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

GRAPHICAL SIMULATOR OF SUPERSCALAR PROCESSORS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. JAKUB HORKÝ**
AUTHOR

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

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

144976

Ústav:	Ústav počítačových systémů (UPSY)
Student:	**Horký Jakub, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Verifikace a testování software
Název:	**Grafický simulátor superskalárních procesorů**
Kategorie:	Počítačová architektura
Akademický rok:	2022/23

Zadání:

1. Seznamte se s architekturou současných superskalárních procesorů.
2. Prostudujte současné grafické simulátory těchto procesorů, zaměřte se především na RISC-V simulátor Jana Vávry.
3. Navrhněte postup pro rozšíření tohoto simulátoru o paměťový subsystém, vykonání částí kódu zapsaného ve vyšších jazycích a sběr výkonnostních metrik.
4. Navržené řešení implementujte.
5. Navrhněte sadu demonstračních úloh, na kterých názorně vysvětlíte fungování procesoru.
6. Vyhodnoťte uživatelskou přívětivost a názornost navržené aplikace.
7. Diskutujte přínos vytvořeného simulátor pro výuku HW kurzů na FIT VUT v Brně.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

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

Vedoucí práce:	**Jaroš Jiří, doc. Ing., Ph.D.**
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1.11.2022
Termín pro odevzdání:	17.5.2023
Datum schválení:	31.10.2022

Abstrakt

Práce se zabývá popisem jednotek v procesoru a způsobem jejich propojení ve skalárním a superskalárním procesoru. Dále se zaměřuje popisem problematiky práce s pamětí a zejména popisem mezi-pamětí. Popisuje fungování překladače z vyššího programovacího jazyka do jazyka symbolických instrukcí. Dále prozkoumává dostupné simulátory procesorů a mezi-pamětí, zaměřuje se zejména na simulátor, na který tato práce navazuje.

Na základě analýzy byly navrženy a následně implementovány rozšíření současného simulátoru o paměťový subsystém, podporu vyššího programovacího jazyka a sbírání většího množství statistik. Závěr práce shrnuje kvalitu implementace a možný přínos simulátoru při výuce předmětu Architektury výpočetních systémů.

Abstract

In this thesis, I firstly focus on functional units inside processors and how they are interconnecetd in scalar and superscalar processor. Then, I describe the memory hiearchy with focus on caches. Next, I describe how compilers do translation from higher level language into assembly. Then, I have a look at available processor simulators and cache simulators and more closely describe the simulator that this thesis is based on.

Thanks to the information from the analysis, I propose possible extensions to the simulator by adding memory subsystem, compiler and gathering more statistics. In the end, I have a look at my implementation and investigate possible benefits to the "Computation Systems Architectures" lectures

Klíčová slova

Superskalární procesor, simulátor, mezi-paměť, překladač.

Keywords

Superscalar processor, simulator, cache, compiler.

Citace

HORKÝ, Jakub. *Grafický simulátor superskalárních procesorů.* Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Jiří Jaroš, Ph.D.

Grafický simulátor superskalárních procesorů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doc. Jaroše. Další informace o simulátoru mi poskytl Jan Vávra. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jakub Horký
9. května 2023

Poděkování

Chtěl bych obzvláště poděkovat vedoucímu této práce, panu doc. Jiřímu Jarošovi, za konzultace a vedení celou diplomovou prací. Dále Ing. Janu Vávrovi za vysvětlení zdrojových kódů a rady do začátku. Mým rodičům za vlídná slova a povzbuzování během celého studia. Mé přítelkyni děkuji za psychickou podporu, která mi pomohla zvládnout tento rok. A nakonec sobě, že jsem to vydržel až do konce.

Obsah

1 Úvod **6**

2 Architektura procesorů **7**
2.1 Funkční jednotky v procesoru 7
2.1.1 Výpočetní jednotky 7
2.1.2 Jednotky pro práci s pamětí 8
2.1.3 Jednotka pro řízení běhu programu 8
2.1.4 Jednotka pro práci s privilegovanými instrukcemi 8
2.1.5 Další jednotky 8
2.1.6 Propojení jednotek ve skalárním procesoru 9
2.2 Superskalární procesor 10
2.2.1 Datové hazardy 10
2.2.2 Tomasulo algoritmus 11
2.3 Další optimalizace výkonu procesoru 11
2.4 Paměti 12
2.4.1 Časová a prostorová lokalita 12
2.4.2 Mezi-paměti 13
2.4.3 Fungování mezi-pamětí 13
2.4.4 Závěr 16

3 Překladače **17**
3.1 Fáze překladu 17
3.2 Syntaxí řízený překlad 18

4 Simulátory **20**
4.1 Simulátory procesorů 20
4.1.1 RIPES 20
4.1.2 QTMips 23
4.1.3 OpendDLX 23
4.1.4 Jupiter 25
4.2 Simulátory mezi-pamětí 26
4.2.1 Simulátor mezi-paměti z washingtonské univerzity 26
4.2.2 Simulátor mezi-paměti z michiganské univerzity 27
4.2.3 Simulátor mezi-paměti z nanyangské univerzity 27
4.2.4 Simulátor mezi-paměti z brněnské univerzity 28
4.3 Shrnutí 29

5 Superskalární simulátor od Jana Vávry **30**

5.1 První spuštění . . . 30
5.2 Pohled v hlavním okně . . . 30
5.3 Analýza funkcí . . . 30
5.4 Analýza kódu . . . 31
5.5 Zdrojové soubory . . . 32

6 Návrhy rozšíření **33**
6.1 Hlavní okno . . . 33
6.2 Návrh mezi-paměti . . . 34
6.2.1 Integrace do kódu . . . 35
6.3 Překladač . . . 35
6.3.1 Definice jazyka . . . 36
6.3.2 Integrace do kódu . . . 37
6.4 Statistiky . . . 37

7 Implementace překladače **38**
7.1 Hlavní logika . . . 38
7.2 Lexikální analyzátor . . . 38
7.3 Syntaktický analyzátor . . . 38
7.3.1 Syntaktický strom . . . 39
7.3.2 Průběh syntaktické analýzy . . . 39
7.4 Sémantický analyzátor . . . 43
7.4.1 Sémantická analýza řídícího příkazu . . . 43
7.4.2 Sémantická analýza definice proměnné . . . 44
7.4.3 Sémantická analýza definice funkce . . . 44
7.4.4 Sémantická analýza příkazu . . . 45
7.5 Generátor tří-adresného kódu . . . 47
7.5.1 Pseudokódy převedení operací do vnitřní reprezentace: . . . 47
7.6 Generátor Cílového kódu . . . 50
7.6.1 Základní fungování . . . 50
7.6.2 Přepis registrů a adres . . . 50
7.7 Grafické rozhraní . . . 51

8 Implementace mezi-paměti **53**
8.1 Implementace bloku mezi-paměti . . . 53
8.2 Implementace kontrolní logiky mezi-paměti . . . 54
8.2.1 Implementace přístupu do mezi-paměti . . . 54
8.3 Napojení do procesoru . . . 54
8.4 Grafické rozhraní . . . 55
8.5 Úprava řízení simulace . . . 55
8.6 Implementace zpoždění . . . 56
8.7 Implementace politik výměny . . . 56
8.8 Nastavitelnost mezi-paměti . . . 57

9 Implementace statistik **58**

10 Zhodnocení implementace **60**
10.1 Ukázkové příklady . . . 60
10.2 Zhodnocení překladače . . . 60

10.3 Zhodnocení mezi-paměti 60
10.4 Zhodnocení statistik 61
10.5 Další možná vylepšení 61

11 Závěr **62**

Literatura **64**

A Obsah přiloženého disku **66**

B User manual **67**

Seznam obrázků

2.1 Instrukce procházející všemi částmi výpočtu 9
2.2 Rozdělení adresy na *Tag*, *Index*, *Offset* 14

3.1 Syntaxí řízený překlad 19

4.1 Hlavní okno simulátoru RIPES 21
4.2 Zobrazení mezi-paměti v simulátoru RIPES 22
4.3 Zobrazení mezi-paměti v simulátoru QTMips 24
4.4 Hlavní okno simulátoru opendlx 25
4.5 Simulátor Jupiter 25
4.6 Simulátor mezi-paměti z washingtonské univerzity 26
4.7 Simulátor z univerzity v Michiganu 27
4.8 Simulátor mezi-paměti z nanyangské univerzity 28
4.9 Simulátor mezi-paměti z brněnské univerzity 29

5.1 Hlavní okno simulátoru 31
5.2 Nastavení simulátoru 32

6.1 Pohled do hlavního okna 33
6.2 Návrh mezi-paměti 34
6.3 Návrh překladače 36
6.4 Návrh statistik 37

7.1 Syntaktická analýza definice proměnné nebo funkce 40
7.2 Pokračování syntaktické analýzy definice pole 41
7.3 Syntaktická analýza příkazu 42
7.4 Sémantická analýza řídícího příkazu 43
7.5 Sémantická analýza definice proměnné 44
7.6 Pokračování sémantické analýzy přiřazení definice pole 44
7.7 Sémantická analýza definice funkce 45
7.8 Sémantická analýza příkazu 46
7.9 Záložka code po implementaci překladače 52

8.1 Grafické zobrazení mezi-paměti 56
8.2 Nastavení mezi-paměti 57

9.1 Vzhled záložky statistik 58

B.1 The main application controls 68
B.2 The Code control buttons 69

B.3 The Code window 70
B.4 The Simulation window 71
B.5 The Simulation control buttons 72
B.6 Cache window described 73
B.7 Statistics window described 74

Kapitola 1

Úvod

Architektury novodobých procesorů se stávají čím dál složitější a správné vysvětlení principů a algoritmů v nich použitých je stále náročnější. Nicméně fungování současných procesorů stále staví na základech a principech vynalezených více než před dvaceti lety.

Tyto principy se snaží na Fakultě Informačních Technologií VUT v Brně vysvětlit předmět Architektury výpočetních systémů (AVS). V rámci uvedeného předmětu jsou vysvětleny základní principy fungování procesorů, které jsou často starší než současní studenti, a tudíž je nutné učit základy názorně a efektivně.

Z toho důvodu vznikl simulátor od Jana Vávry implementující superskalární procesor, který pro jednodušší vysvětlení jeho fungování umožňuje krokování programem a zobrazení stavu funkčních jednotek. V mé práci budu vycházet z uvedeného simulátoru a budu se snažit jej vylepšit, aby studentům přinesl větší užitek.

Na začátku práce se krátce zaměřím na vysvětlení složení a principů fungování superskalárního procesoru. Blíže se zaměřím na jejich práci s pamětí a na fungování mezi-pamětí. Kapitolu 3 věnuji popisu způsobu průběhu překladu z vyššího programovacího jazyka do instrukcí, které je procesor schopný vykonat. V kapitole 4 prozkoumám dostupné simulátory a hlavně se v kapitole 5 pustím do analýzy simulátoru, který budu rozšiřovat.

Dále práce popíše návrh rozšíření v kapitole 6, poté se již pustím do samotného implementování v kapitolách 7 8 a 9, kde popíšu implementaci překladače ze zjednodušeného jazyka C, mezi-paměti a sběru statistik.

Nakonec v kapitole 9 zhodnotím kvalitu těchto rozšíření a jejich možný přínos pro výuku procesorů.

Kapitola 2

Architektura procesorů

Procesor (*CPU - Central Processing Unit*), neboli centrální výpočetní jednotka, představuje základní komponentu počítače, která se stará o řízení celého zařízení. Novodobé procesory vykonávají miliardy operací za sekundu. Tyto operace se dají rozdělit do několika kategorií:

- Výpočetní.
- Práce s pamětí.
- Řízení běhu programu.
- Privilegované operace. [1]

2.1 Funkční jednotky v procesoru

Operace, které procesor vykonává, jsou rozloženy do několika funkčních jednotek, v závislosti na podobnosti operací. Každý procesor nemusí mít všechny tyto jednotky nebo vůbec implementovat všechny tyto operace. Některé procesory mohou sloučit více operací do jedné jednotky, některé je mohou rozdělit. Drobné detaily již závisí na konkrétním procesoru.

2.1.1 Výpočetní jednotky

Základní výpočetní jednotka nacházející se v téměř každém procesoru je Aritmeticko-logická jednotka (*ALU*), která se stará o vykonávaní jednoduchých operací. Mezi ně patří například: Sčítání a odčítání celých čísel, bitové posuvy a výřezy, logické operace, porovnávání apod.

Dále procesory obsahují jednotku vykonávající operaci násobení a dělení celých čísel.

Pro výpočty v plovoucí řádové čárce využívá procesor *FPU* (*Floating point unit*). Tyto jednotky často (ale ne vždy) implementují standard IEEE 754[2]. Pomocí množství vykonaných operací v plovoucí řádové čárce se často porovnává výkon počítačů a hlavně superpočítačů. K tomu se používá termín *FLOPS* (*Floating point operations per second*) nebo spíše častěji *TFLOPS* (Tera *FLOPS* - bilióny operací v plovoucí řádové čárce za sekundu), i když novodobé superpočítače již dosahují řádově vyšších jednotek [15].

Některé novodobé procesory jsou vybaveny vektorovou jednotkou, která v sobě obsahuje několik výpočetních jednotek, čímž umožní vykonat stejnou operaci nad řadou dat. Šířka vektorových operací a podporované operace závisí na architektuře procesorů. Běžné výkonnější procesory s architekturou x86 implementují například rozšíření AVX-512[19], které

[1] Například operace měnící funkcionalitu instrukcí, zakázání/povolení některých instrukcí apod.

pracuje nad daty o šířce až 512 bitů. Pokud tedy takový procesor vykoná jednu vektorovou instrukci nad 16-bitovými daty, provede 32 (512/16) operací. Což je ještě zjednodušeno, protože instrukce může vykonávat spojené operace a tedy jich udělat ještě více (např. VFMADD - Vynásobí a sečte).

2.1.2 Jednotky pro práci s pamětí

Hlavní jednotka pro práci s pamětí je *LSU* (*Load/Store unit*), která řeší výpočet cílové adresy, ukládání a načítání dat z paměti, povolení nebo zakázání těchto přístupů a výslednou komunikaci s pamětí. Uvedená jednotka se skládá z několika vestavěných jednotek. Konkrétně se jedná o:

- Načítací a ukládací vyrovnávací paměti. Ty se starají o uložení probíhajících operací, než se uvolní místo na sběrnici.
- *MAU* (*Memory-access unit*) se stará o komunikaci s pamětí.
- Jednotky pro kontrolu práv přístupů. Těchto jednotek může být více, v závislosti na použitých systémech kontroly (např. virtuální paměť).

Navíc procesory běžně obsahují mezi-paměti, které budou blíže popsány v kapitole 2.4.2

2.1.3 Jednotka pro řízení běhu programu

Řízení běhu programu spočívá ve skocích a větvících instrukcích. Větvení představuje jednu z nejzákladnějších instrukcí procesoru, protože se používají například k implementaci if-then-else příkazů a cyklů. O vyhodnocování, zda se má větvení a následný skok provést a výpočet cílové adresy, se stará *BRU* (*Branch unit*)

2.1.4 Jednotka pro práci s privilegovanými instrukcemi

Procesory často obsahují více privilegovaných módů, které umožňují měnit funkcionalitu v závislosti na aktuálním módu (např. operační systém umožňuje programům provádět operace pouze nad určitým úsekem v paměti, čímž zvyšují ochranu počítače proti škodlivým programům). Privilegované instrukce mění způsob, jakým procesor provádí některé instrukce. Může se jednat o zakázání některých instrukcí, změnu povolených adres při přístupu do paměti, změnu aktuálního privilegovaného módu apod.

2.1.5 Další jednotky

Všechny zatím popsané jednotky přímo vykonávaly instrukce. Nicméně procesor pro své fungování potřebuje i další jednotky.

Načítání instrukcí

Jednu z takovýchto základních jednotek představuje *FU* (*Fetch-Unit*). Tato jednotka se stará o načtení dalších instrukcí z paměti. U super-skalárních procesorů načítá instrukce dříve, než je může vykonat, aby tyto instrukce byly dostupné co nejdříve. V případě programu, který neprovádí skoky nebo větvení bývá tato funkcionalita velmi jednoduchá, protože se načítají další instrukce v programovém pořadí.

Každý program ale obsahuje větvení a to komplikuje toto před-načítání. Aby byla *FU* schopna zásobovat instrukcemi zbytek procesoru využívá *BP* (*Branch Predictor* - prediktor skoků). Prediktor skoků se snaží odhadnout, zda načtená instrukce představuje skok nebo větvení, zda se toto větvení použije a cílovou adresu. V případě správného určování výsledku může razantně snížit penalizaci skoku. Predikce skoků lze implementovat různě. Nejjednodušší prediktory očekávají, že se skok vždy provede, což například může zvednout výkon v cyklech nebo v programech, které na to budou uzpůsobeny. Složitější si pamatují výsledky několika posledních větvení, buď mají uložený výsledek daného větvení, nebo mohou pomocí nich výsledek odhadnout. Jednotka obsahuje seznamy posledních skoků, jejich cílové adresy (ta je většinou neměnná) a řízení predikce skoků (implementující konkrétní algoritmus predikce).

FU stejně jako *LSU* obsahuje jednotku přímo vykonávající přístup do paměti, ke které je často připojena mezi-paměť. Navíc si jednotka ukládá před-načtené instrukce.

Ukládání stavu programu

Procesor sám o sobě pro své potřeby využívá registry (malé paměťové bloky) ukládající stav procesoru, nicméně z pohledu vykonávání programu musí i obsahovat registrové pole, kde si program ukládá data, nad kterými pracuje.

2.1.6 Propojení jednotek ve skalárním procesoru

Jednu z optimalizací, kterou procesory využívají, představuje zřetězování operací. Vykonávání instrukce probíhá v několika fázích: Načtení → dekódování → vykonání → přístup do paměti → uložení výsledku. Uvedené fáze se běžně rozdělují do jednotlivých úrovní procesoru. Jednotlivé úrovně mohou být spojené nebo rozdělené na více části v závislosti na návrhu procesoru a jeho složitosti. Nicméně (téměř) všechny instrukce projdou všemi těmito fázemi a tudíž se dají instrukce vykonávat zřetězeně (jako na obrázku 2.1). Takový procesor vykonává několik instrukcí zároveň. I díky této optimalizaci mohou novodobé procesory dosahovat rychlosti miliard taktů za vteřinu.

Obrázek 2.1: Instrukce procházející všemi částmi výpočtu. Instrukce nepracující s pamětí čekají na uvolnění zápisu výsledku a jsou zdrženy o jeden cyklus (vyobrazeno červeně).

Nicméně skalární procesor vykonává všechny instrukce v pořadí v jakém jsou uvedeny v programu a tudíž nevyužívá plně všechny své jednotky, protože déletrvající instrukce zastavují vykonávání následujících kratších instrukcí. Procesor musí navíc zastavit vykonávání následujících instrukcí, pokud mezi nimi nastává *RAW hazard* - více o hazardech v kapitole 2.2.1).

2.2 Superskalární procesor

Během vykonávání programu skalární procesor v jeden okamžik nevyužívá většinu svých výpočetních jednotek, proto pro zvýšení efektivity začaly vznikat superskalární procesory, které jsou schopné spustit vykonávání instrukcí na více funkčních jednotkách zároveň a umožňují dokončení více něž jedné instrukce za jeden takt hodin. Superskalární procesory často obsahují více funkčních jednotek jednoho druhu.

Superskalární procesor musí zajistit dostatečný přísun instrukcí, aby zvládl plnit funkční jednotky takovým způsobem, že v ideálním případě jsou využity všechny funkční jednotky. Což není vždy možné. Instrukce, které procesor vykonává, mají mezi sebou často závislosti. Překladače programů se snaží dávat instrukce za sebe takovým způsobem, aby limitovaly tyto hazardy. Nicméně užitečný program není možné udělat bez závislostí. Překladače musí překládat programy, aby je více druhů procesorů bylo schopno vykonávat efektivně. Z uvedeného důvodu program nebývá přeložen optimálně na cílovou architekturu. Superskalární procesory tedy vykonávají instrukce mimo pořadí v jakém jsou uvedeny v programu, což může pomoci vyřešit závislosti mezi instrukcemi. Instrukce nezávisející na výsledku aktuálně vykonávaných instrukcí mohou předběhnout instrukce, které na nich závislé jsou.

2.2.1 Datové hazardy

Hazard v procesoru nastává, když se v programu za sebou nachází instrukce přistupující ke stejným registrům nebo místům v paměti. Existuje několik druhů hazardů:

- *WAW* - dvě instrukce po sobě zapisují do stejného registru.
- *RAW* - první instrukce zapisuje do registru, který další instrukce čte.
- *WAR* - první instrukce čte z registru, do kterého následující instrukce zapisuje.
- *RAR* - dvě instrukce po sobě čtou ze stejného registru.

Jelikož skalární procesor vykonává instrukce v pořadí, v jakém jsou v programu, a vždy dokončí maximálně jednu instrukci, tak pro něj *RAR*, *WAR* a *WAW* hazardy nejsou problematické. Buď první instrukce nemění stav procesoru (*RAR*, *WAR*) nebo následující instrukce tuto změnu stejně přepíše (*WAW*). V případě *RAW* hazard musí skalární procesor pozastavit vykonávání všech následujících instrukcí, než se zapisující instrukce dokončí. Uvedená situace může nastat například, když jedna instrukce vykonává dělení, které může trvat desítky cyklů, a následující instrukce s těmito daty pracuje.

Oproti tomu superskalární procesor plní funkční jednotky dalšími instrukcemi [5], které nejsou konfliktní, a může je vykonat dříve. Superskalární procesory umožňují i vykonávání instrukcí mimo pořadí. Kvůli těmto optimalizacím musí superskalární procesor řešit i ty hazardy, které nezpůsobují problém ve skalárním procesoru.

Pro tyto účely se využívá mnoho algoritmů. V rámci předmětu AVS se učí dva základní algoritmy *scoreboarding* a *Tomasuluv algoritmus*. Scoreboarding[16] představuje jednoduší algoritmus, udržuje si informaci o vykonávaných instrukcích, jejich operandech a jejich platností. Tento algoritmus řeší všechny typy hazardů, nicméně stále *WAW* a *WAR* hazardy řeší zastavením vykonávání dalších instrukcí. Oproti tomu *Tomasuluv algoritmus* se umí vypořádat i s těmito hazardy. Navíc je implementován v simulátoru, na který tato práce navazuje, a proto bude popsán blíže.

2.2.2 Tomasulo algoritmus

Tomasulo algoritmus, navržen Robertem Tomasulo v 60. letech, byl poprvé použit v procesorech IBM. Přestože byl tento algoritmus vymyšlen před více než 50 lety, je stále aktuální a principy v něm použité se stále využívají.

Přejmenování registrů

Uvedený algoritmus řeší hazardy, které mohou v procesoru nastat, přejmenováváním použitých registrů (*Register Renaming*) z architekturálních označení na mikro-architekturální označení. Implementace těchto registrů může být implementována buď v *seřazovací paměti* (*Reorder buffer - ROB*), nazývaném také implicitní přejmenovávání [16][17], kde vykonané instrukce mají uložený výsledek v této paměti a pamatují si jaká instrukce je vykonala. Nebo procesor obsahuje oddělené *registrové pole* pro přejmenované registry, případně obsahuje větší *registrové pole* než architektura specifikuje. Získané výsledky jsou tedy uloženy přímo do přejmenovaného registru, což se nazývá explicitní přejmenování[16][17]. V obou případech vyvstává potřeba pamatovat si mapování přejmenovaných registrů na registry architekturální, k čemuž se využívá *RAT* (*Register Alias Table*)

Rezervační stanice

Rezervační stanice se starají o vypouštění instrukcí do funkčních jednotek. Pro tento účel si rezervační stanice udržují instrukce, jejich vstupní operandy, platnost a ukazatel do seřazovací paměti.[16][5] Pokud se architekturální registr namapuje na spekulativní, je platnost tohoto registru nastavena na 0. Pokud jsou data dostupná, platnost se nastaví na 1 a instrukce si uloží hodnotu vstupního operandu. Pokud má instrukce všechna vstupní data platná a volnou funkční jednotku, spustí se vykonání instrukce. Rezervační stanice se snaží upřednostňovat nejstarší instrukce a ty které blokují nejvíce dalších instrukcí.

Komunikační sběrnice

Všechny rezervační stanice v procesoru jsou připojeny na sdílenou sběrnici, kam při dokončení instrukce posílají funkční jednotky jejich výsledek. Rezervační stanice si tudíž mohou uložit výsledek do svých čekajících instrukcí bez nutnosti čtení z registru.

Seřazovací paměť

Seřazovací paměť se používá k uložení instrukcí, než je možné je dokončit. Ukládá si dokončené instrukce a v závislosti na implementaci i výsledky jejich operací. U instrukcí se ukládá zda se vykonávají, zda jsou spekulativní (vykonané na základě predikce skoků), zda je instrukce validní a mapování na přeložené registry. Když je instrukce validní, nevykonává se a není spekulativní, je možné ji dokončit a zapsat její výsledek do architekturálního registru.[16]

2.3 Další optimalizace výkonu procesoru

Vzhledem k přidané logice jsou ale procesory vykonávající instrukce mimo programové pořadí méně používané, než by se mohlo na první pohled zdát. Tato technika optimalizace výkonu se vůbec nepoužívá ve vestavěných zařízeních, která nepotřebují takový výkon, místo toho často cílí na nižší spotřebu a plochu. Uvedené často platí i o mobilních telefonech, kde

Typ paměti	Rychlost přístupu	Velikost
Pevný disk	milióny cyklů	v TB
Operační paměť	stovky cyklů	v GB
Mezi-paměť	jednotky až desítky cyklů	v KB až MB
Registry v procesoru	1 cyklus	většinou v Bytech

Tabulka 2.1: Rychlost a velikost pamětí

se stále hojně využívají procesory vykonávající instrukce v programovém pořadí. Například podle nabídky firmy ARM, předního výrobce procesorů do mobilních telefonů a vestavěných systémů, pouze jejich nejvýkonnější procesory vykonávají instrukce mimo-pořadí [4].

