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

Abstrakt

Práce se zabývá rozšířením simulátoru superskalárního procesoru o webové a programátorské rozhraní. Jsou prozkoumány současné způsoby efektivní implementace procesorů. Uvedu moderní postupy při vývoji webových aplikací. Dále práce prozkoumává existující simulátory, navrhuje vylepšení simulátoru Jana Vávry a Jakuba Horkého.

V poslední části práce je popsána implementace, jsou zhodnoceny přínosy pro použitelnost výsledné aplikace při výuce.

Abstract

This thesis deals with the extension of the superscalar processor simulator with a web and programming interface. Current methods of efficient processor implementation are explored. Modern practices in web application development are presented. Furthermore, the thesis explores existing simulators, proposing improvements to the simulator of Jan Vávra and Jakub Horký.

In the last part of the thesis, the implementation is described, and the benefits for the usability of the resulting application in teaching are evaluated.

Klíčová slova

Simulátor, RISC-V, Superskalární procesor, webová aplikace

Keywords

Simulator, RISC-V, Superscalar processor, web application

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
28. ledna 2024

Poděkování

Rád bych vyjádřil poděkování svému vedoucímu práce, docentu Jarošovi, za jeho vedení a odbornost během celého procesu tvorby této práce. Jeho vedení pro mě bylo přínosné a klíčové pro dosažení úspěchu.

Obsah

Seznam obrázků

Kapitola 1

Úvod

V minulých letech na Fakultě Informačních Technologií VUT v Brně vznikl výukový simulátor superskalárního procesoru pro potřeby předmětu Architektury výpočetních systémů. Simulátor v současné době slouží jako interaktivní a názorná ukázka probíraných principů.

V této práci navážu na předchozí úsilí a pokusím se vylepšit dostupnost a přesnost simulátoru. Mým cílem je vylepšit rozhraní simulátoru, konkrétně vyvinout webovou aplikaci a rozhraní pro příkazovou řádku. Od nových rozhraní očekávám lepší názornost a tedy vetší užitek pro výuku.

V první části práce se zaměřím na principy, na kterých jsou moderní superskalární procesory založeny, především na ty, které jsou v simulátoru demonstrovány. Prozkoumám i metody tvorby webových aplikací a jejich nasazení do provozu.

Následně zanalyzuji dosavadní řešení a navrhnu jejich vylepšení a rozšíření.

V poslední části práce popíšu implementaci a zhodnotím přínos pro výuku.

Kapitola 2

Skalární procesor

Jádro procesoru představuje centrální prvek CPU, který provádí výpočet popsaný instrukcemi určité instrukční sady. Z abstraktního pohledu se procesor nachází v definovaném stavu a vykonáním každé instrukce se dostává do následujícího stavu. Tento stav je představován hodnotami v registrech. Společně se stavem hlavní paměti tvoří stav výpočtu. Instrukční sada tvoří *programátorův model CPU*.

Programátorův model CPU je abstraktní reprezentace funkcionality a chování procesoru. Je navržena tak, aby usnadnila programátorův proces psaní kódu. Příklady instrukční sady jsou RISC-V, nebo Intel 64.

Instrukční sada nepopisuje jak má vypadat implementace procesoru, ale pouze popisuje *prostředí a zdroje*, které program může využít. Této volnosti využívají implementace, které mohou zpracovávat více instrukcí současně, i mimo původní pořadí programu, případně i spekulativně. Návrháři instrukční [11]

V této kapitole popíšu různé techniky využívané v efektivních implementacích jader procesorů. Krátce se budu věnovat skalárním procesorům, ale většina kapitoly se bude věnovat přímo superskalárním procesorům.

2.1 Složky výpočetního jádra

Instrukční sada definuje počet instrukcí nutných k určitému výpočtu, implementace procesoru ale rozhoduje o jeho celkovém výkonu.

Jednoduchá implementace dělí datovou cestu do následujících stádií:

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 datová cesta je prováděna během jednoho cyklu. Stádia mohou sdílet hardware. Ne každá instrukce potřebuje všechny stádia, například uložení do paměti nemá poslední fázi zápisu do registru.

Tato stádia jsou vhodně zvolena, jejich provedení může být nezávislé. Jejich nezávislost dovoluje vydělit pro každou část speciální hardware a jejich výpočet *paralelizovat*. Vznikne řetězená linka. [12]

Bez řetězení by se musel zvolit takt procesoru podle nejpomalejší instrukce.

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.

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 dekódovány binární instrukce a převedeny na interní kódy, které CPU může vykonat. Tato fáze také zahrnuje identifikaci operačního kódu a přiřazení a načtení příslušných registru a operandů k provedení dané instrukce.

Při zpracovávání skokových instrukcí se zároveň se čtením počítá cílová adresa skoku. Podmínka je vypočtena ve fázi execute, čímž v případě, že se skok má uskutečnit, vznikne pokuta 1 cyklus.

U procesorů se zpracováním mimo pořadí zde dojde k přejmenování registrů. Architektury CISC v této fázi dekódují instrukci na jednu nebo více mikroinstrukcí. Dále zpracovávají mikroinstrukce.

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.

Paměťové instrukce zde vypočítají adresu pro přístup do paměti.

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

Tabulka 2.1: Vybrané instrukce a jejich latence v cyklech architektury Intel Ice Lake. Předpokládá se, že neproběhnou přístupy do paměti nevzniknou výjimky. [6]

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 $sin(x)$

Memory Access (MA)

Fáze "memory access"v CPU zahrnuje přístup k 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.

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

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. 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. Detailnější popis vyrovnávací paměti je uveden v sekci 2.2.

