Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Grafický simulátor RISC-V procesoru

Bc. Monika Kopřivová

Diplomová práce
2023

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Monika Kopřivová**
Osobní číslo:	**I21277**
Studijní program:	**N0613A140007 Informační technologie**
Téma práce:	**Grafický simulátor RISC-V procesoru**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování

Cílem diplomové práce je naprogramovat jednoduchý simulátor jádra RISC-V procesoru Potato v jazyce C#. Simulátor bude umožňovat simulaci programu napsaném v jazyce assembler, výsledky jednotlivých operací bude graficky zobrazovat včetně zobrazení obsahu registrů procesoru a paměti programu. V teoretické části práce bude popsána stručně architektura procesoru, instrukční soubor RV32I a bude provedena rešerše podobných simulátorů. V praktické části bude naprogramován simulátor a ověřena jeho funkčnost na ukázkách.

Rozsah pracovní zprávy: **40-50 normostran**
Rozsah grafických prací:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

PINKER, Jiří a Martin POUPA. Číslicové systémy a jazyk VHDL. Praha: BEN – technická literatura, 2006. ISBN 80-7300-198-5
VIRIUS, Miroslav. Programování v C#: od základů k profesionálnímu použití. Praha: Grada Publishing, 2021. Knihovna programátora (Grada). ISBN 978-80-271-1216-6
BARTÁK, Jiří a Marcela ZACHARIÁŠOVÁ. Model procesoru RISC-V. 2015.

Vedoucí diplomové práce: **doc. Ing. Michael Bažant, Ph.D.**
Katedra softwarových technologií

Datum zadání diplomové práce: **8. listopadu 2022**
Termín odevzdání diplomové práce: **19. května 2023**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

prof. Ing. Antonín Kavička, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 30. listopadu 2022

Prohlašuji:

Tuto práci jsem vypracovala samostatně. Veškeré literární prameny a informace, které jsem v práci využila, jsou uvedeny v seznamu použité literatury.

Byla jsem seznámena s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 17. 5. 2023

Bc. Monika Kopřivová

PODĚKOVÁNÍ

Tímto bych chtěla poděkovat svému vedoucímu práce doc. Ing. Michaelu Bažantovi, Ph.D. a konzultantovi práce Ing. Miroslavu Dvořákovi, Dipl.tech. za ochotu, pomoc a vstřícný přístup při vedení této diplomové práce. Děkuji také svým rodičům, partnerovi a přátelům za trpělivost a podporu během celého studia.

ANOTACE

V diplomové práci je řešena problematika vývoje vlastního simulátoru jádra RISC-V procesoru Potato. Architektura procesoru bude společně s instrukčním souborem RV32I popsána v teoretické části práce. Výsledná aplikace bude umožňovat simulaci programu napsaném v jazyce assembler. Konkrétně se bude jednat o načtení instrukcí, jejich zkompilování a zpracování, přičemž aplikace bude výsledky jednotlivých operací graficky zobrazovat.

KLÍČOVÁ SLOVA

Grafický simulátor, RISC-V procesor, instrukční soubor RV32I, C#, Potato.

TITLE

RISC-V processor graphics simulator

ANNOTATION

The thesis deals with the development of the own simulator of the RISC-V core of the Potato processor. The architecture of the processor together with the RV32I instruction set will be described in the theoretical part of the thesis. The resulting application will allow the simulation of a program written in assembly language. Specifically, it will involve instruction fetching, compiling and processing, with the application displaying the results of each operation graphically.

KEYWORDS

Graphics simulator, RISC-V processor, RV32I instruction set, C#, Potato.

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM ZKRATEK

ALU	Arithmetic Logic Unit
CISC	Complex Instruction Set Computing
CLI	Command Line Interface
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
IDE	Integrated Development Environment
ISA	Instruction Set Architecture
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
RISC-V	Reduced Instruction Set Computing – Five
RTL	Register Transfer Level
SQL	Structured Query Language
UART	Universal Asynchronous Receiver-Transmitter
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High-Speed Integrated Circuit
WPF	Windows Presentation Foundation
XAML	Extensible Application Markup Language
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

ÚVOD

Hlavním cílem diplomové práce je vytvoření plně funkční a uživatelsky přívětivé aplikace, která bude sloužit jako simulátor jádra RISC-V procesoru Potato. Tato aplikace je nazvaná PotatoSim a měla by uživatelům posloužit jako pomůcka pro obeznámení se s funkčností jádra procesoru.

Cílovou skupinou uživatelů jsou pak hlavně začátečníci, kteří se s tím, jak funguje jádro procesoru teprve seznamují. I z tohoto důvodu byl zvolen právě RISC-V procesor Potato, protože je v celku jednoduchý, avšak pro začátečníky plně dostačující. Uživatelům bude tedy umožněno vzdělávat se pomocí této aplikace, aniž by měli jakékoliv předchozí znalosti z této oblasti.

Aplikace by měla sloužit jako simulační nástroj, ve kterém uživatel může zadávat instrukce, vyplňovat registry, a především pak spouštět samotnou simulaci, která uživatele obeznámí s tím, jak se instrukce načítají, ukládají do paměti, dekódují a následně se pak provádějí příslušné operace nad těmito instrukcemi a výsledky těchto operací jsou posléze zapsány do registrů. Celý simulátor bude graficky znázorněn v jednom hlavním okně, které bude rozděleno do několika logicky uspořádaných částí. Uživatel zde bude mít možnost vyzkoušet zapsání vlastních instrukcí nebo jejich načtení ze souboru, součástí této práce je také vytvoření několika vzorových ukázek instrukcí, které budou určeny především pro uživatele, kteří nemají s psaním instrukcí žádné předchozí zkušenosti. Dále pak uživatel bude moci spustit simulaci, a to buď jako celkovou animaci, nebo si pro lepší učení může simulaci procházet postupně po jednotlivých krocích.

Teoretická část práce se poté věnuje seznámení se s architekturou procesoru, s použitým instrukčním souborem RV32I a také s jazykem VHDL. Dále je v práci provedena rešerše obdobných aplikací a popsán rozdíl mezi těmito aplikacemi a aplikací vytvořenou v této práci. V neposlední řadě jsou zde pak popsány i technologie, které byly při vývoji aplikace využity.

V praktické části je detailně popsán návrh a také prostředí naprogramované aplikace. Také je zde popis kompilátoru RISC-V procesoru a také je tu část věnující se přepisu VHDL kódu do jazyka C#. A nakonec se v této části práce nachází i instalační a uživatelské příručky spolu s různými návody a ukázkovými úlohami.

1 ARCHITEKTURA PROCESORŮ

Tato část práce je zaměřena na vysvětlení architektury procesorů a popis základních komponent. Dále jsou zde popsány základní instrukční sady CISC a RISC a v neposlední řadě je zde popsána architektura instrukční sady RISC-V procesoru.

1.1 Architektura procesoru

Architektura procesoru udává způsob, jakým jsou navrženy jednotlivé části procesoru a jak spolu tyto části komunikují. Do architektury procesoru lze zahrnout procesorové jádro, paměťový řadič, cache, vstupně-výstupní řadič a sběrnice. [1][3]

Architektura procesoru také určuje, jak jsou zpracovávány instrukce, jakým způsobem jsou ukládána data a jak jsou tato data přenášena mezi různými částmi procesoru a paměti. Různé architektury procesorů pak mohou mít různé vlastnosti, jako jsou například rychlost zpracování, podpora multithreadingu nebo počet instrukcí podporovaných procesorem. [1][2]

V různých typech zařízeních, jako jsou ku příkladu osobní počítače, vestavěné systémy, mobilní zařízení a servery, jsou používány různé architektury. Například procesory využívané v mobilních telefonech bývají často navrženy tak, aby byly co nejúspornější, zatímco procesory pro servery musí být spolehlivé a schopné zpracovávat velké množství dat. Architektura procesoru je klíčovou součástí celého počítačového systému a má velký vliv na výkon a efektivitu systému. [1][2][3]

Základními komponentami, které utváří strukturu procesorů, jsou registry, ALU a řadič. Avšak ve vnitřní struktuře současných procesorů mohou být i další jednotky podporující nové technologie pro práci s instrukcemi a s daty, jejichž úkolem je co možná nejoptimálněji zvýšit jeho stabilitu a operační rychlost bez nutnosti použití hrubé síly. Veškeré obvody nacházející se uvnitř procesoru mají za cíl optimalizovat práci s instrukcemi a daty, či snižovat vyzářené ztrátové teplo a spotřebu elektrické energie. Základními jednotkami procesoru jsou však již výše uvedené registry, ALU a řadič. [5]

Registry mohou být ve dvou formách, buď jako pole statických paměťových prvků nebo jako RAM, která může být buď dynamického nebo statického typu. Je zřejmé, že mikroprocesor, který využívá techniky dynamické paměti, musí být nepřetržitě řízen hodinovými impulsy kvůli zachování integrity dat. Zatímco tam, kde jsou všechny registry implementovány pomocí statických buněk, může být tok hodinových pulsů zastaven, aniž by došlo ke ztrátě uložených informací. Často jsou tak registry rychlé statické paměti typu RAM s malou kapacitou uvnitř jádra procesoru. Registry pak slouží především k uchování aktuálních

instrukcí, operandů, adres v paměti, mezivýsledků a výsledků matematických a logických operací. [4][5]

Úkolem ALU je na základě řídících signálů z řadiče CPU provádět matematické a logické operace, přičemž ALU obvykle poskytuje minimálně možnosti sčítání, odčítání, OR, AND a doplňování. Pro práci s reálnými čísly s plovoucí řádovou čárkou bývají v procesoru integrovány výpočetní jednotky FPU. [4][5]

Řadič má za úkol číst operandy v podobě dat a čísel a instrukce z operační paměti, následně je dekódovat a na základě toho poté provádět mikroprogram a generovat řídící signály, tedy řídit činnost ostatních jednotek v procesoru a počítači. Řídící jednotka tedy manipuluje s registry a ALU. Řídicí jednotka využívá princip známý jako mikroprogramování, aby interpretovala každou instrukci a posléze pak přiměla procesor provést požadovanou operaci. Mikroprogramování spočívá v tom, že aby byla provedena jedna instrukce, musí se projít několika kroky. Úplnou posloupnost kroků spojených s jednou konkrétní instrukcí lze považovat za malý program. Mikroprogram pro každou instrukci je uložen v paměti a po přijetí konkrétní instrukce je vyvolán příslušný mikroprogram. [4][5]

1.2 Architektura instrukční sady

Termín, který se často používá zaměnitelně s pojmem architektura počítače, je architektura instrukční sady (ISA). ISA definuje formáty instrukcí, operační kódy instrukcí, registry, instrukční a datovou paměť, jaký mají prováděné instrukce efekt na registry a paměť a v neposlední řadě definuje i algoritmus pro řízení provádění instrukcí. [1]

V dnešní době se používají dva typy architektury instrukční sady, kterými jsou RISC a CISC. ISA je konkrétní způsob komunikace procesoru s programátorem. Nejstarší počítače měly architekturu CISC, ale v 80. letech 20. století byla vyvinuta architektura RISC, která měla překonat rostoucí složitost procesorů CISC.