Nicméně se využívají i jiné optimalizační techniky. Hlavní takovou technikou jsou více-jádrové procesory. Tyto procesory spojují více samostatných jednotek schopných vykonávat programy (včetně načítání programu, dat a udržování stavu programu) nezávisle na ostatních jádrech.

Podobná technika je *multi-threading*, kde více programů (většinou 2) sdílí jedny zdroje. Tento program se nazývá *vlákno* a v případě, že jedno z *vláken* nemůže vykonávat své instrukce kvůli hazardům nebo protože čeká na výsledek z paměti, přepne se vykonávání na druhé vlákno, čímž se udržuje větší vytížení procesoru. Výkon *více-vláknových* procesorů hodně závisí na rychlosti přepínání vláken (několik cyklů, jeden cyklus, vlákna se mohou vykonávat zároveň), ale hlavně závisí na podpoře operačním systémem, který na tyto procesory musí dávat aplikace, jež zvládnou využít zdroje efektivně.

2.4 Paměti

Data v počítačovém systému se ukládají na trvalé úložiště (např. *SSD* nebo *pevný disk*), nicméně trvalé úložiště je moc pomalé, aby procesor mohl pracovat přímo nad ním a proto se programy před spuštěním nahrají do operační paměti, kde se vykonávají. Operační paměť je rychlejší než trvalé úložiště, ale také o dost menší. Ale i operační paměť pracuje příliš pomalu, protože přístup do operační paměti trvá stovky procesorových cyklů. Z toho důvodu vznikly *mezi-paměti* [18], které jsou ještě rychlejší, ale o to menší. Často tak malé, že se do nich nevejde celý program.

Důvodem pro zvětšující se rychlost mezi-pamětí je jednak technologie uložení dat, ale hlavně vzdálenost paměti od procesoru. Důvod, proč se nepoužívá stejná technologie na operační paměť i na mezi-paměti, představuje příliš vysoká cena tohoto řešení. Uvedený přístup se jeví navíc zbytečným protože by přinesl minimální navýšení výkonu.

2.4.1 Časová a prostorová lokalita

Jak jsem se již zmínil, mezi-paměti jsou často menší než programy a data, ke kterým přistupují. Nicméně to nevadí z důvodu časové a prostorové lokality v programech, čehož mezi-paměti využívají. Programy jsou vykonávány sekvenčně a jednotlivé kusy kódu bývají tedy často blízko sebe. Navíc se v kódu často objevují cykly, tudíž je pravděpodobné, že jednou zpracovaný kód se bude vykonávat znovu. To samé platí o zpracovávaných datech. Program často přistupuje ke stejným proměnným v určité části programu a k datům blízko sebe (např. pole). Jako příklad by bylo možné uvést tento jednoduchý cyklus:

```
for(i = 0; i<array.size(); i++) {
    sum += array[i]
}
```

Tento cyklus provádí stále stejné instrukce a pouze se postupně posouvá dále v poli a přičítá jeho data do jedné proměnné. *Mezi-paměť* může uvedený cyklus výrazně zrychlit.

2.4.2 Mezi-paměti

Mezi-paměť je umístěna na výstupu procesoru v několika úrovních [13]. V závislosti na složitosti procesoru a požadovaném výkonu se přidává další úroveň, ale maximálně se používají tři úrovně [2].

První úroveň mezi-paměti (*L1 cache*) bývá umístěna přímo v procesoru a nabízí nejrychlejší přístup k datům, tudíž musí být malá. Tato mezi-paměť bývá rozdělena na instrukční a datovou, protože dva zdroje přístupu do paměti pracují nad rozdílnou částí paměti.

Druhá úroveň mezi-paměti (*L2 cache*) bývá umístěna na stejném čipu jako procesor. Nenabízí již tak rychlý přístup, ale oproti tomu o dost větší prostor pro data. Druhá úroveň se zaměřuje, aby pokud možno již obsahovala data, ke kterým se přistupuje.

Třetí úroveň mezi-paměti (*L-3 cache*) nejčastěji využívají více-jádrové systémy, kde je tato mezi-paměť sdílená mezi všemi jádry, což zrychluje přístup, pokud více-jader pracuje nad stejnými daty a zároveň zajišťuje koherenci dat mezi těmito jádry.

2.4.3 Fungování mezi-pamětí

V této části bude vysvětlena funkcionalita datové mezi-paměti. Stejné principy se využívají i na instrukční *mezi-paměti* s tím rozdílem, že do instrukční paměti nelze zapisovat.

Základní prvky

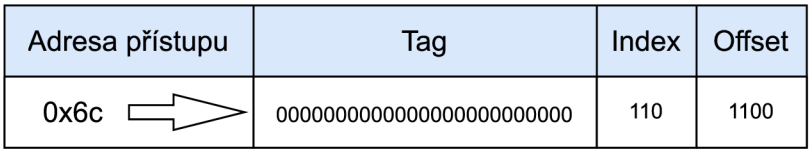
Mezi-paměti si udržují poslední používaná data. Data jsou rozdělená do bloků (nebo také angl. *cache-line*), kde jeden blok je nejmenší adresovatelný prvek touto mezi-pamětí. Typicky bývá velikost jednoho bloku mezi-paměti mezi 16B a 64B[18]. Procesor může přistupovat k menším datům než je blok, ale mezi-paměť pracuje s daty po celém bloku. Aby mezi-paměť věděla, která data jsou platná, obsahuje *Valid bit*. Tento bit se při spuštění nastavuje na 0 a na hodnotu 1 se nastaví až po načtení dat. Pokud procesor požaduje data, která jsou již v mezi-paměti načtená (mají *Valid bit* nastavený na hodnotu 1), pošle je procesoru a s pamětí není potřeba komunikovat. Takový případ se označuje jako *cache hit*. Pokud mezi-paměť data neobsahuje, načte je z paměti (nebo z následující mezi-paměti), čemuž se říká *cache miss*. V případě, že procesor data do mezi-paměti zapíše, u dané bloku se nastaví *Dirty bit* na hodnotu 1. Značí tím, že daná data byla změněna, a blok neobsahuje stejnou hodnotu jako v paměti. Dále si mezi-paměť ukládá adresu daných dat. K tomu využívá *Tag* a *Index*, které jsou získané z adresy.

[2] Některé procesory obsahují i čtyři úrovně mezi-pamětí pro zrychlení grafických aplikací. Nicméně většina článků, stránek apod. se zmiňuje o třech úrovních, což je mnohem běžnější praxe. I zmínky o čtyřech úrovních na fórech je označovali za nestandardní.

Při přístupu do mezi-paměti se adresa rozdělí na tři části - *Tag*, *Index* a *Offset*, kde:

- *Index* je adresa bloku v mezi-paměti.
- *Offset* adresuje data uvnitř bloku.
- *Tag* slouží k rozlišení adres, které se namapují na stejný index.

Jejich velikost závisí na velikosti mezi-paměti. Obrázek 2.2 vyobrazuje způsob dělení adresy.



Obrázek 2.2: Rozdělení adresy na *Tag*, *Index*, *Offset*

Asociativita

Mezi-paměti se dají rozdělit do tří kategorií podle způsobu adresování dat uvnitř mezi-paměti, tedy podle asociativity[13]:

- Přímo mapovaná.
- Plně asociativní.
- N-cestná asociativita.

V **přímo mapované** mezi-paměti má každá adresa své konkrétní místo, kam může být namapována. Podle dekódovaného *Indexu* se namapuje adresa na konkrétní blok mezi-paměti. Není možné použít jiný blok, i kdyby byl volný.

Oproti tomu stojí **plně asociativní** mezi-paměť, kde se každá adresa může namapovat na jakýkoliv blok. V této implementaci má *Index* velikost nula (nepoužívá se). Oproti přímo mapované sice nabízí mnohem větší úspěšnost přístupů, nicméně je mnohem složitější na implementaci, protože mezi-paměť musí porovnávat *Tag* z přistupované adresy s *Tagem* každého bloku v mezi-paměti, což zvyšuje složitost implementace.

Mezi nimi samozřejmě existuje i kompromis v podobě **N-cestné** mezi-paměti, kde N značí, kolik adres se může namapovat na jeden *Index*. V tomto případě se nejprve namapuje *Index* z adresy na *Index* v mezi-paměti a poté výběr funguje stejně jako u plně asociativní.

Politiky výměny

Pokud přistupovaná adresa v paměti není a namapované místo je volné, mezi-paměť data načte. Problém nastává v situaci, kdy namapované místo volné není. V případě **přímo mapované** mezi-paměti je postup jednoduchý: Pokud daný blok obsahuje přepsaná data (má nastavený *Dirty* bit na hodnotu 1), nejdříve se uloží do paměti, a poté se načte blok nový. Pokud obsahuje data nepozměněná, rovnou se přepíše.

Protože v případě **plně asociativní** nebo **N-cestné** mezi-paměti, více adres sdílí jeden *Index*, musí nejprve vybrat blok, který bude nahrazen. Existují různé politiky výměny [3] oběti (*victim*), neboli bloku, který se má nahradit. Zde zmíním pár jednoduchých z nich,

ale některé složitější politiky využívají podobné principy.

Náhodná oběť
Nejjednodušší politika, která vybere náhodně svoji oběť.

Politika FIFO
Politika FIFO vybírá jako oběť blok, který je v mezi-paměti nejdéle. Tato politika je velmi jednoduchá na implementaci, nicméně nebere v potaz, jakým způsobem kód s daty pracuje, a tedy může vyměnit často používaný blok.

Politika LRU
Relativně často používaná a jednoduchá politika, která si udržuje informaci o pořadí posledních přístupů k jednotlivým blokům. Když musí vybrat oběť, vybere nejdéle nepoužitý blok (*Least Recently Used*). Uvedená politika vylepšuje FIFO o ochranu posledně používaných bloků, ale má velký problém, pokud program pracuje s daty většími než mezi-paměť, protože každý přístup způsobí výpadek. Tomuto chování se v angličtině říká *Trashing* [3].

Politika LFU
Politika vybírá jako oběť nejméně často používaný blok (*Least Frequently Used*). Přístup udržuje často používané bloky uložené. Nevýhodu představuje, že se špatně vypořádá s tokem programu, kde dříve často používané kusy paměti již program nepotřebuje používat. Takové bloky zůstanou uloženy a blokují místo dalším. Proto se spíše využívají verze tohoto algoritmu, které tento problém řeší [3].

Přístupy do paměti

Další nastavení, jež mezi-paměti mohou mít, ovlivňují jejich přístup k zápisům dat. Pokud již mezi-paměť obsahuje daný blok, může se zachovat dvěma způsoby:

- Data se ihned propíší do paměti (*Write-through*).
- Data se propíší do paměti až při výměně této cesty (*Write-back*).

Pokud se data propíší do paměti ihned, nemusí si mezi-paměť udržovat informaci, zda data nejsou změněna (*Dirty bit*).

Pokud mezi-paměť daný blok neobsahuje, opět se může zachovat dvěma způsoby:

- *Mezi-paměť* počká, než se daný blok načte, a pak data uloží(*Write-allocate*).
- *Mezi-paměť* rovnou zapíše do paměti.

L2 mezi-paměti

Mezi paměti na druhé úrovni (a případně dalších) již nemají tak tvrdé nároky na rychlost odpovědi, což umožňuje větší velikost. Nicméně při použití více úrovní je potřeba určit, jak bude každá další úroveň pracovat s daty na nižších úrovních. Konkrétně existují dvě hlavní možnosti [20]:

- Inkluzivní: Obsahuje všechna data jako nižší úrovně.
- Exkluzivní: Obsahuje jen data, co se nenachází na nižší úrovni.

Exkluzivní mezi-paměť se na první pohled může jevit jako lepší volba. Protože si další úrovně nepamatují stejná data jako na nižších úrovních, celá kapacita se využije na další data. Ovšem uvedené s sebou přináší mnohem složitější implementaci při více-jádrových systémech, kde se musí udržovat koherence mezi-pamětí mezi jádry. Kvůli složitější kontrolní logice má rovněž delší dobu odpovědi[20]. Díky větší velikosti má větší úspěšnost.

Obě z těchto variant využívají navíc malou mezi-paměť (*victim cache*, nebo *victim buffer*) na oběti první mezi-paměti.

Je těžké určit lepší z těchto přístupů. Každý má své výhody a stále se využívají.

2.4.4 Závěr

Cílem kapitoly bylo vysvětlení fungování procesoru a poukázání na jeho složitost. Člověk, který počítačům rozumí, může pochopit fungování procesorů bez názorných ukázek. Mnoha studentům ale může dělat problémy si danou problematiku představit, a proto je vhodné názorné vysvětlení. Což byl důvod vzniku simulátoru Jana Vávry i této navazující práce.

Kapitola 3

Překladače

V kapitole bude popsáno fungování překladačů. Vycházet budu převážně z knihy *Elements of Compiler Design* [9].

Psaní kódu přímo v jazyce symbolických instrukcí představuje velmi náročný a zdlouhavý proces. Pro jeden typ procesoru bychom teoreticky byli schopni napsat nejoptimálnější kód, který by využíval plně cílenou architekturu. Takový vývoj by ale byl příliš nákladný, proto se v podstatě nikdy nevyplatí. Nejenom že by vývoj byl pomalý, navíc bychom byli závislí na používání stejného typu procesoru.

Z toho důvodu se využívají překladače, které obecně přeloží kód z jednoho jazyka do druhého, a přitom zachovají jeho funkcionalitu. Většinou se překladače využívají na přeložení programu napsaném ve vyšším programovacím jazyce (C, C++, Pascal atd.) na cílovou architekturu. Díky tomu můžeme stejný kód využít na více architekturách a není nutné znát tyto architektury tak důkladně. Kód napsaný ve vyšším programovacím jazyce je mnohem čitelnější a rychlejší na vývoj. Jeden překladač může podporovat více zdrojových jazyků, protože některé části překladu lze použít znovu.

3.1 Fáze překladu

Během překladu kódu zpracovávají překladače program v několika fázích: [9]

- Lexikální analýza.
- Syntaktická analýza.
- Sémantická analýza.
- Generování vnitřního kódu.
- Optimalizace vnitřního kódu.
- Generování cílového kódu.

Lexikální analýza zpracovává program a rozděluje jej do jednotlivých *lexémů* (slov). Zpracování provádí procházením znak po znaku a podle dané sekvence znaků odděluje jednotlivé *lexémy*. Z těchto *lexémů* vytváří *tokeny*, které jednotně reprezentují *lexémy*. V případě potřeby přidává k *lexémům* další charakterizující atributy. Vzniklé *tokeny* následně posílá do syntaktické analýzy.

Syntaktická analýza zkoumá strukturu *tokenů* a vyhodnocuje, zda splňují syntax konkrétního jazyka. Syntax jazyka je dána gramatickými pravidly, ze kterých se vytvoří

syntaktický analyzátor. Ten transformuje daný program pomocí těchto pravidel k utvoření syntaktického stromu. V případě, že se syntaktický strom nepodaří sestavit, překlad selže, protože vstupní program není validní. Když se strom podaří sestavit, je program syntakticky správně.

Sémantická analýza zkoumá, zda program splňuje sémantiku daného jazyka. Konkrétně se jedná převážně o typovou kontrolu. Chyby v této části mohou způsobit selhání překladu nebo pouze varování, kdy se překladač s touto chybou vypořádá automatickou konverzí.

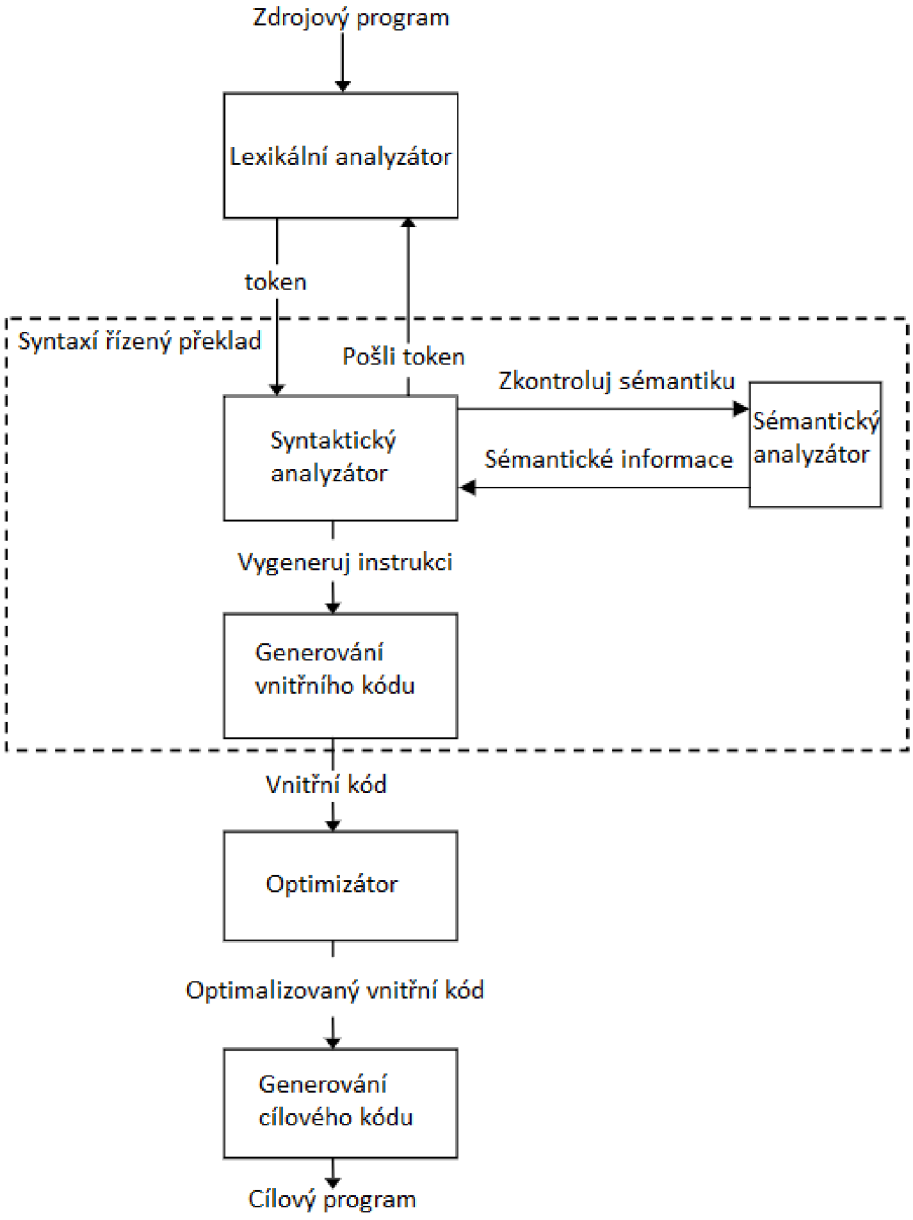
Generování vnitřního kódu převádí syntaktický strom do vnitřního kódu reprezentující stejnou funkcionalitu jako vstupní kód, ale většinou v podobě *tří adresného kódu*. Ten znázorňuje každý příkaz vstupního kódu v podobě krátké sekvence jednoduchých instrukcí. Tří-adresný kód nezávisí na cílové architektuře.

Optimalizace vnitřního kódu zpracovává vygenerovaný vnitřní kód a snaží se instrukce poskládat, aby maximalizovala výkon. Tato fáze je často aplikována opakovaně, jelikož některé optimalizace mohou vytvořit prostor pro optimalizace nové. Mohou být všeobecné nebo cílené na konkrétní architekturu. Například přeskládání operací, aby bylo možné využít speciální instrukce dané architektury, nebo plné využití jejich registrů.

Generování cílového kódu převádí optimalizovaný kód na cílovou architekturu. Z toho důvodu musí být cílová architektura překladači známá. V této části využívá informace o počtu registrů, dostupných instrukcích apod.

3.2 Syntaxí řízený překlad

Během překládání programu nemusí být uvedené fáze vykonávány postupně, ale často se překrývají z důvodu rychlejšího překladu. Syntaktická struktura obsahuje nejvíce informací pro překlad, proto je hlavním prvkem překladu a řídí jej. V průběhu zpracovávání programu si syntaktická analýza žádá od lexikální analýzy další *tokeny*. Ty po zpracování předá sémantické analýze, která je zpracuje a přidá případné doplňující charakteristiky. Výsledný strom zpracuje generátoru vnitřního kódu a dále již kód putuje ve fázích, jak bylo výše popsáno. Celou strukturu je možné si prohlédnout na obrázku 3.1.



Obrázek 3.1: Syntaxí řízený překlad [9]

Kapitola 4

Simulátory

Pro efektivní učení a vysvětlení principů procesorových architektur se jeví užitečné ukázat principy jejich fungování na konkrétních příkladech s interaktivním přístupem. K tomu slouží simulátory, což jsou programy, které se chovají jako cílový procesor. Vizuální simulátory například ukazují svůj stav a umožňují krokování. V této kapitole se podívám na volně dostupné simulátory, a poté v kapitole 5 prozkoumám simulátor, na který tato práce navazuje.

4.1 Simulátory procesorů

4.1.1 RIPES

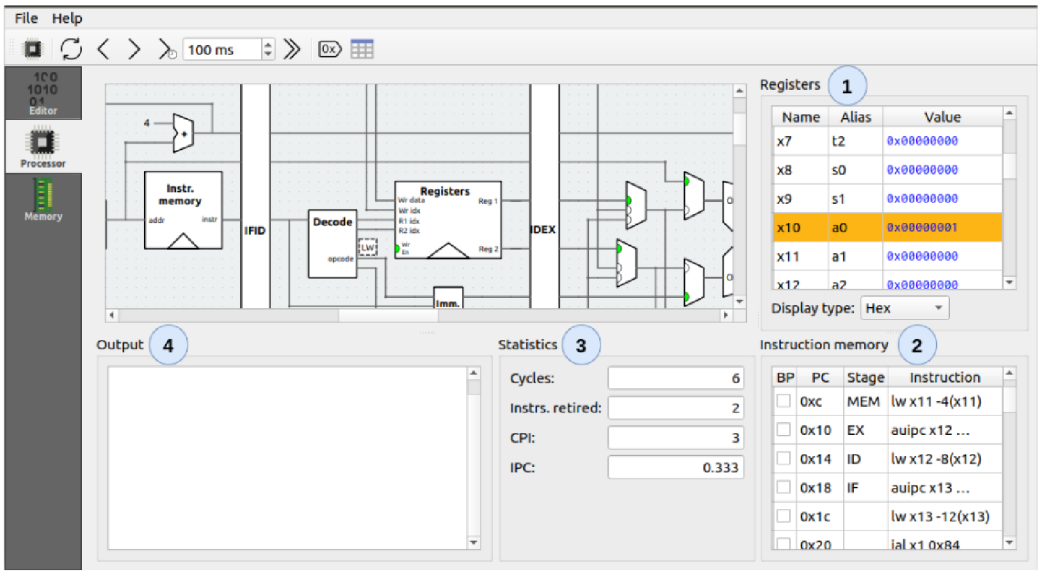
RIPES [11] je grafický simulátor a editor kódu jazyka symbolických instrukcí architektury RISC-V, který umožňuje i psaní kódu v jazyce C, pokud simulátor detekuje překladač z jazyka C do RISC-V, případně můžeme cestu k překladači zadat.

Zobrazení v hlavním okně

Simulátor v hlavním okně (na obrázku 4.1) umožňuje simulaci krok po kroku, s nastavením doby trvání jednoho kroku v milisekundách. V levé části okna je přepínání mezi zobrazením pohledu na Procesor/Editor/Paměť/Mezi-paměť/Vstup-výstup.

Uprostřed okna lze vidět architekturu procesoru s jednotlivými funkčními jednotkami a propojeními mezi nimi, rozložené až na multiplexory, které ukazují, jaký vstup do jednotky se právě vybírá, což ukazuje zelený bod na vstupu multiplexoru. Na propojení mezi jednotkami je možné kliknout, což dané propojení zvýrazní a zjednoduší přehlednost propojení jednotek. RIPES simuluje celé propojení procesoru, a tudíž lze v každém bodě programu zjistit, jakou hodnotu obsahuje port jednotky najetím na něj myší.

V pravé části okna se v horní části nachází obsah registrů, v dolní části okna je instrukční paměť. Toto okno zobrazuje adresu instrukce a její zápis v jazyce symbolických instrukcí. Navíc umožňuje nastavení instrukce, na které se simulátor zastaví před jejím vykonáním (*breakpoint*). Pokud se simulace přehrává krok po kroku, zobrazuje v jaké fázi vykonávání se jaká instrukce nachází. Tato data ukládá a případně umožňuje jejich zpětné zobrazení.



Obrázek 4.1: Hlavní okno simulátoru RIPES

RIPES umožňuje vypisování hodnot na výstup programu pomocí instrukce *ecall*, která zavolá funkci, jež vypíše data na výstupní okno. Nakonec RIPES obsahuje několik statistických informací:

- Počet vykonaných instrukcí.
- Kolik cyklů program trval.
- Počet cyklů na vykonání jedné instrukce (*CPI*).
- Počet instrukcí na jeden cyklus (*IPC*).

