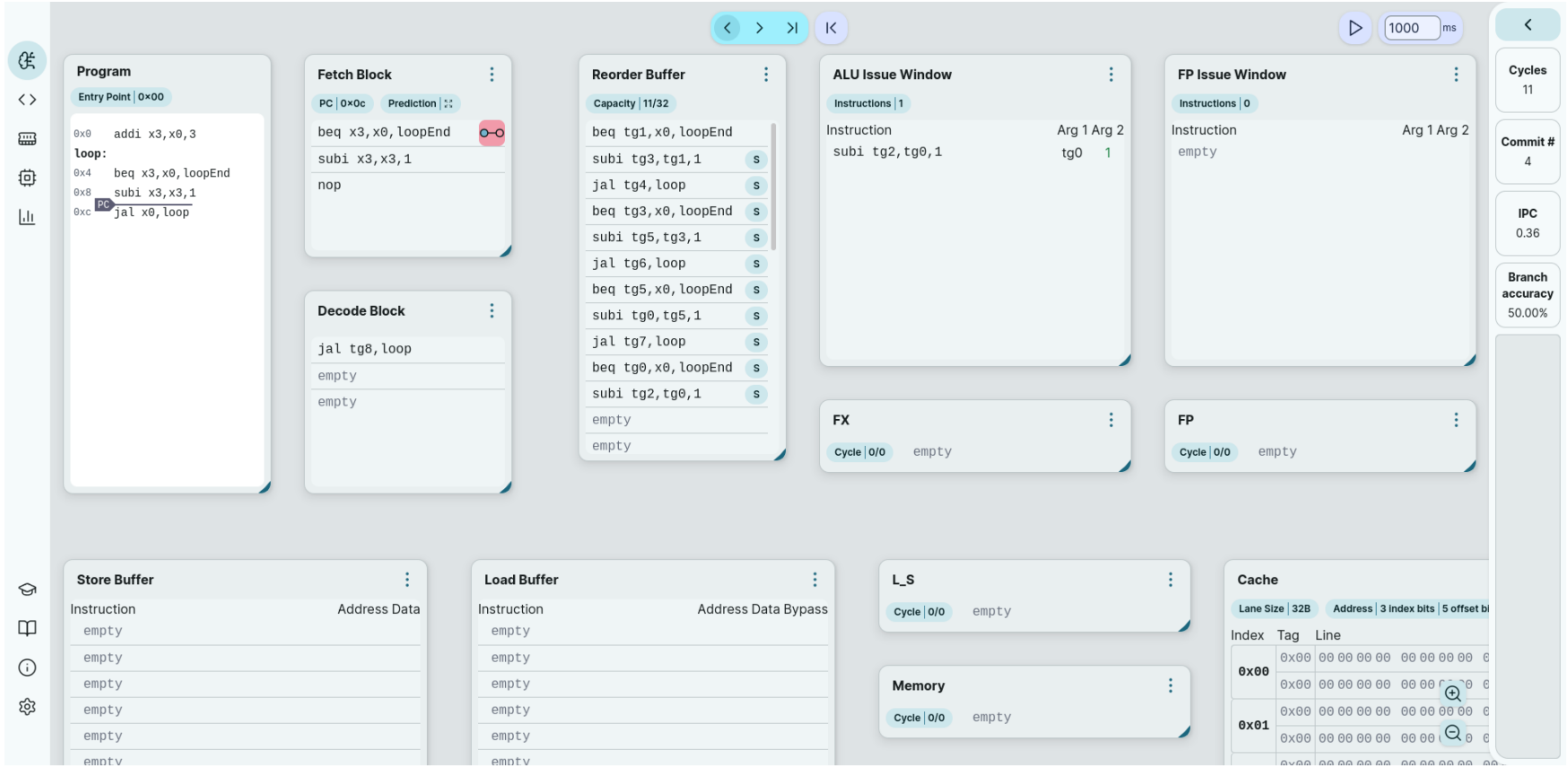
```
├── Readme.md
├── .gitlab-ci.yml
├── .gitlab
├── Literature
│   └── Web_Based_Simulator_Of_Superscalar_Processors.pdf
├── Thesis
└── Sources
    ├── docker-compose.yml
    ├── build_container.sh
    ├── stop_container.sh
    ├── run_container.sh
    └── frontend
        ├── tailwind.config.ts
        ├── package-lock.json
        ├── Dockerfile
        ├── .env.example
        ├── tsconfig.json
        ├── Readme.md
        ├── biome.json
        ├── package.json
        ├── next-env.d.ts
        ├── .gitignore
        ├── next.config.mjs
        ├── public
        └── src
            ├── app
            │   └── ...