1.2.1 CISC

Hlavní myšlenka procesorů s instrukční sadou CISC spočívá v tom, že všechny operace načítání, vyhodnocování a ukládání, lze provádět za pomoci jediné instrukce. Hlavním cílem přístupu CISC je tedy minimalizovat počet instrukcí na program. Tím se však zvyšuje počet cyklů na jednu instrukci. V procesoru typu CISC provádí každá instrukce takové množství akcí, že její dokončení může trvat několik taktů. Zároveň mohou mít instrukce v procesoru s instrukční sadou CISC proměnnou délku, což prodlužuje dobu zpracování. [6]

1.2.2 RISC

Hlavním cílem procesorů, které mají instrukční sadu RISC je zjednodušení hardwaru pomocí instrukční sady, která se skládá z několika základních kroků pro operace typu načítání, vyhodnocování a ukládání. Nejde tedy o to mít méně instrukcí, ale spíše o to, jakým způsobem se tyto instrukce používají. Tento přístup snižuje počet cyklů na jednu instrukci, avšak za cenu zvýšení počtu instrukcí na program. Obecně lze říci, že RISC je mnohými považován za vylepšení oproti CISC. Argumentem, proč je RISC oproti CISC lepší, je, že jeho méně komplikovaná sada instrukcí usnadňuje, zlevňuje a zrychluje návrh procesoru. V procesoru RISC je velikost paměti pro každou instrukci pevně daná, což usnadňuje dekódování a provádění těchto instrukcí. [6]

1.2.3 Porovnání RISC a CISC

Účelem obou výše uvedených architektur je zvýšit výkon procesoru, ale tohoto cíle se snaží každá z nich dosáhnout různými způsoby. Jak již bylo zmíněno výše v textu, instrukční sada RISC je vylepšením oproti CISC. Hlavní rozdíl mezi architekturou RISC a CISC spočívá v tom, že stroje založené na RISC vykonávají jednu instrukci za takt, zatímco v procesoru CISC dokončení jedné instrukce trvá několik taktů. Rozdíl je pak také ve velikosti instrukce, kdy v procesoru RISC má každá instrukce pevnou velikost paměti, na rozdíl od CISC, kde mohou mít instrukce proměnnou délku. Hlavní rozdíly jsou pro přehlednost shrnuty v níže uvedené tabulce. [6]

Tabulka 1 - Porovnání RISC a CISC

RISC	CISC
Důraz na software	Důraz na hardware
Malý počet instrukcí	Velký počet instrukcí
Instrukce pro jeden takt	Instrukce může trvat několik taktů
Jednoduché instrukce s pevnou délkou	Složité instrukce s proměnnou délkou
Intenzivní využívání paměti RAM	Efektivnější využití paměti RAM

1.3 RISC-V

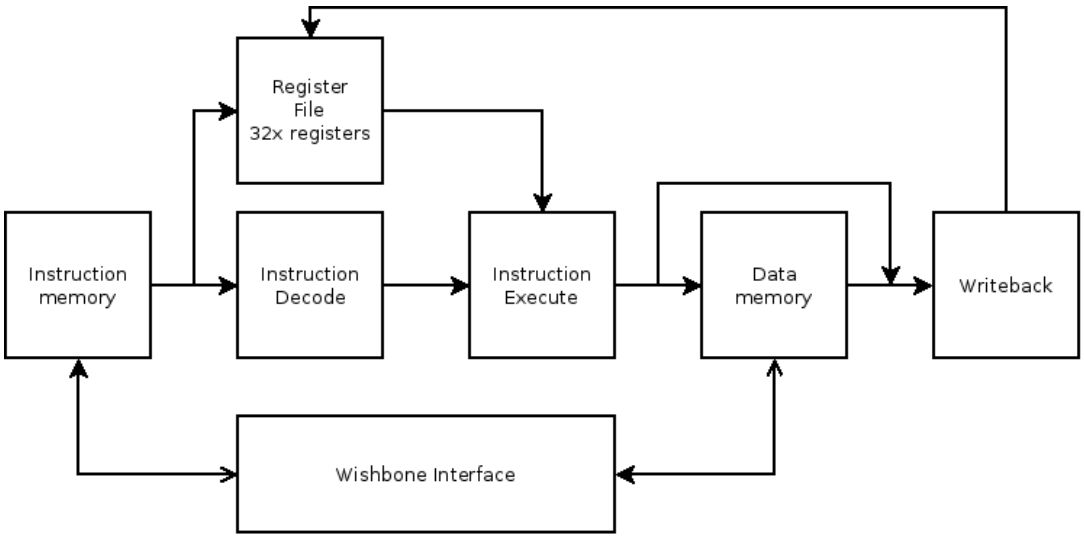
RISC-V je podobně jako RISC a CISC architektura instrukční sady (ISA). S tím rozdílem, že se jedná o volně dostupnou a bezplatnou architekturu instrukční sady, což znamená, že se na jejím vývoji může podílet kdokoli a její používání je zdarma. To je důležité, protože to umožňuje i menším výrobcům zařízení vytvářet hardware, aniž by museli platit licenční

poplatky. RICS-V byla navržena tak, aby měla malý instrukční soubor s pevně daným základem spolu s modulárními, avšak pevně stanovenými standardními rozšířeními, která dobře fungují s většinou kódu. To ponechává dostatek prostoru pro rozšíření pro specifické aplikace, aby bylo možné vytvořit vlastní procesory se specifickou pracovní zátěží. [6]

Mezi hlavní výhody RICS-V oproti konvenčním architekturám RICS a CISC patří flexibilita, inovace a snížení nákladů. Flexibilita v tomto případě znamená možnost přizpůsobení procesoru. Umožňuje tedy konfigurovat čipové sady tak, aby byly velké, malé, výkonné nebo lehké v závislosti na konkrétních požadavcích zařízení. Tato architektura je inovační proto, že společnosti mohou implementovat minimální instrukční sadu a k tomu poté využívat vlastní nebo definovaná rozšíření a vytvářet tak moderní procesory. A ke snížení nákladů a rychlejšímu uvedení na trh přispívá to, že opakované použití duševního vlastnictví spolu s otevřeným zdrojovým kódem vede ke snížení nákladů na vývoj a umožňuje společnostem rychleji uvést svůj návrh na trh. [6]

1.3.1 RISC-V Potato

Praktickou částí práce je naprogramování simulátoru jádra RISC-V procesoru Potato. Je tedy třeba zmínit se zde i o tomto procesoru. Potato je jednoduchý procesor RISC-V napsaný v jazyce VHDL pro použití v obvodech FPGA. Implementuje 32bitovou celočíselnou podmnožinu specifikace RISC-V. Mezi jeho významné funkce patří podpora kompletní 32bitové základní celočíselné ISA RISC-V (RV32I) verze 2.0, dále podpora velké části strojového režimu definovaného v RISC-V Privileged Architecture. Mezi výhody patří i podpora až 8 individuálně maskovatelných externích přerušení (IRQ), dále pak pětistupňová klasická RISC pipeline, volitelná instrukční cache a podpora sběrnice Wishbone. [7]



Obrázek 1 - Potato procesor [7]

2 INSTRUKČNÍ SOUBOR RV32I

Jedním z témat, kterými se zabývala předešlá kapitola je architektura instrukční sady RISC-V, s tímto tématem úzce souvisí i instrukční soubor RV32I, jehož popisu se věnuje tato část práce. V této části práce je popsán již zmíněný instrukční soubor a možné formáty instrukcí.

2.1 Popis instrukčního souboru RV32I

Instrukční soubor RV32I je základním instrukčním souborem architektury RISC-V pro 32bitové procesory. V základní ISA RV32I existují čtyři formáty instrukcí jádra (R/I/S/U), přičemž dva z těchto formátů se mohou ještě dále rozdělit dle významu, rozlišujeme tedy celkem šest formátů. Všechny mají pevnou délku 32 bitů a musí být v paměti zarovnány na čtyřbajtovou hranici.

- R-type: využívá se pro aritmetické a logické instrukce. Obsahuje operační kód, tři registry, z nichž dva jsou zdrojové a jeden cílový, a kód funkce.
- I-type: používá se pro instrukce, které pracují s konstantami. Obsahuje operační kód, dva registry, konstantu a kód funkce.
- S-type: používá se pro instrukce, kterou jsou určeny pro práci s pamětí, tedy ukládají hodnoty do paměti. Obsahuje operační kód, dva registry, hodnotu konstanty a kód funkce.
- B-type: využívá se pro instrukce pro řízení toku programu, jako jsou například podmíněné skoky. Obsahuje operační kód, dva registry, konstantu pro výpočet podmínky a kód funkce (3 bity)
- U-type: slouží pro instrukce pro práci s pamětí, které načítají hodnotu z paměti. Obsahuje operační kód, jeden registr a konstantu.
- J-type: používá se pro instrukce pro řízení toku programu, jako jsou nepodmíněné skoky. Obsahuje operační kód, jeden registr a konstantu pro výpočet adresy skoku. [8][9]

2.1.1 R-type instrukce

R-type instrukce jsou aritmetické a logické instrukce, které operují nad registry procesoru RISC-V. Tento formát je používán pro instrukce jako je například add, sub nebo and. Formát R-type instrukcí pak vypadá následovně:

Tabulka 2 - R-type instrukce

funct7	rs2	rs1	funct3	rd	opcode
7 bitů	5 bitů	5 bitů	3 bity	5 bitů	7 bitů

Přičemž každé pole je chápáno jako neznaménkové číslo. Neznaménkové číslo je číslo, které neobsahuje znaménko, obvykle reprezentuje kladné hodnoty. Pole opcode identifikuje operaci, která má být instrukcí provedena, pole rs1 a rs2 slouží pro čísla registru prvního a druhého operandu. Pole rd je pak určeno pro číslo registru, do kterého bude uložen výsledek operace. Blok funct3 identifikuje operaci, která bude instrukcí provedena (např. sčítání, odčítání, logické operace atd.) a blok funct7 je tu z toho důvodu, že některé R-type instrukce mají rozšířenou verzi, která používá těchto 7 bitů pro další parametry. [9]

Seznam instrukcí v R-type formátu je následující:

- ADD: sečtení hodnot v registrech rs1 a rs2 a uložení výsledku do registru rd
- SUB: odečtení hodnoty registru rs2 od hodnoty registru rs1 a uložení výsledku do registru rd
- SLT: porovnání hodnoty v registru rs1 s hodnotou v registru rs2 a uložení výsledku do registru rd
- SLTU: porovnání hodnoty v registru rs1 s hodnotou v registru rs2 a uložení výsledku do registru rd, na rozdíl od SLT však slouží pro neznaménková čísla.
- XOR: bitový xor hodnoty v registrech rs1 a rs2 a uložení výsledku do registru rd
- OR: bitový součet hodnoty v registrech rs1 a rs2 a uložení výsledku do registru rd
- AND: bitový součin hodnoty v registrech rs1 a rs2 a uložení výsledku do registru rd
- SLL: bitový posun hodnoty registru rs1 o počet bitů určený hodnotou v registru rs2 a uložení výsledku do registru rd
- SRL: logický bitový posun hodnoty registru rs1 o počet bitů určený hodnotou v registru rs2 a uložení výsledku do registru rd, jsou zde doplněny nulové bity
- SRA: aritmetický bitový posun hodnoty registru rs1 o počet bitů určený hodnotou v registru rs2 a uložení výsledku do registru rd, jsou zde doplněny znaménkové bity [8][10]

2.1.2 I-type instrukce

I-type je další z formátů instrukcí používaný v architektuře RISC-V. Instrukce v tomto formátu pracují s okamžitými hodnotami, které jsou přímo zakódovány v instrukci. Formát I-type se skládá z následujících částí:

Tabulka 3 - I-type instrukce

imm	rs1	funct3	rd	opcode
12 bitů	5 bitů	3 bity	5 bitů	7 bitů

Okamžité hodnoty v I-type instrukcích jsou zpravidla sign-extended, což znamená, že jsou automaticky doplněny znaménkovými bity, aby zůstala zachována správná hodnota. Typicky se využívají pro aritmetické operace, jako je například sčítání, odečítání, logické a bitové operace, ale mohou sloužit také pro načítání hodnot z paměti nebo uložení hodnoty do paměti.