Obsažené architektury

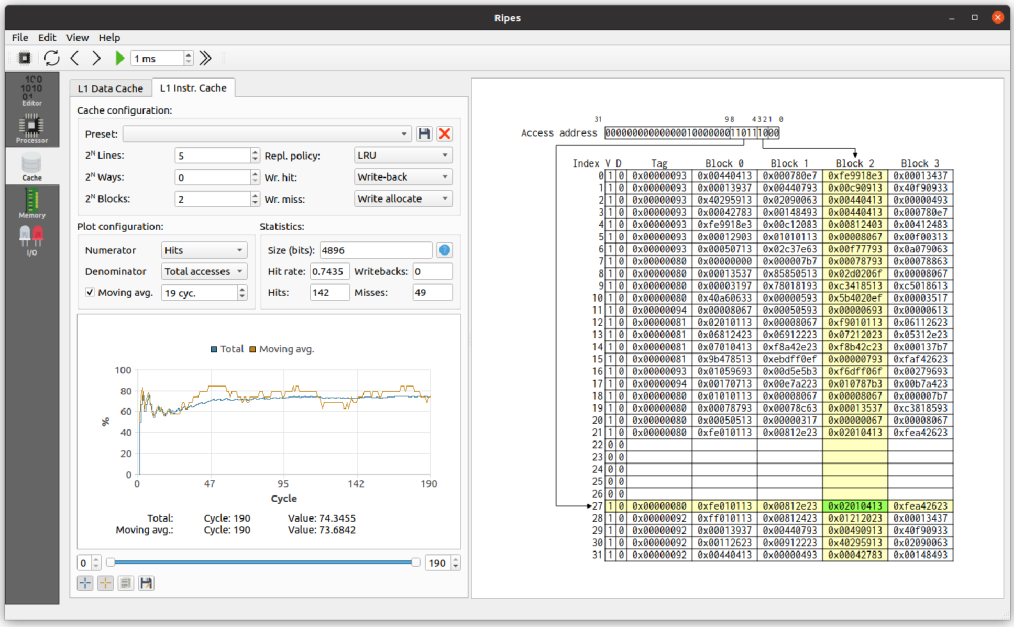
RIPES umožňuje výběr několika různě složitých RISC-V procesorů, které vykonávají instrukce v programovém pořadí. Nejjednodušší procesor vykonává vše v jednom cyklu. Poté obsahuje tři procesory o pěti fázích, s různou složitostí pomocné logiky:

- Nedetekující hazardy.
- Bez předávání výsledků, když nastane hazard.
- S předáváním výsledků, když hazard nastane.

Nejsložitější procesor je šesti-fázový a umožňuje vykonání dvou instrukcí v jednom cyklu.

Zobrazení paměti

V levé části okna je možné pohled přepnout na paměť. Okno paměti zobrazuje seznam sekcí v paměti a obsah konkrétních paměťových buněk s možností přepnutí se na jednotlivé sekce v programu a nebo na adresu na které je uložena hodnota z registru.



Obrázek 4.2: Zobrazení mezi-paměti v simulátoru RIPES

Zobrazení mezi-paměti

RIPES obsahuje L1 mezi-paměti pro data i pro instrukce. Na obrázku 4.2 je okno RIPES v pohledu na mezi-paměť. V levé části umožňuje nastavení mezi-pamětí a zobrazení statistik mezi-paměti. RIPES umožňuje nastavovat:

- Počty bloků.
- Asociativita.
- Velikost bloků.
- Politiku výměny bloků.
- Nastavení propisování do paměti.

Zobrazení statistik umožňuje vykreslení grafu se statistikami v čase a vypisuje celkové statistiky v aktuálním cyklu vykonávaní. RIPES sbírá tyto statistiky pro mezi-paměti:

- Počet přístupů do paměti.
- Počet úspěšných přístupů z procesoru do mezi-paměti.
- Počet neúspěšných přístupů z procesoru do mezi-paměti.
- Úspěšnost využití mezi-paměti.

V pravé části zobrazuje mezi-paměť jako tabulku se všemi hodnotami, které se v ní nachází. S aktuální přistupovanou adresou rozdělenou na jednotlivé části, se kterými mezi-paměť pracuje, a s vyznačenou buňkou v mezi-paměti, jež je právě přistupována.

Vstupně-výstupní okno

RIPES také obsahuje několik periferních zařízení s možností jejich připojení k procesoru, ovládání přes procesor a zobrazení ve vstupně-výstupním okně. Tyto zařízení si lze definovat vlastní pomocí QtWidgets. Nicméně simulátor obsahuje i pár před-připravených zařízení, konkrétně:

- Pole vypínačů.
- Led pole.
- Směrový ovladač.

Shrnutí

RIPES simulátor obsahuje kompletní vysvětlení fungování jednoduššího procesorového systému, včetně mezi-pamětí, překladače a vstupně výstupních operací. Obsahuje velkou část funkcionality, která představuje cílem i této práce, a navíc některé další funkcionality. RIPES obsahuje pouze jednoduchý procesor, ale nejspíše by bylo možné architekturu procesoru rozšiřovat nebo měnit, ale bylo by to náročné. Vytvoření a simulace procesoru je vytvořena pomocí nástroje, který vznikl z projektu RIPES. Některá rozšíření, například další statistiky a více mezi-pamětí, by pravděpodobně nebylo možné upravovat.

4.1.2 QTMips

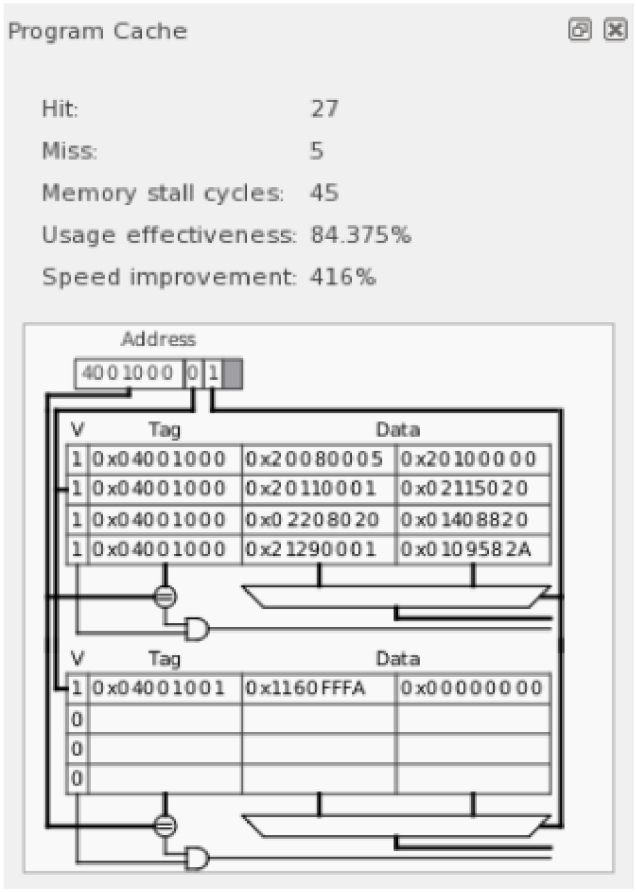
QTMips[8] představuje simulátor pro procesory s architekturou MIPS vyvinutý na ČVUT v rámci diplomové práce. Simulátor umožňuje výběr několika jednoduchých MIPS procesorů, kde nejsložitější nabízí zřetězení s detekcí hazardů a mezi-paměťmi. V hlavním okně se nachází procesor rozdělený na jednotlivé stupně. Nad každým stupněm je název právě vykonávané instrukce v dané fázi. Na levé straně se nachází obsah registrů a v pravo kód programu s adresami instrukcí. Simulátor zobrazuje propojené jednotky, včetně propojujících komparátorů a multiplexorů, kde některé výstupní porty obsahují hodnotu na nich. Toto zobrazení až na logické obvody je použité i v rámci zobrazení mezi-pamětí, které je na obrázku 4.3.

Simulátor také obsahuje nastavitelné instrukční a datové mezi-paměti stejným způsobem jako RIPES. Opět propojené až na logické součástky. Mezi-paměti sbírají navíc výkonové statistiky:

- Počet úspěšných přístupů z procesoru do mezi-paměti.
- Počty neúspěšných přístupů z procesoru do mezi-paměti.
- Počet cyklů na kolik mezi-paměť zastavila procesor.
- Úspěšnost využití mezi-paměti.
- Zrychlení získané mezi-pamětí.

4.1.3 OpendDLX

Simulátor OpenDLX[14] implementuje jednoduchý *MIPS* procesor zpracovávající instrukce v pěti fázích. OpenDLX nezobrazuje architekturu, ale místo toho obsahuje podrobné informace, v jaké části výpočtu se nachází konkrétní instrukce v daném cyklu vykonávání programu.



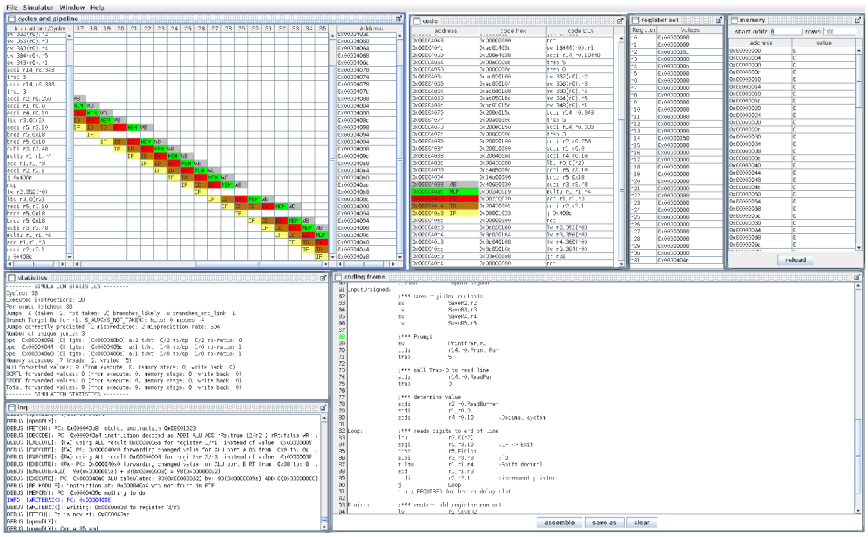
Obrázek 4.3: Zobrazení mezi-paměti v simulátoru QTMips

Dále OpenDLX obsahuje okno se statistikami vykonávání programu:

- Počet načtených instrukcí.
- Počet dokončených instrukcí.
- Úspěšnost predikce skoku.
- Počet skoků.
- Počet přístupů do paměti.
- Počet cyklů.

Tento simulátor nezobrazuje vizuálně činnost jednotek, jako některé ostatní simulátory, ale vypisuje vykonané operace do okna **log**. Výpisy obsahují informace, že jednotka dokončila výpočet, prováděla danou operaci a její výsledek, ale i například, že daná jednotka neprováděla žádnou operaci.

Simulátor také obsahuje okno k napsání kódu v jazyce symbolických instrukcí architektury *MIPS*

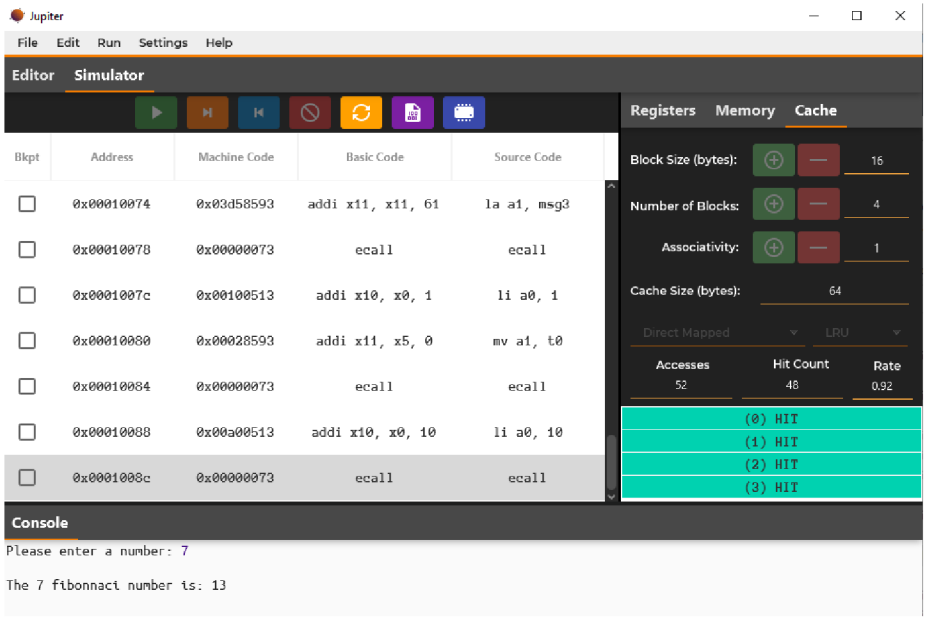


Obrázek 4.4: Hlavní okno simulátoru opendlx

4.1.4 Jupiter

Jupiter[6] je otevřený RISC-V simulátor, který implementuje základní 32-bitovou instrukční sadu (RV32I) a rozšíření: Násobení (M) a operace v plovoucí řádové čárce (F), s tím, že se připravuje nová verze s dalšími rozšířeními.

Tento simulátor se zaměřuje na práci s kódem. Umožňuje kompilaci a simulaci velkých souborů i složek. Obsahuje podporu výpisu do konzole pomocí *ecall* RISC-V instrukce. Navíc umožňuje nastavení parametrů mezi-paměti, ale nezobrazuje informace o obsahu mezi-paměti, pouze informuje zda byl přístup úspěšný.



Obrázek 4.5: Simulátor Jupiter

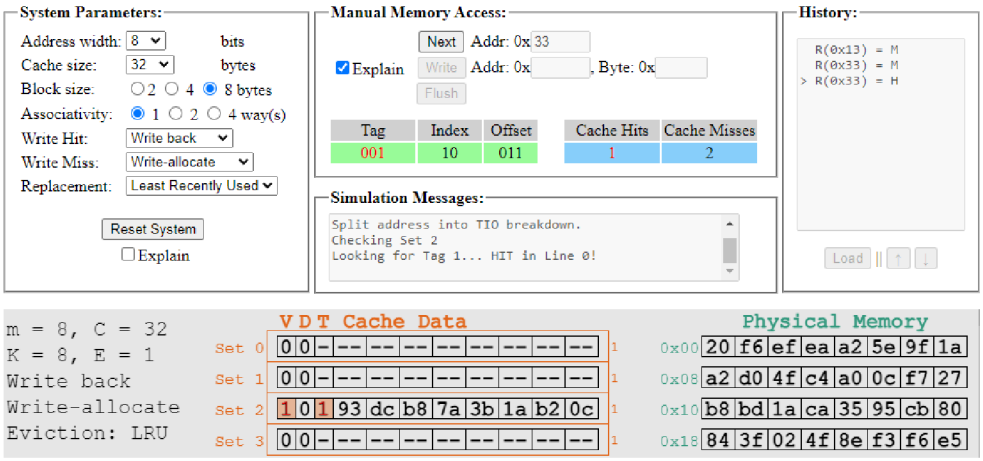
Uvedený simulátor se může hodit na zkoumání toku programu a vysvětlení programování v jazyku symbolických instrukcí RISC-V. Pro účely této práce Jupiter ale nenabízí nic užitečného, co by již nenabízely další simulátory.

4.2 Simulátory mezi-pamětí

Většina simulátorů procesorů, které jsem zde popsal, obsahovaly i mezi-paměti. Jelikož se tato práce zaměřuje na přidání paměťového subsystému do simulátoru procesoru, zaměřil jsem se i na simulátory mezi-pamětí. Všechny zkoumané simulátory, které v této části popíšu, pochází z univerzit, což může značit užitečnost grafických simulátoru pro výuku.

4.2.1 Simulátor mezi-paměti z washingtonské univerzity

Webový simulátor [1] z washingtonské univerzity má velmi jednoduchý vzhled, díky čemuž se používá intuitivně.



Obrázek 4.6: Simulátor mezi-paměti z washingtonské univerzity

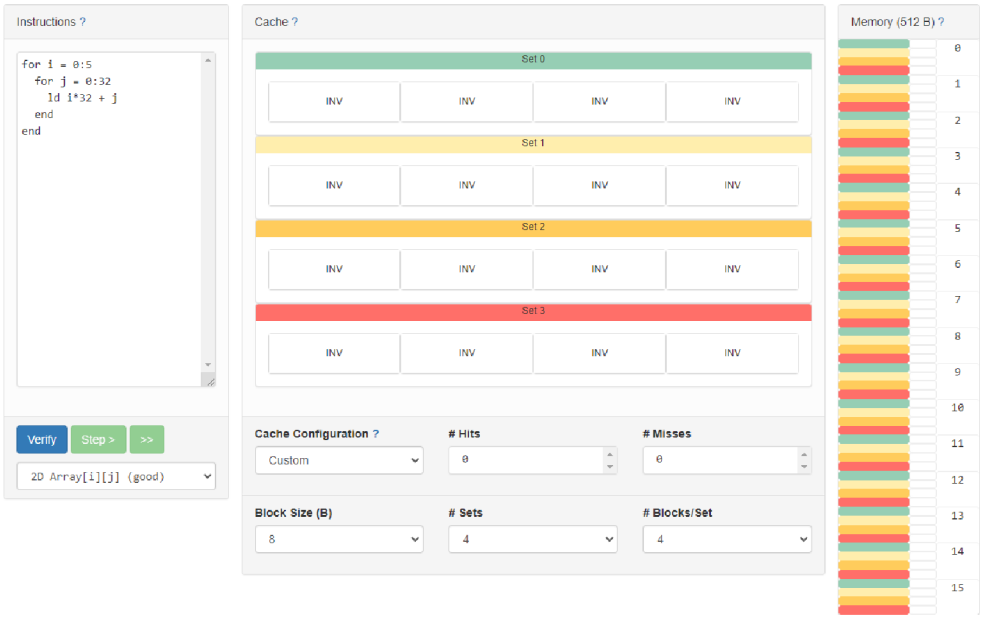
Umožňuje nastavování:

- Velikosti mezi-paměti a jejích bloků.
- Asociativity.
- Chování při (ne)úspěšném přístupu do mezi-paměti.
- Politiky vyměňování: FIFO, LRU, Cyklicky.

Na začátku je možné si vybrat nastavení mezi-paměti, a pak vygenerovat systém, což nám zároveň vygeneruje obsah paměti o velikosti adresního prostoru, který byl zadán. Poté je možné provést čtení/zápis mezi-paměti nebo vymazání obsahu mezi-paměti. U generování i u přístupu do mezi-paměti umožňuje zadat, zda chceme vysvětlit, co mezi-paměť dělá a projít si její činnost krok po kroku. Navíc obsahuje historií operací což umožňuje si příklad projít znovu.

4.2.2 Simulátor mezi-paměti z michiganské univerzity

Webový simulátor[7] z michiganské university na první pohled vypadá lépe. Tento simulátor nepracuje s daty, ale zaměřuje se pouze na adresy. Umožňuje vytvářet jednoduchý kód, který obsahuje pouze operace načtení/uložení do paměti a cyklus *for*. Všechny bloky v paměti, která má 512B, jsou zbarveny stejnou barvou jako bloky v mezi paměti, takže je zřejmé mapování mezi bloky mezi-paměti a bloky paměti.



Obrázek 4.7: Simulátor z univerzity v Michiganu

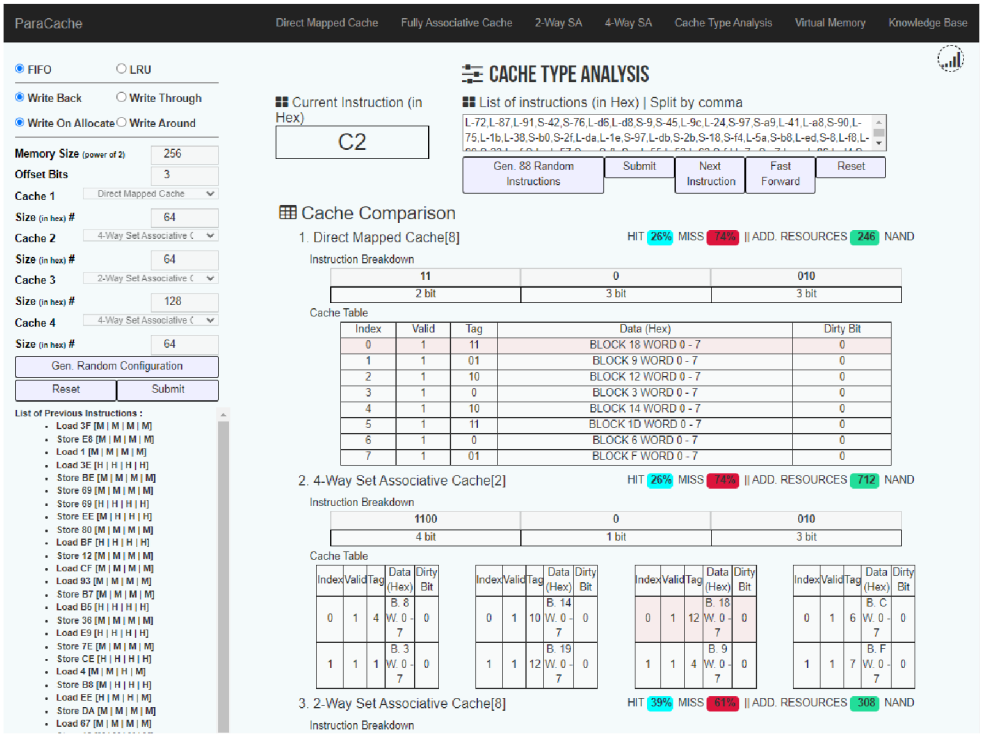
Simulátor umožňuje nastavení:

- Velikosti a počtu bloků.
- Asociativity.

Při krokování programu, vybrané místo v mezi-paměti blikne. Tento simulátor používá pouze *LRU* politiku výběru oběti.

4.2.3 Simulátor mezi-paměti z nanyangské univerzity

Webový simulátor[10] z nanyangské univerzity, který je možné si prohlédnout na obrázku 4.8, umožňuje podobné nastavování mezi-pamětí jako simulátor z washingtonské univerzity, ale poskytuje větší výběr velikosti paměti.



Obrázek 4.8: Simulátor mezi-paměti z nanyangské univerzity

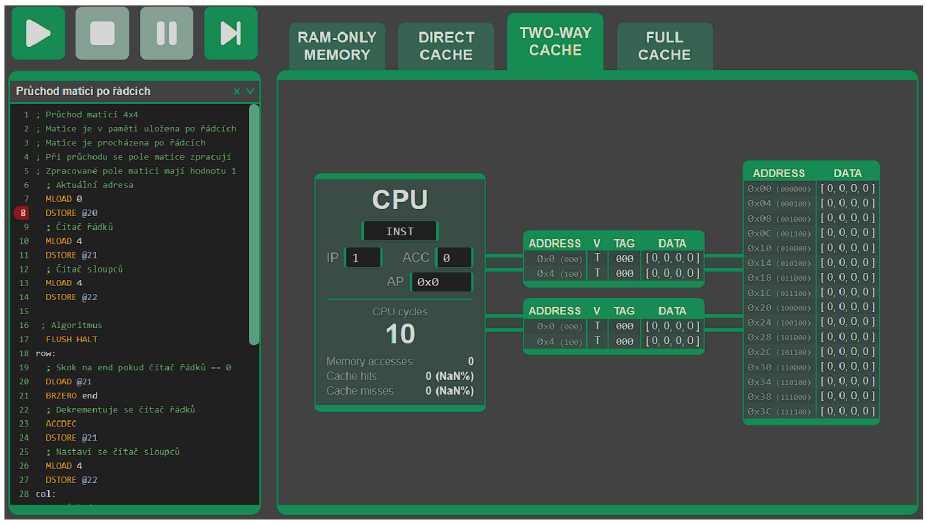
Stejně jako simulátor z waschingtonské univerzity umožňuje krok po kroku vysvětlovat činnost mezi-paměti, ale barvy zobrazení způsobují, že vysvětlivky jsou někdy nečitelné. Nicméně umožňuje zvolení čtyř mezi-pamětí s různými konfiguracemi a porovnání jejich výkonu na stejných datech, což ostatní simulátory nemají. Navíc je k tomuto simulátoru přiložena skvělá prezentace, která vysvětluje principy mezi-paměti a virtuální paměti.

4.2.4 Simulátor mezi-paměti z brněnské univerzity

Webový simulátor z brněnské univerzity [12] (na obrázku 4.9), obsahuje jednoduchou malou mezi-paměť s možným nastavením mezi:

- Přimo mapovanou.
- Dvou-cestnou.
- Plně asociativní.

Oproti předchozím simulátorům má jednoduché nastavení i malou velikost, ale jeho přednost spočívá v ukázce funkcionality, kde mezi-paměť dělá svoji činnost krok po kroku se zvýrazněním právě aktualizovaných bloků. Rovněž obsahuje jednoduchý jazyk pro tvorbu programů ukazujících funkci mezi-paměti a několik ukázek základních algoritmů, jako například procházení matic.



Obrázek 4.9: Simulátor mezi-paměti z brněnské univerzity

4.3 Shrnutí

Ze všech simulátorů mě nejvíce zaujal simulátor RIPES, který obsahuje v podstatě všechnu funkcionalitu, která je cílem této práce, a ještě něco navíc. Největší jeho nevýhodu představuje, že pro účely předmětu AVS obsahuje příliš jednoduché procesory a že je napsaný v c++. Nicméně v dalším pokračování práce může posloužit jako dobrá inspirace. Zároveň po vyzkoušení simulátorů mezi-pamětí je zřejmé, že přímé vysvětlení fungování mezi-paměti krok po kroku již také existuje. Pouze RIPES ale kombinuje oba přístupy dohromady.

Kapitola 5

Superskalární simulátor od Jana Vávry

V této kapitole se budu zabývat prací Jana Vávry [16], na niž navazuje moje diplomová práce, grafickým simulátorem superskalárního procesoru. Konkrétně se zaměřím na funkcionalitu obsaženou v simulátoru a způsob jeho implementace.