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.

Jsou zde řešeny výjimky, které jsou do této fáze propagovány. 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.

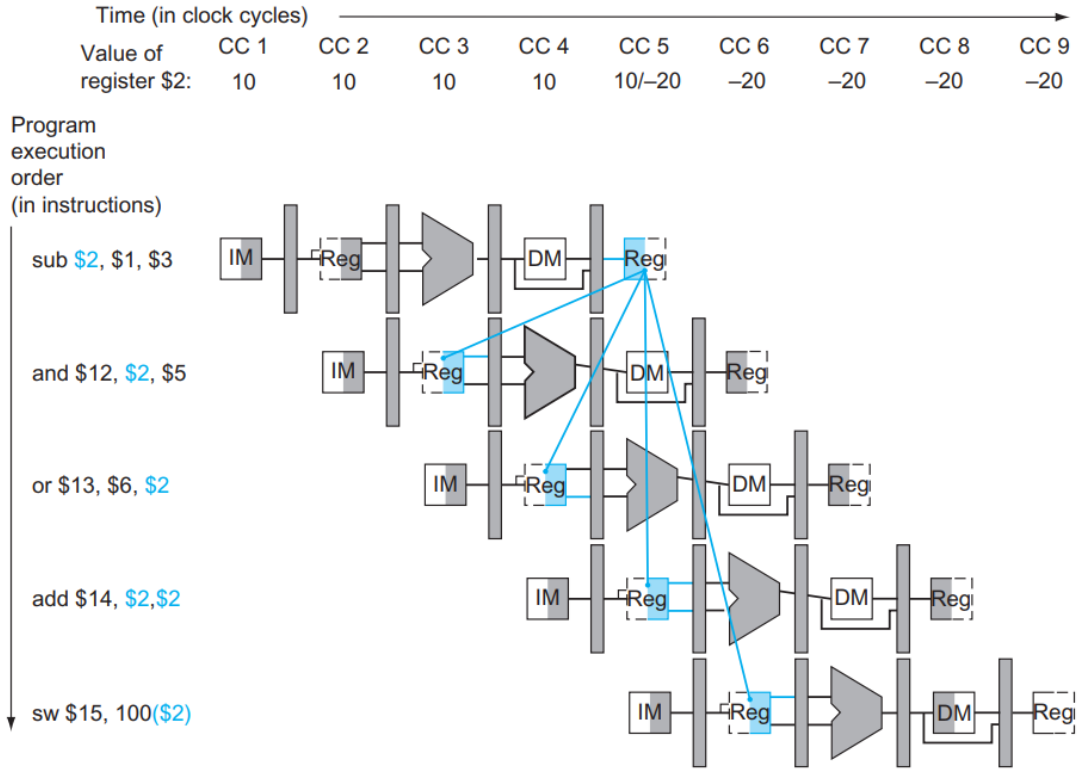
Kromě výše uvedených komplikací je v realitě propustnost snižována zastavováním linky (*stall*). Důvodem zastavování linky jsou *konflikty*. [12]

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. 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.



Obrázek 2.1: Sekvence instrukcí vykonávaných ve skalární lince. Zvýrazněné závislosti mezi registry ukazují, že hodnota `$2` čtená instrukcí `and` není výsledkem předchozí instrukce. [12]

2.1.3 Další části procesoru

Byly zmíněny hlavní komponenty procesoru, ty ale nemohou pracovat izolovaně. V jádře se především nachází řídící logika (control unit). Tato jednotka pomocí signálů ovládá datovou cestu, paměťová a vstupně-výstupní zařízení. Jednoduchá řídící logika může být implementována jako konečné stavové řízení. [12]

Modul Memory Access bývá napojen na hlavní paměť, cache a tabulku stránek. Write Back je napojen na registrové pole.

Další části jádra procesoru slouží k optimalizaci těchto základních funkcí. Prediktory budou probrány v sekci 3.4.

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.

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. [11]

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.

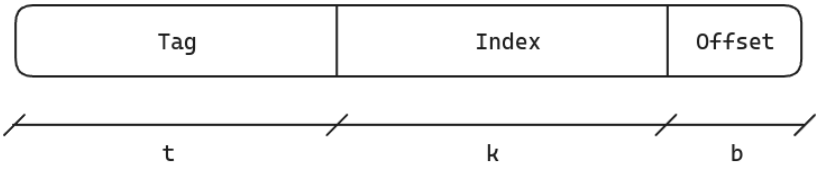
Zajímavé jsou krajní případy. Pokud $m = 1$, potom se cache nazývá přímo mapovaná. Pokud m odpovídá kapacitě cache, potom se cache nazývá plně asociativní.

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.



Obrázek 2.2: Adresa paměti a její části. Prvních `t` bitů je použito jako *tag* k vyhledání v rámci skupiny. Následujících `k` bitů adresuje skupinu a posledních `b` bitů adresuje obsah bloku.

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. [11]

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.