Prvních 12 bitů je zde tvořeno okamžitou hodnotou, která je použita pro operaci. Dalších 5 bitů určuje zdrojový registr a následně je zde i 5 bitů určujících cílový registr. Posledních 7 bitů určuje kód operace a pak jsou zde 3 bity pro specifikaci funkce operace, ty slouží pro určení, jaký typ operace bude proveden. [9]

I-type formát má následující seznam instrukcí:

- ADDI: přičte okamžitou hodnotu k hodnotě v registru rs1 a výsledek uloží do registru rd.
- SLTI: pokud je hodnota v registru rs1 menší než okamžitá hodnota, uloží se do registru rd hodnota 1, jinak se tam uloží hodnota 0.
- SLTIU: pokud je hodnota v registru rs1 menší než okamžitá hodnota, uloží se do registru rd hodnota 1, jinak se tam uloží hodnota 0. Toto porovnávání je prováděno jako porovnání neznaménkových celých čísel.
- XORI: provede bitovou XOR operaci mezi hodnotou v registru rs1 a okamžitou hodnotou a výsledek uloží do registru rd.
- ORI: provede operaci bitového součtu mezi hodnotou v registru rs1 a okamžitou hodnotou a výsledek uloží do registru rd.
- ANDI: provede operaci bitového součinu mezi hodnotou v registru rs1 a okamžitou hodnotou a výsledek uloží do registru rd.
- SLLI: logicky posune hodnotu v registru rs1 o několik bitů doleva na základě okamžité hodnoty a výsledek uloží do registru rd.
- SRLI: logicky posune hodnotu v registru rs1 o několik bitů doprava podle okamžité hodnoty a výsledek uloží do registru rd. Pokud je hodnota v registru rs1 znaménková, doplní se posunované bity na nejvyšší bitové pozice podle nejvyššího bitu.

- SRAI: aritmeticky posune hodnotu v registru rs1 o několik bitů doprava dle okamžité hodnoty a výsledek uloží do registru rd. Posunované bity se doplňují podle hodnoty nejvyššího bitu v registru rs1.
- LW: načte 32bitové znaménkové celé číslo z paměti na pozici určenou hodnotou v registru rs1, přičte okamžitou hodnotu a uloží ho do registru rd.
- LB: načte byte z paměti na pozici určené hodnotou v registru rs1, přičte okamžitou hodnotu a uloží ho jako znaménkové celé číslo do registru rd.
- LH: načte 16-bitové znaménkové celé číslo z paměti na pozici určenou hodnotou v registru rs1, přičte okamžitou hodnotu a uloží ho do registru rd.
- LBU: načte byte z paměti na pozici určené hodnotou v registru rs1, přičte okamžitou hodnotu a uloží ho jako neznaménkové celé číslo do registru rd.
- LHU: načte 16-bitové neznaménkové celé číslo z paměti na pozici určenou hodnotou v registru rs1, přičte okamžitou hodnotu a uloží ho do registru rd. [8][10]

2.1.3 S-type instrukce

Dalším z formátů instrukcí používaných v architektuře RISC-V je S-type. Instrukce v tomto formátu pracují s okamžitými hodnotami, které jsou přímo zakódovány v instrukci. Formát I-type se skládá z následujících částí:

Tabulka 4 - S-type instrukce

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bitů	5 bitů	5 bitů	3 bity	5 bitů	7 bitů

S-type formát je využíván pro instrukce, které provádějí operace se dvěma registry a offsetem, který je uložen v dolních 12 bitech instrukce. Tento formát se skládá z bloku opcode obsahujícího kód operace, bloků rs1 a rs2, které určují cílový a zdrojový registr. Poté z části imm, která určuje offset, který je uložen v dolních 12 bitech instrukce a z bloku pro funkční kód. [9]

Příklady instrukcí S-type:

- SB: ukládá hodnotu z registru rs2 na adresu určenou součtem hodnoty v registru rs1 a imm.
- SH: ukládá hodnotu z registru rs2 na adresu určenou součtem hodnoty v registru rs1 a imm, přičemž hodnota je zarovnána na sudou adresu.
- SW: ukládá hodnotu z registru rs2 na adresu určenou součtem hodnoty v registru rs1, přičemž hodnota je zarovnána na násobek 4 bajtů. [8][10]

2.1.4 B-type instrukce

Dalším formátem používaným v architektuře RISC-V je B-type. B-type formát je použit pro instrukce větvení a slouží k podmíněnému přesunu programu na jinou část kódu za splnění určité podmínky. Podmínka se testuje na základě hodnoty v registru a offsetu, který je uložen v instrukci. Podobně jako u S-type formátu je offset uložen v dolních 12 bitech instrukce a rozšířen na 32 bitů. [9][8]

Výpočet adresy, na kterou má být program přesunut, je určován na základě registru Program Counter. Když nedojde ke skoku, je PC zvětšen o 4 bajty, je tedy uskutečněn posun o jednu instrukci vpřed. V případě že dojde ke skoku, je výpočet následující adresy PC změněn tím způsobem, že je přičtena hodnota imm vynásobená 4. Přičemž blok imm určuje počet instrukcí, o které se máme posunout, buď vpřed pokud je číslo kladné, nebo dozadu, pokud je číslo záporné. [9][8]

Vzhledem k tomu, že RISC-V podporuje rozšíření pro 16bitové instrukce, musí být posun vykonán přes půlslova, je tomu tak i v případě, že nejsou přítomny žádné 16bitové instrukce nebo instrukce s volitelnou délkou. [9][8]

Seznam instrukcí (podmínek větvení) v B-type je následující:

- BEQ: podmínka je splněna, pokud se hodnoty v registrech rs1 a rs2 rovnají.
- BNE: podmínka je splněna, pokud se hodnoty v registrech rs1 a rs2 liší.
- BLT: podmínka je splněna, pokud je hodnota v registru rs1 menší než hodnota v registru rs2.
- BGE podmínka je splněna, pokud je hodnota v registru rs1 větší nebo rovna hodnotě v registru rs2.
- BLTU: podmínka je splněna, pokud je hodnota v registru rs1 menší než hodnota v registru rs2, přičemž není použito znaménkové rozšíření.
- BGEU: podmínka je splněna, pokud je hodnota v registru rs1 větší nebo rovna hodnotě v registru rs2, přičemž není použito znaménkové rozšíření. [8][10]

2.1.5 U-type instrukce

Dalším z formátů používaných v architektuře RISC-V je U-type. U-type instrukce jsou určeny pro přímé adresování větších částí paměti. Mohou být využity k nastavení horních 20 bitů registru na konstantní hodnotu, což může být užitečné například pro inicializaci ukazatele na paměťovou oblast. [9]

Formát instrukcí je následující:

Tabulka 5 - U-type

imm	rd	opcode
20 bitů	5 bitů	7 bitů

Blok opcode určuje operační kód a má délku 7 bitů, blok imm je okamžitá hodnota, která má délku 20 bitů a obsahuje adresu v paměti a blok rd označuje cílový registr. [9]

Seznam instrukcí v U-type je následující:

- AUIPC: nastaví horních 20 bitů registru rd na hodnotu aktuální adresy, ke které přičte okamžitou hodnotu imm.
- LUI: nastaví horních 20 bitů registru rd na okamžitou hodnotu imm a dolních 12 bitů na nulu. [8][10]

2.1.6 J-type instrukce

Poslední z formátů používaných v architektuře RISC-V je J-type, formát instrukce je v tomto případě podobný jako u instrukcí U-type. Tyto instrukce jsou používány k nepodmíněnému skoku na adresu uloženou v instrukci a uložení návratové adresy do registru.[9]

Instrukce J-type je pouze jedna:

- JAL: provede nepodmíněný skok na adresu, která je vypočtena jako součet aktuální adresy a okamžité hodnoty imm a poté uloží návratovou adresu do registru rd, aby bylo možné se vrátit na správné místo. [8][10]

3 VHDL

Tato kapitola se věnuje popisu jazyka VHDL. Je zde představen jazyk VHDL, zmíněny výhody a nevýhody jeho používání a popsány základní elementy, objekty a datové typy tohoto jazyka. A také je zde v krátkosti zmíněno porovnání s jazykem Verilog.

3.1 Popis jazyka VHDL

Jazyk VHDL je jazyk navržený speciálně pro účely popisu a simulace rozsáhlých číslicových obvodů a systémů. Jde tedy o jazyk určený pro popis hardwaru, přičemž jazyky pro popis hardwaru jsou označovány zkratkou HDL. Jazyk VHDL byl vyvinut ministerstvem obrany, které chtělo dokumentovat návrh obvodů, v roce 1980. VHDL se používá k vyjádření smíšených signálových a digitálních systémů, jako jsou integrované obvody a programovatelná hradlová pole (FPGA). Jazyk VHDL také využíváme k zápisu textových modelů, které popisují nebo vyjadřují logické obvody. [18][19]

Při popisu číslicového systému pomocí jazyka VHDL je třeba myslet na to, že popisujeme číslicový systém, který bude později realizován v hardware. To znamená, že vytvořený kód bude muset projít syntézou, jejímž výsledkem bude zapojení z hradel a klopných obvodů určené pro programovatelný logický obvod nebo případně pro jiný typ zákaznického obvodu. Konstrukce, které jsou v jazyce VHDL při popisu navrhovaného číslicového systému vytvářeny, musí tedy být většinou syntetizovatelné. Výjimkou jsou testovací programy nebo modely určené pouze pro simulaci na simulátoru. [18][19]

Jednou z výhod tohoto jazyka jsou jeho obsáhlé vyjadřovací schopnosti a značná nezávislost číslicového systému, který je popsán jazykem VHDL, na cílové technologii jeho realizace nebo výroby. Zásadní výhodou tohoto jazyka z hlediska využití návrhu systému je, že umožňuje ověřit a modelovat chování základního systému dříve, než nástroje pro syntézu převedou návrh na skutečná hradla a vodiče. Mezi další výhody tohoto jazyka patří podpora opakované použitelnosti a sdílení kódu, projekty v jazyce VHDL jsou přenositelné, což znamená, že lze vygenerovat projekt pro jednu bázi prvků a poté jej přenést na jinou bázi prvků. Jednou z dalších výhod je také to, že projekty v tomto jazyce jsou víceúčelové, což znamená, že je projekt vytvořen jednou a blok výpočtů může být využit v různých dalších projektech. [19][20]

Kromě výše uvedených výhod má jazyk VHDL i určité nevýhody. Za nevýhodu by se například dalo považovat, že je tento jazyk trochu obtížnější na naučení, protože vyžaduje specifické znalosti struktury a syntaxe jazyka. [20]

3.2 Základní elementy jazyka VHDL

Kód jakéhokoli obvodu napsaný jazykem VHDL je rozdělen na dvě samostatné části. Na jedné straně je entita, která specifikuje vstupní a výstupní porty obvodu. Na druhé straně je architektura popisující chování tohoto obvodu, přičemž architektura musí být spojena s entitou. Pro přiřazení architektury k entitě se používá element konfigurace. Ke stejné entitě je možné přiřadit i několik architektur, takže programátor si může vybrat jednu z dostupných.