5.1 První spuštění

Simulátor lze stáhnout ze stránky uvedené v Readme souboru zdrojových kódů tohoto projektu. Nicméně v rámci prvního spuštění jsem se rozhodl si simulátor sestavit na svém počítači, abych mohl pokračovat dále ve vývoji. Simulátor je implementován v jazyce Java15 v prostředí Intellij-idea s grafickým prostředím implementovaným pomocí JavaFX16.

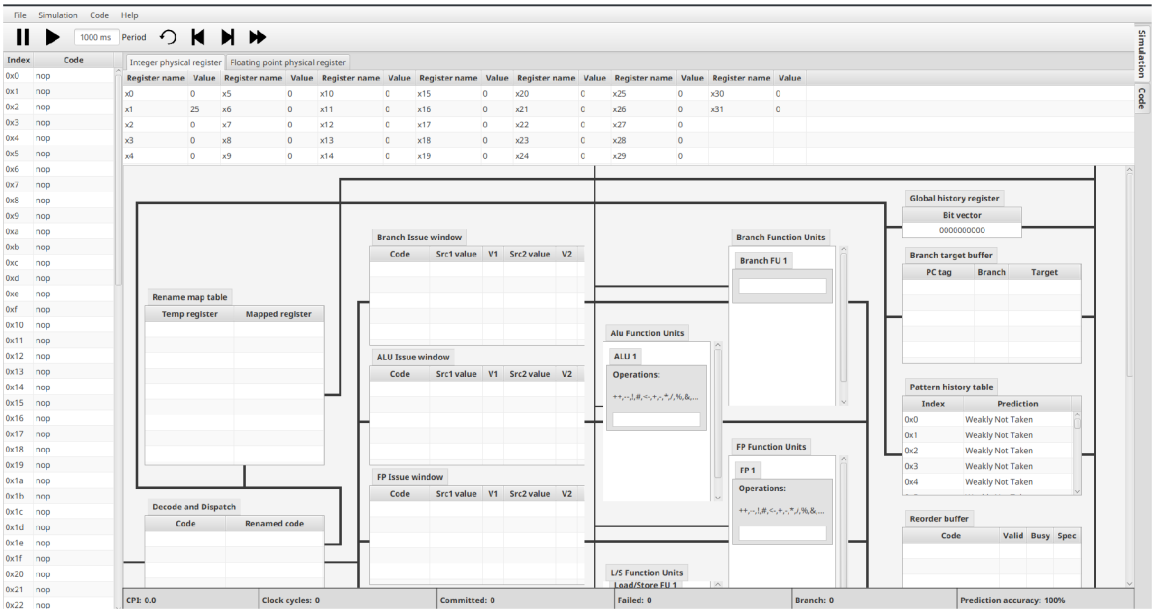
Během prvního spuštění jsem narazil na řadu problémů, které většinou vzešly z toho, že jsem pečlivě nenásledoval informace v Readme souboru, jelikož jsem je často špatně pochopil. V rámci prvního spuštění jsem i tedy upravoval Readme soubor, aby bylo více jasné, jakým způsobem lze tento program zprovoznit.

5.2 Pohled v hlavním okně

Program se spustí v okně s propojenými funkčními jednotkami superskalárního procesoru (obrázek 5.1). Na levé straně se nachází prázdný kód programu, ve vrchní části okna obsah registrů a nad nimi menu na spuštění simulace. Simulaci je možné spustit na celém programu nebo udělat pouze jeden krok. Zároveň umožňuje automatické krokování simulace po čase zadaném uživatelem, se základním nastavením na jednu sekundu. Dále simulátor nabízí možnost dělat i zpětné kroky.

5.3 Analýza funkcí

Fungování simulátoru si lze vyzkoušet na ukázkových programech, které se nacházejí v menu v sekci *code→examples* napsaných v jazyce symbolických instrukcí architektury RISC-V. Umožňuje rovněž měnit nastavení simulace v menu *simulation→configuration*: Počty funkčních jednotek, množství načítaných a dokončovaných instrukcí, velikosti v prediktoru skoků a velikosti *reorder* a *Load-Store bufferu*. Menu nastavení je vyobrazeno na obrázku 5.2



Obrázek 5.1: Hlavní okno simulátoru

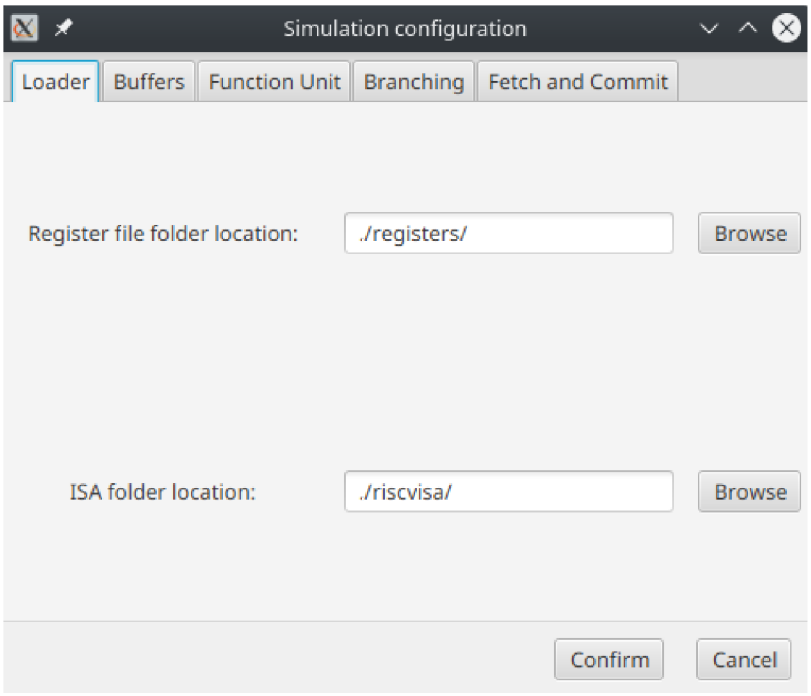
Simulátor také obsahuje možnost napsání programu v assembleru architektury RISC-V, k čemuž se dostaneme po kliknutí na záložku *code* v pravé částí okna. Toto okno umožňuje uložení a načtení programu, otevření více záložek s programy a překlad, nebo spíše analýzu, aktuálního vybraného kódu. Průběh analýzy se vypisuje v dolní části okna v sekci *console*, kde se zobrazí řádky s chybami a informace o druhu chybu (např. neznámá instrukce). Dostupné instrukce na dané architektuře lze zobrazit v menu sekci *help→Instruction list*

5.4 Analýza kódu

Zdrojové soubory simulátoru se nachází v repozitáři ve složce *Source*. V této složce je možné najít:

- examples
- registers
- riscvisa
- gradle
- src
- testfiles

První tři z uvedených složek přímo reflektují sekce objevující se v programu. Ve složce *examples* se nachází ukázkové programy. Složka *registers* obsahuje nastavení celočíselných a desetinných registrových polí. A *riscvisa* obsahuje seznam podporovaných (RISC-V) instrukcí s jejich zápisem v jazyce symbolických instrukcí, prováděnou operací a typy operandů. Pro sestavení programu se využívá systém *gradle*. Tento systém zároveň pracuje se složkou *testfiles* pro automatické testování.



Obrázek 5.2: Nastavení simulátoru

5.5 Zdrojové soubory

Hlavní část implementace se nachází ve složce *src/main/java/com.gradle.superscalarsim*. Složka obsahuje:

- blocks - implementace funkčních jednotek (*ALU*, *FPU*, *Branch* a *L/S*).
- code - parser a interprety kódu.
- di - vkládání závislostí (dependency injection).
- enums - enumerace a další definice.
- loader - načítání hlavního okna, registrů a instrukcí.
- models - Soubory s třídami, které obsahují a předávají stav simulace.
- ui - grafické rozhraní simulátoru.
- SimulationMain - hlavní třída simulátoru.

Kód je přehledně rozdělený do souborů, kde každý soubor obsahuje jednu třídu a ty obsahují všechny potřebné funkce pro přístup k vnitřním proměnným a prací s nimi. Zároveň kód obsahuje komentáře ve stylu pro *Doxygen*.

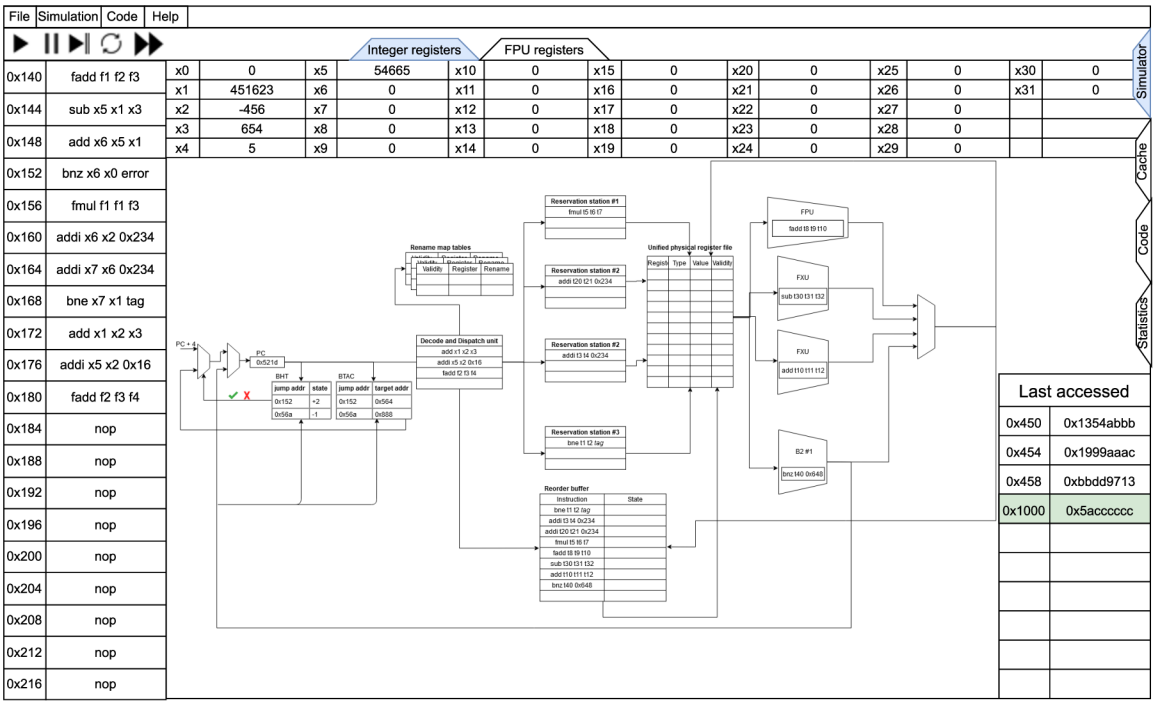
Kapitola 6

Návrhy rozšíření

Pro lepší využití v rámci předmětu AVS navrhuji rozšířit simulátor o lepší práci s pamětí. Hlavní část bude představovat přidání mezi-paměti. Mezi-paměť si zároveň bude sbírat své statistiky a tím se rozšíří funkcionalita sběru statistik. Pro lepší vysvětlení překladačů, a hlavně jednodušší tvorbu ukázkových příkladů, by mohl simulátor přijímat program v jazyce C nebo jemu podobném.

6.1 Hlavní okno

V hlavním okně, jehož návrh znázorňuje obrázek 6.1, bude minimální rozdíl, konkrétně půjde o přidání tabulky X posledních přístupů do paměti a záložek mezi-paměti a statistik.



Obrázek 6.1: Pohled do hlavního okna

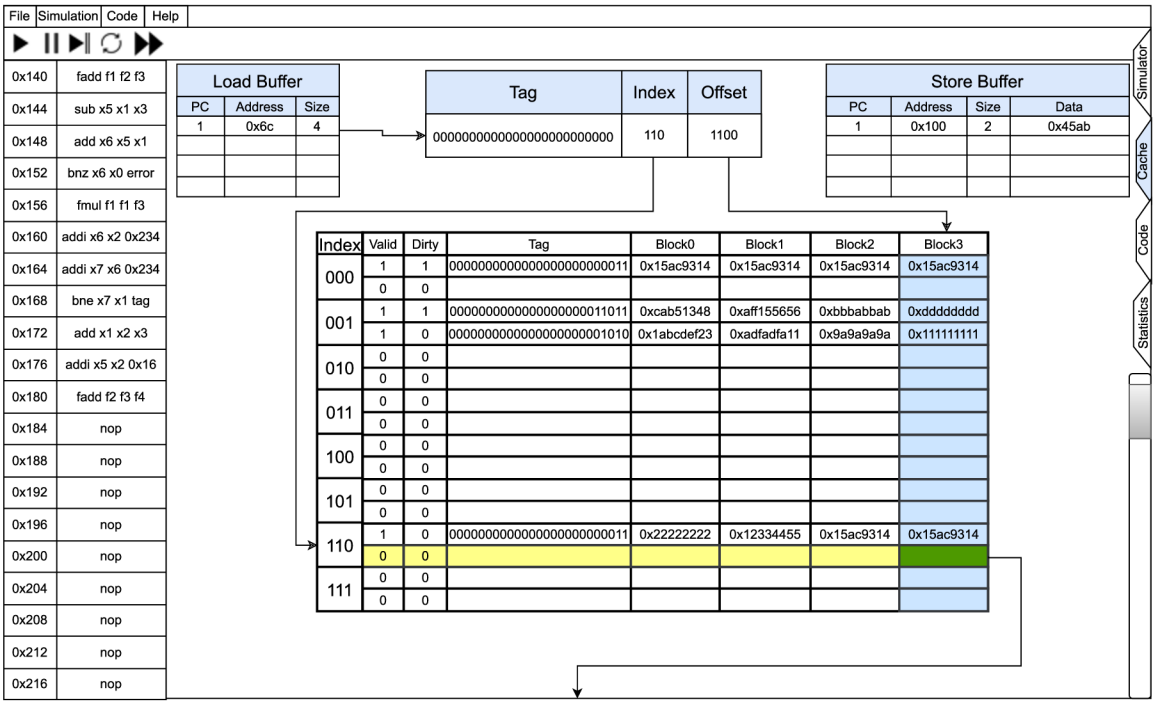
Zobrazení celé paměti v hlavním okně simulace by zabíralo moc místa nebo by bylo nepřehledné. Zobrazující komponenta by navíc musela umět zpracovat data paměti i z mezi-

paměti a zobrazit jen ty nejaktuálnější. Proto jsem se raději rozhodl implementovat přístup, kdy budu ukládat pouze poslední dokončené přístupy do paměti. Čímž bude lépe vizualizován chod programu. Data v této tabulce budou seřazena podle adresy a poslední přístup bude podbarvený, aby bylo zřejmé, která adresa se změnila.

Pro tuto tabulku budu získávat data z dokončených instrukcí. Ta si navíc bude muset pamatovat svůj předchozí obsah pro možnost zpětného krokování, což půjde implementovat jednoduše zásobníkem.

6.2 Návrh mezi-paměti

Rozšíření simulátoru o paměťový subsystém bude spočívat hlavně v implementaci mezi-paměti. Mezi-paměť bude v rámci simulátoru umístěna na vlastní záložce. V rámci tohoto okna budou zobrazeny také *L/S Buffery* z *Load/Store* jednotky [1] a tabulka s pamětí.



Obrázek 6.2: Návrh mezi-paměti

Implementace mezi-pamětí bude podporovat široké nastavení pro možnosti pokusů nad daným kódem. Konkrétně se bude jednat o:

- Velikosti: počet bloků a velikost jednoho bloku [2].
- Asociativitu: možnost nastavení na násobky 2.
- Přístup k *hit/miss* zápisu.

[1] Pro jednoduchost zobrazení a jednodušší implementaci bude mezi-paměť podporovat pouze jednu *Load/Store* jednotku. Toto by nemělo omezit hlavní cíl rozšíření, tedy vysvětlení fungování mezi-pamětí a jak by s nimi měly programy pracovat.

[2] Počítá se s tím, že pro velké velikosti bude zobrazení nepřehledné. V tomto případě již nebude potřeba potřeba zkoumat mezi-paměť samotnou, ale spíše její statistiky.

6.2.1 Integrace do kódu

Definice třídy mezi-paměti by měla být vložena ve zdrojovém kódu ve složce *blocks* podobně jako jiné jednotky. Implementaci svých funkcí bude mít ve složce *code*. Mezi-paměť ale nebude nic interpretovat, takže oproti jiným jednotkám nebude mít soubory pro interpretaci.

Tato jednotka bude obsahovat funkce:

- Uložit data: Adresa, šířka.
- Načíst data: Adresa, šířka.
- Simulovat zpět.
- Získat statistiky.
- Získat obsah.

Mezi-paměť bude možné simulovat dvěma způsoby. V hlavním okně simulace se bude chovat jako paměť a pouze vrátí data a zpoždění způsobené tímto přístupem (stále si bude sbírat statistiky). Při přepnutí do okna mezi-paměti bude umožňovat krokování. Toto krokování bude řízeno z okna simulace mezi-paměti, která bude vykreslovat její činnost.

Napojení třídy mezi-paměti do kódu simulátoru bude na stejném místě, jako je v tuto chvíli napojená paměť.

6.3 Překladač

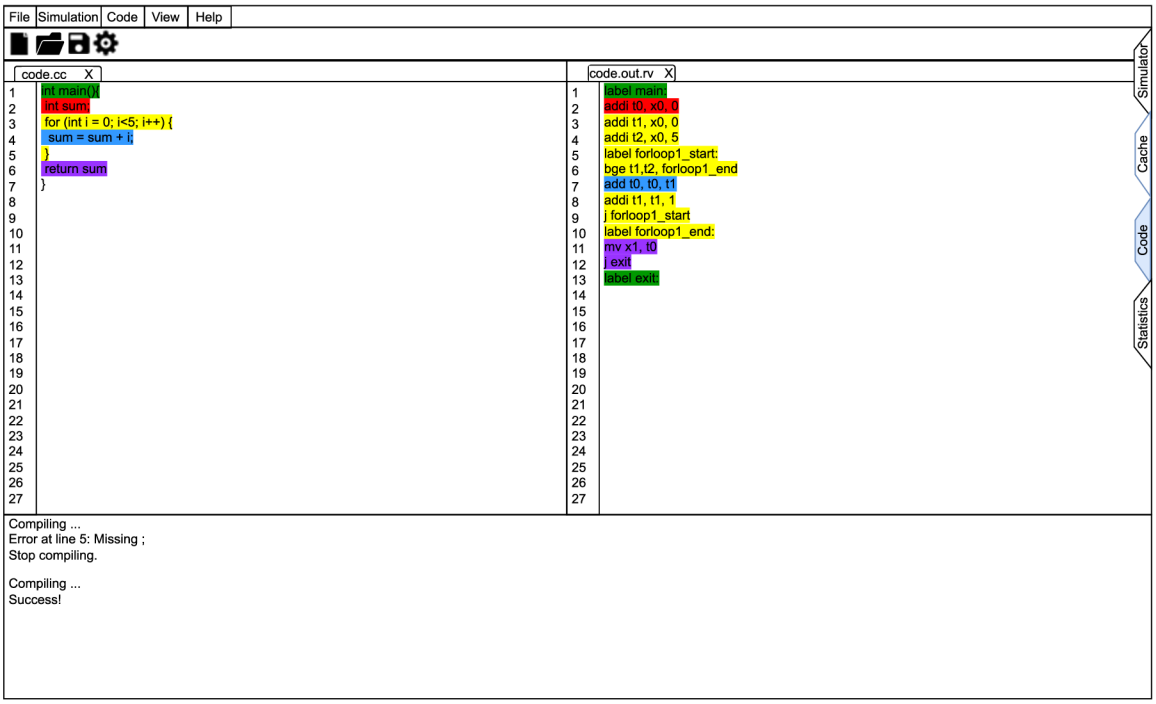
Rozšíření záložky *code* bude spočívat v překladači z vyššího programovacího jazyka do jazyka symbolických instrukcí. Konkretně jsem volil mezi dvěma možnostmi:

- Běžně používané překladače (např. Clang, gcc).
- Vlastní překladač z jazyka podobného C.

Hlavní přednost, kterou by měl překladač umět, představuje podbarvení každé důležité části kódu stejnou barvou ve vyšším programovacím jazyce i ve vygenerovaném jazyce symbolických instrukcí. Důvodem je lepší vysvětlení činnosti překladače a jednodušší porozumění jaké operace musí procesor udělat pro vykonání daného programu. Příklad jednoduchého programu s jedním cyklem si lze prohlédnout na obrázku 6.3.

Běžně používané překladače

Běžně používané překladače umožňují výběr cílové architektury a vygenerování assembleru, což by umožňovalo případnou podporu více architektur než RISC-V. Nicméně, aby celý simulátor podporoval jiné jazyky, je potřeba také definovat interpretaci těchto instrukcí, která je již definovaná pro RISC-V. Zároveň by bylo potřeba vygenerovaný *ASM* soubor ještě zpracovat kvůli správnému podbarvování textu a odstranění nadbytečností, které simulátor nepotřebuje nebo nepodporuje, přičemž by toto nejspíše bylo částečně jiné pro každou architekturu.



Obrázek 6.3: Návrh překladače

Vlastní překladač

Oproti tomu vlastní překladač umožní přirozené propojení překládaného a přeloženého kódu. Další velká výhoda vlastního překladače by mohla být možnost podpory téměř jakékoliv architektury. Zaměřený bude na RISC-V, který je v současnosti v simulátoru implementován. Při definici vlastních instrukcí, by měly být použity automaticky. Poslední výhodu tohoto řešení představuje jeho přenositelnost na jiné systémy.

Výběr překladače

Je samozřejmé, že běžně používané překladače zvládnou vyprodukovat kvalitnější kód než vlastní překladač, nicméně cílem není vyprodukovat co nejlepší kód, ale vysvětlit jeho fungování. A proto z důvodu většího množství výhod zvolím vlastní překladač.

6.3.1 Definice jazyka

Podpora celého jazyka C by byla zbytečně náročná a přinesla by málo užitku navíc. Z toho důvodu je potřeba vybrat důležité aspekty jazyka C. Ořezané C bude podporovat:

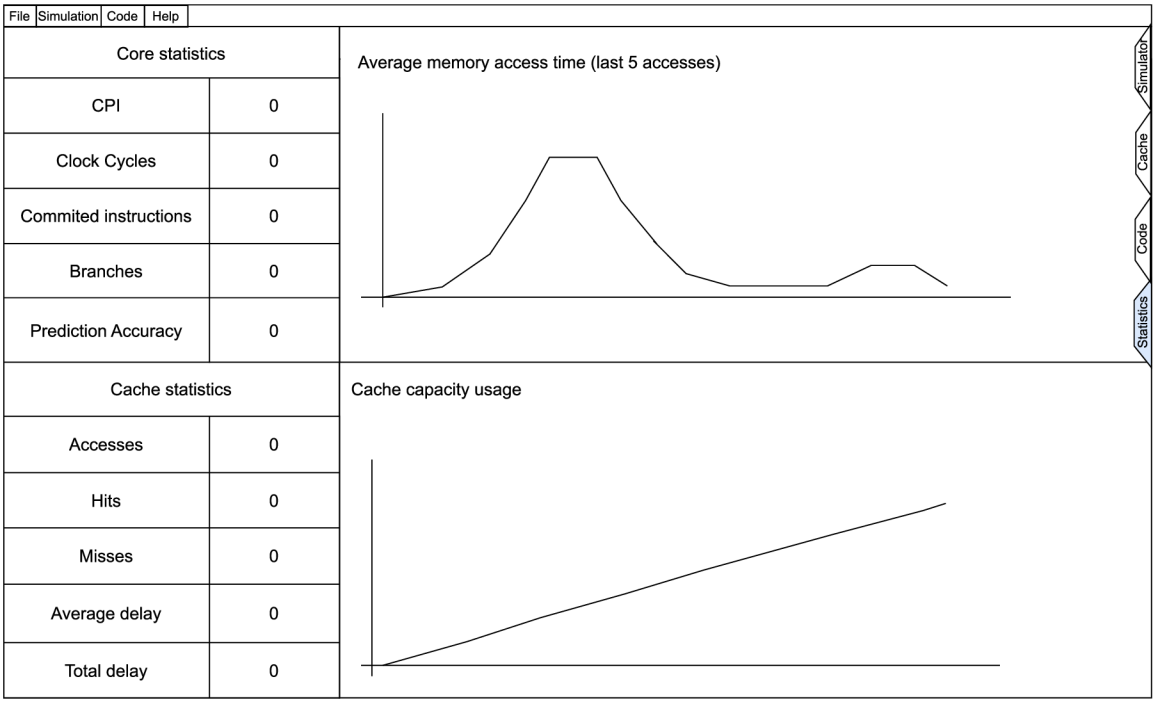
- typy: int, float
- řízení: for, while, if-then-else
- operace: +,-,*,/,%, », »>, «, <, >, ==, >=, <=, !=
- funkce
- main: musí být implementován, nepodporuje argumenty
- 1D pole se statickou šířkou

6.3.2 Integrace do kódu

Integrace do kódu bude poměrně jednoduchá, protože pouze rozšířím funkcionalitu současného překladače. Konkrétně rozšířím zobrazení v *ui/codeWindowController* a samotnou funkcionalitu v části *code* → hlavně v *CodeParser*.

6.4 Statistiky

Rozšíření o mezi-paměť s sebou přináší další statistiky, které je možné, a dokonce vhodné sbírat. Pro lepší zobrazení statistik vznikne nová záložka *Statistics*. Zde budou statistiky z procesoru, které jsou již v simulátoru, a navíc nové statistiky z mezi-paměti. RIPES [11] simulátor obsahoval vyobrazení stavu mezi-paměti v čase do grafu, což mě velmi zaujalo, protože se jedná velmi užitečný způsob vizualizace statistiky, tudíž grafy také zahrnu do statistik. Návrh vzhledu statistik si můžete prohlédnout na obrázku 6.4.