            ├── components
            │   └── ...
            └── lib
                └── ...
```


__tests__
...
constant
...
styles
...
.swc
plugins
...
test
...
simulator
build.gradle
Dockerfile
Readme.md
gradlew
settings.gradle
scripts
install.sh
run.sh
runProd.sh
testExamples.sh
src
test
...
jmh
...
main
...
examples
simulations
...
cpuConfigurations
...
asmPrograms
...
memory
...
cPrograms
...

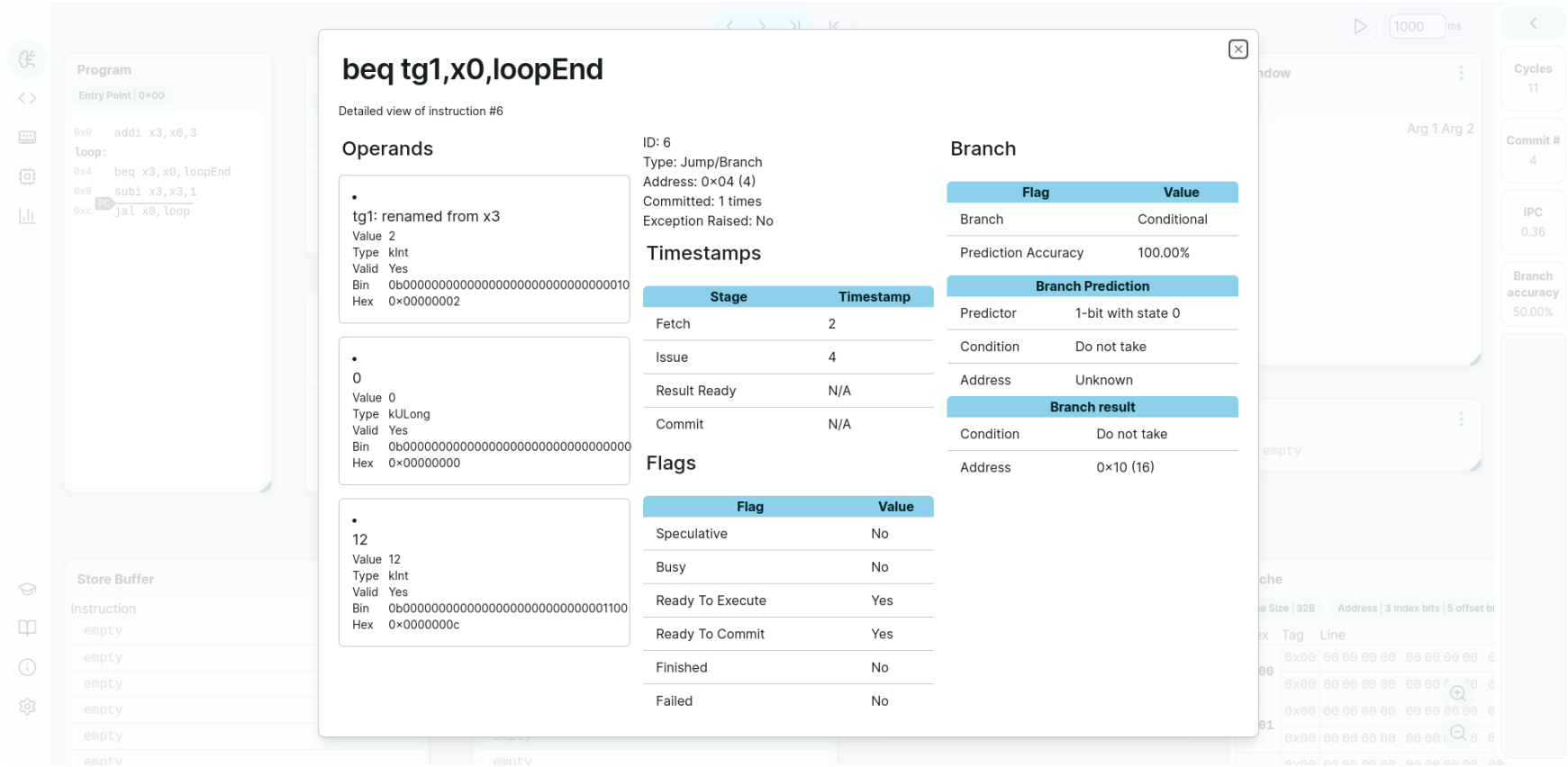
Příloha G

Galerie webové aplikace

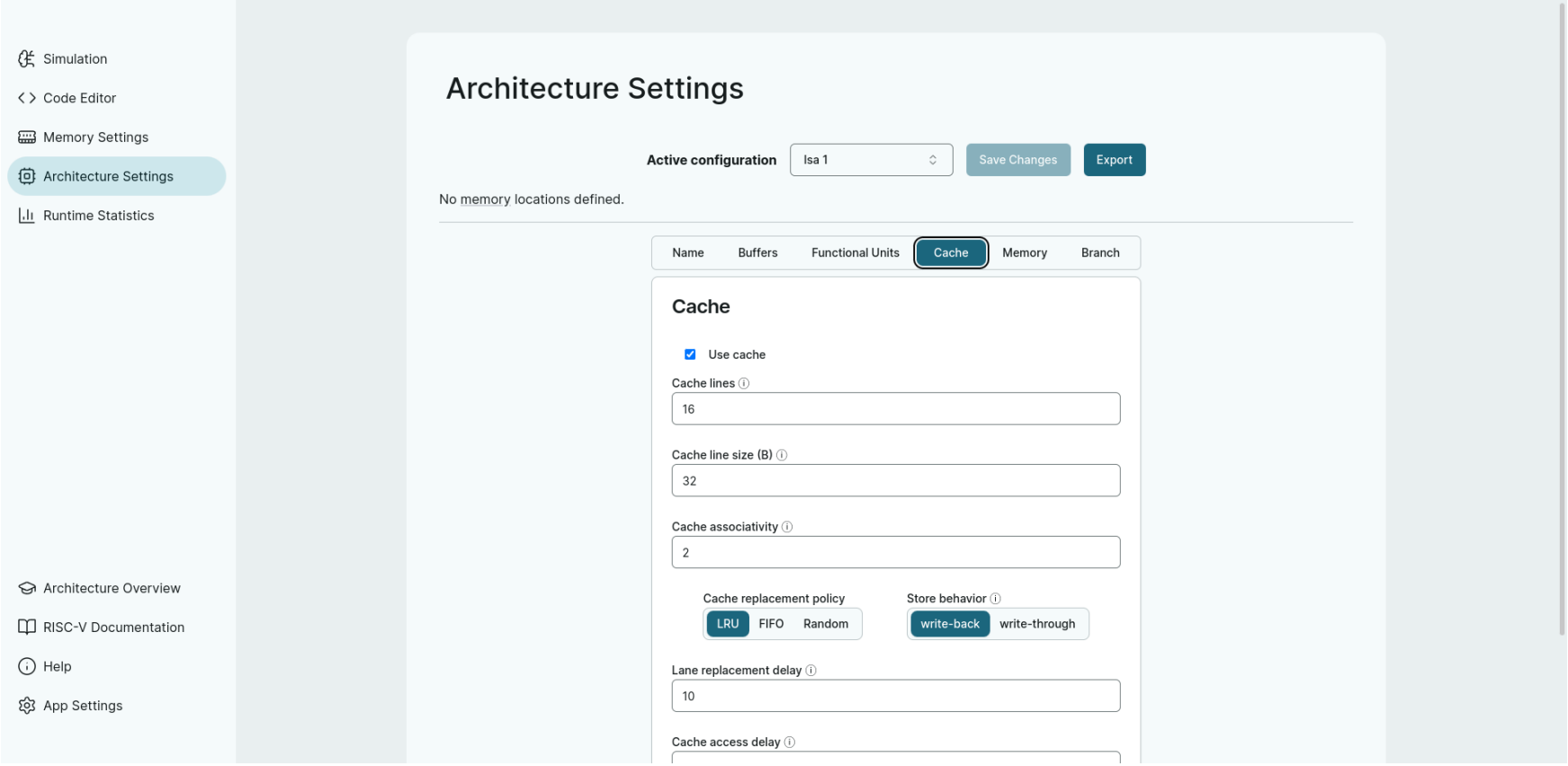
Obrázek G.1: caption



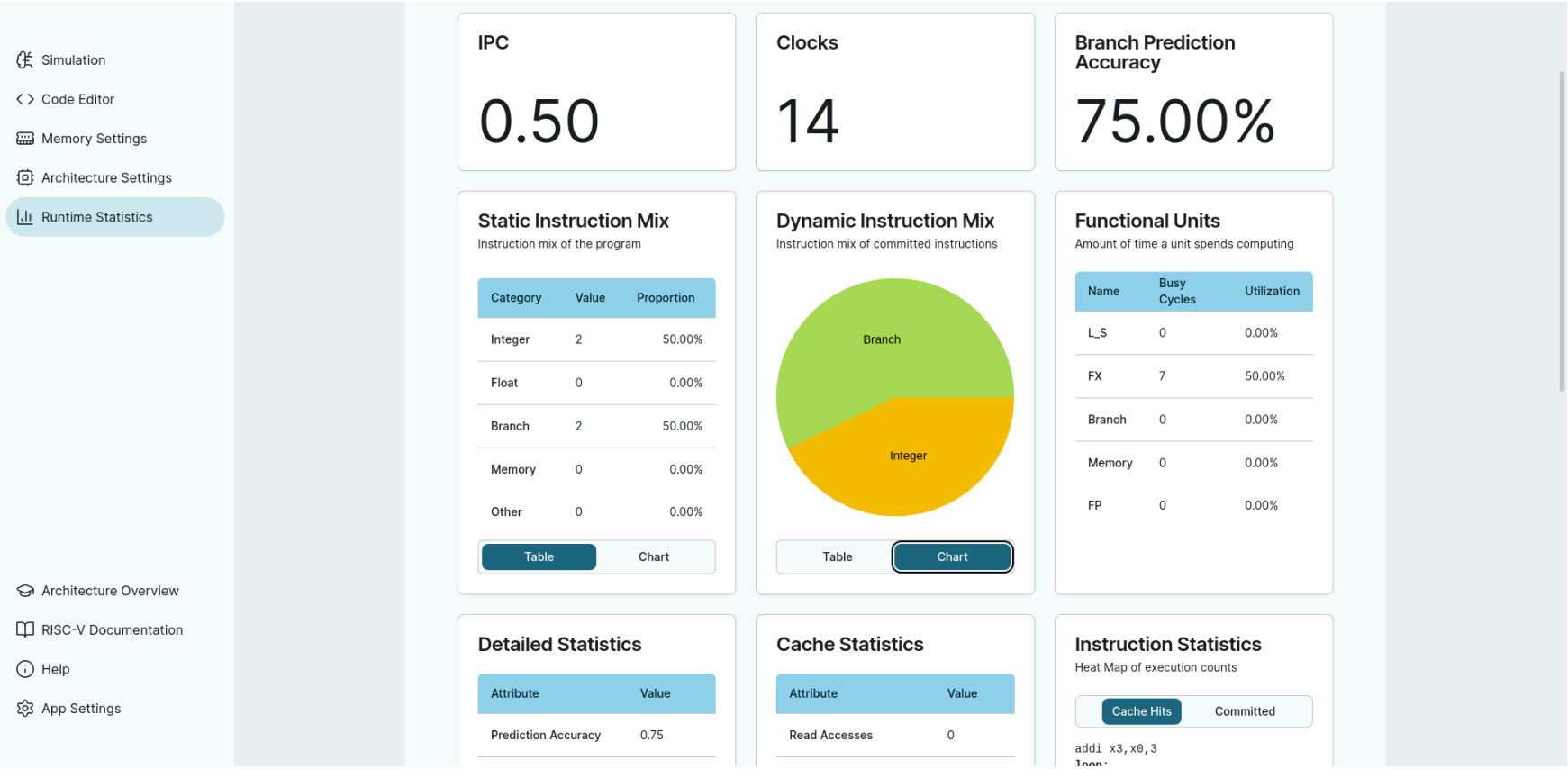
Obrázek G.2: Hlavní okno aplikace – procesor.