3.2.1 Entita

Jak již bylo zmíněno, entita slouží k určení vstupních a výstupních portů obvodu. Entita má obvykle jeden nebo více portů, které mohou být vstupy, výstupy, vstupy-výstupy nebo vyrovnávací paměť. Vstupní porty lze pouze číst, nelze je tedy uvnitř architektury měnit. Naproti tomu výstupní porty lze zase pouze zapisovat, ale nelze je číst. V případě, že je třeba výstupní port číst, například pokud je zapotřebí rozhodnout o jeho hodnotě, nebo vstupní port zapsat, musí být port inicializován jako vstupně-výstupní nebo jako vyrovnávací paměť. [21]

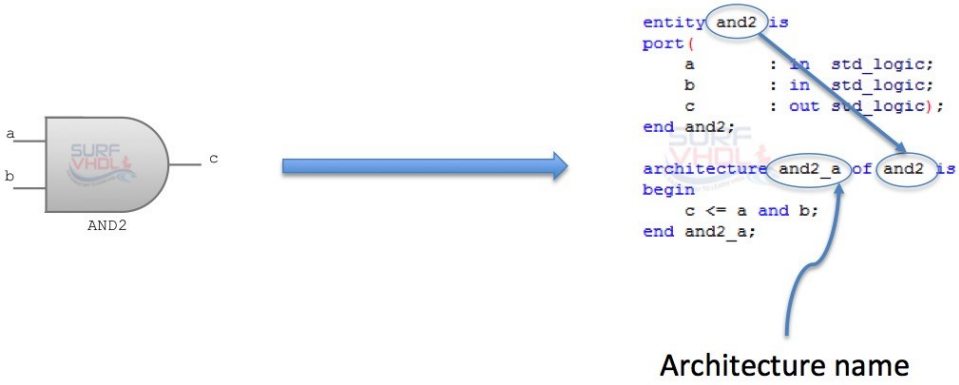
Entita také může obsahovat sadu obecných hodnot, které se používají k deklaraci vlastností a konstant obvodu. Entita také může být generická, v takovém případě musí být deklarována před porty. Generické hodnoty mohou mít vícenásobné využití. Mohou být využity k definici zpoždění signálů a hodinových cyklů nebo je lze použít také jako konstanty, které se budou používat uvnitř architektury. Tyto konstanty pak pomáhají dělat kód srozumitelnějším, přenositelnějším a udržovatelnějším. Generické parametry však nejsou nutné, proto obvod, který je nepotřebuje, jednoduše v deklaraci entity žádný generický příkaz nemá. [21][22]

Obrázek 2 - VHDL entita [22]

3.2.2 Architektura a konfigurace

Architektura je vlastní popis návrhu sloužící k popisu fungování obvodu. Může obsahovat jak souběžné, tak sekvenční příkazy. Architektura označuje způsob fungování obvodu tím, že obsahuje sadu vnitřních signálů, funkcí a procedur, její popis může být buď strukturální nebo behaviorální. Architektura zahrnuje popis bloků systému, vazby mezi nimi a časování signálů.

Architektura se vždy vztahuje k entitě a popisuje její chování. Každá entita musí mít alespoň jednu architekturu, ale je možné k jedné entitě přiřadit i několik architektur. O přiřazení architektury k entitě se stará konfigurace. Ta zároveň i definuje, jak je propojena hierarchie návrhu. [21][22]



Obrázek 3 - Ukázka propojení architektury a entity [22]

3.3 Typy modelovacích stylů v jazyce VHDL

Architekturu lze zapsat jedním ze tří základních stylů modelování. Tyto tři modelovací styly dělíme na datový tok, behaviorální styl a strukturální styl. Rozdíl mezi těmito styly je založen na typu použitých souběžných příkazů. Styl datového toku používá pouze souběžné příkazy pro přiřazení signálů. Behaviorální styl používá pouze procesní příkazy a strukturální styl používá pouze příkazy pro vytváření instancí komponent. [20][22]

3.3.1 Strukturální styl

Strukturální modelování slouží k určení funkčnosti a struktury obvodu, obsahuje deklarace signálů, instancí komponent a map portů v instanci komponenty. Na této úrovni je entita implementována v podobě logických hradel v závislosti na zvolené technologii. Návrh na této úrovni je podobný popisu návrhu z hlediska logického diagramu na úrovni hradel, je silně závislý na technologii. Kód není přenositelný mezi různými technologiemi a je velmi podobný jazyku assembleru. [22]

Jedná se v podstatě o modulární návrh, kdy dojde při návrhu složitého projektu k jeho rozdělení na dva nebo více jednodušších návrhů. Výhody modulárního návrhu ve VHDL jsou dost podobné výhodám, které poskytuje modulární návrh nebo objektově orientovaný návrh u vyšších počítačových jazyků. Modulární návrhy umožňují zabalit nízkoúrovňové funkce do modulů a tento přístup také umožňuje opakované použití návrhu. [22]

3.3.2 Behaviorální styl

Behaviorální styl se používá k postupnému provádění příkazů a ukazuje, jak se systém chová podle aktuálního příkazu. Jedná se o nejvyšší úroveň abstrakce, kterou VHDL poskytuje. Entitu lze implementovat z hlediska požadovaného algoritmu návrhu bez ohledu na detaily hardwarové implementace. Návrh na této úrovni je velice podobný programovacímu jazyku. Tento styl modelování popisuje, jak by se měl obvod chovat, o skutečné implementaci obvodu pak rozhoduje nástroj pro syntézu VHDL. Behaviorální model je často používán při návrhu testovacích designů, protože testovací návrh se nezajímá o hardwarovou realizaci. [22]

3.3.3 Styl datového toku

Tento styl modelování lze popsat jako logicky výraz. Ukazuje, jak data proudí od vstupu k výstupu a funguje na principu souběžného provádění. Na této úrovni je entita navržena tím způsobem, že je specifikován tok dat. Návrhář si je vědom toho, jak data proudí mezi hardwarovými registry a jak jsou zpracovávána v návrhu. Tento styl kódování se nazývá RTL neboli logika přenosu registrů. Modelování VHDL-RTL je zcela nezávislé na technologii popisu. Efektivní návrh RTL lze přenést na jinou technologii bez úpravy nebo pouze s menší úpravou kódu. [22]

3.4 VHDL objekty

Jazyk VHDL používá tři typy objektů. Těmito typy jsou konstanty, proměnné a signály. Prvním typem objektu jsou konstanty. To jsou objekty, které mají počáteční hodnotu, která jim byla přiřazena před simulací. Deklarace konstanty jí tedy vždy musí přiřadit hodnotu. Tato hodnota nesmí být během syntézy nebo provozu obvodu nikdy změněna. Konstanty mohou být deklarovány před začátkem architektury anebo před začátkem procesu. [20][21]

Druhým typem objektu jsou proměnné. Jsou to objekty, které nabývají jediné hodnoty, jež se může během simulace nebo procesu měnit za pomoci příkazu přiřazení. Deklarace proměnné jí může, ale nemusí přiřadit hodnotu. Deklarace může proběhnout buď před začátkem architektury anebo před začátkem procesu. Proměnné se obvykle používají jako indexy, především ve smyčkách, nebo k přebírání hodnot, které umožňují modelovat jiné komponenty. Je důležité podotknout, že proměnné nepředstavují fyzická spojení nebo paměťové prvky.[20][21]

Posledním typem objektu jsou signály. Jedná se o objekty, které představují paměťové prvky nebo spojení mezi dílčími obvody. Na rozdíl od konstant a proměnných lze signály syntetizovat. Což znamená, že signál ve zdrojovém kódu VHDL lze fyzicky převést na paměťový prvek ve

finálním obvodu. Musí být deklarovány před začátkem architektury. Příkladem signálů mohou být třeba porty entity, které jsou při deklaraci implicitně deklarovány jako signály, protože představují fyzická spojení v obvodu. [20][21]

3.5 Datové typy a operátory v jazyce VHDL

Jazyk VHDL umožňuje používat předdefinované typy, nebo uživatelsky definované typy. Přičemž předdefinované typy lze dělit na skalární a složené. Mezi skalární typy patří například celá čísla, čísla s pohyblivou desetinou čárkou, výčtové typy nebo fyzické datové typy. Fyzické datové typy obsahují hodnoty reprezentující měření fyzikální veličiny jako je například čas, délka nebo proud. Fyzický typ poté slouží vyjádření množství vztaženého k základní jednotce. Hodnota fyzického typu je tedy celočíselný násobek základní jednotky fyzického typu. [18][20][21]

Příkladem složených typů jsou pak pole nebo záznamy, přičemž záznamy se používají pro určení jednoho nebo více prvků, kdy každý z prvků má jiné jméno a jiný typ. Na rozdíl od pole, které je homogenní, může záznam obsahovat prvky různých typů. Při přístupu k záznamu lze přistupovat buď k jednotlivým prvkům záznamu nebo k záznamu jako k celku. [18]

Jazyk VHDL také definuje několik typů operátorů. Mezi základní typy operátorů patří logické operátory, relační operátory, aritmetické operátory a operátory posunu. Logické operátory slouží k řízení toku programu. Lze je použít k vytvoření kombinační logiky, pokud se kombinují se signály nebo proměnnými. Mezi logické operátory patří and, or, nand, nor, xor, xnor a not. Relační operátory se pak používají k porovnávání dvou operandů stejného datového typu a získaný výsledek je vždy typu boolean, patří sem =, /=, <=, >, >. Aritmetické operátory slouží k provádění aritmetických operací a příkladem může být třeba +,-,*,/. Operátory posunu jsou sil, srl, sla, sra, rol a ror a používají se k bitové manipulaci s daty posunutím a otočením bitů prvního operandu doprava nebo doleva. [18][20]

3.6 Porovnání s jazykem Verilog

Verilog je také jazyk HDL pro popis elektronických obvodů a systémů. Používá se při simulaci i syntéze hardwaru. K popisu elektronických systémů a obvodů používá textový formát. Základními stavebními prvky jazyka Verilog jsou moduly, které poskytují informace o vstupních a výstupních portech a skrývají vnitřní implementaci. [19]

Pokud by měly být rozebrány některé z rozdílů mezi Verilogem a VHDL, tak co se týče využití, platí, že VHDL je populárnější v Evropě a Verilog zase naopak v USA. VHDL je syntaxí celkem podobný jazyku Pascal, zatímco Verilog připomíná C. VHDL má přísnější

pravidla syntaxe a na rozdíl od Verilogu jsou jeho zápisy podstatně složitější. Verilog je novějším z těchto dvou jazyků, byl představen v roce 1984. Oba dva tyto jazyky jsou v současnosti velice populární a záleží především na osobních preferencích každého jedince, který z těchto jazyků si zvolí. [19][23]

VHDL:

```
process ({S0,S1},A,B,C,D)
begin
   case {S0,S1}, is
      when "00" => Y <= A;
      when "01" => Y <= B;
      when "10" => Y <= C;
      when "11" => Y <= D;
      when others => Y <= A;
   end case;
end process;
```

Verilog:

```

always @({S0,S1}, A, B, C, D)
      case ({S0,S1})
         2'b00: Y = A;
         2'b01: Y = B;
         2'b10: Y = C;
         2'b11: Y = D;
      endcase

```

Obrázek 4 - Porovnání Verilogu a VHDL [23]

4 REŠERŠE PODOBNÝCH APLIKACÍ

Další část práce se věnuje popisu podobných aplikací. Pro tento účel byly vybrány tři na internetu dostupné aplikace: emulsiV, RISC-V Interpreter a BRISC-V Simulator. První z aplikací byla zvolena z toho důvodu, že se jedná o poměrně dobře zpracovaný simulátor, který částečně posloužil jako inspirace při tvorbě vlastní aplikace.

Další dvě aplikace byly zvoleny, protože se jedná o vcelku jednoduché simulátory, které funkčností připomínají aplikaci vytvářenou v rámci této práce. Přičemž jsou však mezi těmito dvěma aplikacemi patrné rozdíly, z toho důvodu byly do této rešerše zahrnuty obě. Všechny tři aplikace jsou pak v této kapitole popsány a zhodnoceny.