Obrázek 6.4: Návrh statistik

Kapitola 7

Implementace překladače

Nejprve jsem se rozhodl implementovat překladač, protože nejméně zasahuje do aktuálního kódu. Pro přidání překladače bylo potřeba upravit aktuální funkcionalitu překladače, který v současnosti kontroluje, zda kód splňuje syntax symbolických instrukcí. Dále bylo potřeba upravit grafické rozhraní.

7.1 Hlavní logika

Při spuštění překladu se vytvoří třída překladače a předá se jí zdrojový soubor, který ho bude zpracovávat syntaxí řízeným překladem (stejně jako bylo vysvětleno v kapitole 3).

Hlavní komponentu v tomto případě tvoří *Syntaktický analyzátor* 7.3, ale implementaci komponent popíšu podle postupu zpracování vstupního programu.

7.2 Lexikální analyzátor

Lexikální analyzátor přijme celý soubor, zpracovává ho a postupně ho rozděluje na lexémy. Konkrétně dostane již zpracovaný soubor načtený po řádcích do pole řetězců. Třída *CompilerParser* obsahuje funkce:

- New: (String[] vstupni soubor)
- GetNextToken: (),(Token)

Jeho hlavní práce spočívá ve funkci *GetNextToken*, který zpracovává vstup následujícím způsobem:

- Bílé znaky: Oddělovač, pošle se jako *token*, více bílých znaků za sebou se odstraňují.
- Speciální znaky (/,*,+,-,%,&,|,!,=,(,),;,[,],,): Oddělovač, pošle se jako *token*, posílá se každý znak.
- Ostatní znaky: Posílají se jako celé slovo, hranice určují oddělovače, nebo konec/začátek řádku.

7.3 Syntaktický analyzátor

Syntaktický analyzátor je hlavní komponentou řídící celý překlad. Z *tokenů*, které získává z *lexikálního analyzátoru*, se postupně vytváří syntaktický strom.

Vždy, když se dokončí podstrom až na listy (jeden příkaz), předá se daný podstrom *sémantickému analyzátoru*. Řídící prvky se pošlou k sémantické analýze, bez zpracovaného podstromu. Tedy v případě řídících příkazů (If, While, For) a funkcí se předá odděleně řídící část a tělo.

Podstrom doplněný o informace ze *sémantického analyzátoru* se dále předá generátoru tříadresního kódu.

Dokončený program se předá generátoru cílového kódu, který celý program převede na cílové instrukce a vytvoří jejich finální sekvenci.

Třída syntaktického analyzátoru obsahuje:

- New: (Lexikální analyzátor, Sémantický analyzátor, Generátor tří adresného kódu, Optimalizátor, Generátor kódu cílové architektury)
- Compile: (), ((String, Integer)[] Code)

7.3.1 Syntaktický strom

Syntaktický strom má tyto vlastnosti:

- Kořen: Vše definované v této části je globální.
- Každý uzel obsahuje číslo řádku, typ uzlu a hodnotu.
- Neomezený počet podstromů.
- Každý uzel obsahuje odkaz na nadřazený uzel, kromě kořene.
- Listmy jsou jednotlivé *Tokeny* a v případě řídících prvků jeho části.
- Každý uzel obsahuje vygenerovaný kód v tomto uzlu.

Hodnota v uzlu stromu je pořadí řídícího prvku, v případě listu konkrétní *token*.

7.3.2 Průběh syntaktické analýzy

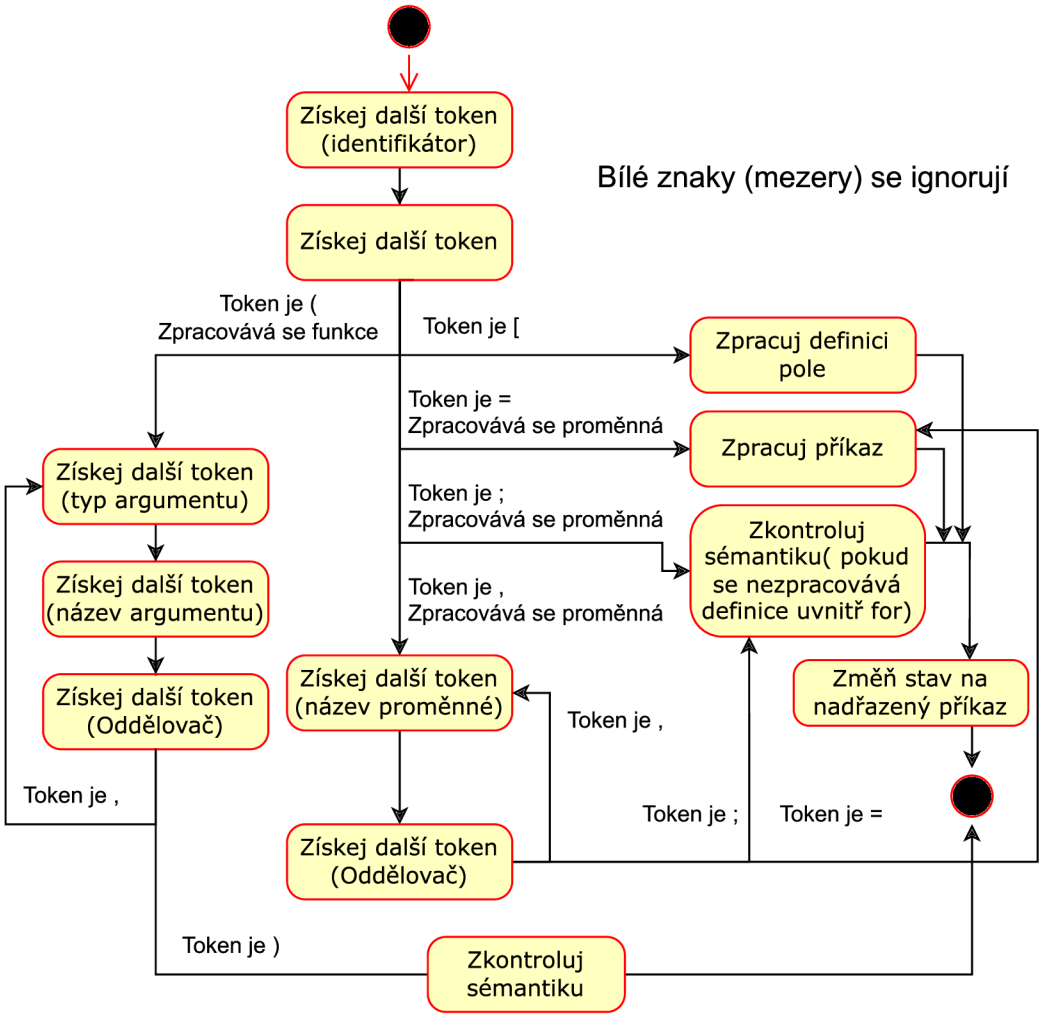
Většinu druhů příkazů je možné rozlišit již podle prvního slova v příkaze. Podle prvního slova se tedy vybere, jaký druh příkazu se zpracovává.

Zpracování řídícího příkazu

Zpracování řídícího příkazu se řídí podle toho, zda se zpracovává cyklus *For* nebo ne. Z pohledu správnosti syntaktické analýzy není rozdíl mezi příkazy *If* a *While*. Pro analýzu jednotlivých částí se využívá funkce zpracování příkazu a zpracování definice, které jsou vysvětleny dále.

Zpracování definice proměnné nebo funkce

Podle třetího *tokenu* je možné rozlišit, zda se zpracovává definice funkce nebo proměnné, případně, zda jde o definici pole. Obrázek 7.1 zobrazuje graf zpracování proměnné a funkce. Na obrázku 7.2 graf pokračuje zpracováním definice pole.



Obrázek 7.1: Syntaktická analýza definice proměnné nebo funkce

Zpracování příkazu *return*

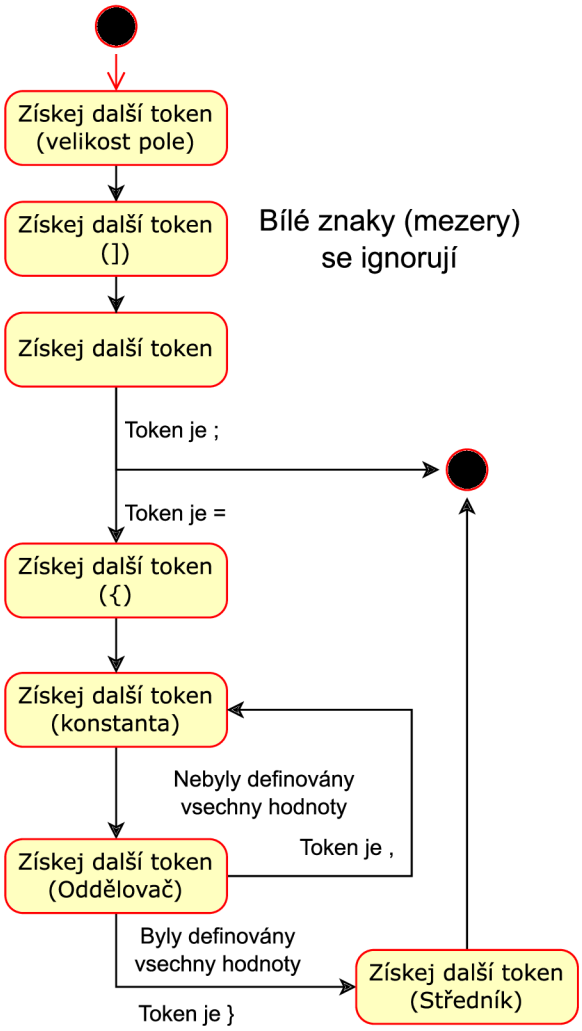
Příkaz *return* se zpracovává velmi jednoduchým způsobem, zavoláním zpracování běžného příkazu.

Zpracování příkazu *else*

Zkontroluje se, zda aktuální stav zpracovávání programu je uvnitř příkazu *If* (tedy, že již byla dokončena větev *then*). V takovém případě se nastaví aktuálně zpracovávaný příkaz na If-else.

V případě, že se nacházíme ve stavu *If* (a očekává se *else*) a příkaz *else* nepřijde, příkaz *If* se ukončí a přesune se zpracovávání na nadřazený příkaz.

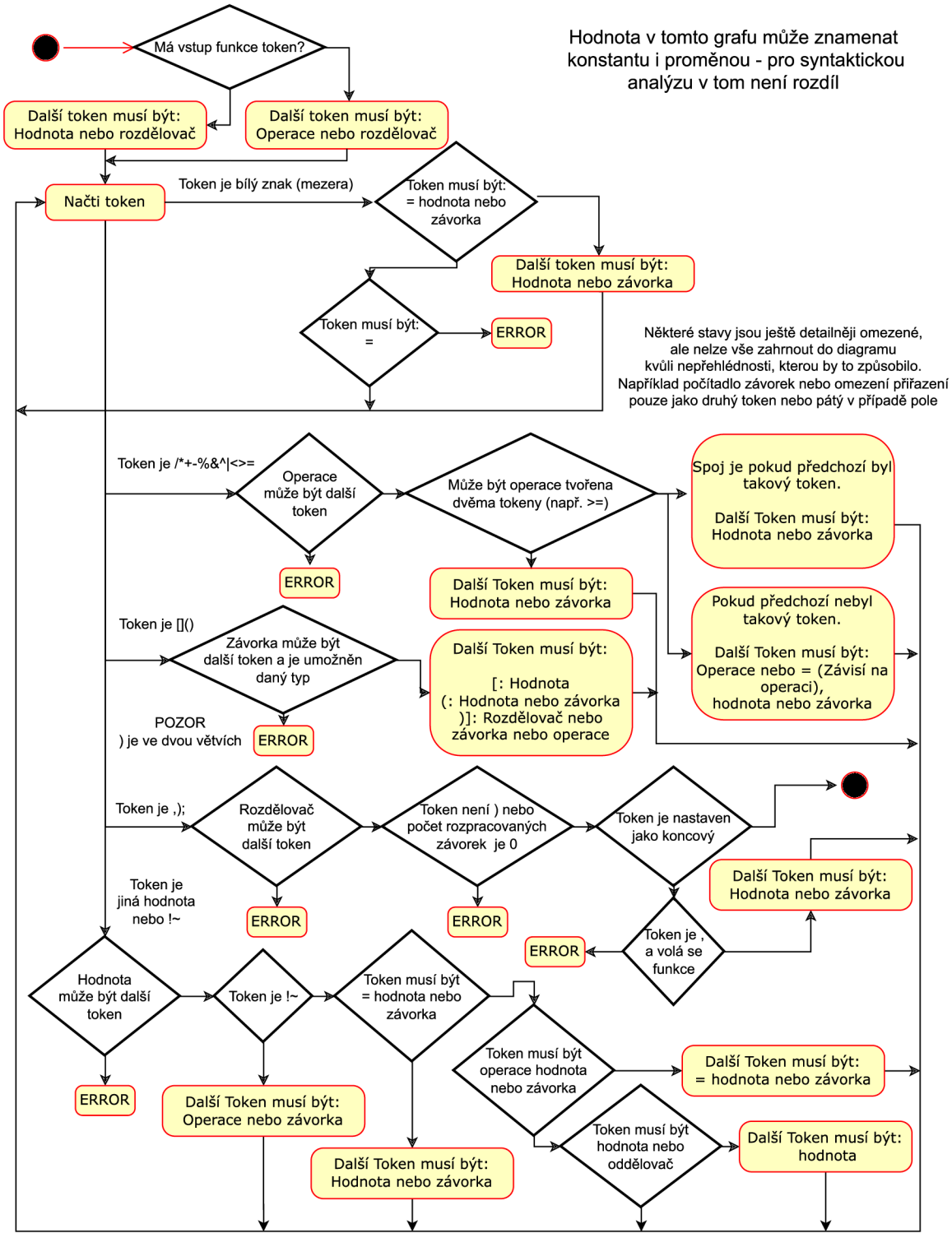
V případě, že se objeví příkaz *else* v jiném stavu než *If*, vyvolá se chyba překladu.



Obrázek 7.2: Pokračování syntaktické analýzy definice pole

Zpracování běžného příkazu

Zpracování běžného příkazu si ukládá aktuální stav zpracovávání, který určuje jaký *token* je povolený jako další. Každý *token* se vkládá do aktuálního stromu jako dítě. Z *tokenů* se tedy utvoří pole. Detailně postup ukazuje v grafu 7.3.



Obrázek 7.3: Syntaktická analýza příkazu

Na konci vytvoření příkazu se aktuální stav přesune na nadřazený prvek. Pokud se zpracovává příkaz v rámci jiného příkazu (*if*, *while*, *for*, *return*, *definice*), nevyvolá se zpracování syntaktické analýzy. Pokud je to uvnitř těla programu, vykoná se *Sémantická analýza* a zpracovávání se přesune na nadřazený řídící prvek.

Zpracování složené závorky

V případě otevírací složené závorky *{* se zkontroluje zda aktuální stav umožňuje více podpříkazů zabalených pod složené závorky a v případě že ano, nastaví se příznak zpracovávání více příkazů. Pokud již byl příznak nastaven, vyvolá se chyba překladu.

V případě uzavírací složené závorky *}* se zkontroluje příznak zpracovávání více příkazů. Pokud je nastavený, ukončí zpracovávání aktuálně zpracovávaného řídícího prvku a přesune se na nadřazený příkaz. Pokud příznak nastaven nebyl, vyvolá se chyba překladu.

7.4 Sémantický analyzátor

Sémantický analyzátor zpracovává příchozí podstromy, kontroluje datové typy, definici proměnných a funkcí a příchozí strom transformuje do hierarchie například podle priority operací.

Z toho důvodu si udržuje tří rozměrnou tabulku symbolů:

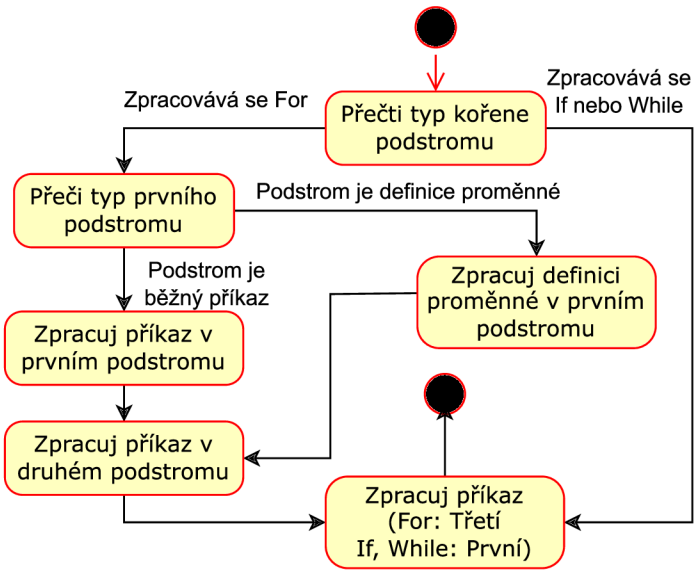
- 1. rozměr: ID nadřazeného uzlu: určuje rozsah proměnných.
- 2. rozměr: Název proměnné.
- 3. rozměr: Typ proměnné, ID uzlu proměnné.

Dále si udržuje dvourozměrnou tabulku funkcí:

- 1. rozměr: Název funkce.
- 2. rozměr: ID uzlu, typ výstupu funkce, typy vstupů funkce.

7.4.1 Sémantická analýza řídícího příkazu

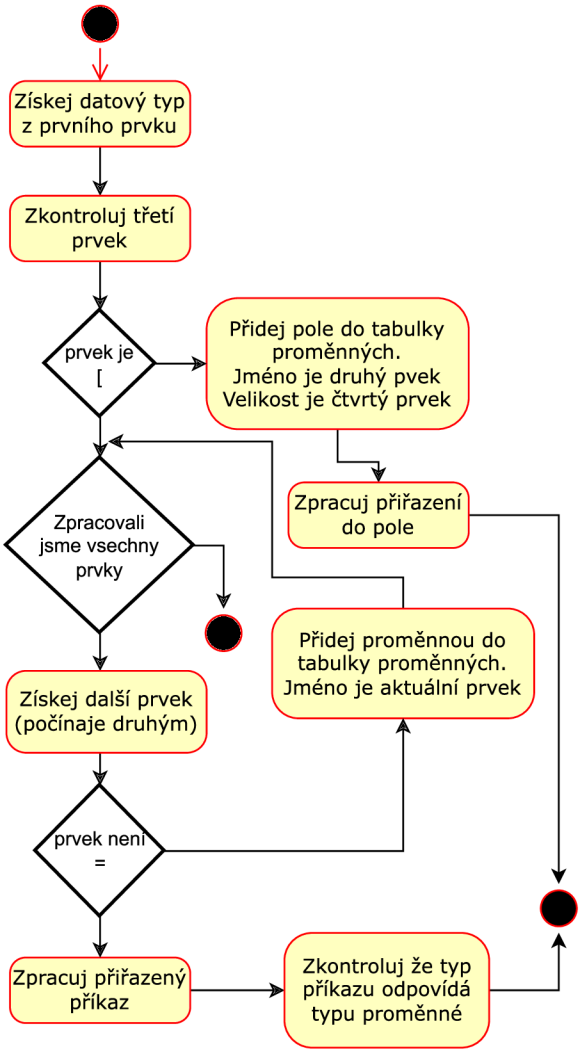
Zpracování řídícího prvku využívá zpracování definice i příkazů a stará se pouze o správné vytváření podstromů.



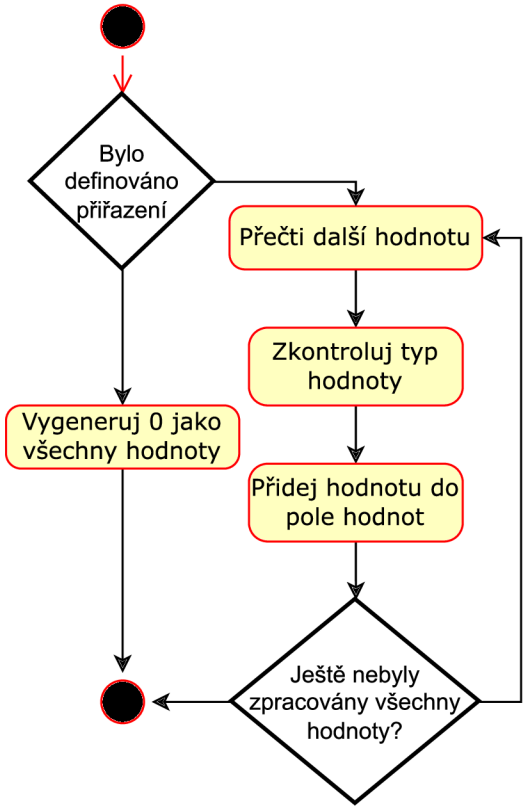
Obrázek 7.4: Sémantická analýza řídícího příkazu

7.4.2 Sémantická analýza definice proměnné

Sémantická analýza při definici proměnné zkontroluje, že proměnná již není definovaná v daném kontextu, a typ přiřazované hodnoty, poté si proměnnou uloží. Přesněji sémantická analýza postupuje podle grafu na obrázku 7.5, zpracování definice pole dále pokračuje podle grafu na obrázku 7.6



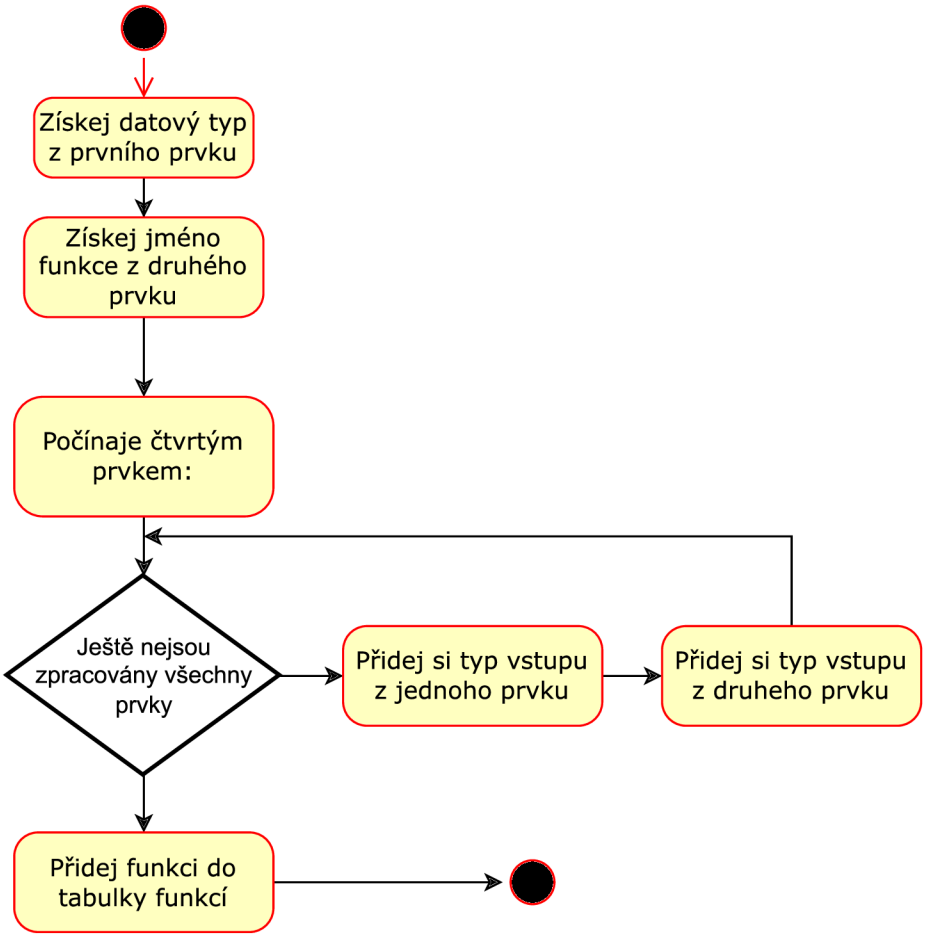
Obrázek 7.5: Sémantická analýza definice proměnné



Obrázek 7.6: Pokračování sémantické analýzy přiřazení definice pole

7.4.3 Sémantická analýza definice funkce

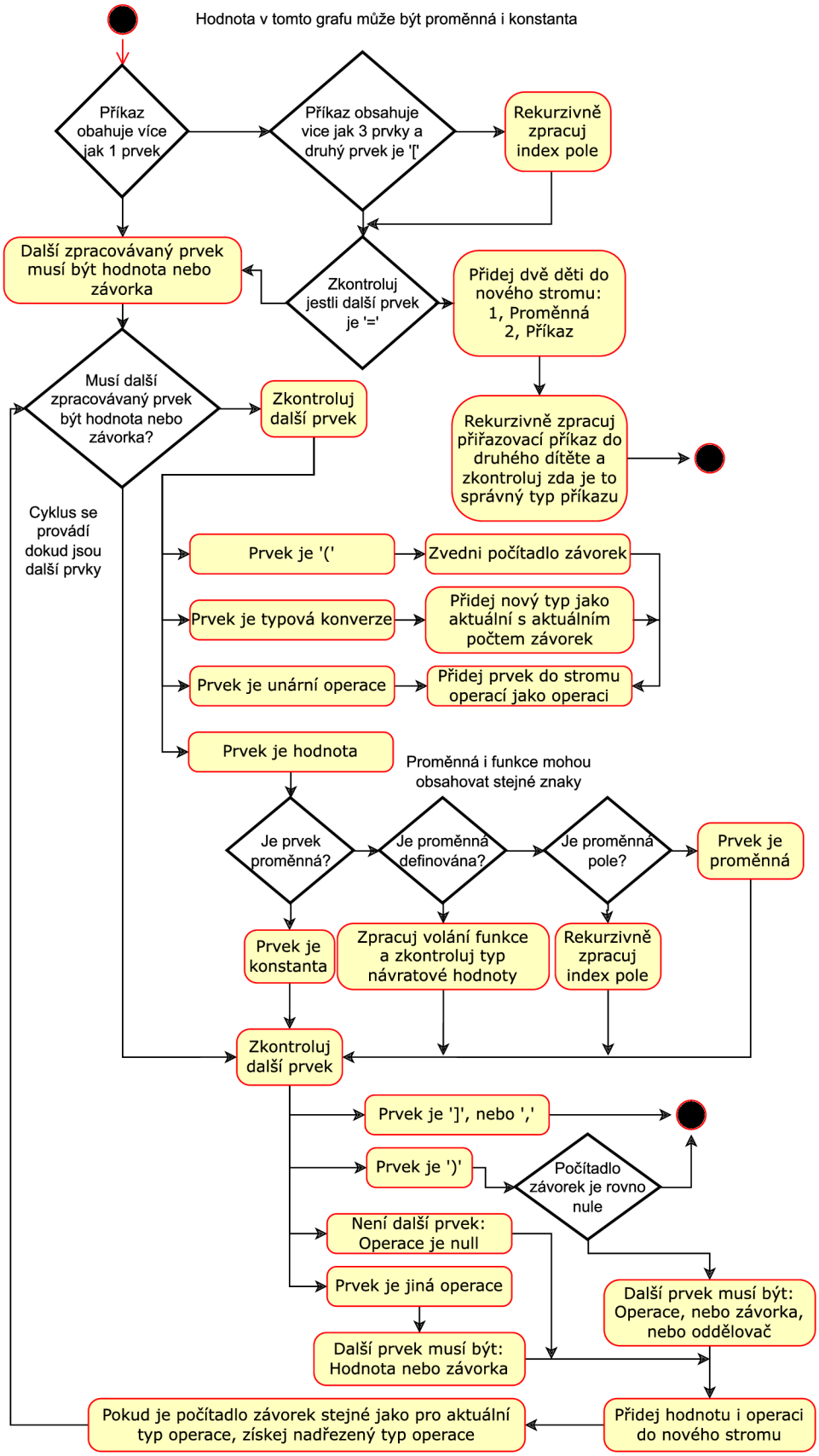
Při definici funkce sémantická analýza zpracuje všechny typy a názvy vstupních argumentů, uloží si pořadí v argumentech a návratovou hodnotu funkce. Přesněji probíhá tato analýza podle grafu na obrázku 7.7



Obrázek 7.7: Sémantická analýza definice funkce

7.4.4 Sémantická analýza příkazu

Sémantická analýza příkazu představuje stejně jako u syntaktické analýzy nejsložitější část na sémantické analýze. Analyzátor v této části kontroluje typ příkazu a transformuje vstupní pole *tokenů* na strom podle priority operací. Výsledný podstrom má v kořenu operaci, která se provede jako poslední. Blíže k listům se nachází hodnoty a operace, jež se provedou dříve. Zpracování sémantické analýzy příkazu (bez vytvoření stromu operací) je na grafu na obrázku 7.8.