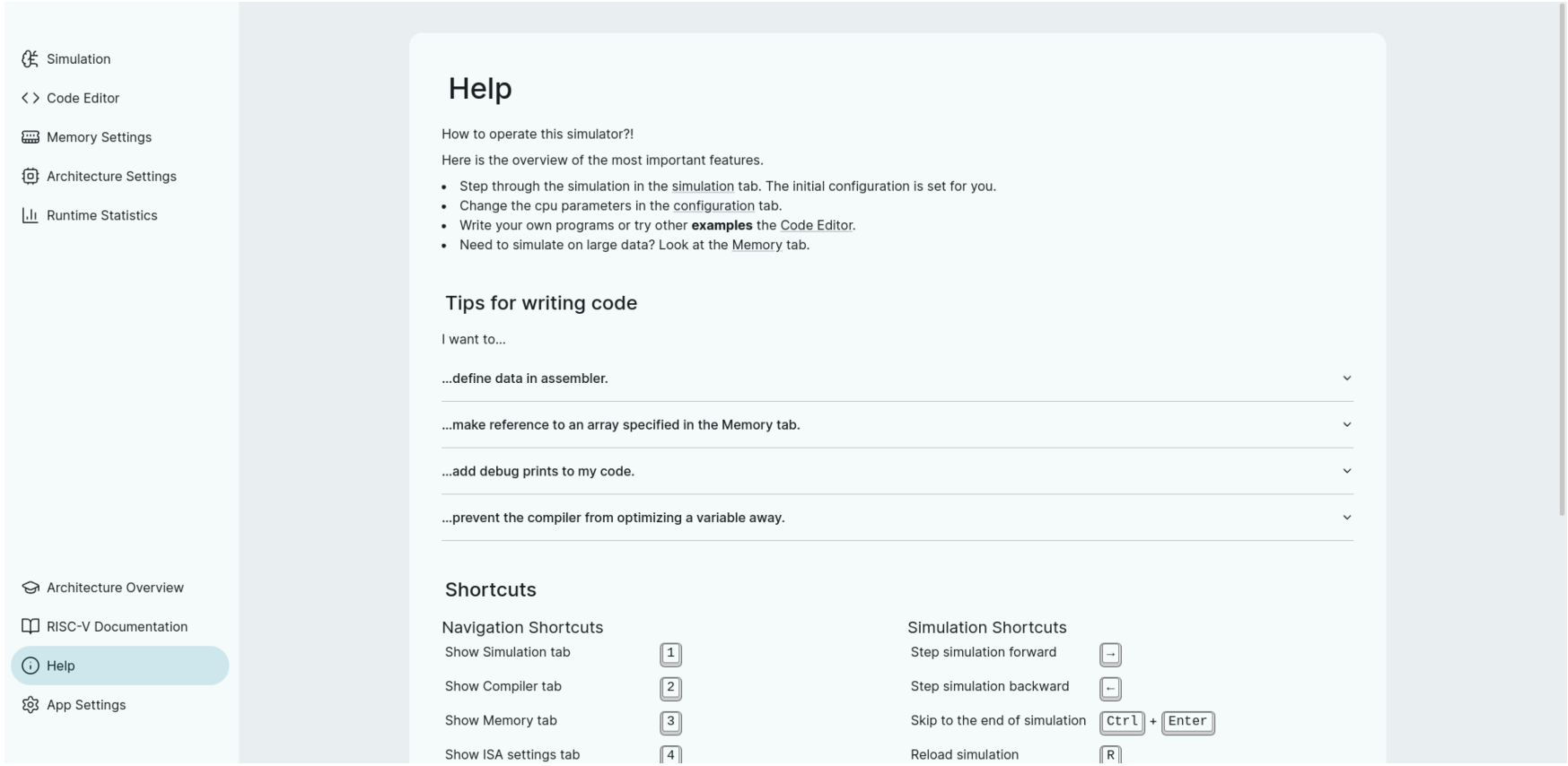
Obrázek G.3: Detail konkrétní instrukce.