4.1 Aplikace emulsiV

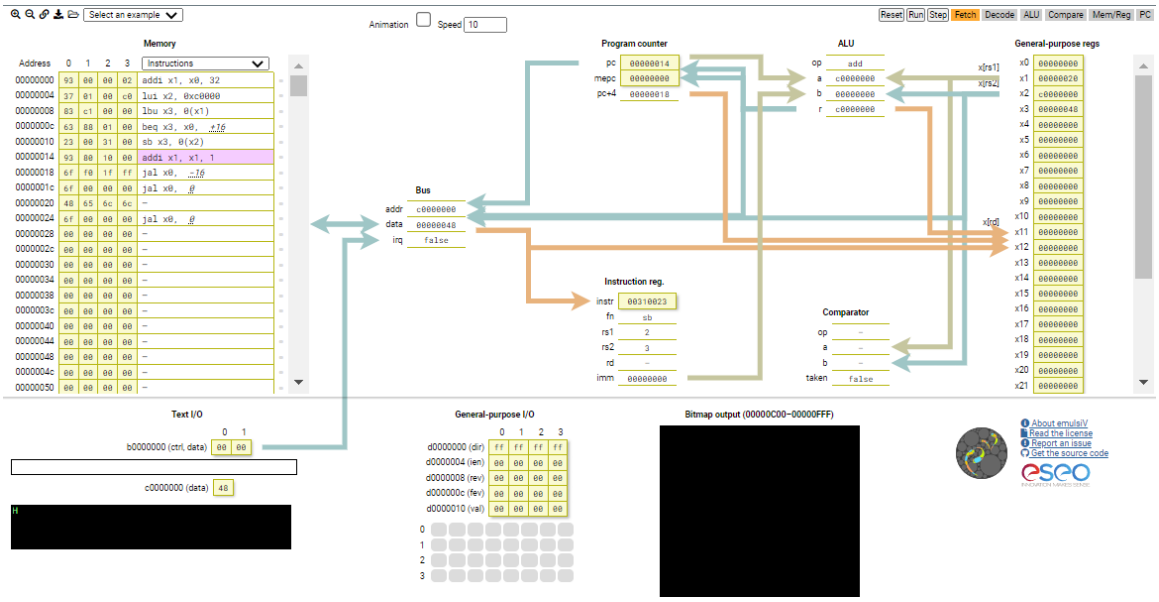
První aplikací, která byla zvolena pro rešerši, je emulsiV. Jedná se o simulátor volně dostupný ve formě webové stránky. Tento simulátor byl vytvořen na škole École supérieure d'électronique de l'Ouest (ESEO) ve Francii. Při návrhu a vytváření vlastní aplikace byl právě tento simulátor použit jako vzor.

4.1.1 Popis

Jedná se o skutečně dobře zpracovaný simulátor, který nabízí uživateli spoustu možností. Simulátor je velice detailně propracovaný a na první pohled i graficky přívětivý. Co se týče samotné simulace, jsou zde už při otevření dostupné vzorové instrukce, tudíž je tento simulátor vhodný i pro uživatele, kteří si pouze chtějí vyzkoušet, jak simulace funguje, ale nejsou obeznámeni se zápisem instrukcí.

Pro ovládání se tu nachází horní menu obsahující několik tlačítek, kterými je možné spustit animaci, určit její rychlost nebo ji případně krokovat. V tomto horním menu se nachází i možnost výběru vzorového příkladu a také tlačítka pro nahrání či uložení instrukcí. Simulátor je poté rozdělen na vrchní a spodní část, přičemž pro účely simulace je důležitá především ta vrchní. Ta je dále pomyslně rozdělena do levé části, kde jsou vidět instrukce, prostřední části, která slouží k zobrazování mezivýpočtů a pravé části, kde se nachází pole s registry. Tyto tři části jsou pak opticky propojeny šipkami.

Jak již bylo zmíněno, tlačítko pro spuštění se nachází v horním menu aplikace. Po spuštění jsou postupně načítány instrukce a prováděny výpočty. V každém kroku simulace se barevně zvýrazňují pole, s kterými se v danou chvíli zrovna pracuje. Lze tedy vidět, která instrukce je v danou chvíli zpracovávána.



Obrázek 5 - emulsiV

4.1.2 Hodnocení

Jedná se o velmi dobře propracovaný simulátor. Design této aplikace je také velice přívětivý a dobře zpracovaný. Aplikace nabízí vzorové příklady, díky čemuž je vhodná i pro začínající uživatele. Pro takové uživatele by však mohla být aplikace trochu komplikovaná, protože je zde zobrazováno opravdu velké množství informací, přičemž některé z těchto informací nejsou až tak důležité a jejich zobrazování může spíše zhoršovat celkovou přehlednost aplikace.

I samotná simulace je velice dobře propracována. Zvýrazňování částí, se kterými se zrovna pracuje pomáhá uživateli zorientovat se, jak simulace probíhá. Výhodou aplikace je také možnost simulaci buď spustit celou nebo ji postupně krokovat. Přičemž pokud je zvolena možnost animace, je zde povoleno nastavit si i její rychlost. Animace by byla možná trochu přehlednější, kdyby výchozí rychlost byla nastavena trochu pomalejší, ale vzhledem k tomu, že si rychlost může uživatel přizpůsobit, tak se nejedná o nijak závažný problém. Při zvolení možnosti krokování, jsou jednotlivé kroky nastaveny vždy po celých instrukcích, což je možná až moc velký krok. Bylo by zřejmě přehlednější, kdyby během jednotlivých kroků byly zobrazovány jednotlivé výpočty prováděné s danou instrukcí.

4.2 RISC-V Interpreter

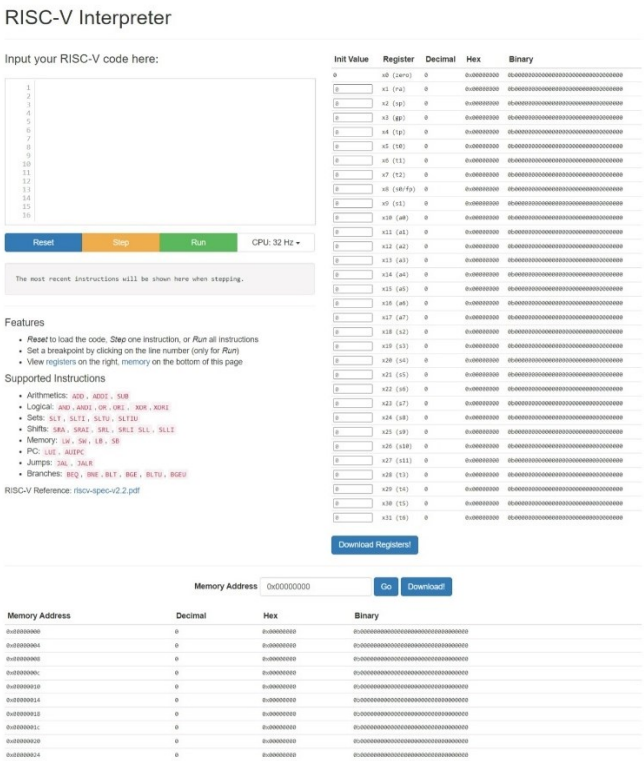
Druhou aplikací, na kterou se tato rešerše zaměřuje je RISC-V Interpreter. I v tomto případě se jedná se o simulátor volně dostupný ve formě webové stránky. Také tento simulátor byl vytvořen na škole, konkrétně na škole Cornell Ann S. Bowers College of Computing and Information Science v USA.

4.2.1 Popis

Tento simulátor je již na první pohled jednodušší než ten předchozí. I tento simulátor je uživatelsky přívětivý a graficky dobře zpracovaný. Jeho jednoduchost může být pro některé uživatele výhodou, nejsou zde sice zobrazovány jednotlivé operace, které jsou s instrukcí prováděny, ale hlavní informace zde viditelné jsou. Zda je zobrazované množství informací dostačující už záleží na samotném uživateli a způsobu, jakým chce aplikaci využívat.

Celkově je aplikace rozdělena do tří částí. V levé části je pole pro zadávání instrukcí, tlačítka pro spuštění a krokování simulace a část s popisem simulátoru. Tato část popisuje funkčnost jednotlivých tlačítek a podporované instrukce, přičemž se zde nachází i odkaz na manuál k RISC-V instrukčnímu setu. V pravé části aplikace se pak nachází pole obsahující registry, kde je pro každý registr vidět jeho hodnota v decimálním, binárním i hexadecimálním čísle. Ve spodní části je pak blok obsahující paměť, i zde je zobrazení obsahu paměti v decimálním, hexadecimálním i binárním čísle.

Pokud jde o spuštění simulace, jsou zde již zmíněná tlačítka. V tomto případě tlačítko pro krokování opět prochází krokově po jednotlivých instrukcích. Tlačítko pro spuštění je tu však odlišné od předchozího simulátoru. Zde tlačítko provede všechny instrukce najednou, je však možné si v poli pro zadávání instrukcí vložit breakpoint. Poslední tlačítko pak slouží pro celkový reset aplikace.



Obrázek 6 - RISC-V interpreter

4.2.2 Hodnocení

Jedná se o vcelku jednoduchý, avšak plně funkční a graficky přívětivý simulátor. Výhodou této aplikace je popis nacházející se v levé části. Tento popis pomáhá uživateli pochopit, jak fungují jednotlivá tlačítka a v případě, že je uživatel úplným začátečníkem a nezná jednotlivé instrukce, je zde uveden odkaz na manuál, který může být pro takového uživatele také velmi užitečný. Trochu zde chybí možnost nahrání nebo uložení instrukcí do souboru, ale naopak jsou zde možná trochu nadbytečná tlačítka pro stažení registrů a paměti. Také by zde mohl být k dispozici vzorový příklad instrukcí. Možná trochu zbytečné je v tomto případě zobrazení jak decimálních, hexadecimálních tak i binárních hodnot čísla u registrů a paměti, zde by možná stačila pouze jedna z variant. Celkově je ale aplikace velice přehledná.

U samotné simulace je možnost krokování v tomto případě opět po jednotlivých instrukcích, vzhledem k tomu že tento simulátor ale nezobrazuje jednotlivé operace s instrukcí, je takto nastavená velikost kroku úplně ideální. Tlačítko pro spuštění pak spustí všechny instrukce najednou, zde by možná bylo vhodnější, kdyby se při stisku tohoto tlačítka také postupovalo po jednotlivých instrukcích.

4.3 BRISC-V Simulator

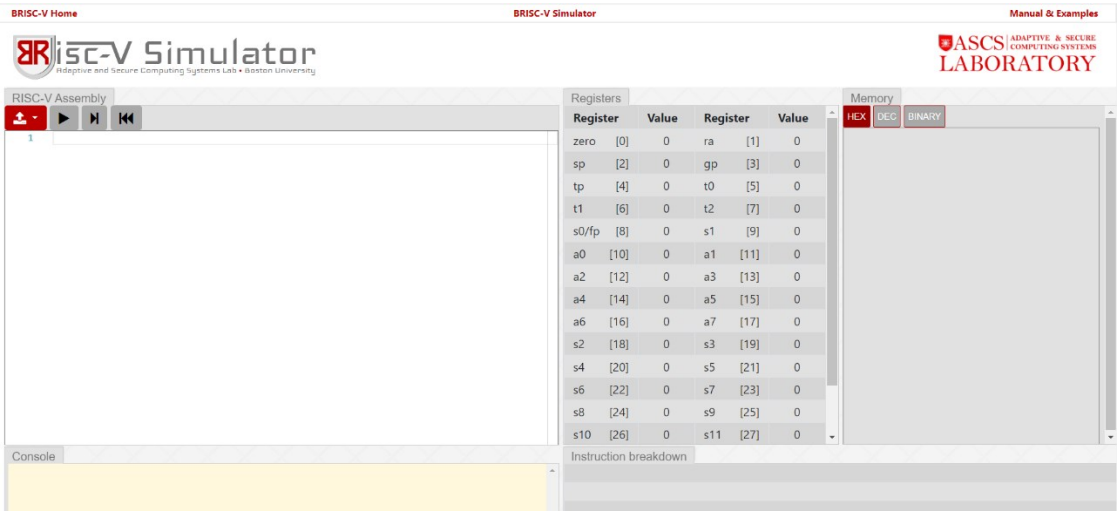
Poslední aplikací, která je zde v rámci rešerše popsána je BRISC-V Simulator. Opět jde o simulátor, který je volně dostupný ve formě webové stránky. Přičemž i tento simulátor byl vytvořen v laboratoři ASCS Laboratory, která je součástí univerzity Ira A. Fulton Schools of Engineering v USA.