Obrázek 7.8: Sémantická analýza příkazu

7.5 Generátor tří-adresného kódu

Syntaktický podstrom s doplněnými informacemi od syntaktické analýzy se předá generátoru tří-adresného kódu, který daný podstrom (příkaz) převede na posloupnost tří-adresných operací. Konkrétně se jedná o tyto operace:

- Load TargetReg (AddressReg + offset)
- Store (AddressReg + offset) SourceReg
- Arithmetic TargetReg=Source1 operace Source2
- Label Name
- Branch (Podminka) TargetLabel
- Jump TargetLabel
- LoadImm Registr Hodnota
- Zálohuj registry
- Obnov registry

Všechny tyto operace navíc obsahují:

- Typ vstupu a výstupu
- Číslo řádku z kterého byl příkaz vygenerován
- Velikost proměnné (pole)
- Pořadí argumentu funkce (0 pokud to není argument, -1 pro poslední argument)

7.5.1 Pseudokódy převedení operací do vnitřní reprezentace:

```
    VYGENEROVANA INICIALIZACE CYKLU - Pouzij docasny registr
podminka_cyklu:
    VYGENEROVANA PODMINKA CYKLU - Pouzij docasny registr
    branch (Neplati podminka) konec_cyklu
        VYGENEROVANE TELO CYKLU - Pouzij docasny registr
        VYGENEROVANY KONEC CYKLU - Pouzij docasny registr
    jump podminka_cyklu
konec_cyklu:
```

Výpis 7.1: Pseudokód cyklu For převedeného vnitřní reprezentace

```
podminka_cyklu:
    VYGENEROVANA PODMINKA CYKLU - Pouzij docasny registr
    branch (Neplati podminka) konec_cyklu
        VYGENEROVANE TELO CYKLU - Pouzij docasny registr
    jump podminka_cyklu
konec_cyklu:
```

Výpis 7.2: Pseudokód cyklu While převedeného do vnitřní reprezentace

```
        VYGENEROVANA PODMINKA IFU - Pouzij docasny registr
        branch (Neplati podminka) vetev_else
            VYGENEROVANE TELO VETVE THEN - Pouzij docasny registr
            jump konec_ifu
    vetev_else:
            VYGENEROVANE TELO VETVE ELSE - Pouzij docasny registr
    konec_ifu:
```

Výpis 7.3: Pseudokód příkazu If-then-else převedeného do vnitřní reprezentace

```
        VYGENEROVANA PODMINKU IFU - Pouzij docasny registr
        branch (Neplati podminka) konec_ifu
            VYGENEROVANE TELO VETVE THEN - Pouzij docasny registr
    konec_ifu:
```

Výpis 7.4: Pseudokód příkazu If-then převedeného do vnitřní reprezentace

```
        VYGENEROVANY PRIKAZ NAVRATOVE HODNOTY - Pouzij navratovy registr
        jump konec_funkce_<nazev>
```

Výpis 7.5: Pseudokód příkazu Return převedeného do vnitřní reprezentace

```
    funkce_<nazev>:
        Zazalohuj registry
        VYGENEROVANE TELO FUNKCE - Pouzij docasny registr
    konec_funkce_<nazev>:
        Obnov registry
        Obnov Stackpointer
        jump navratovyRegistr
```

Výpis 7.6: Pseudokód funkce převedené do vnitřní reprezentace

```
    funkce_main:
        VYGENEROVANE TELO FUNKCE - Pouzij docasny registr
    konec_funkce_main:
        jump konec_programu
```

Výpis 7.7: Pseudokód funkce main převedené do vnitřní reprezentace

```
    For (Vsechny vstupni hodnoty):
        VYGENEROVANA INICALIZACNI HODNOTA - Pouzij docasny registr
        For(Vsechny nazvy promennych):
            Store (promenna) docasnyRegistr
```

Výpis 7.8: Pseudokód definice proměnné převedené do vnitřní reprezentace

```
function generovaniPrikazu(cilovyRegistr, aktualniUzel):
    Switch (Typ uzlu):
        Prikaz nebo zavorka:
            Rekurzivne zpracuj dite
        Konstanta:
            //Nacti konstantu
```

```
    LoadImm cilovyRegistr hodnotaUzlu
Promenna:
    //Nacti promennou z jeji adresy
    Load (promenna) cilovyRegistr
Pole:
    Rekurzivne zpracuj index - Pouzij docasnyRegistr
    // Vynasob index velikosti datoveho typu (zatim pouze int a float)
    Mul docasnyRegistr * 4
    //Nacti prvek pole z jeho adresy
    Load (promenna+docasnyRegistr) cilovyRegistr
Operace:
    If (binarni operace):
        Rekurzivne zpracuj dite1 - Pouzij docasnyRegistr1
        Rekurzivne zpracuj dite2 - Pouzij docasnyRegistr2
        //Proved binarni operaci
        Operace cilovyRegistr docasnyRegistr1 docasnyRegistr2
    else:
        Rekurzivne zpracuj dite - Pouzij docasnyRegistr
        //Proved unarni operaci
        Operace cilovyRegistr docasnyRegistr
Prirazeni:
        Rekurzivne zpracuj dite2 - Pouzij docasnyRegistr1
        Ziskej promennou z dite1
        If (Promenna je pole):
            Rekurzivne zpracuj index - Pouzij docasnyRegistr2
            // Vynasob index velikosti datoveho typu
            Mul docasnyRegistr2 = docasnyRegistr2 * 4
            //Uloz hodnotu na adresu promenne
            Store (promenna+docasnyRegistr2) docasnyRegistr1
        else:
            //Uloz hodnotu na adresu promenne
            Store (promenna+docasnyRegistr2) docasnyRegistr1
Zavolani funkce:
    For (Vsechny argumenty):
        Rekurzivne zpracuj argument - Pouzij docasnyRegistr
        // Uloz argument na vrsek zasobniku
        Store (stackpointer+vrsekZasobniku) docasnyRegistr
    //Zazalohuj zasobnik
    Store (stackpointer+vrsekZasobniku) stackpointer
    //Posun zasobnik
    Add stackpointer=stackpointer+vrsekZasobniku
    //Skoc na funkci
    jump funkce
    //Presun hodnotu do spravneho registru
    move cilovyRegistr navratovyRegistr
```

Výpis 7.9: Pseudokód převedení příkazu do vnitřní reprezentace

7.6 Generátor Cílového kódu

Všechny instrukce implementované v procesoru mají definováno: Jak mají být vykonané procesorem, vstupní a výstupní typy a syntax. Tedy vše, co generátor cílového kódu potřebuje znát, aby tyto instrukce mohl použít. Generátor cílového kódu si při vytvoření načte všechny instrukce podporované procesorem a převede je na operace ve vnitřní reprezentaci. Dále zpracuje definici všech registrových polí (v tuto chvíli procesor obsahuje dvě registrová pole: Celočíselné a pro čísla s plovoucí řádovou čárkou).

7.6.1 Základní fungování

Generátor cílového kódu se zavolá na dokončený program a začne ho zpracovávat postupně po jedné instrukci, ve výjimečném případě po dvou. Každá instrukce ve vnitřní reprezentaci projde všechny převedené instrukce z cílové architektury, než najde takovou, která vykonává stejnou operaci. Nicméně některé operace z vnitřní reprezentace nejsou definované v procesoru. V takovém případě musí generátor operaci nahradit pomocí dostupných operací.

Generátor cílového kódu se umí vypořádat s těmito problémy:

- Instrukce z překladače nepotřebuje třetí hodnotu a instrukce procesoru bere jako poslední argument konstantu. V takovém případě vloží do konstanty hodnotu 0.
- Zálohování a obnovení registrů se v této části zatím neřeší, budou zpracovány později.
- Procesor neobsahuje instrukci pro porovnání hodnot (<,>,== atd.). V takovém případě se prozkoumá, zda je další instrukce podmíněný skok a vykonává stejnou operaci. Pokud se toto nepodaří namapovat, zkusí zaměnit operandy.
- Neexistuje instrukce načtení konstanty nebo přesunutí hodnoty v registru. V takovém případě se použije instrukce sečtení s konstantou.

Během mapování instrukce se zároveň převedou všechny proměnné (k nim se přistupuje pouze pomocí operací pracujících s pamětí) na adresování pomocí zásobníku. Z tohoto důvodu si musí generátor cílového kódu pamatovat adresu proměnných a velikost každého rámce, aby mohl proměnným dávat správnou adresu. Nicméně tato adresa je zatím pouze dočasná, protože stále neproběhlo zálohování a obnova registrů, která bude vyžadovat místo na zásobníku.

7.6.2 Přepis registrů a adres

Po namapování všech operací na cílový kód je potřeba namapovat všechny registry na cílovou architekturu. Překladač si rezervuje několik registrů pro své základní potřeby:

- Registr x0 je v architektuře vždy rovný 0. Překladač jej tedy využívá, když potřebuje 0.
- Registr x1 je použit pro návrat hodnoty z funkce.
- Registr x2 je použit jako zásobník.
- Registr x3 se používá jako adresa pro návrat z funkce.
- Všechny registry navíc jsou použity jako dočasné registry.

Funkce přejmenovávání registrů vyhledá registr pomocí regulárního výrazu a nahradí za název použitý v registrovém poli. V této části zároveň i probíhá zálohování a obnova registrů. Pokud zpracovávaná instrukce představuje zálohování registrů, začne překladač zaznamenávat použité registry a instrukce do té doby, než se začne zpracovávat obnovení registrů. V tu chvíli se uloží všechny registry na zásobník a obnoví se ze zásobníku. Mezi ně se vloží všechny instrukce, které se zpracovaly mezi zálohou a obnovou, a pokud přistupovaly k proměnným, aktualizuje jejich adresu podle množství uložených registrů.

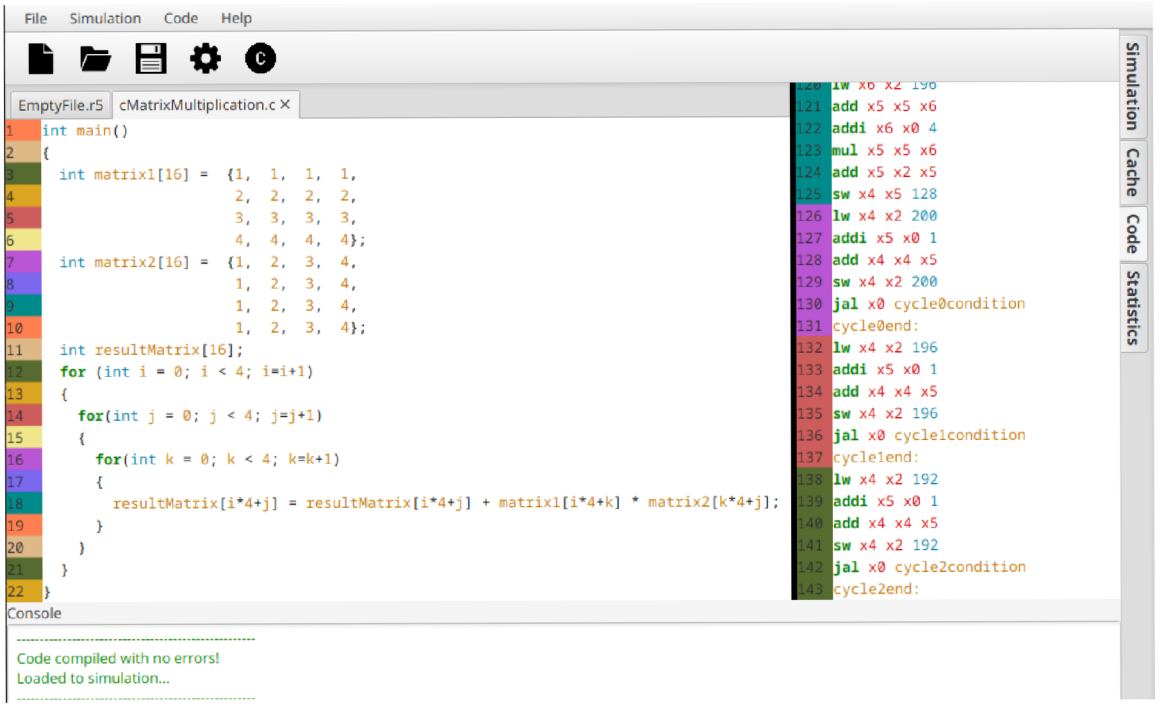
7.7 Grafické rozhraní

Implementace grafického rozhraní o překladač spočívala v rozšíření záložky *code*. V této záložce byl z původního simulátoru již naimplementován jednoduchý editor na tvorbu programu v jazyce symbolických instrukcí. Ten podporuje zvýrazňování napsaného kódu, otevírání několika souborů v záložkách, kontrolu napsaného kódu a jeho nahrání do instrukční paměti procesoru.

Přesněji jsem pro aktualizaci grafického rozhraní o překladač provedl tyto kroky:

1. Přepínání pohledu mezi jazykem C a jazykem symbolických instrukcí.
2. Další záložku editoru v C pohledu, do které se nahraje přeložený kód.
3. Zvýraznění čísla řádku barvou podle čísla v editoru. Řádky v přeloženém kódu mají stejnou barvu jako řádek ze kterého byly přeloženy.
4. Kompilace C souboru provede kompilaci pomocí C překladače a použije původní kompilaci kódu k nahrání do procesoru.
5. Zvýrazňování syntaxe C kódu.

Snímek obrazovky záložky kód po úpravách obsahuje obrázek 7.9.



Obrázek 7.9: Záložka code po implementaci překladače

Kapitola 8

Implementace mezi-paměti

Jak již bylo popsáno v kapitole 6, další rozšíření simulátoru spočívá v implementaci mezi-paměti a zobrazení paměti. Oproti překladači byla implementace mezi-paměti samotné poměrně jednoduchá. Složité bylo napojení mezi-paměti do původního kódu simulátoru. Implementace mezi-paměti probíhala následovně:

1. Implementace bloku mezi-paměti.
2. Implementace náhodné politiky výběru.
3. Implementace kontrolní logiky mezi-paměti.
4. Napojení do procesoru.
5. Grafické rozhraní.
6. Řízení simulace.
7. Implementace zpoždění.
8. Implementace zpětné simulace.
9. LRU a FIFO politiky výměny.
10. Nastavitelnost mezi-paměti.

8.1 Implementace bloku mezi-paměti

Blok mezi-paměti představuje základní prvek udržující informace o stavu mezi-paměti. *Blok* je implementovaný jako pole *Integer* hodnot, *Boolean* hodnot pro bity *Dirty* a *Valid*, *Long* hodnota pro *Tag*. Dále si udržuje začínající adresu uložených dat a několik pomocných proměnných pro grafické rozhraní a historie pro zpětnou simulaci.

Hlavní funkce implementované *blokem*, mimo funkcí nastavující nebo získávající hodnoty výše zmíněné, jsou:

- getData: Načte data z *bloku* podle zadané adresy a velikosti - zarovnaný přístup.
- setData: Nastaví data do *bloku* podle zadané adresy a velikosti - zarovnaný přístup.

Tyto funkce přistupují k jednotlivým prvkům tak, že z dané adresy určí index bloku držící data a vymaskují z ní hodnotu o dané velikosti.

8.2 Implementace kontrolní logiky mezi-paměti

Kontrolní logika mezi-paměti je zodpovědná za komunikaci s pamětí a přístupu k blokům. Při vytvoření si mezi-paměť vytvoří dvourozměrné pole bloků, kde první rozměr je určen *indexem* a v druhém jsou *bloky* se stejnou *asociativitou*. Dále si mezi-paměť udržuje informace o jejím nastavení: Asociativitu, počet bloků, velikost bloků, implementaci politiky výměny, nastavení chování při zápisu.

8.2.1 Implementace přístupu do mezi-paměti

Stejně jako u implementace bloku jsou nejdůležitější funkce pro přístupy do paměti, které již podporují nezarovnaný přístup i přes dva bloky:

```
Rozdel adresu na Tag, Index a Offset
For (Blok na indexu):
    If (Tag bloku == Tag z adresy):
        Proved pristup
        Return
//Blok s danou adresou neni v mezi-pameti
Vyber obetni blok
If (Obet obsahuje pozmenena data): //ma nastaveny dirty bit
    Uloz obetni blok do pameti
Nacti cilovoy blok z pameti: Valid 1, Dirty 0
Proved pristup
```

Výpis 8.1: Pseudokód přístupu do mezi-paměti

```
If (Zpracovava se ulozeni dat AND je nastaveno propisovani do pameti):
    Nastav bloku priznak, ze ma pozmenena data //Dirty bit

If (Pristup je zarovnany):
    Proved pristup do bloku
ElseIf (Pristup je nezarovnany, ale v jednom bloku):
    Proved pristup do bloku po Bytu
Else:
    Proved pristup do bloku po Bytu dokud se pristupuje do stejneho bloku
    Rekurzivne zpracuj zbytek pozadavku
```

Výpis 8.2: Pseudokód provedení přístupu do bloku

8.3 Napojení do procesoru

Napojení do procesoru jsem plánoval provést stejně, jako byl přístup do paměti. Nicméně během napojování se projevil jeden zásadní problém. Simulátor nezpracovával ukládání dat v jednotce pro přístup do paměti (*MAU*) a místo toho provedl přístup do paměti během dokončení instrukce v seřazovací paměti. Simulátor měl sice správně výsledky, ale tento přístup neodpovídá realitě, protože reálný procesor uvedený přístup samozřejmě musí vykonat. Tudíž jsem musel upravit kód, aby se ukládání dat zpracovávalo přes *MAU*, což s sebou přineslo několik problémů. Mimo zavedení nových chyb, které bylo potřeba opravit,

se jednalo třeba o úpravu velké části testů simulace, protože se opravou změnilo chování procesoru.

V tuto chvíli již bylo možné připojit mezi-paměť na stejném místě jako byla připojená paměť, ale bylo potřeba upravit přístup k datům. K původní paměti simulátor přistupoval po Bytu, což s mezi-pamětí nešlo. Kvůli lepší práci s kontrolní logikou a výpočtu zpoždění se přístup k mezi-paměti provádí celý najednou.

8.4 Grafické rozhraní

Pro práci s grafickým rozhraním bylo potřeba rozšířit mezi-paměť o zapamatování posledního provedeného přístupu, který obsahuje tyto informace:

- *Tag*, *index* a *offset*.
- Zda daný blok (nebo bloky) byl načtený v mezi-paměti.
- Pořadí přistoupeného bloku v mezi-paměti.
- Typ přístupu.
- Data (pokud se jedná o uložení dat).

Grafické rozhraní jsem vytvářel se stejnou myšlenkou, kterou jsem uplatnil při návrhu v kapitole 6, zobrazení vyrovnávacích pamětí pro ukládání a načítání dat, zobrazenou mezi-paměť a poslední přístup do ní. Nicméně během vytváření se ukázalo zbývající místo po straně záložky mezi-paměti. Z toho důvodu jsem přesunul zobrazení paměti, které podle plánu mělo být v tabulce pod mezi-pamětí, do seznamu na pravou stranu obrazovky.

Po každém přístupu do paměti se aktualizuje zobrazení mezi-paměti i paměti. Pro mezi-paměť se navíc ukáže, ke které buňce mezi-paměti se přistupovalo naposledy. Zda již data v mezi-paměti byla (*Hit*) nebo ne (*Miss*) lze poznat podle barvy.

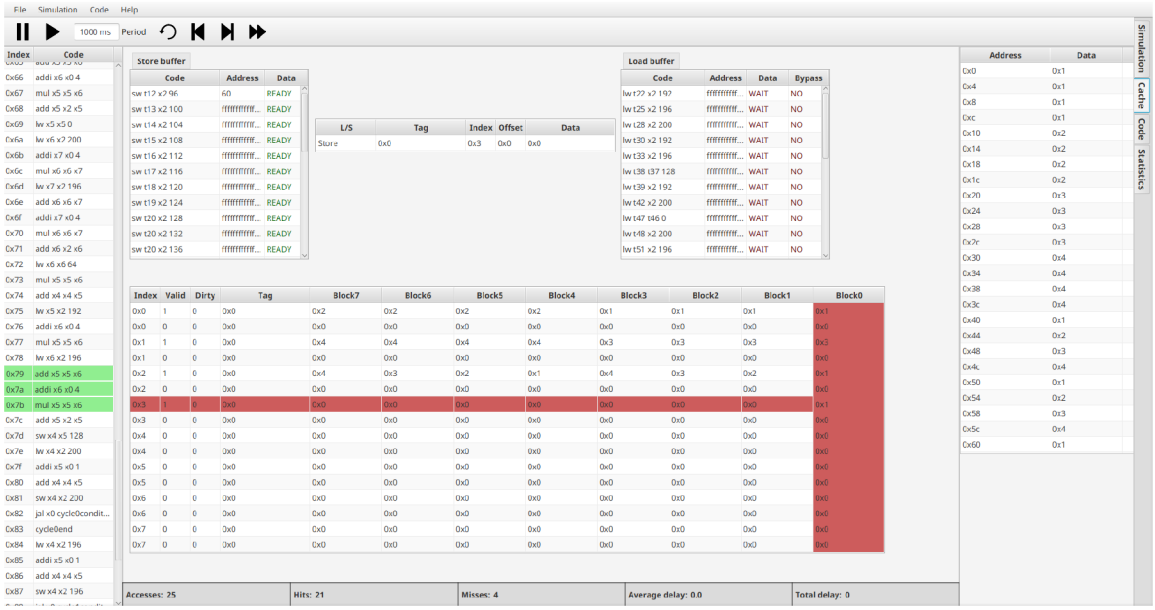
Výsledné grafické rozhraní pro mezi-paměť je na snímku obrazovky na obrázku 8.1

Do okna simulátoru navíc přibyla záložka posledních přístupu do paměti na místě registrových polí. Podle plánu měla být tato tabulka umístěna přímo v okně simulace, ale toto umístění lépe využilo prostor na obrazovce.

8.5 Úprava řízení simulace

Během zkoušení simulace v záložce mezi-pamětí se projevilo, že je potřeba upravit řízení simulace. Krokování po jednom cyklu často způsobovalo, že bylo potřeba odklikat několik kroků, než nastal další přístup do paměti, kvůli čemuž se s programem špatně pracovalo. Tudíž jsem upravil řízení simulace, aby krokovalo po přístupech do paměti. Krokování v pohledu na mezi-paměť tedy spustí simulaci a v momentě, kdy zaznamená přístup do paměti, pozastaví vykonávání programu.

Podobným způsobem bylo také potřeba upravit krokování v pravidelných intervalech. V základním pohledu simulace se krokování provádělo uvnitř simulátoru a pouze se informovalo grafické rozhraní, aby znovu vykreslilo stav procesoru. Nicméně pro pravidelné krokování po přístupech do paměti jsem se rozhodl využít již naimplementované krokování z předchozího kroku. Každých X milisekund (interval zadá uživatel) se spustí zpracování dalšího kroku.