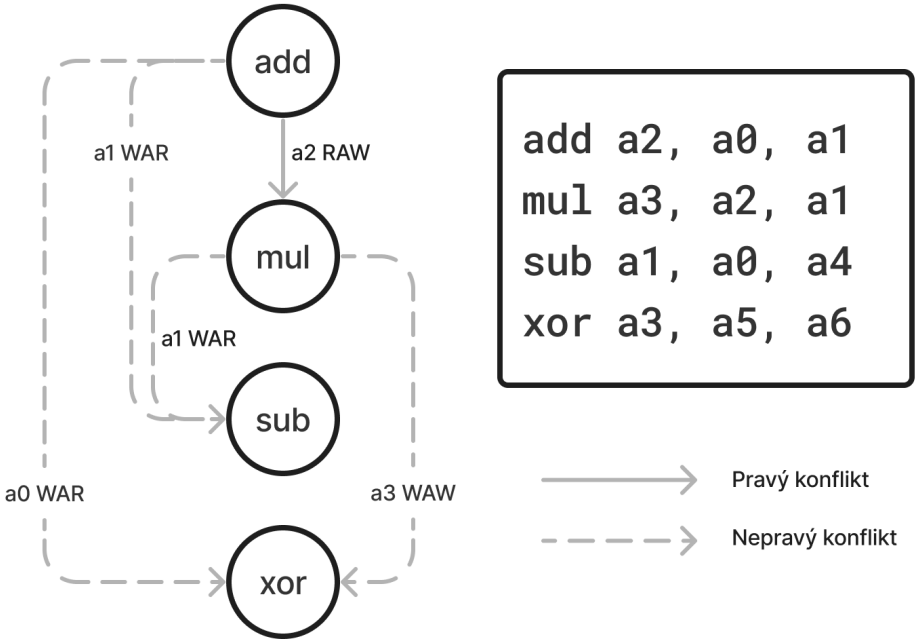
3.1 Konflikty

Kapitola o skalárních procesorech 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. 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.



Obrázek 3.1: Program a jeho datové závislosti zobrazené jako graf. Nepravé konflikty jsou zobrazeny přerušovanou čarou, pravé konflikty plnou čarou. Jediná pravá závislost je mezi instrukcemi `add` a `mul`, ostatní dvojice instrukcí mohou být vykonány v libovolném pořadí (za předpokladu přejmenování registrů).

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.

Pokud dojde ke zpracovávání nesprávné instrukce, musí být z linky odstraněna.

Linka je zastavena, dokud není vypočítána adresa následující instrukce, proto je žádoucí tento výpočet urychlit. Techniky pro eliminaci pokut skokových instrukcí zahrnují predikci skoků a předsazení výpočtu podmínky v lince.

3.1.2 Strukturní konflikty

Ke strukturnímu konfliktu dochází, pokud vykonání dvou instrukcí vyžaduje stejný prostředek. Prostředkem je myšlen hardwarový modul, například funkční jednotka, 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. Tento koncept uvádím v kapitole o superskalárních procesorech, protože jsou to úzce spojené koncepty.

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*. 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 vydání 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. 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,
- *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.

```
add x2, x1, x1
sub x2, x2, x3
mul x4, x5, x2
shr x5, x1, x4
```

(a) Původní jména operandů.

```
add t0, x1, x1
sub t1, t0, x3
mul t2, x5, t1
shr t3, x1, t2
```

(b) Jména operandů po přejmenování.

Obrázek 3.2: Přejmenování registrů. Vstupní registry používají nejaktuálnější přejmenování. Výstupní registry vytvoří nové přejmenování. Přejmenování probíhá v pořadí programu.

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,

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

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 klasickým výpočtem se zkontroluje, zda byla predikce správná.

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 o předvídání skoků 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. [10]

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. [15]

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. [15]

Další strategie mohou využít směru skoku (skok dopředu nebo dozadu), nebo vzdálenosti skoku. Směr skoku je zajímavým ukazatelem, protože v mnoha programech velkou část

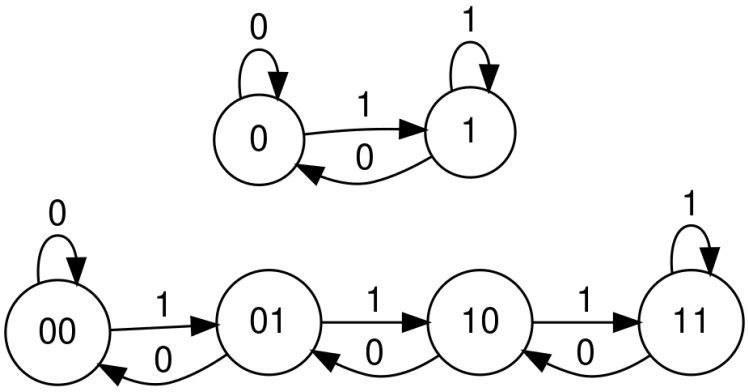
skoků směrem zpět tvoří smyčky, které typicky provádějí velký počet iterací. Nevýhodou je, že předpověď cílové adresy a podmínky nemůže být provedena paralelně.

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. [12]

Dynamické prediktory se umí naučit různé vzory. 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. Větší automaty nejsou běžné. Stavy představují saturační čítač. Z aktuálního stavu je odvozena předpověď (skočit nebo neskočit).



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

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. [19]

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. [17]

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 instrukčních slov. Typicky se jedná o minimální množinu instrukcí, ve které je možné popsat výpočty a práci s hardwarem. Do instrukční sady se dostávají i další instrukce[1], často z důvodu výkonu.

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.

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.

[1] Zajímavostí 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.

Tabulka 4.1: Příklady pseudoinstrukcí a jejich odpovídajících reálných instrukcí. Pseudoinstrukce lépe vyjadřují význam operace, zároveň nijak nezatěžují hardware. Často obsahují implicitní argument.

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

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[2]. 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*.

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

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

obsloužení jádrem. Obdobně jako instrukce `SYSCALL` z ISA x86, argumenty jsou definovány podle používaného ABI.

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.

Tabulka 4.2: Nejvýznamnější rozšíření instrukční sady RISC-V. Celý výčet je k dispozici ve specifikaci RISC-V [17].