4.3.1 Popis

V tomto případě se nejedná o RISC-V simulátor, ale o BRISC-V. Tato skutečnost však nijak neovlivňuje samotnou funkčnost simulátoru, proto byl i tento simulátor zařazen do rešerše. BRISC-V je parametrizovaná sada modulů pro průzkum návrhového prostoru s architekturami založenými na ISA RISC-V. BRISC-V se skládá z řady různých architektur procesorů, simulátoru a nástroje pro vizuální generování verilogových souborů, který je určen pro vzdělávací a výzkumné projekty. [24]

Na první pohled se jedná o vcelku jednoduchý simulátor. Oproti předchozím dvěma simulátorům je po stránce designu trochu méně přívětivý, ale i přesto je pro uživatele přehledný a snadno ovladatelný. Je rozdělen do tří hlavních částí, nalevo se nachází blok pro zápis instrukcí, vedle něho poté registry a vpravo je zobrazena paměť. Je zde také možnost načtení instrukcí, spuštění, krokování a resetování. Velice důležitý je pak odkaz na manuál a příklady,

který se nachází úplně vpravo nahoře. Ten uživatele přesměruje na stránku, kde se nachází vzorové příklady pro simulaci a také velmi podrobný tutoriál, jak se simulátorem pracovat.



Obrázek 7 - BRISC-V simulator

4.3.2 Hodnocení

Tento simulátor je vcelku jednoduchý a přehledný. Jednou ze záporných stránek tohoto simulátoru by mohl být jeho design, který není úplně moderní a hezký, ale vzhled není až tak podstatný, navíc záleží na osobních preferencích každého uživatele, zda se mu tento vzhled líbí či nikoliv. Větší nevýhodou, opět související se vzhledem a rozvržením aplikace, je umístění odkazu na příklady a tutoriál. Tento odkaz by měl být rozhodně větší nebo by na něho mělo být nějakým způsobem lépe upozorněno, aby se nestalo, že ho uživatel přehlédne.

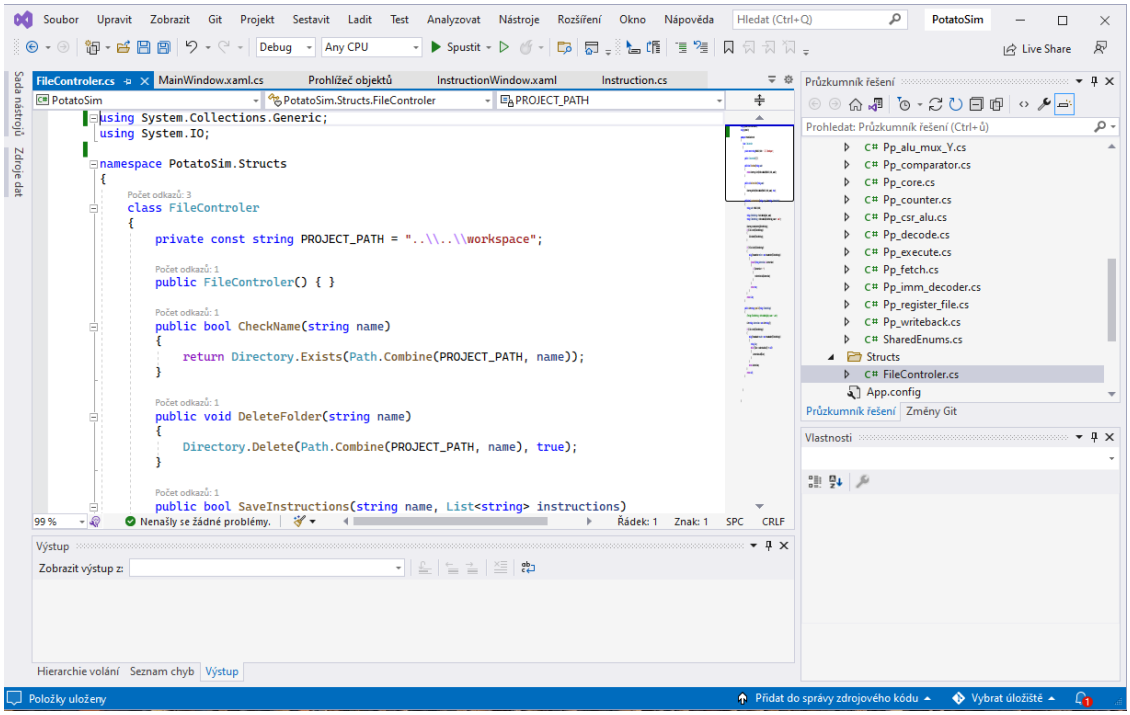
Samotná simulace je pak jednoduše ovladatelná, a právě díky již zmíněnému tutoriálu a příkladům zvládne tento simulátor ovládat i úplný začátečník. Právě tutoriál je hlavní výhodou tohoto simulátoru, je v něm totiž podrobný popis, jak simulátor funguje a jak s ním zacházet. Pokud si uživatel tento tutoriál prostuduje a využije přiložených příkladů, může si simulátor vyzkoušet, aniž by měl jakékoliv předchozí zkušenosti.

5 POUŽITÉ TECHNOLOGIE

Tato část práce je věnována technologiím, které byly použity při vytváření aplikace. Bude zde popsáno vývojové prostředí Visual Studio, programovací jazyk C# a framework pro tvorbu formulářových aplikací WPF.

5.1 Vývojové prostředí Visual Studio

Visual Studio je integrované vývojové prostředí, které se používá při vývoji softwaru pro platformu Microsoft Windows, právě společností Microsoft bylo toto IDE vyvinuto. Jde o velmi rozšířené vývojové prostředí, které poskytuje uživatelům velkou řadu nástrojů pro vývoj softwaru v jazycích jako je C#, C++, Visual Basic, F# a mnoho dalších. Visual Studio je výborným nástrojem nejen pro vývoj softwaru pro platformu Microsoft Windows, ale je možné ho využít i pro vývoj mobilních aplikací, cloudu a webových služeb. Toto IDE také obsahuje řadu nástrojů pro správu verzí, testování a ladění kódu. Visual Studio podporuje integraci s různými systémy pro správu verzí jako jsou Git, Subversion a Team Foundation Server. První verze Visual Studia byla vydána v roce 1997 a byla pojmenována jako Visual Studio 97 s verzí číslo 5.0. Nejnovější verze sady Visual Studio je 2022 verze 17.0.11. Tato verze je pak k dispozici ve třech různých edicích. [11]



Obrázek 8 - Visual studio

První dostupnou edicí je Community. Jedná se o bezplatnou verzi, která je primárně určena pro akademické účely, individuální vývojáře a malé týmy. Tato verze nabízí základní nástroje

pro vývoj softwaru v různých jazycích, webových technologiích a mobilních platformách, dostupnými funkcemi je docela podobná edici Professional. Tato edice však vzhledem k tomu, že je bezplatná, obsahuje určitá omezení. Například není dovoleno tuto edici používat v organizacích, kde je více než 250 počítačů a roční tržby překračují 1 milion USD. [11]

Druhou variantou je již zmíněná edice Professional, která je o něco málo rozšířenější než edice Community. Jedná se o komerční verzi sady Visual Studio. Tato edice poskytuje například podporu pro úpravy XML a XSLT a obsahuje nástroje jako je Server Explorer, dále umožňuje integraci s Microsoft SQL Serverem, také obsahuje například podporu pro testování aplikací, vývoj pro cloudové platformy a podporu pro různé další pluginy a rozšíření. Společnost Microsoft poskytuje tuto edici v bezplatné zkušební verzi. Hlavním cílem této edice je poskytovat flexibilitu a s ní spojené profesionální vývojářské nástroje pro vytváření různých typů aplikací, produktivitu, což zahrnuje výkonné funkce jako CodeLens a v neposlední řadě poskytuje i možnost spolupráce pomocí agilních nástrojů pro plánování projektů. [11]

Poslední verzí je Visual Studio Enterprise, jde o nejvyšší verzi Visual Studia, která nabízí nejvíce nástrojů a fukncí pro vývoj softwaru. Jedná se o integrované komplexní řešení určené pro velké firmy a organizace, které potřebují výkonný nástroj pro vývoj softwaru s pokročilými funkcemi a možnostmi. Společnost Microsoft poskytuje 90denní bezplatnou zkušební verzi této edice. Hlavní výhodou verze Enterprise je vysoká škálovatelnost a poskytování vysoce kvalitního softwaru. [11]

Celkově lze říci, že Visual Studio je všestranný nástroj pro vývoj softwaru, který nabízí řadu funkcí a nástrojů pro vývoj aplikací pro různé platformy. Mezi některé z hlavních výhod Visual Studia patří rychlé ladění, důkladné testování, správa verzí, spolupráce s využitím Live Share, která umožňuje celému týmu upravovat a ladit kód v reálném čase bez ohledu na jazyk nebo platformu. [12]

5.2 Jazyk C#

Jazyk C# je moderní programovací jazyk vyvinutý firmou Microsoft. Jedná se o jednoduchý, objektově orientovaný jazyk, který je odvozen z C, C++ a Javy. C# je široce používaný vývojáři pro tvorbu Windows aplikací, her a webových stránek. Tento programovací jazyk byl vyvinut Andersem Hejlsbergem a jeho týmem při vývoji .Net Framework. Je určen pro CLI, která se skládá ze spustitelného kódu a běhového prostředí, které umožňuje použití různých vysokoúrovňových jazyků na různých počítačových platformách a architekturách. Jedná se o univerzální programovací jazyk, který je strukturovaný a komponentově orientovaný. Zároveň je poměrně snadný na naučení a vytváří efektivní programy. [13]

Hlavními výhodami jazyka C# jsou jeho přehlednost, jednoduchost, silná typová kontrola, garbage collector a to, že nepovoluje přímou manipulaci s pamětí. Byl založen na základě současných trendů a je tedy velmi výkonný a jednoduchý pro vytváření interoperabilních, škálovatelných a robustních aplikací. C# je objektově orientovaný jazyk, což znamená, že podporuje zapouzdření dat, dědičnost, polymorfismus a rozhraní. Zároveň zavádí struktury, které umožňují, aby se z primitivních typů staly objekty. Mezi další výhody tohoto jazyka patří typová bezpečnost, škálovatelnost a interoperabilita. [14]

5.3 Windows Presentation Foundation

Windows Presentation Foundation neboli WPF je výkonný framework pro vytváření aplikací Windows. Nejprve byl představen ve verzi .NET Framework 3.0 a poté byly v následujících verzích .NET Framework přidány další funkce. [15]

Framework disponuje velkým množstvím hotových komponent, ze kterých je možné formulář jednoduše poskládat. Jedná se především o různá tlačítka, posuvníky, pole, popisky a další komponenty. Pokud by při vytváření aplikace nestačily pouze tyto komponenty, tak nám WPF samozřejmě umožňuje i vytváření vlastních ovládacích prvků. Ve WPF jsou jednotlivé prvky uživatelského rozhraní navrženy v jazyce XAML a jejich chování je pak implementováno v procedurálních jazycích, jako například v jazyce C#. Je tedy snadné oddělovat chování od kódu návrháře. [15][16]

```
<Window x:Class="PotatoSim.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        mc:Ignorable="d"
        Title="PotatoSim" Height="650" Width="1200" MinHeight="650" MinWidth="1200">
    <Window.Resources>
        <ControlTemplate x:Key="ButtonControlTemplate" TargetType="{x:Type Button}">
            <Border x:Name="border" BorderBrush="{TemplateBinding BorderBrush}" BorderThickness="{TemplateBinding Bord
                <ContentPresenter x:Name="contentPresenter" ContentTemplate="{TemplateBinding ContentTemplate}" Conter
            </Border>
            <ControlTemplate.Triggers>
                <Trigger Property="Button.IsDefaulted" Value="True">
                    <Setter Property="BorderBrush" TargetName="border" Value="{DynamicResource {x:Static SystemColors.
                </Trigger>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter Property="Background" TargetName="border" Value="#3902db"/>
                    <Setter Property="BorderBrush" TargetName="border" Value="#2a059c"/>
                </Trigger>
                <Trigger Property="IsPressed" Value="True">
                    <Setter Property="Background" TargetName="border" Value="#3902db"/>
                    <Setter Property="BorderBrush" TargetName="border" Value="#2a059c"/>
                </Trigger>
            </ControlTemplate.Triggers>
```

Obrázek 9 - Ukázka XAML

Díky komponentové architektuře se s veškerými grafickými prvky pracuje jako se samostatnými objekty. Pracuje se tak nejen s prvky používanými ve formulářích, ale i s grafickými objekty. Tudíž je možné i grafickému objektu, jako například čtverci, přidat určité chování. Například je možné přidat grafickému objektu funkci se změnou barvy při najetí

myši, aniž by bylo zapotřebí složitých výpočtů. Právě díky tomuto chování je práce s WPF poměrně snadná.