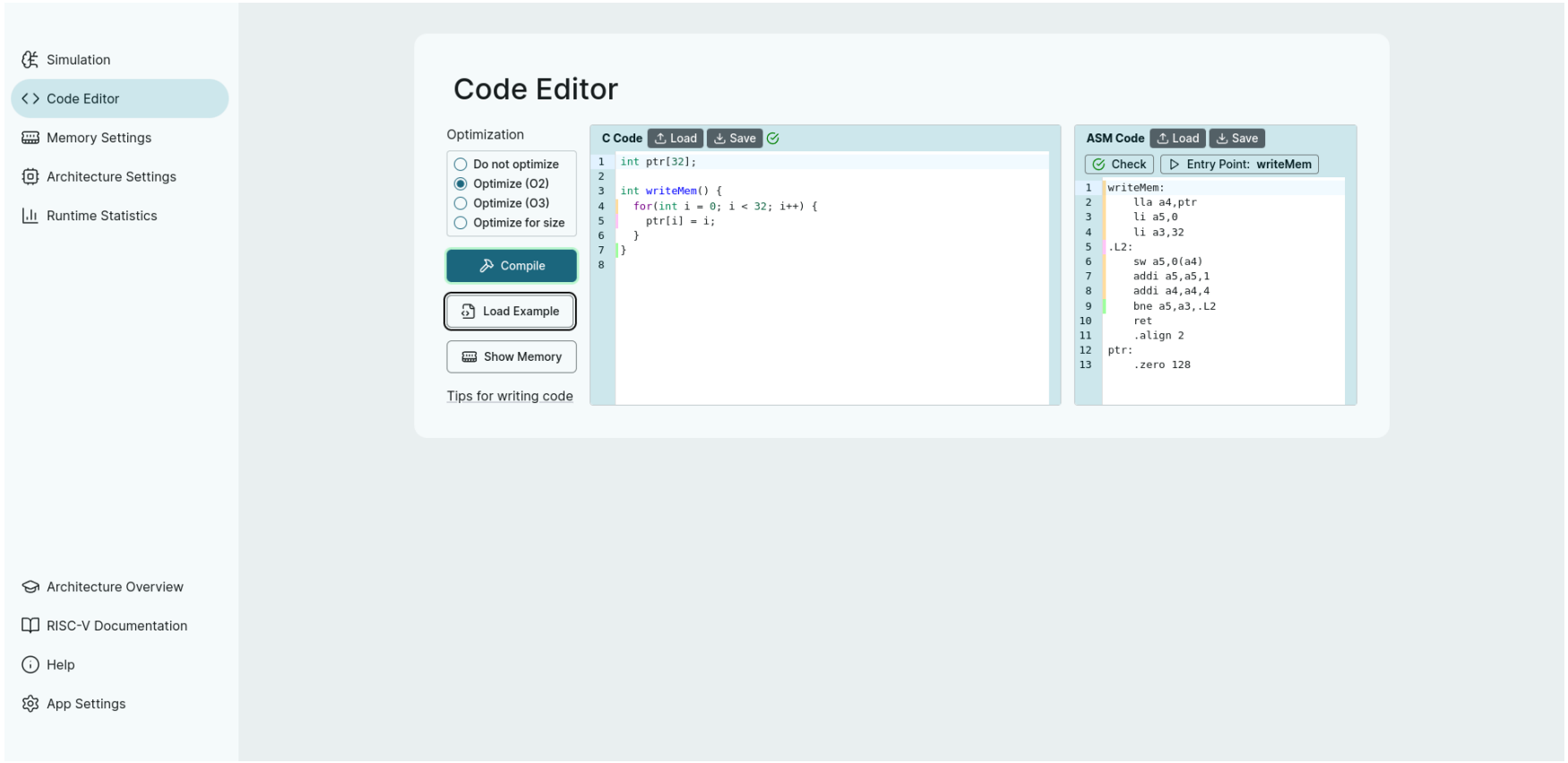
Obrázek G.4: Konfigurační menu.



Obrázek G.5: Běhové statistiky.



Obrázek G.6: Návod k použití.



Obrázek G.7: Editor kódu.