Obrázek 8.1: Grafické zobrazení mezi-paměti

8.6 Implementace zpoždění

V reálném procesoru mezi-paměť značně urychluje vykonávání programu, nicméně pro maximalizaci výkonu je potřeba využívat mezi-paměť co nejlépe, jinak může mezi-paměť i uškodit. Stejnou funkcionalitu bylo nutné i přidat do simulátoru. Proto jsem rozšířil mezi-paměť o schopnost výpočtu zpoždění daného přístupu.

Zpoždění má statickou složku, která je nastavitelná pro načítání i ukládání dat, a dynamickou složku, což je zpoždění způsobené výměnou bloku. Do každého načítání dat se tedy přidá i čas pro výměnu bloku v případě, že blok není obsažený v mezi-paměti. Navíc je možné nastavit, zda se má zpoždění započítat do ukládání dat. V případě že tato možnost není nastavená, je potřeba do načítání dat také započítat zbývající čas pro načtení bloku z paměti po předchozím ukládání dat. Jeden přístup do paměti tedy může mít započítané zpoždění i několika výměn bloků, které s tímto blokem nesouvisely, protože bylo potřeba obsloužit je dříve.

Samotné vykonání zpoždění se řeší v jednotce pro přístup do paměti, která již měla naimplementované zpoždění. Nicméně to bylo nutné upravit. *MAU* nejdříve vykoná své zpoždění. Poté provede přístup do mezi-paměti, ze které získá zpoždění daného přístupu, a poté toto zpoždění vykoná.

8.7 Implementace politik výměny

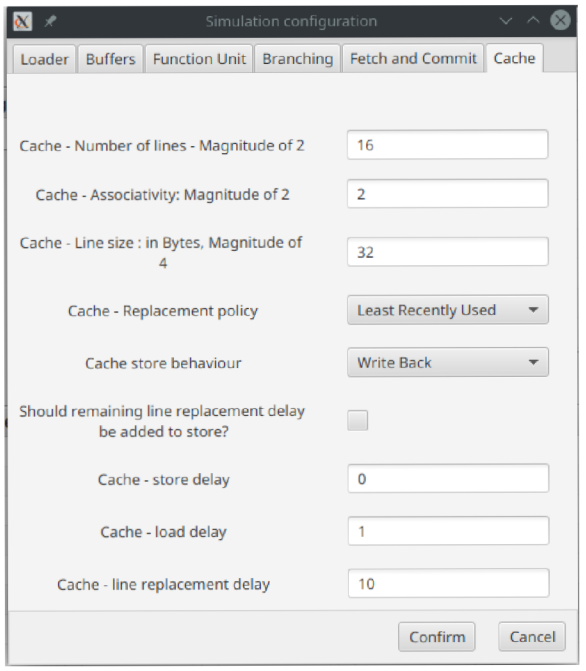
Cílem implementace politiky výměny bylo jednotné použití mezi-pamětí nezávislé na vybrané politice. Z toho důvodu pracuje se základní třídou, která definuje pouze prototypy funkcí. Politika výměny si určí, jakou informaci potřebuje. Například náhodná politika výměny nepotřebuje žádné informace.

Všechny politiky výměny proto implementují tyto funkce:

- Aktualizuj politiku: Zavolá se s každým přístupem do bloku.
- Získej blok k výměně: Zavolá se vždy, když je potřeba vyměnit blok.
- Obnov historii: Používá se pro zpětnou simulaci.

8.8 Nastavitelnost mezi-paměti

Posledním krokem bylo umožnit nastavování konfigurace mezi-paměti. Z toho důvodu přibyla v nastavení záložka pro mezi-paměť (obrázek 8.2 ukazuje základní nastavení).



Obrázek 8.2: Nastavení mezi-paměti

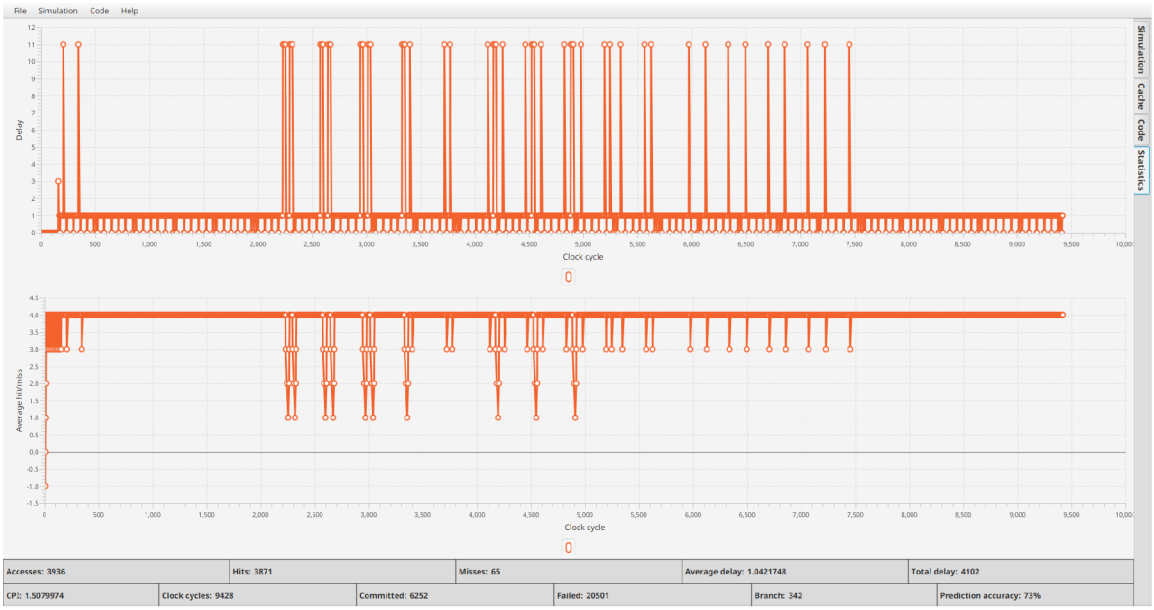
Celkově mezi-paměť umožňuje tyto nastavení:

- Počet bloků.
- Asociativita.
- Velikost bloku.
- Politika výměny.
- Chování při zápisu.
- Přidání času výměny bloku k uložení dat.
- Zpoždění ukládání dat.
- Zpoždění načtení dat.
- Doba výměny bloku.

Kapitola 9

Implementace statistik

Rozšíření statistik spočívalo v přidání sběru informací, jakým způsobem program pracuje s mezi-pamětí. Podobně jako u implementace mezi-paměti se projevilo, že na reálné obrazovce je nutné počítat s jiným rozvržením prostoru, než při tvorbě návrhů. Základní nápad, jak byl popsán v kapitole 6, zůstal stejný. Pouze rozvržení obrazovky vypadá rozdílně. Výsledný vzhled záložky statistik se nachází na obrázku 9.1 a obsahuje dva grafy s informacemi o průběhu programu. Navíc obsahuje lišty se statistikami, které se zobrazují i v záložce simulace respektive v záložce mezi-paměti.



Obrázek 9.1: Vzhled záložky statistik

Nově se sbírají tyto statistiky z mezi-paměti:

- Počet přístupů.
- Počet úspěšných přístupů do mezi-paměti.
- Počet neúspěšných přístupů do mezi-paměti.
- Průměrné zpoždění způsobené přístupem do mezi-paměti.

- Celkové zpoždění způsobené přístupem do mezi-paměti.
- Graf zpoždění způsobené přístupem vykreslený v čase.
- Graf posledních čtyř přístupů vykreslený v čase.

Nejužitečnější ze statistik podle mého názoru budou grafy, kde je možné vysledovat, jakým způsobem program pracuje s pamětí. V ideálním případě by graf posledních čtyř přístupů (spodní graf na obrázku 9.1) měl ukazovat stabilní stav přístupů k datům, které se již v mezi-paměti nacházejí, nebo občasný pokles z důvodu výměny bloku. Podobně by měl vypadat i graf zpoždění (vrchní graf na obrázku 9.1), nicméně tam navíc může být ukázáno, v jaké části programu probíhá více operací načítání/ukládání dat (za předpokladu, že mají viditelně rozdílné zpoždění).

Kapitola 10

Zhodnocení implementace

Cílem práce bylo rozšířit simulátor superskalárních procesorů, aby bylo možné jednodušeji vysvětlit principy fungování procesoru, jakým způsobem vykonává programy, které programátor napíše.

10.1 Ukázkové příklady

K tomu účelu jsem zároveň vytvořil ukázkové příklady ukazující některé základní principy. Jsou to dva programy, které vykonávají násobení matic. V prvním příkladě se násobí matice 4×4 a v druhém matice 5×5, aby bylo možné jednodušeji ukázat užitečnost zarovnávání dat. Dále se jedná o program implementující výpočet Fibonnaciho posloupnosti pomocí rekurze. Poslední ukázkový příklad implementuje bublinové řazení pole (*bubble sort*). Rychlost tohoto řazení výrazně ovlivňuje, zda se celé pole vejde do mezi-paměti, což z něj tvoří ideální ukázkový příklad. Během těchto programů zároveň vzniká řada situací v procesoru, jako jsou povedené a nepovedené predikce skoků, předbíhání ukládání dat načítáním dat, vykonání velkého množství operací při dlouhém čekání na data (závislé na nastavení zpoždění z mezi-paměti) apod.

10.2 Zhodnocení překladače

Vytváření příkladů v jazyce symbolických instrukcí je sice možné, nicméně jedná se o zdlouhavý proces náchylnější k chybám, a složitější pro studenty na pochopení, co daný algoritmus dělá. Přidání překladače by mělo zjednodušit tvorbu příkladů a pomoci studentům pochopit daný kód. Dle mého názoru se jako velmi užitečné projeví zabarvování řádků v cílovém programu jako ve zdrojovém kódu, jelikož pro mě samotného to bylo nápomocné během opravování chyb, které se během zkoušení ukázkových příkladů objevily.

Protože překladač neoptimalizuje svůj kód, vykonává řadu zbytečných přístupů do paměti, což výrazně prodlužuje délku programu. Je sice možné, že se to projeví jako užitečné (např. by to studentům mohlo pomoci pochopit, co vše překladač dělá, aby urychlil vykonávání programu), ale pro řadu věcí to může být limitující.

10.3 Zhodnocení mezi-paměti

V rámci předmětu AVS se často využívá násobení matic a zkoumání jakým způsobem je možné optimalizovat kód, aby dobře pracoval s mezi-pamětí. V tomto předmětu se sice

implementují programy, které zpracovávají mnohem větší data, než by bylo možné odsimulovat v simulátoru. Očekávám, že v rámci simulátoru se budou spouštět mnohem menší programy, než jsou reálné aplikace, protože danou problematiku je možné vysvětlit i na mnohem jednodušších příkladech.

Jako užitečné by se mohlo projevit krokování po operacích přistupujících do paměti, kde je jasněji viditelné jakým způsobem program pracuje s pamětí, ale i v tomto ohledu může chybějící optimalizace v překladači trochu kazit tuto funkcionalitu. Dále může být mírně limitující omezení pouze jedné jednotky pro přístup do paměti (*MAU*) s *mezi-pamětí*. Přesněji řečeno *mezi-paměť* funguje i s více *MAU*, nicméně zpoždění způsobené výměnou bloku může způsobit nežádoucí chování (větší zpoždění, než by se dalo čekat). Tento problém se dá obejít nastavením doby výměny bloku na 0. V takovém případě sice není zřejmá výhoda mezi-paměti, ale stále je možné ukázat výhodu více *MAU*.

10.4 Zhodnocení statistik

Myslím, že nové okno se statistikami není oproti překladači a mezi-paměti sice tak důležité, ale i tak může být užitečné k názorné ukázce jak neoptimální práce s pamětí může negativně ovlivnit chování programu. Jak jsem se již zmínil v kapitole 9, nejspíše nejužitečnější budou grafy pro vizualizaci průběhu programu. V současnosti jsou v tomto okně pouze dva grafy, ale v budoucnu by jich mohlo být více, protože implementace překladače umožní vytvářet složitější programy ke spuštění v simulátoru, než bylo doteď možné (nebo spíše, než byl kdo ochotný vytvořit).

10.5 Další možná vylepšení

Studenti využívající tento simulátor by mohli těžit z dalších rozšíření. Mezi dalšími možnými rozšířeními simulátoru by mohla být například:

- Optimalizace v překladači.
- Doplnění o složitější operace.
- Implementace vláken.
- Doplnění mezi-paměti o přesnější práci s více *MAU*.
- Ukládání konfigurace.
- Přidání adresy k instrukcím.
- Oddalování/přibližování okna simulace.
- Oddělení záložky do samostatného okna.

Kapitola 11

Závěr

V rámci této práce jsem popsal základní jednotky, které se v procesoru nacházejí a jakým způsobem funguje superskalární procesor vykonávající instrukce mimo programové pořadí. Blíže jsem se zaměřil na fungování mezi-paměti, jelikož implementace paměťového subsystému byla důležitou součástí mé práce. Dále jsem popsal k čemu slouží překladače a jak funguje implementace překladače řízeného syntaktickou analýzou.

V poslední části, než jsem se pustil do návrhu rozšíření simulátoru, jsem prozkoumal další existující simulátory procesorů a mezi-pamětí. Během tohoto procesu jsem objevil simulátor, o kterém si myslím, že by mohl být užitečný ve vysvětlování fungování procesoru, nicméně i tento simulátor měl své nevýhody. Hlavní nevýhodou byla chybějící implementace složitějších procesorů. Simulátor, který jsem v rámci této práce rozšiřoval, neměl problém se složitostí procesoru, ale na druhou stranu v něm chyběly užitečné části, které v jiných simulátorech již existovaly, nicméně některé z těchto chybějících prvků jsem dále implementoval.

Součástí mých rozšíření byla implementace překladače ze zjednodušeného jazyka C, mezi-paměť a rozšíření statistik o data z mezi-paměti. Všeobecně mnou implementovaná rozšíření vypadala jinak než podle návrhu, nicméně změny vedly k lepšímu využití prostoru a příjemnější práci se simulátorem.

Nejspíše nejdůležitější součástí implementace se stal překladač, což se projevilo při testování simulátoru s přeloženým programem z překladače. Díky přeloženým programům jsem objevil hned několik chyb, kterých by bylo bez překladače složitější dosáhnout, protože to vyžadovalo dostatečně dlouhý program. Navíc zobrazeni všech operací, které musí procesor vykonat pro zpracování programu, může studentům pomoci lépe pochopit chování jejich kódu.

Implementace mezi-paměti přináší další benefit této práce pro studenty. V rámci předmětu AVS se snaží studenti co nejefektivněji využít procesor, aby dosáhli co nejrychlejšího vykonání jejich programu, který často souvisí se zpracováním dat (např. násobením matic). Uvedené zásadně souvisí se způsobem, jakým jejich program pracuje s mezi-pamětí. Simulátor sice nebude schopný zpracovat tak velké programy, nicméně stejné principy lze vysvětlit i na mnohem menších příkladech. Domnívám se, že k vysvětlení způsobu fungování mezi-paměti se obzvlášť bude hodit krokování po operacích přistupujících do paměti.

Posledním rozšířením simulátoru implementovaným v této práci bylo sbírání statistik z mezi-paměti. Dle mého názoru nejužitečnější bude zobrazení grafů, jelikož na příliš dlouhém programu, již nebude možné sledovat program pomocí krokování. Z důvodu zpoždění přístupu do paměti a velkého množství přístupů do paměti se program může vykonávat i několik tisíců cyklů.

Během používání programu vyučující nebo studenti možná objeví další rozšíření, která by pro ně byla užitečná a která by uvítali v tomto programu (o některých jsem se již zmínil). Doufám, že někdo další převezme štafetu a rozhodne se rozšířit tento simulátor o další prvky a zpříjemní život nové řadě studentů.

Literatura

[1] *351 Cache Simulator* [online]. University of Washington [cit. 2022-12-30]. Dostupné z: https://courses.cs.washington.edu/courses/cse351/cachesim/.

[2] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*. 2019, s. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[3] Akanksha Jain, C. L. *Cache Replacement Policies*. Morgan & Claypool publishers, 2019. ISBN 978-3-031-00634-0. Dostupné z: https://par.nsf.gov/servlets/purl/10113803.

[4] ARM. *Processors* [online]. [cit. 2023-4-29]. Dostupné z: https://developer.arm.com/Processors.

[5] Baer, J.-L. *Microprocessor architecture : from simple pipelines to chip multiprocessors*. New York: Cambridge University Press, 2010. ISBN 978-0-521-76992-1.

[6] Castellanos, A. *Jupiter, RISC-V Assembler and Runtime Simulator* [online]. 2022 [cit. 2022-12-30]. Dostupné z: https://github.com/andrescv/Jupiter.

[7] Harrison Davis, S. N. *Cache simulator* [online]. [cit. 2022-12-30]. Dostupné z: https://vhosts.eecs.umich.edu/370simulators/cache/simulator.html.

[8] Kočí, K. *Graphicall CPU Simulator with Cache Visualization*. May 2018. Diploma thesis. Faculty of Electrical Engineering ČVUT in Prague.

[9] Meduna, A. *Elements of Compiler Design*. 1st Edition. Auerbach Publications, 2007. ISBN 9780429120336.

[10] Paramita, A. *Paracache* [online]. [cit. 2022-12-30]. Dostupné z: https://personal.ntu.edu.sg/smitha/ParaCache/Paracache/dmc.html.

[11] Petersen, M. B. *Ripes Introduction* [online]. 2022 [cit. 2022-12-30]. Dostupné z: https://github.com/mortbopet/Ripes/blob/master/docs/introduction.md.

[12] Peřina, D. *Vizualizace činnosti vyrovnávacích pamětí procesoru*. 2022 [cit. 2023-01-22]. Dostupné z: https://mrazekv-students.github.io/bp22_cpu_perina/.

[13] Rajaei, H. An Empirical Study for Multi-Level Cache Associativity. *2020 ASEE Virtual Annual Conference Content Access*. 2020, č. 31008. DOI: 10.18260/1-2–34115. Dostupné z: https://peer.asee.org/an-empirical-study-for-multilevel-cache-associativity.pdf.

[14] Stefan Metzlaff, N. K.-L. *DLXMIPS processor simulator* [online]. University of Augsburg, 2013 [cit. 2022-12-30]. Dostupné z: https://github.com/smetzlaff/openDLX.

[15] Strohmaier, E., Donggara, J., Simon, H. a Meuer, M. *November 2022 | TOP500* [online]. [cit. 2022-01-18]. Dostupné z: https://www.top500.org/lists/top500/2022/11/.

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

[17] Wang, C., Li, X., Zhang, J., Zhou, X. a Nie, X. MP-Tomasulo. *ACM Transactions on Architecture and Code Optimization.* 2013, sv. 10, č. 2. DOI: 10.1145/2459316.2459320. ISSN 1544-3566.

[18] Webb, K. *Cache Architecture and Design* [online]. [cit. 2022-01-18]. Dostupné z: https://www.cs.swarthmore.edu/~kwebb/cs31/f18/memhierarchy/caching.html.

[19] Yin, Z., Zhang, T., Müller, A., Liu, H., Wei, Y. et al. Efficient Parallel Sort on AVX-512-Based Multi-Core and Many-Core Architectures. In: *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS).* 2019, s. 168–176. DOI: 10.1109/HPCC/SmartCity/DSS.2019.00038.

[20] Zheng, Y., Davis, B. a Jordan, M. Performance evaluation of exclusive cache hierarchies. In: *IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software, 2004.* 2004, s. 89–96. DOI: 10.1109/ISPASS.2004.1291359.

Příloha A

Obsah přiloženého disku

DVD disk obsahuje tyto soubory a složky:

- `./Source` - Obsahuje zdrojové soubory aplikace s Readme.md souborem obsahující návod jak je sestavit.
- `./Thesis` - Složka obsahuje LaTeXzdrojové soubory použité k vygenerování tohoto dokumentu a tento dokument.
- `./Application` [1] - Aplikace samotná s Readme.md souborem obsahujícím návod ke spuštění.

[1] Současná verze aplikace je také dostupná na https://nextcloud.fit.vutbr.cz/s/JcAkwNx8etioQbX

Příloha B

User manual

This chapter explains how to install, use, and configure the RISC-V simulator.

Installing dependencies

Linux

You need to have JDK 15 or higher installed on your system. If you don't have JDK 15 installed, follow the steps below:

Ubuntu

You will need at least `openjdk-15-jre` and `openjdk-15-jdk` packages. To install then use the following commands:

```
sudo apt-get install openjdk-15-jre
sudo apt-get install openjdk-15-jdk
```

Usually, after installing these packages they should be set as default. To check the set version, use:

```
java --version
```

If not, use `update-alternatives` to change the default version of java.

```
sudo update-alternatives --config java
```

Arch Linux/Manjaro

To install required packages, use the following command:

```
sudo pacman -S jre15-openjdk-headless jre15-openjdk
sudo pacman -S jdk15-openjdk openjdk15-doc openjdk15-src
```

After that, you need to set the default JDK using the following command:

```
sudo archlinux-java set java-15-openjdk
```

To check which version is active you can use `archlinux-java status` or `java --version`.

All versions

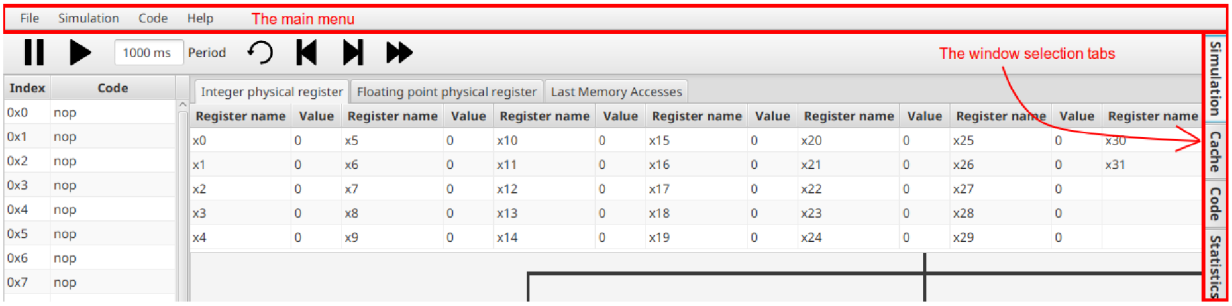
After installing the required packages, you need to run the `run.sh` script. Most of the time, you will need to set up permission to be able to run this script. Use `chmod +x run.sh` to make it runnable. After that, you'll be able to run the application.

Windows

Download and install the Java SE Development Kit 16.0.1 from the official Oracle site: `https://www.oracle.com/java/technologies/javase-jdk16-downloads.html` and after that, run the 'run.bat' file.

Application window

The simulator consists of four main windows that are linked to the control tabs on the right side of the screen, being the Simulation, Cache, Statistics and Code windows. The Simulation window can be seen at the start of the application. It is used for controlling and the visualization of the simulation. The Cache window visualises the state of cache and is used to control simulation per Load/Store access. The statistics windows shows statistics gathered throughout the simulation. The Code window serves as an input for user-made source code, written in subset of C or following the ISA loaded by the application. There is also the main menu on the top of the screen where users can configure the simulator properties, load existing examples, and find out which instructions got loaded.



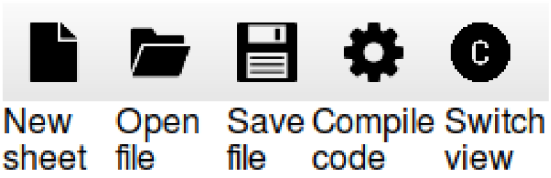
Obrázek B.1: The main application controls

Code window

The Code window is used for entering source code and loading it into the simulation logic. For entering the source code, please use the Code text area in the middle. The code window menu is located at the top of the window, providing controls for file manipulation and compiling the source code. At the bottom, there is a console output, for the result of the compilation, which on success, loads the code into the simulation. The code windows default view contains window for compiled C-code, but that window can be hidden by clicking on the *change view* button. When the C-program is compiled, the lines' color in the compiled code match the lines's color from which it was generated.