Zkratka	Popis rozšíření
M	Celočíselné násobení a dělení
A	Atomické instrukce
F	podpora čísel *single* podle standardu IEEE 754-2008[3]
D	podpora čísel *double* podle standardu IEEE 754-2008

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 "RVWMO"(*RISC-V Weak Memory Ordering*). 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 ten, že uživatelé mohou začít aplikaci okamžitě používat. Vývoj webové aplikace je také rychlejší a levnější.

S rostoucí popularitou, implementací nových standardů a vývojem rámcových řešení podíl webových aplikací stále roste.

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. Tento protokol aplikační vrstvy je postaven na TCP, poskytuje tedy spolehlivý přenos. Jedná se o protokol typu request/response.

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. [5]

Server odpovídá zprávou response, která obsahuje statusový kód, hlavičky a data určitého typu.

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.

S vyvíjejícími se požadavky na web se protokol HTTP vyvíjel do verze 2 a 3. Firma Google vyvinula 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 může pro své cíle vyžadovat kontext. Příkladem kontextu může být identita uživatele pro autorizaci nebo personalizaci.

Mechanismus cookies slouží ke komunikaci kontextu při dotazu. Cookie je textový token vytvořený serverem, přenášený v hlavičce každého dotazu. Cookies jsou typicky využity k ustanovení sezení (*session*) – sekvence dotazů sdílející kontext. 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 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 a zasílání formulářů a prací s různými API. Webová API například umožňují skriptům pracovat se souborovým systémem, nebo komunikovat pomocí HTTP.

Příklady použití API k manipulaci DOM jsou volání jako `element.appendChild`, nebo `querySelector`. Při vývoji složitých aplikací se ale typicky tato volání nepoužívají přímo, ale abstrahovaně, prostřednictvím knihoven. Mezi nejznámější patří React, Angular, a Vue.js.

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 programovým 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 (změna parametrů, zánik 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.

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

Obrázek 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.

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

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 výkon aplikací oproti řešení v čistém JavaScriptu a velikost knihovny, kterou je nutné stáhnout při návštěvě stránky (130 kB kódu).

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

5.2.2 Next.js

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

Next.js poskytuje implementaci serveru. Dostupné cesty stránek jsou definovány strukturou souborů (file-system based router). Je podporováno i dynamické směrování.

Next.js používá hybridní přístup k renderování. Stránky jsou pokud možno alespoň částečně staticky renderované na serveru. Pokud stránka obsahuje dynamický obsah, je dodatečně renderovaný na straně klienta. Po přijetí statické stránky je *hydratována*, čímž se stává interaktivní. Pouze první stránka je stažena ze serveru – následující navigace jsou vykonány JavaScriptem.

Framework se stará o cachování, přednačítání zdrojů a další optimalizace s cílem zvýšit výkon aplikací. Významně zjednodušuje nasazení aplikace do provozu.

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. [8]

Použitelnost je ovlivněna mnoha faktory, například intuitivnost, spolehlivost, nebo úsilí nutné k používání aplikace.

Estetická příjemnost, uspořádanost a čitelnost rozhraní jsou také důležitými faktory pro jejich pozitivní vnímání. [14]

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. Analýza probíhá z perspektivy nového uživatele a jeho schopnosti dosáhnout svých cílů používáním aplikace. Metodu je možné začít využívat již v raných fázích vývoje, jakmile je k dispozici prototyp. [13]

Vyhodnocení vykonávají specialisté (vývojáři a odborníci na danou doménu). 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.

Výstupem analýzy je seznam prvků rozhraní, které mohou být pro nové uživatele představovat překážky.

5.3.2 Dostupnost

Dostupnost v kontextu webu se zaměřuje na podporu široké škály možností interakce s webovými aplikacemi. Důležitou skupinou jsou invalidní uživatelé a uživatelé mobilních zařízení. Standardizační organizace World Wide Web Consortium (W3C) vydává směrnice *Web Content Accessibility Guidelines*[2] určené pro vývoj dostupných aplikací.

Velká část implementace 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.

Důležité je také možnost ovládat celou aplikaci pomocí klávesnice.

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

Accessible Rich Internet Applications (ARIA) je skupina atributů, kterými lze sémanticky anotovat HTML značky. 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.

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 pomáhá vývojářům identifikovat potenciální problémy a optimalizovat užitečnost aplikace.

Uživatelé vnímají vizuální odezvu jako okamžitou, pokud se odehraje do 30 ms. Vnímaná kvalita významně klesá, pokud je odezva vyšší než 100 ms. [9]

Kvalitativní metody

Vnímání produktu jeho uživatelem je velmi subjektivní.

Poskytnutí formuláře nebo dotazníku pro zpětnou vazbu je nejjednodušší způsob implementace 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 měřitelná analýzou sezení. Metoda Real user monitoring (RUM) instrumentuje aplikaci a tvoří záznam akcí v čase. Poznatky o nedostatcích lze získat pasivním porovnáním akcí úspěšných a neúspěšných uživatelů.

Kvantitativní metody

Web Vitals je soubor metrik vyjadřujících kvalitu uživatelského zážitku z pohledu rychlosti načítání a výkonu při interakci. Metriky odrážejí architekturu aplikace i infrastrukturu, na které je aplikace nasazena.

[2] 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. Za dobrý výsledek se považuje hodnota menší než 2,5 s.

Cumulative Layout Shift (CLS) měří míru neočekávaných změn pozic prvků rozhraní. Tyto posuny jsou způsobeny postupným načítáním obsahu (typicky obrázků, reklam a jiného dynamického obsahu). Velká nestabilita rozvržení stránky způsobuje frustraci uživatele a možnost kliknout na nechtěný prvek rozhraní.