Před vznikem WPF byl používán převážně formulářový framework Windows Forms, tento framework je sice starší, ale stále je často využívaný a je tedy součástí .NET frameworku. V současnosti se využívá jak WPF tak Windows Forms, avšak vzhledem k tomu, že WPF bylo vytvořeno, aby odstranilo některé nedostatky Windows Forms, je technologicky mnohem pokročilejší. I když je stále mnoho existujících aplikací, které používají Windows Forms a stále vznikají i nové aplikace ve Windows Form, je v současnosti již lepší vyvíjet právě pomocí WPF. [16]

Pokud jde o porovnání WPF a Windows Form jednou z hlavních výhod WPF je, že je jeho uživatelské rozhraní založeno na vektorové grafice narozdíl od uživatelského rozhraní Windows Forms. Díky tomu umožňuje WPF plynule škálovat komponenty uživatelského rozhraní, aniž by docházelo k problémům se zkreslením velikosti. Mezi další výhody oproti Windows Form patří možnost snadného přizpůsobení ovládacích prvků. Další výhodou je také rychlost vykreslování, která je ve WPF vyšší než ve Windows Form. [17]

6 NÁVRH APLIKACE

Tato kapitola se zabývá popisem základní analýzy a návrhu aplikace. Jsou zde popsány požadavky na vzhled a funkčnost výsledné aplikace. Dále je zde v jednoduchosti popsán UML diagram tříd a v neposlední řadě tato kapitola popisuje i adresáře, které se nacházejí v projektu finální aplikace.

6.1 Požadavky na aplikaci

V této části jsou vyjmenovány a popsány základní funkční a nefunkční požadavky, které jsou kladeny na výslednou aplikaci. Mezi nejdůležitější z požadavků patří především tyto:

- Funkční požadavky
    - Možnost uložení či načtení instrukcí
    - Možnost zapsání vlastních instrukcí
    - Možnost spuštění simulace
    - Zamezení simulace při nevalidních instrukcích
    - Možnost zápisu do registrů
    - Možnost krokování simulace
    - Nastavení času pro animaci
- Nefunkční požadavky
    - Napsána v jazyce C#, framework WPF
    - Pro systémy Windows

Výsledná aplikace bude sloužit jako jednoduchý grafický simulátor jádra RISC-V procesoru Potato. Nejdůležitějším úkolem aplikace je tedy možnost spouštění a ovládání simulace. S tímto bodem se pojí hned několik funkčních požadavků.

Aby bylo možné vůbec spustit simulaci, je zapotřebí dovolit uživateli zápis instrukcí. Instrukce může uživatel buď do aplikace zapsat sám nebo je nahrát ze souboru. K zápisu a načítání instrukcí se poté pojí požadavek na jejich validaci. Zároveň by měl mít uživatel možnost si zapsané instrukce uložit do souboru. Pokud zapsané instrukce nebudou ve správném formátu, nemělo by uživateli být dovoleno spuštění simulace.

Mezi povolené instrukce patří základní instrukce z instrukční sady RV32I, přičemž tato instrukční sada a jednotlivé formáty instrukcí, které obsahuje, je popsána výše v této práci.

Požadavkem tedy je, aby aplikace uměla pracovat s instrukcemi právě z této instrukční sady. Zároveň by aplikace měla být schopna jednotlivé instrukce dekódovat a zobrazovat informace o dané instrukci, například o který z typů instrukce jde.

Před spuštěním simulace by měl uživatel zároveň mít možnost zapisovat hodnoty do registrů, s kterými se bude během simulace pracovat. I zde se předpokládá, že uživateli bude povolen zápis pouze validních hodnot.

Samotná simulace by měla uživateli dovolit spuštění buď jako animace nebo postupné krokování. Pokud bude uživatel spouštět animaci, souvisí s jejím spuštěním požadavek na nastavení času pro animaci. Čímž se rozumí čas pro vykonání jednotlivých kroků animace. V případě spuštěné animace by uživatel měl mít zároveň možnost animaci pozastavit a poté případně znovu spustit. Zároveň by zde měla být i možnost resetování animace, čímž se rozumí vrácení animace zpět na úplný začátek. Tato možnost by měla fungovat i v případě, že uživatel zvolí možnost simulaci postupně krokovat.

Postupné krokování simulace by mělo být velmi podobné animaci, i v tomto případě by měl uživatel vidět jednotlivé operace, které se dějí s instrukcí. Jak již bylo zmíněno, i v tomto případě by mělo dojít před spuštěním krokování ke kontrole, zda jsou všechny instrukce ve validním stavu. Zároveň by aplikace měla umožňovat přechod mezi animací a postupným krokováním, pokud se uživatel v průběhu simulace rozhodne změnit variantu, jak simulace probíhá.

Nejdůležitějším nefunkčním požadavkem je pak jazyk, ve kterém bude aplikace napsána. Zvoleným jazykem je C# konkrétně s grafickou nadstavbou WPF, tento jazyk byl zvolen z toho důvodu, že se jedná o moderní programovací jazyk. Nadstavba WPF je pak jednou z nejlepších pro tvorbu grafických aplikací, což je pro výslednou aplikaci velmi důležité. Se zvoleným programovacím jazykem také velmi úzce souvisí i požadavek na operační systém, kterým je v tomto případě Microsoft Windows.

6.2 Popis tříd

Tato část práce je věnována popisu použitých tříd výsledné aplikace. Pro náhled do tříd slouží UML diagram tříd, který poskytuje zajímavý pohled na datovou strukturu aplikace, operace a souvislosti mezi různými objekty v aplikaci. Vzhledem k velkému počtu tříd je diagram rozdělen na dva samostatné diagramy. Prvním z diagramů je diagram znázorňující třídy, které slouží jako třídy popisující procesor a pomocnou třídu pro práci se soubory. Druhý diagram poté zobrazuje třídy, které zobrazují grafickou část aplikace.

6.2.1 Popis tříd reprezentujících procesor

Tento diagram, který je přiložený jako příloha A na konci této práce, popisuje strukturu tříd, které slouží k popisu procesoru a také pomocné třídy, která slouží pro práci se soubory. Celkem je v aplikaci použito sedmnáct tříd pro popis jednotlivých částí procesoru a jedna třída pro výčet.

Většina těchto tříd jsou původně třídy jazyka VHDL, které byly přepsány do jazyka C#. Podrobnějšímu přepisu těchto tříd se věnuje následující kapitola. Ve stručnosti každá z těchto tříd většinou obsahuje atributy, které jsou zde pro uložení portů, dále tyto třídy mohou obsahovat generické atributy. Poté tyto třídy obsahují funkčnost v závislosti na tom, která část procesoru je danou třídou znázorněna. A v neposlední řadě tyto třídy také obsahují ve většině případů testovací metodu, která testuje funkčnost dané třídy.

Dalšími třídami jsou zde třídy pro popis instrukcí. Těmito třídami jsou třída Instr_code a datová třída Instruction. Ve třídě Instr_code dochází ke konverzi instrukce. Jsou zde metody pro každý možný typ instrukce, přičemž v těchto metodách je instrukce rozdělena na jednotlivé části, dekódována a informace o ní jsou uloženy do již zmíněné datové třídy Instruction. Obě tyto třídy se pak často využívají i v grafické části aplikace.

Jednou z dalších tříd zobrazených v tomto diagramu je třída SharedEnums. Jedná se o třídu, která v sobě obsahuje čtyři výčtové typy. Konkrétně jde o výčtový typ sloužící pro ALU, dále pak o výčtový typ, který je využíván muxy, také výčtový typ pro komparátor a posledním výčtem je výčet pro mód zápisu.

Dále je v diagramu zobrazena třída FileController sloužící jako pomocná třída pro veškerou práci se soubory v aplikaci, ať jde o načítání nebo jejich ukládání. Atributem této třídy je konstanta s cestou k ukázkovým úlohám s instrukcemi. Tato třída také obsahuje čtyři metody, jejichž výpis je možné vidět na obrázku v příloze A. Tyto metody slouží ke kontrole, zda již existuje složka se zadaným jménem, dále je tu metoda pro smazání složky a poté dvě nejdůležitější metody. Těmi jsou metoda pro uložení instrukcí do souboru a metoda pro načtení instrukcí ze souboru.

6.2.2 Popis grafických tříd

Výsledná práce obsahuje celkem čtyři grafické třídy, přičemž zde budou popsány tři z nich. To je z toho důvodu, že třída App je výchozí třídou aplikace, která ale neobsahuje žádnou vlastní logiku, která by zde mohla být popsána, budou popsány tedy pouze zbývající tři třídy. Jedná se o třídy, které reprezentují různá grafická okna použitá v aplikaci. Vzhledem k velikosti obrázku je tento diagram, stejně jako i diagram tříd popisujících procesor, přiložen jako příloha na konci

práce, konkrétně jde o přílohu B. Veškerá grafická okna jsou spouštěna z hlavní grafické třídy s názvem MainWindow.

Jedná se o hlavní okno aplikace, které zobrazuje menu a plochu pro simulaci, která je rozdělena do několika částí. Samotnému popisu GUI simulátoru je věnována jedna z následujících kapitol této práce. Z této třídy jsou také volány metody ze třídy FileController a také jsou zde používány třídy pro popis procesoru. Jedná se o třídu zodpovědnou za veškerou práci s grafickými prvky, a především je tato třída zodpovědná za simulaci procesoru. Nachází se zde velké množství atributů, které jsou zapotřebí pro správný chod aplikace jako například kolekce obsahující instrukce, obsah registrů a obsahy informační a datové paměti, nebo třeba časovač, který slouží pro animaci. Toto je pouze malý výčet atributů, všechny atributy je možné si prohlédnou v přiloženém diagramu.

Většina metod, které jsou obsaženy v této třídě slouží pro simulaci jádra procesoru. Jedná se například o metody Simulation, SetLabelsBackgroundToDefault, SetLabelsContentToDefault, a metody SimulationStep1 až SimulationStep6. Také tu je spousta různých eventů, které jsou přiřazeny grafickým objektům (např. tlačítkům), čímž je definováno jejich chování při různých událostech. Takovým eventem je například metoda Event_Animate, která se vykonává při kliknutí myší na tlačítko Animate. Vzhledem k tomu, že vyvíjená aplikace je z převážné části založena na práci s grafickým prostředím, je v této hlavní grafické třídě obsaženo skutečně velké množství metod a různých eventů pro obsluhu událostí.