The window button functions are from the left as follows:

- New - Opens a new sheet for the source code



Obrázek B.2: The Code control buttons

- Open - Opens a file dialogue to selected the file to be loaded into the code window. If the selected sheet is empty, it will load into the selected one. Otherwise, a new sheet with file contents will be opened.
- Save as/Save - The selected sheet will be saved. If not saved before, a file dialogue opens, expecting the user to specify the path and filename. If the sheet has been saved before, changes will be saved into that file.
- Compile - Takes the selected sheet and tries to compile it and load it into the simulation. If an error arises, the output is written into the Console window. The file will be compiled as C program if the file ends with *.c* extension. Otherwise assembly loader will be used.
- Change view - Switches between C and assembly view (shows/hides compiled assembly code)

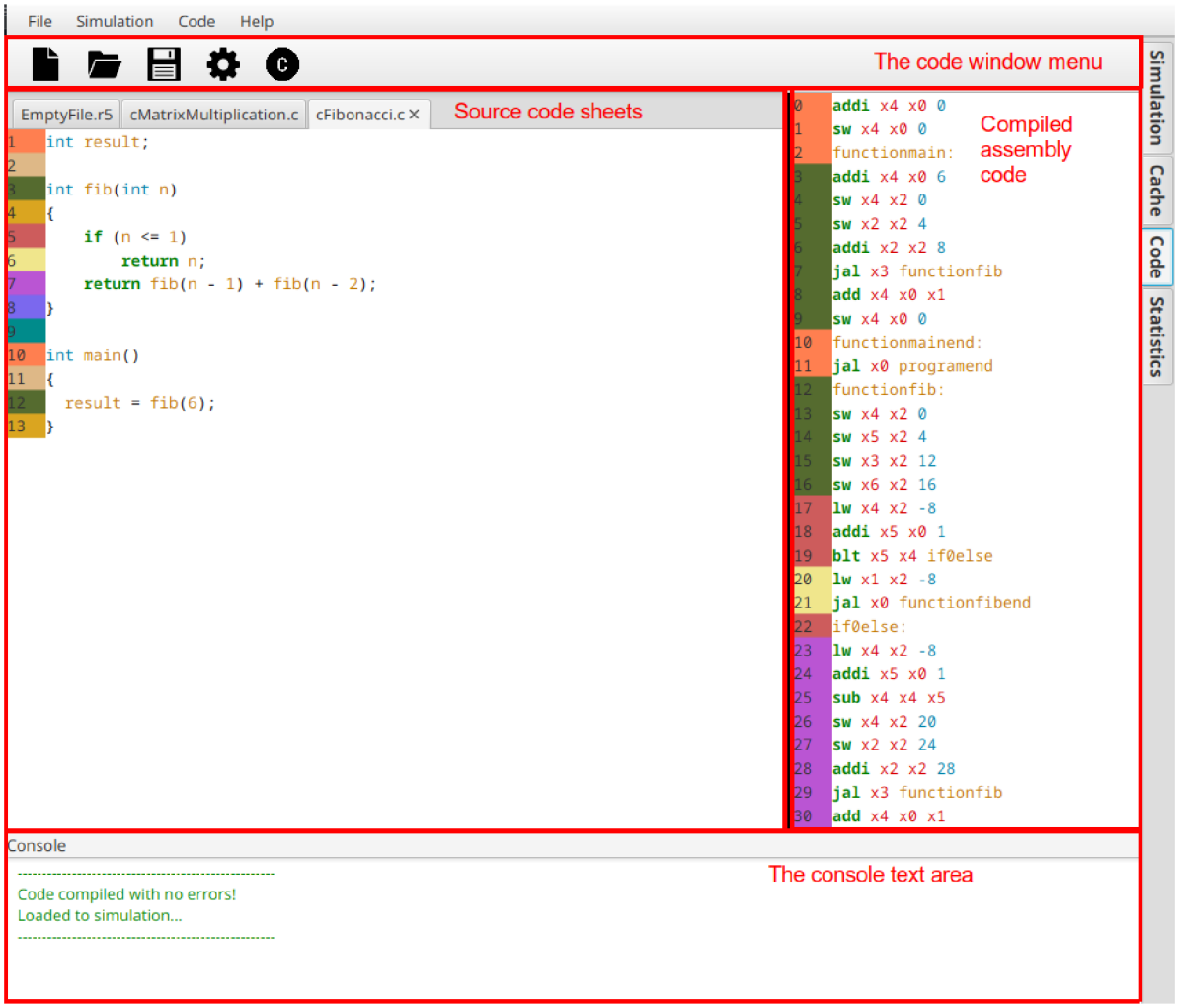
The Code window sheets support basic highlighting.
Highlighting for C-code is as follows:

- Green - Represents keywords.
- Red - Represents the registers loaded into the simulator.
- Blue - Represents data type.
- Orange - Represents variable name, function name or constant
- Black - Represents operation or bracket

Highlighting for assembly code is as follows:

- Green - Represents the instructions loaded into the simulator.
- Red - Represents the registers loaded into the simulator.
- Blue - Represents numerical values.
- Orange - Represents labels. List of acceptable labels if formed dynamically based on the source code
- Grey - Represents comments

The list of allowed instructions and their syntax can be found inside the *Help->Instruction list* in the main menu.



Obrázek B.3: The Code window

Operations supported by the compiler

The compiler supports subset of C language. Support for some operations depends on their support by processor. That is especially true for comparison and logical operations. Because by default the processor doesn't implement them. So comparison operatos can be used by default only as the last operation in the condition (e.g. $4 < 5+3$).

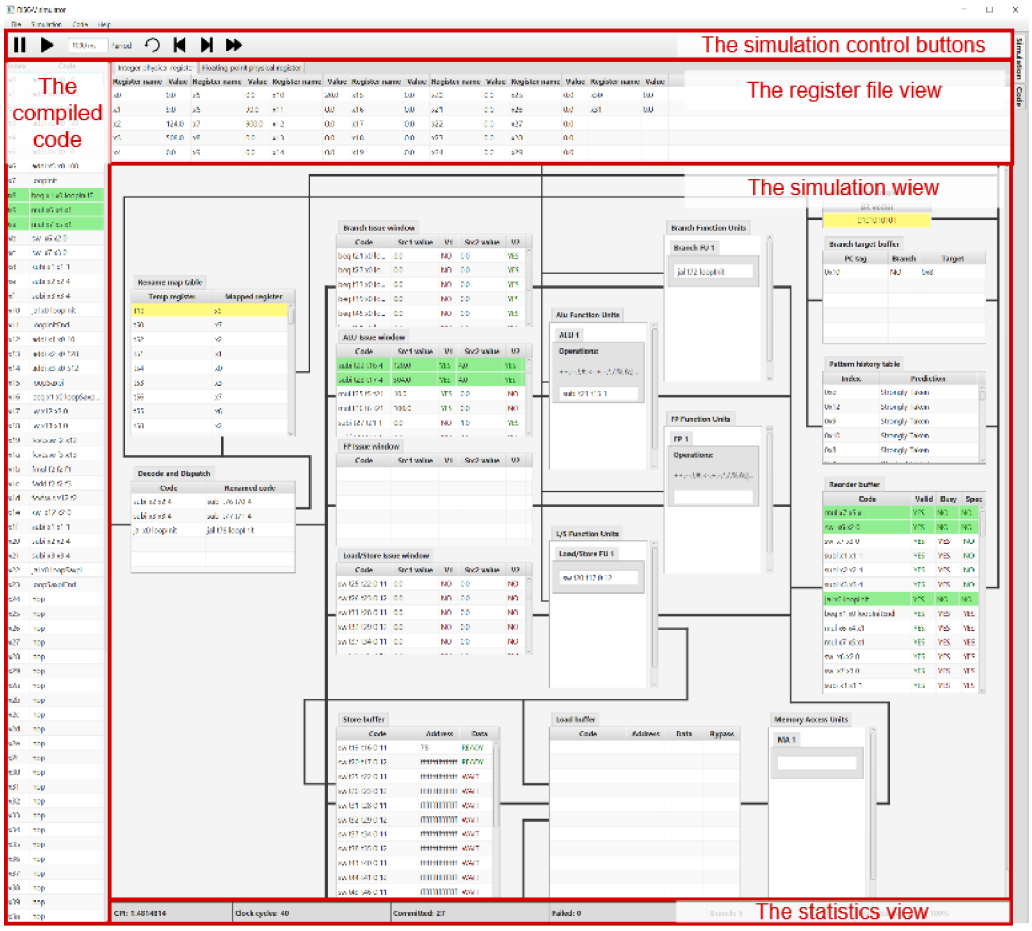
These features are implemented by the compiler:

- Data types: Integer and Float
- Control statements: For, While, If
- Functions: Must be defined before use; Allows recursion; Can use void as return type.
- Main: Must be defined, doesn't allow input arguments
- Conversion between data types
- Curly brackets allowed only in control statements and function definitions

- Global variables
- 1D arrays

Simulation window

In the simulation window, users can simulate their code after a successful compilation. If the user is inside the Code window during the compilation, a dialogue will ask him, if they want to move to this window. Inside the window, the user can find:

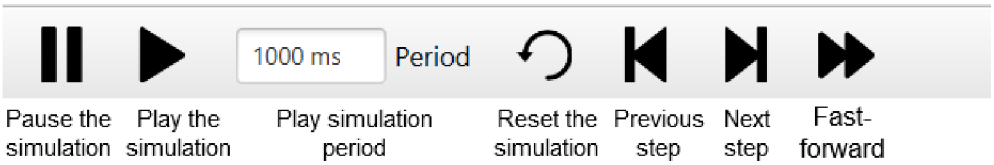


Obrázek B.4: The Simulation window

- Control menu - Placed on the very top of the window, with controls to the simulation
- Register file window - Situated on the top of the screen, displaying all loaded registers.
- Code window - Situated on the right, showing the compiled code.
- Simulation view - Showing all the blocks of the simulation
- Statistics - At the bottom, showing the gathered statistics from the run

The control menu button functions are from the left as follows:

- Pause - Stops the periodic execution.
- Play - Steps the simulation at fixed intervals. The period can be changed in the Period text field located also in the menu
- Reset - Resets the simulation to the initial state
- Previous step - Simulator will take one step back in the simulation. If the simulation is in the initial state, nothing happens.
- Next step - Simulator will take one step forward in the simulation.
- Fast-Forward - Simulator will simulate the whole input source code and shows the result. (WARNING: may take some time)



Obrázek B.5: The Simulation control buttons

As soon as the simulation reaches the end, the user is notified using the dialogue window.

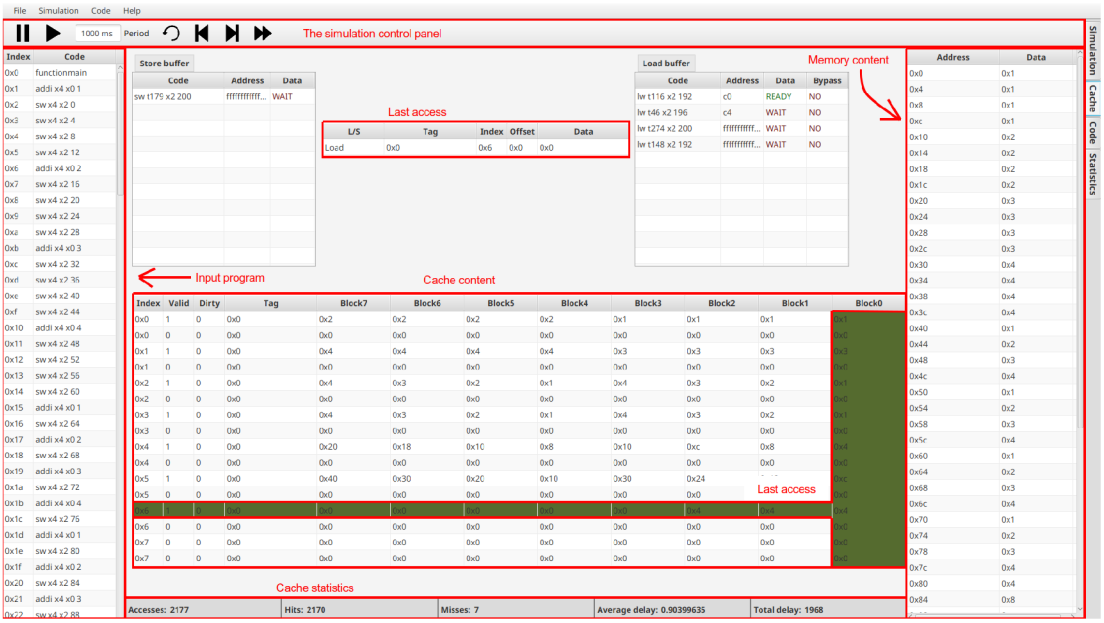
Cache window

The cache window is used for displaying cache and memory contents and accesses to cache from the core. In the middle-top of the window it displays last executed access with contents of Load/Store buffer. In the middle-bottom there is table displaying cache contents with last access highilghted (green/red for hit/miss). On the right there is a list displaying contents of memory in 4 Byte words. On the bottom of the window is a list of cache statistics gathered during simulation.

The cache changes behaviour of simulation control, specifically the step size. In cache window the steping is done per memory access. If multiple accesses occur in the same cycle, the simulation stops after the access from last *MAU*.

The buttons on simulation control panel in cache window (shown in picture B.5) are:

- Pause - Stops the periodic execution.
- Play - Steps the simulation at fixed intervals. The period can be changed in the Period text field located also in the menu. Step is a memory access.
- Reset - Resets the simulation to the initial state
- Previous step - Simulator will take one step back in the simulation. If the simulation is in the initial state, nothing happens. Step is a memory access.
- Next step - Simulator will take one step forward in the simulation. Step is a memory access.
- Fast-Forward - Simulator will simulate the whole input source code and shows the result. (WARNING: may take some time)



Obrázek B.6: Cache window described

Statistics window

The statistics windows is used to display statistics gathered during simulation. On the bottom are the same statistics as are shown in simulation and cache windows. On top there are two graphs displaying cache accesses through the execution of the program.

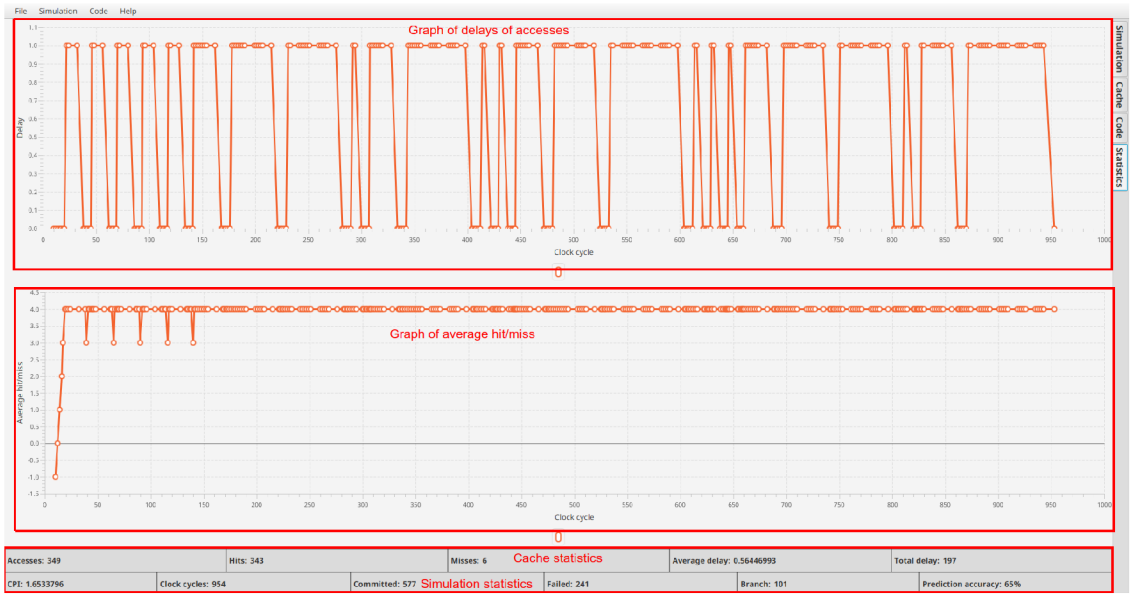
The top graph shows delay of Load/Store acceses. The bottom graph show average of last 8 accesses:

- Positive for more hits.
- Negative for more misses.

Basic configuration

The configuration inside the simulator can be accessed by locating the *Simulation->Configuration* in the main menu. A dialogue will pop out showing multiple tabs featuring different configuration options.

- Loader - Configuration of the locations of the register file and ISA folders
- Buffers - Configuration for buffer sizes
- Function Unit - Configuration of function units for different issue windows
- Branching - Configuration of the branch units
- Fetch and Commit - Configuration of the amount of instruction fetched/committed in one cycle
- Cache - Configuration of the cache sizes, access delays and other cache behaviour



Obrázek B.7: Statistics window described

The function unit configuration tab offers additional configuration options. The users can create their own function unit or edit the existing ones here. The function unit configuration tab contains additional tabs for each issue window plus the load buffer. Each sub-tab has a table, showing all the existing function units and at the bottom options for modifying the table. The options are:

- Add - A pop-up window will show up, with a form for creating a new function unit. The field delay is required.
- Edit - A pop-up window will show up, with a filled form from the selected function unit.
- Remove - Removes all selected field in the table

Each of the pop-up windows has a Confirm button and a Cancel button. The Confirm button either adds new correctly configured unit into the list or edits selected function unit with new values. The Cancel button closes the window without committing the changes. The ALU and Floating-Point function units have one extra field in the creation pop-up window, where operators can be specified. The list of allowed binary operators is as follows:

- Add (+)
- Subtract (-)
- Multiply (*)
- Divide (/)
- Modulo (+)
- AND (&)

- OR (|)
- Arithmetic shift right (»>)
- Logical shift right (»)
- Logical shift left («)
- Less than or equal (<=)
- Greater than or equal (>=)
- Equal (==)
- Less than (<)
- Greater than (>)

List of allowed unary operators is as follows:

- Increment (++)
- Decrement (–)
- NOT (!)
- Squared (#)
- Assign (<-)

The comparator operators are used only for the ALU function unit, while they return 1 if the condition is true or 0 if it is false. The assign operator is mainly used for the convert functions between integer and floating-point values.

When the user wants to confirm his changes, there is a confirm button inside the configuration window, which will reflect the changes and reset the simulation. If the user wants to exit without confirming the changes, there is a cancel button at the bottom of the window.

Advanced configurations

Apart from the configurations done inside the simulation, users can extend, modify, or change completely the register files and instructions used inside the application. The default configuration files can be found in `./riscisa` and `./registers` folder. These folders are free to be configured or copied to create different configurations. The configuration files are written in JSON with a fixed structure of the object names, which every file in a certain category must follow.

Register file

The JSON of the register file structure can be seen in listing B.1. Each register file object must have the `name`, `dataType`, and `registerList`. The `name` specifies the display name that is used in the Register file window. The `dataType` is used for comparing with the instruction argument data types during the compilation. The data type can be either integer (`kInt`), long integer (`kLong`), float (`kFloat`), or double (`kDouble`).

```
{
  "name": "Integer physical register",
  "dataType": "kInt",
  "registerList": [
    {"name": "x0", "isConstant": true,"value": 0},
    {"name": "x1", "isConstant": false,"value": 0},
    {"name": "x2", "isConstant": false,"value": 0},
  ]
}
```

Výpis B.1: Register file example

The register list must be composed of one or more register objects. The Register object can be seen in listing B.2. It has 3 objects, a `name`, an `isConstant` flag, and a `value`. The `name` is used to address the specified register in the source code, and the user can also see it in the Register file window. The flag `isConstant` is telling the simulator, whether the register value is read-only (flag set to true) or read/write (flag set to true). In the `value`, the user can specify the initial value for a certain register.

```
{
    "name": "x1",
    "isConstant": false,
    "value": 0
}
```

Výpis B.2: Single register example

Instruction

The instruction is also a JSON object. The structure can be seen in listing B.3.

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

Výpis B.3: Instruction example

The name is used to address certain instructions when writing the source code. The `instructionType` is to specify the type of the instruction. The simulator uses three types, being arithmetic (`kArithmetic`), load

The `inputDataType` is specifying the data type of input arguments. It is used to validate the data type of an argument if it is the register or used to cast the argument if it is an immediate value. The `outputDataType` is used to specify the data type of the output register argument.

The `instructionSyntax` is telling the parser the syntax of the instruction with the types of each argument. The arguments are either destination register (`rd`), source register (`rsX`, where `X` is a natural number), or an immediate value (`immX`, where `X` is a natural

number). Lastly, the interpretableAs is used to tell the interpreter how arguments should be processed.

The `interpretableAs` field has different syntax depending on the type of instruction. Examples can be found in the `./riscisa` folder. The syntax uses the abbreviations used by the `instructionSyntax` to link arguments together. The syntax can be divided into the 3 categories.

Arithmetic expressions that are similar to the Java expression ($rd = rs1 - rs2 + +$, Increments the value inside the rs2). The arithmetic expressions allows brackets ($rd = (rs1 - rs2) * rs3$), and also indexing separate bits ($rd[5:0] = rs1[31:25]$), where the range needs to be specified in the "downto"fashion. See the FX and FP instructions in *Help->Instruction list* for more examples.

Branch instruction syntax has two versions: unconditional jump (by specifying `jump` in the `interpretableAs` field), or conditional jump, where the syntax is as follows: "`(unsigned|signed):`*compareExpression*". The first part of the conditional jump syntax tells the interpreter, whether the expression should be evaluated as signed or unsigned. The second part is the condition, where compare operators and either the register, the immediate value, or the numerical constants are allowed. An example of such a condition would be $rs1 == 0$. See the branch instructions in the *Help->Instruction list* for more examples.

The load/store instruction syntax is different depending, whether it is a store instruction or a load instruction. The load instruction syntax is "`load` *dataType*`:(signed|unsigned)` *what where offset*", where `load` literal specifies the load instruction, dataType specifies how many bytes are loaded from the memory (allowed values: `byte, half, word, doubleword` for the integer values, `float, double` for floating-point values). The signed or unsigned literals tells the interpreter, whether the loaded value should be signed or unsigned. The *what*, *where* and *offset* can be either the register, the immediate value, or the numerical constant, where *what* is the value to be saved, *where* points to the certain place in the memory, and *offset* specifies the offset from the *where* value. The store instruction has similar syntax being "`store` *dataType what where offset*", where the `store` literal specifies the store instruction and other arguments has the same rules as in the load syntax.

Use cases

I want to run a simulation

1. Run the application
2. Move to the Code window using the tabs in the top right corner
3. Either write a new code or load the existing one
4. Press the Compile button (cog icon in the menu)
5. Confirm the pop-up dialogue
6. Use the buttons in the simulation window menu to control the simulation

I want to open the existing code

Expecting you are already running the application.

1. Move to the Code window using the tabs in top right corner
2. Click the open icon (the folder icon in the menu) or press CTRL+S combination
3. Select the file, which you want to load
4. Confirm the dialogue

I want to save my new code

Expecting you are already running the application.

1. Move to the Code window using the tabs in the top right corner
2. Click the save button (the floppy disk icon in the menu) or press CTRL+S combination
3. Enter the name of the file
4. Specify the path, where to store your source code
5. Confirm the dialogue

I want to save my existing code

Expecting you are already running the application.

1. Move to the Code window using the tabs in top right corner
2. Click the save button (the floppy disk icon in the menu) or press CTRL+S combination

I want to simulate at certain period

Expecting you are already running the application and code has been compiled.

1. Move to the Simulation window using the tabs in the top right corner
2. In the simulation menu, specify your period in the Period text field (supported units: s, ms)
3. Press the play button (the play icon in the simulation menu)
4. If simulation needs to be stopped, press either the play button or stop button (the pause icon)

I want to see each step of the simulation

Expecting you are already running the application and code has been compiled.

1. Move to the Simulation window using the tabs in the top right corner
2. Use the Previous and the Next step buttons to carefully observe the state in each step (the fourth and the fifth buttons in the Simulation window)

I want to see the result state of my source code

Expecting you are already running the application and the code has been compiled.

1. Move to the Simulation window using the tabs in the top right corner
2. Use the Fast-forward button to see the end state of the simulation (the sixth button in the Simulation window)

I want to edit the buffer sizes

Expecting you are already running the application.

1. Navigate to the very top menu
2. Go to *Simulation->Configuration*
3. In the configuration select the Buffers tab
4. Specify the desired sizes
5. Click the confirm button at the bottom of the configuration window

I want to edit the branch configurations

Expecting you are already running the application.

1. Navigate to the very top menu
2. Go to *Simulation->Configuration*
3. In the configuration select the Branching tab
4. Set the desired BTB size
5. Change the GHR and the PHT sizes
6. Change the predictor type in the selector
7. Change the initial state of all predictors
8. Click the confirm button at the bottom of the configuration window

I want to edit the commit or the fetch size

Expecting you are already running the application.

1. Navigate to the very top menu
2. Go to *Simulation->Configuration*
3. In the configuration select the Fetch and Commit tab
4. Change the fetch size and the number of ways text field
5. Change the commit size in the text field
6. Click the confirm button at the bottom of the configuration window

I want to add new function unit

Expecting you are already running the application.

1. Navigate to the very top menu
2. Go to *Simulation->Configuration*
3. In the configuration select the Function unit tab
4. Select the function unit list from the sub-tabs to which you want to add the new function unit
5. Press the Add button in the Function unit sub-tab
6. Set the Function unit name
7. If configuring ALU or FP units, write down allowed instruction split by the coma
8. Specify the delay of the unit
9. Create the unit by clicking Confirm button
10. To confirm your changes into the simulation, press Confirm button in the configuration dialog

I want to edit existing function unit

Expecting you are already running the application.

1. Navigate to the very top menu
2. Go to *Simulation->Configuration*
3. In the configuration select the Function unit tab
4. Select the function unit list from the sub-tabs in which you want to edit the function unit
5. Select the function unit from the table that you want to edit
6. Press the Edit button in the Function unit sub-tab
7. Edit the values in the forms
8. Press the Confirm button
9. When done editing, pres the confirm button in the configuration window

I want to delete function unit

Expecting you are already running the application.

1. Navigate to the very top menu
2. Go to *Simulation->Configuration*
3. In the configuration select the Function unit tab
4. Select the function unit list from the sub-tabs in which you want to delete the function unit
5. Select the function unit from the table that you want to delete
6. Press the Delete button in the Function unit sub-tab
7. When done deleting the function units, pres the confirm button in the configuration window

I want to change cache configuration

Expecting you are already running the application.

1. Navigate to the very top menu
2. Go to *Simulation->Configuration*
3. In the configuration select the Cache tab
4. Change configuration which you wish to change - must follow constraints written in the description
5. Press confirm button

I want to display contents of memory

Expecting you are already running the application.

1. Navigate to the right of the screen to window selection tabs
2. Go to *cache*
3. On the right of the screen is contents of memory, In the middle-bottom there is contents of cache

I want to display whole content of memory with cache data inside the memory

Expecting you are already running the application.

1. Navigate to the very top menu
2. Go to *Simulation->Configuration*

3. In the configuration select the Cache tab
4. Navigate to *Cache store behaviour*
5. Select *write through*
6. Run your program
7. Navigate to the right of the screen to window selection tabs
8. Go to *cache*
9. On the right of the screen is contents of memory, In the middle-bottom there is contents of cache