Interaction to Next Paint (INP) měří responzivitu aplikace jako dobu mezi interakcí a prezentací dalšího vyrenderované obrazovky. Za dobrou hodnotu se považuje 200 ms. Některé akce přirozeně trvají delší dobu. V takovém případě je nutné prezentovat 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.

[18]

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*) využívající knihovny jako například Django, Express a Ruby on Rails.

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. Data jsou zpracovávána do nového stavu DOM, který je prohlížečem prezentován uživateli.

5.4.1 Globální stav aplikace

Globální stav jsou data dostupná ze všech částí aplikace. Využívá se ke zpřístupnění dat, se kterými pracuje velká část aplikace a ke sdílení dat mezi moduly. Příklady globálně ukládaných informací jsou data o sezení, uživatelská data a konfigurace aplikace.

Dříve zmíněná knihovna React podporuje práci s globálním stavem ve formě `Context API`. Vytvořený *kontext* je zpřístupněn všem komponentům hlouběji ve stromu.

Redux je zástupcem knihoven pro správu 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 jeho akcích.

Ekosystém nástrojů a knihoven ulehčuje vývoj aplikace. Ladící konzole dovoluje prohlížení stavu, zaznamenává všechny změny stavu a umožňuje změny přehrát. Jiné knihovny například ulehčují perzistenci stavu napříč sezeními.

5.5 Vývojové praktiky

Při vývoji webových aplikací je klíčové implementovat moderní vývojové praktiky, které zajišťují efektivní správu kódu, plynulý vývoj, komunikaci a nasazení aplikace do provozu.

Git, systém pro správu verzí kódu, hraje klíčovou roli ve sledování a dokumentování změn v kódu. Hlavním konceptem gitu je *commit*, záznam stavu kódu. Každý commit je uzlem v grafu, hrany vyjadřují jejich následnost. Události jako například vznik nového commitu, nebo schválení kódu mohou být spouštěčem pro automatizované testování a nasazení.

Continuous Integration a Continuous deployment (CI/CD) je dalším důležitým konceptem. Díky CI probíhají automatizované akce, například sestavení projektu a testování, což zvyšuje jistotu, že nový kód dosahuje dobré kvality. Jsou generována hlášení o výsledcích běhů. Automatizace snižuje pravděpodobnost lidské chyby.

Využití knihoven zrychluje vývoj, populární knihovny zároveň bývají robustní a dobře navržené. Manažer balíčků umožňuje velmi jednoduchou správu závislostí projektu.

5.5.1 Nasazení

Proces nasazování představuje klíčový 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 tento problém řeší tak, že celé prostředí popíše předpisem pro jeho vytvoření. 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, představuje žádné nebo minimální změny v kódu nebo 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)

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

V této kapitole se budu zabývat analýzou simulátoru, který byl výsledkem diplomových prací, na které navazuji [7, 16]. Poskytnu přehled o fungování systému, zaměřím se na jednotlivé moduly a funkce simulátoru, pro které v následující kapitole navrhnu zlepšení.

V současném stavu má simulátor podobu desktopové aplikace s grafickým uživatelským rozhraním. Celý systém je implementován v programovacím jazyce Java s použitím knihovny JavaFX. Aplikace nemá programátorské rozhraní.

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 behaviorální bloky. 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. Příklad 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.

Tento výraz je ve fázi execute interpretován v metodách objektů představující příslušné funkční jednotky. Jedná se o poměrně komplexní interpret s precedenční analýzou operátorů. Implementace obsahuje precedenční tabulku a operátor přiřazení řeší zvlášť.

```
{
  "name": "add",
  "instructionType": "kArithmetic",
  "inputDataType": "kInt",
  "outputDataType": "kInt",
  "instructionSyntax": "add rd rs1 rs2",
  "interpretableAs": "rd=rs1+rs2;"
}
```

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

Každý druh operací 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, proto velikost stavu s délkou simulace značně roste. Simulace například udržuje seznam všech vydaných instrukcí a historii všech paměťových transakcí. Dostatečně dlouhá simulace musí 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 6.2) poskytuje vhled do stavu celého procesoru a prvky k ovládání simulace. 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 6.2). Detailnější pohled poskytuje záložka *Statistics*, kterou je možné vidět na obrázku 6.3. Zde se nachází více číselných statistik a dva grafy vývoje statistik v čase.

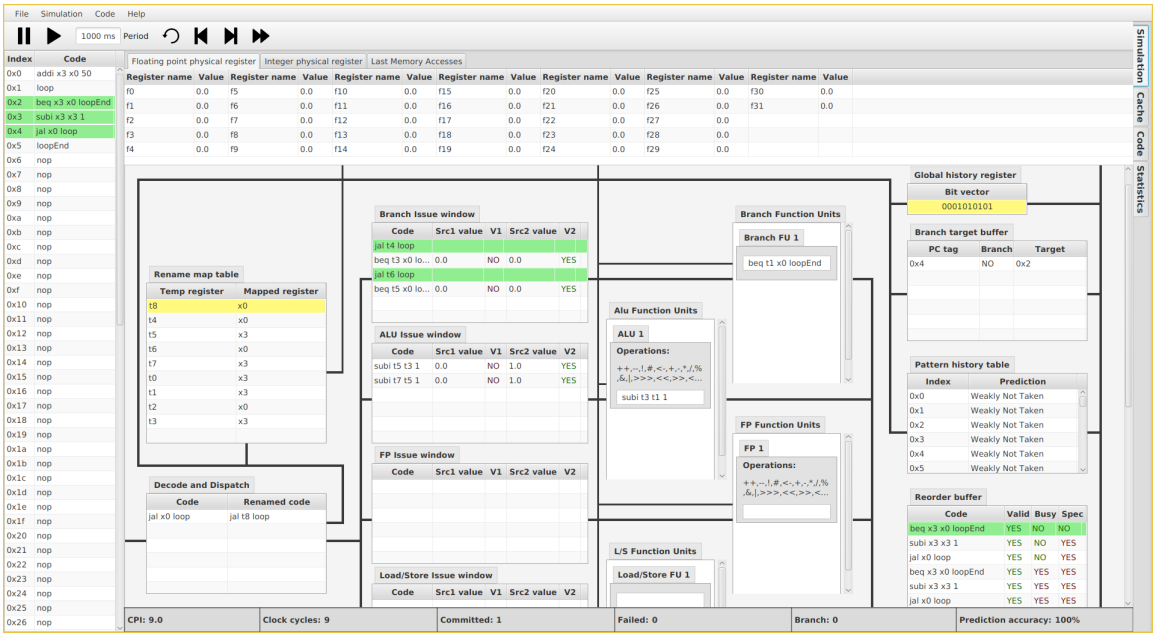
Chybí mi zde více statistik, například instrukční mix (poměr typů instrukcí v kódu), nebo vytíženost funkčních jednotek. Prezentace statistik také není příliš názorná.

6.7 Editor kódu a kompilátor

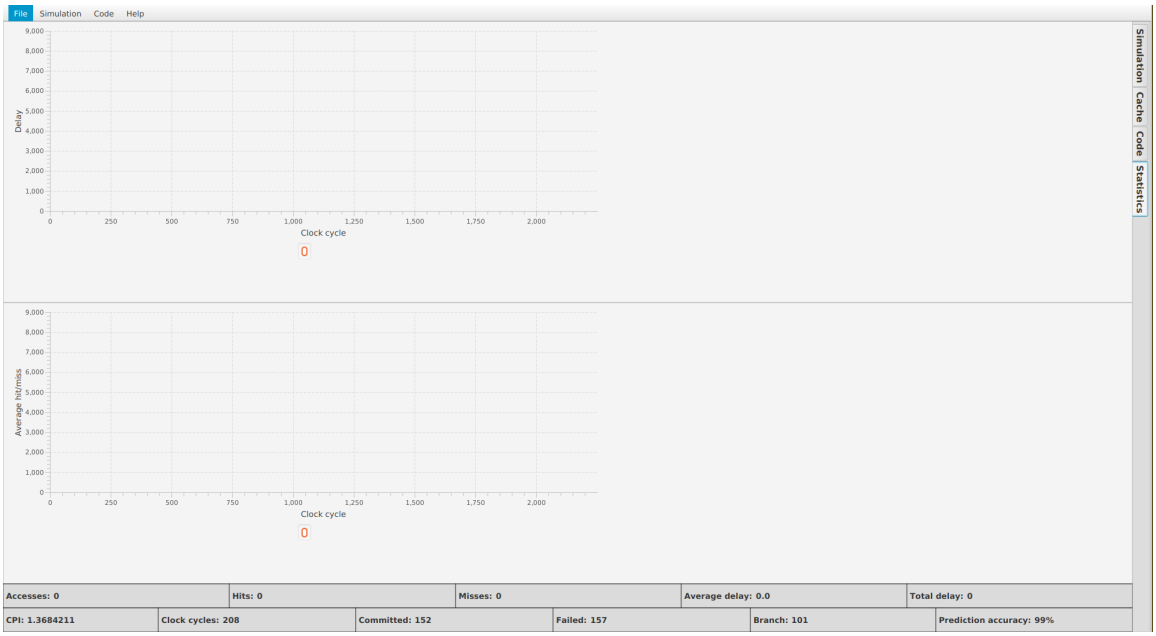
Aplikace umožňuje vytvářet vlastní kódy v assembleru RISC-V, nebo v jazyce C, který je do assembleru následně přeložen. Kód lze následně přenést do simulátoru.

Editor (na obrázku 6.4) je praktický. Zajímavou vlastností je vizualizace vztahu mezi kódem v jazyce C a assemblerem. Je možné nahrát jeden z mnoha předpřipravených příkladů kódu, což snižuje úsilí nutné k vyzkoušení simulátoru.

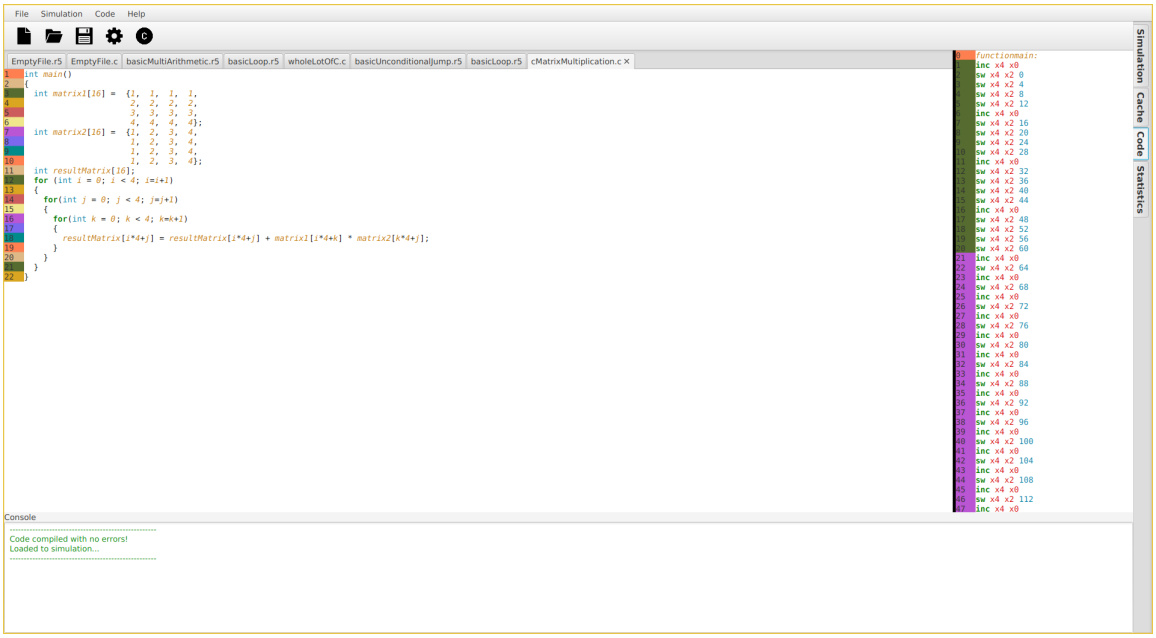
Mít překlad plně pod svou kontrolou je velkou výhodou, zejména pokud simulátor nepodporuje všechny instrukce a direktivy, které běžný překladač může vygenerovat, nebo



Obrázek 6.2: Hlavní okno aplikace v průběhu simulace. Kód, registry a buffery jsou reprezentovány tabulkami. Horní lišta umožňuje spustit celou simulaci, nebo ji postupně krokovat.



Obrázek 6.3: Okno pro zobrazení statistik simulace. Ve spodní části se nachází tabulka.



Obrázek 6.4: Editor kódu zabudovaný v aplikaci. Obsahuje vizuální pomůcky v podobě zvýraznění syntaxe programů a zvýraznění souvisejících částí kódu.

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 [7], 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 Návrh 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 ="
},
```

Obrázek 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 podle současné implementace s bezstavovými dotazy, musel by být stav procesoru přenášen spolu s dotazem. Logika zpětné simulace je však velmi složitá, výrazně zvětšuje velikost stavu a obsahuje těžko odhalitelné chyby.

Navrhuji jiný přístup. Stav v libovolném čase n lze vypočítat ze startovací konfigurace, za předpokladu, že je simulace deterministická. Dotaz tedy nemusí přenášet stav simulace, ale pouze původní konfiguraci. Veškerá zpětná simulace může být realizována dopřednou simulací z výchozího stavu.

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.

Rizikem tohoto přístupu je latence při interaktivní simulaci – délka výpočtu následujícího stavu bude růst lineárně s aktuální pozicí v simulaci, protože celá simulace musí proběhnout znovu. V případě, že odpověď serveru nebude spolehlivě dosahovat interaktivní rychlosti, je možné implementovat cache rozpočítaných simulací.

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 datové 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*, 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).

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).

7.2.10 Zavádění dat do paměti procesu

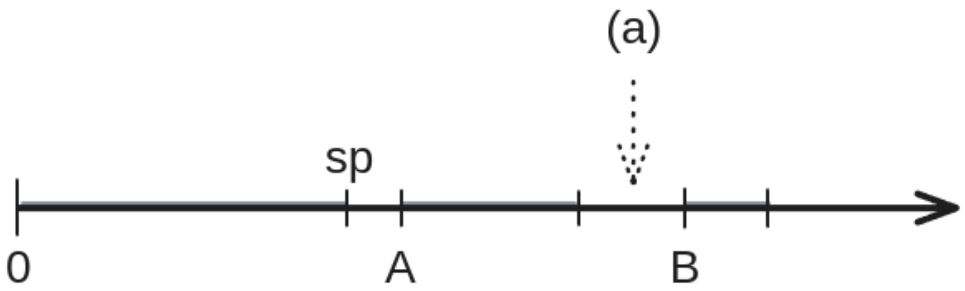
Data definovaná v konfiguraci i v assembleru direktivami[1] musí být staticky alokována v hlavní paměti procesoru. Alokace proběhne podle požadavků na datový typ a *zarovnání*.

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 návěští (labels), která do této doby byla používaná pouze pro adresy skoků. Hodnoty návěští 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 7.2.



Obrázek 7.2: Schéma rozložení dat v paměti procesoru. Počáteční adresy jsou vyhrazeny pro zásobník volání. Následně jsou alokována jednotlivá pole. Prázdné místo mezi bloky (zvýrazněno jako *(a)* může vznikat požadavkem na zarovnání počátku pole v paměti.

7.2.11 Ladící výstupy v průběhu simulace

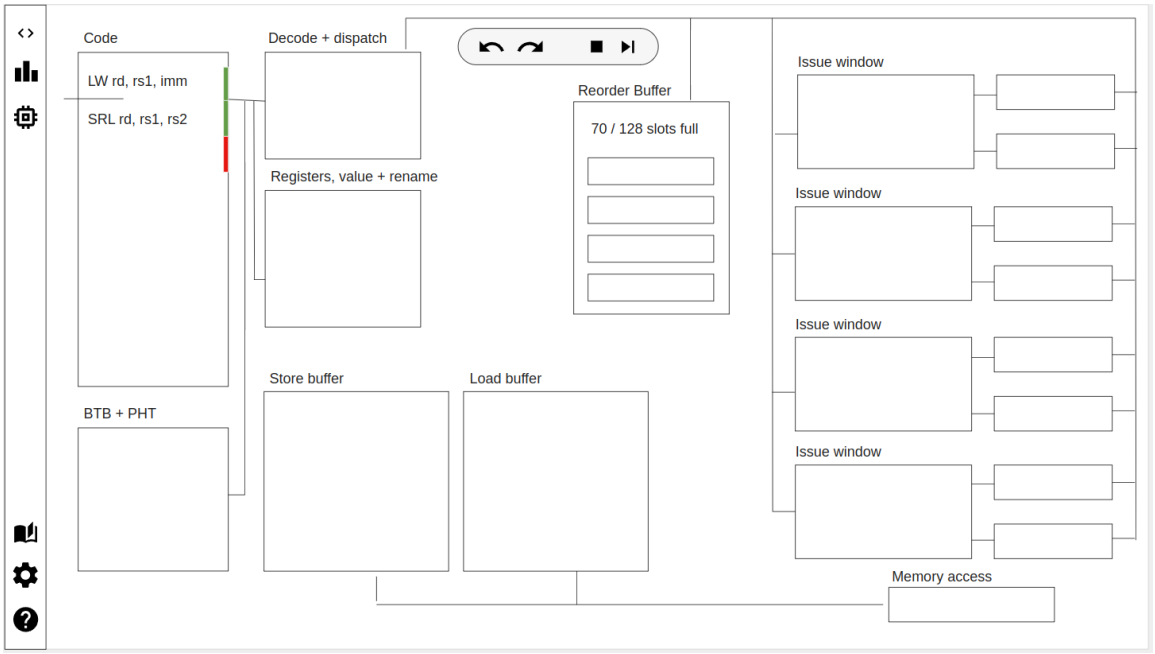
7.3 Návrh webové aplikace

Rozhraní bude v anglickém jazyce. Cílovou skupinou pro aplikaci jsou studenti vysokoškolského kurzu AVS, angličtina je pro studium předpokladem. Angličtina také dovolí projekt používat v mezinárodním prostředí. Vícejazyčná podpora vyžaduje mnoho práce, i za použití internacionalizačních knihoven jako `i18next`.

Obrázek 7.3 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.

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.

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



Obrázek 7.3: Schéma grafické reprezentace stavu simulace. Na levé straně jsou odkazy na ostatní okna aplikace.

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.

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. Syntax a případné chyby v kódu budou v textovém poli znázorněny. Přeložený kód bude barevně vizualizovat vztah řádků původního a přeloženého programu podobným způsobem.

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

7.3.2 Výukové materiály

Cílovými uživateli aplikace budou studenti kurzu AVS. AVS se nezaměřuje na konkrétní instrukční sadu, proto bude pohodlné poskytnout základní informace o sadě RISC-V v podobě vysvětlujícího textu a tabulek. Popis může používat odborné termíny a může se opřít o znalosti již nabyté v průběhu předmětu AVS.

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

Závěr

V této semestrální práci jsem prozkoumal některé známé metody efektivní implementace superskalárních procesorů. Konkrétněji jsem se zaměřil na instrukční sadu RISC-V, kterou simulátor využívá. Také jsem uvedl teorií implementace webových aplikací s přihlédnutím k návrhu uživatelských rozhraní.

Provedl jsem důkladnou analýzu stávajícího simulátoru, zhodnotil jeho klady i nedostatky a navrhl jsem mnoho zlepšení, jak z pohledu implementace simulace, tak použitelnosti jeho rozhraní.

Aplikace je v současné podobě provozuschopná. Na webové stránce je možné provozovat interaktivní simulaci, měnit konfiguraci, editovat kód a prohlížet statistiky o simulaci. Příloha A uvádí tabulku s přehledem jednotlivých modulů aplikace a můj přínos.

Další vývoj bude spočívat v implementaci chybějících částí systému. Nejdůležitější chybějící funkcí je rozhraní příkazové řádky. Další úkoly na letní část práce jsou implementace ladících výstupů, implementace chybějících prvků uživatelského rozhraní simulátoru, vylepšení testů a oprava chyb v logice a optimalizace výkonu. V posledních týdnech proběhne uživatelské testování, na základě kterého provedu úpravy. Výsledky testování také vyhodnotím.

Literatura

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

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

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

[7] Horký, J. *Grafický simulátor superskalárních procesorů*. Brno: [b.n.], 2023. Dostupné z: https://www.vut.cz/www_base/zav_prace_soubor_verejne.php?file_id=252467.

[8] Jordan, P. Human factors for pleasure in product use. *Applied Ergonomics*. 1998, sv. 29, č. 1, s. 25–33. DOI: https://doi.org/10.1016/S0003-6870(97)00022-7. ISSN 0003-6870. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0003687097000227.

[9] 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. DOI: 10.1145/2611387. ISSN 1544-3558. Dostupné z: https://doi.org/10.1145/2611387.

[10] 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.

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

[12] 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.

[13] 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. DOI: 10.1145/223355.223735. ISBN 0897917553. Dostupné z: https://doi.org/10.1145/223355.223735.

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

[15] 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.

[16] Vávra, J. *Graphical Simulator of Superscalar Processors*. Brno: [b.n.], 2021. Dostupné z: https://www.fit.vut.cz/study/thesis-file/21991/21991.pdf.

[17] 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.

[18] 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. DOI: 10.1109/QoMEX55416.2022.9900881.

[19] 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. DOI: 10.1145/123465.123475. ISBN 0897914600. Dostupné z: https://doi.org/10.1145/123465.123475.

Příloha A

Přehled převzaté práce

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

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

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