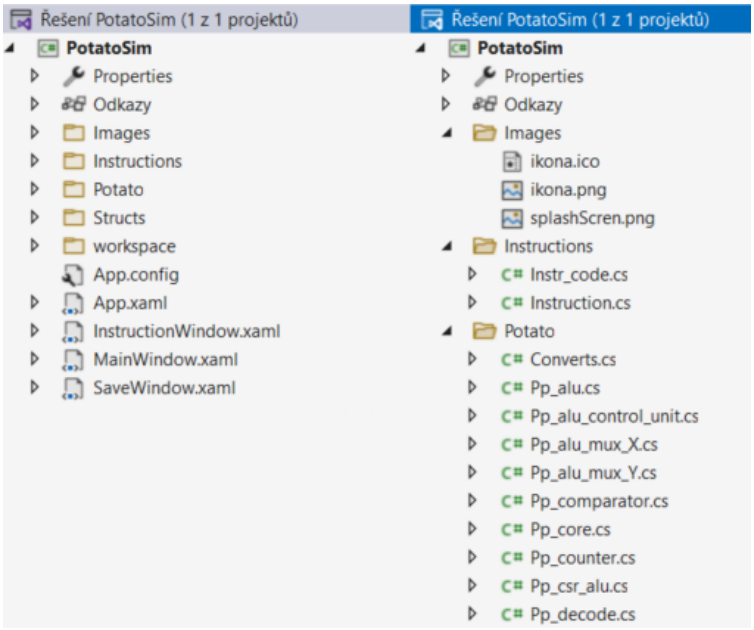
Další třídou je třída InstructionWindow, která slouží pouze jako třída pro zobrazování informací a nemá tedy žádný uživatelský vstup. Jedná se o třídu, která na základě atributu instruction, který je nastaven v konstruktoru této třídy, zobrazuje informace o příslušné instrukci. Atribut instruction je typu Instruction, což je datová třída popsána v předchozí podkapitole. Tato datová třída obsahuje veškeré informace o instrukci, takže v tomto grafickém okně jsou tyto informace již pouze zobrazeny. Kromě nastavení jednotlivých zobrazovaných polí tedy tato třída neobsahuje žádnou další logiku.

Poslední důležitou grafickou třídou je třída SaveWindow. Tato třída je využívána pro ukládání instrukcí do souboru. Konkrétně je v této třídě pole pro název ukládaného souboru a dvě tlačítka pro uložení a zrušení ukládání. Tato třída je poté využívána v hlavní grafické třídě MainWindow, kde se na základě výstupu z třídy SaveWindow buď uloží nebo neuloží instrukce do souboru. Samotná třída SaveWindow má pak kromě konstruktoru pouze dvě jednoduché metody, a to metodu OKButton_Click, která je zavolána při kliknutí na tlačítko OK a nastaví výsledek dialogu na kladnou hodnotu. Druhou metodou je pak metoda TextChangedEventHandler, která kontroluje, aby délka zadávaného textu byla delší než čtyři

znaky. Pokud je délka kratší, tak není uživateli umožněno kliknout na tlačítko OK, ve chvíli, kdy je délka zadávaného názvu dlouhá alespoň čtyři znaky, je uživateli tlačítko zpřístupněno. Tato metoda je typu event a provádí se pokaždé, když je jakkoliv změněn text v příslušném textovém poli pro zadávání názvu.

6.3 Popis jednotlivých adresářů

V této části je popsána adresářová struktura projektu, kde jsou popsány jednotlivé adresáře, popřípadě obsah daných adresářů. Hlavní adresářová struktura kromě složek obsahuje také veškeré zdrojové kódy pro grafické třídy, přičemž tyto třídy byly již popsány. Adresářová struktura obsahuje celkem pět složek.



Obrázek 10 - Adresářová struktura

První složkou v adresářové struktuře je složka Images, kde jsou uloženy veškeré obrázky, které jsou použity v grafických prvcích aplikace. Například je zde obrázek, který slouží pro úvodní načítací obrazovku aplikace (splashScreen). Dále je tu také samotná ikona aplikace.

Další složkou je složka Instructions, která obsahuje dvě již výše zmíněné třídy pro práci s instrukcemi. Tyto třídy jsou umístěny mimo hlavní složku, protože se nejedná o třídy, které by přímo popisovaly chování některé části procesoru.

Hlavní složkou této aplikace je složka Potato, která obsahuje třídy reprezentující procesor. Jsou zde umístěny veškeré třídy, které popisují některou z částí procesoru. A také pomocné třídy, se kterými se v třídách pro popis procesoru pracuje. Pomocnou třídou je myšlena například třída SharedEnums obsahující výčtové typy. Tento adresář obsahuje nejvíce tříd ze všech adresářů, protože je zde obsažena v podstatě veškerá logika chování procesoru, jedná se

tedy o nejdůležitější z adresářů. Podrobnější popis některých těchto tříd je obsažen v následující kapitole, která se věnuje přepisu kódu z jazyka VHDL do C#.

Posledními dvěma složkami adresářové struktury jsou složka structs, která obsahuje pouze třídu FileController, která slouží pro práci se soubory a složka workspace. Tento adresář slouží jako uložiště instrukcí. Přičemž každý soubor s instrukcemi je uložen ve složce pod svým jménem. V této složce jsou obsaženy i ukázkové soubory s instrukcemi.

7 PŘEPIS VHDL KÓDU NA C#

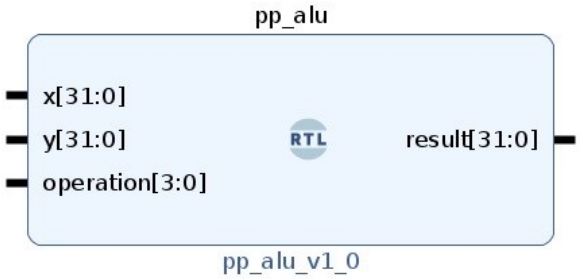
Tato kapitola popisuje způsob přepisu VHDL kódu do jazyka C#. Na zvolených ukázkách je popsán způsob převodu datových typů, jednotlivých komponent a způsob převodu mapování. Také je zde stručně popsán kompilátor RISC-V procesoru.

7.1 Přepis kódu pp_alu.vhd

Základní princip přepisu kódu bude popsán na nejdůležitější komponentě a to ALU (Aritmeticko Logická Jednotka). Jak bylo popsáno již výše v kapitole věnující se popisu jazyka VHDL, tak základním prvkem VHDL je entita. Tato entita má název pp_alu a proto je vytvořena v jazyce C# třída se stejným názvem (static class Pp_alu). Třída je typu static, neboť pro třídy popisující procesor byl zvolen statický přístup.

Tento přístup byl zvolen především z toho důvodu, že se jedná o třídy, od kterých nechceme v projektu vytvářet instance, jsou to třídy obsahující pouze pomocné metody. Tyto metody jsou posléze v ostatních třídách volány pomocí názvu třídy a tečkové notace. Navíc jsou díky tomuto přístupu jednotlivé třídy v paměti zavedeny po celou dobu běhu aplikace. V tomto případě díky použití statiky dochází i k zpřehlednění kódu.

V entitě jsou definovány v bloku port vstupně výstupní signály této entity. Entitu si je možné představit jako stavební blok RISC-V procesoru, viz obrázek číslo 11. V tomto případě se jedná o signály x, y, operation a result. Vstupní signály x, y jsou typu std_logic_vector (31 downto 0), jelikož se jedná o 32bitovou hodnotu, tak je jí v jazyce C# přiřazen typ uint. Dalším vstupním signálem je signál operation, který je typu alu_operation. Tento typ může nabývat specifických hodnot, proto mu bude v jazyce C# přiřazen typ enum. V tomto výčtovém typu pak budou definovány všechny stavy, kterých může signál nabývat. Posledním signálem je signál výstupní, označený jako result. Jelikož je stejného typu jako x a y, je mu přiřazen stejný typ uint.



Obrázek 11 - pp_alu

Tyto vstupně výstupní signály jsou v jazyce C# definovány jako atributy třídy a pomocí public metod get a set jsou jejich hodnoty čteny a zapisovány. Daná třída také obsahuje statický konstruktor, který slouží k počátečnímu nastavení hodnot a vyloučení neurčitých stavů.

Ve VHDL bloku architektura je pak popsána funkcionalita této entity. V této entitě je použit behaviorální popis, tento popis je založen na popisu chování logického obvodu jako reakce obvodu na vstupní signály. Toto bývá označováno jako vysoká úroveň abstrakce, kdy není popisována vnitřní struktura logického obvodu, ale pouze jeho chování. Více dopodrobna je behaviorální styl popsán v kapitole zabývající se popisem jazyka VHDL.

V tomto bloku se nachází jeden proces s názvem calculate, který má vstupní signály x, y a operation. Tento proces si lze představit jako metodu v jazyce C# (static void calculate()), která je volána pokaždé, když se jakkoliv změní hodnota libovolného parametru (x, y nebo operation). V rámci třídy Pp_alu je toto chování realizováno voláním metody calculate() při každém nastavení atributu pomocí set, viz ukázka kódu na obrázku číslo 12.

```
public static uint X
{
    set { X = value; Calculate(); } //set method
}
```

Obrázek 12 - Ukázka C# kódu pp_alu

Na základě hodnoty operation je v metodě calculate vypočítána výsledná hodnota result. Hodnota operation nabývá 10 hodnot, proto je použita konstrukce switch-case. V kódu se vyskytují dva typy proměnných, těmi jsou signed a unsigned. Typ unsigned je kladné číslo o velikosti 32 bitů, které v tomto případě nabývá hodnot v rozsahu 0 až 2^{32}.

Pro účely ověření funkčnosti každé entity je vytvořena v každé třídě funkce test, která ověřuje základní funkčnost třídy. Toto je řešeno pomocí vyvolání a zachycení uživatelských výjimek.

7.2 Přepis kódu pp_csr_alu.vhdl

Tato entita byla zvolena z důvodu popisu dalších funkcionalit jazyka VHDL. Jmenovitě se jedná o datový objekt signal.

```
port(
    x,y : in std_logic_vector(31 downto 0);
)

signal a,b : std_logic_vector(31 downto 0);
a <= x;
b <= std_logic_vector(resize(unsigned(immediate), b'length));
```

Obrázek 13 - Ukázka VHDL kódu pp_csr_alu

Pro tento datový objekt neexistuje v jazyce C# žádný podobný ekvivalent. Na první pohled by se mohlo zdát, že datový typ signal je pouhé přiřazení hodnoty jedné proměnné jiné proměnné, tak jak to je v jazyce C# běžné. Zde se však jedná o kontinuální přiřazení, což znamená, že kdykoliv se změní hodnota x tak se automaticky změní hodnota proměnné a. Zde se nabízí použití C# ukazatelů, které jsou však v tomto případě nevhodné, neboť během přiřazení se budou zároveň provádět i modifikace proměnných.

Z tohoto důvodu je zde vytvořena metoda přiřazení, nazvaná jako signals(), která je vždy volána při modifikaci atributu v metodě set, podobně je řešen i process.

```
public static uint X
{
    set { X = value; Signals(); Calculate(); } //set method
}
```

Obrázek 14 - Ukázka C# kódu pp_csr_alu

V entitě se nachází jeden proces s názvem calculate, který je řešen shodným způsobem jako u ALU, tedy je volán pokaždé, když se jakýmkoliv způsobem změní hodnota libovolného z parametrů.

Ještě je třeba zde popsat dvě vstupní proměnné. První proměnná use_immediate je typu std_logic, takže se jedná o binární proměnnou. V jazyce C# je proto definována jako typ bool a může tedy nabývat hodnot true nebo false. Druhá proměnná immediate je typu std_logic_vector(4 downto 0), což je 5-ti bitový typ. Pro jednotnost lze použít typ uint a v metodě set provést oříznutí vstupní hodnoty pomocí logického součinu.

```
public static uint Immediate
{
    set { immediate = value & 0x1F; Signals(); Calculate(); } //set method
}
```

Obrázek 15 – Ukázka C# kódu immediate

7.3 Přepis kódu pp_execute.vhd

Na poslední entitě je zde vysvětleno mapování, respektive vzájemné vnořování entit.

```
alu_instance: entity pp_alu
    port map(
        result => alu_result,
        x => alu_x,
        y => alu_y,
        operation => alu_op
    );
```

Obrázek 16 - Zjednodušená entita pp_execute v jazyceVHDL

V této entitě je vytvořena instance entity pp_alu a následně jsou namapovány jednotlivé vstupně výstupní signály. Zde opět musí být zaručeno, že jakákoliv změna hodnoty vstupního signálu se musí okamžitě projevit na změně hodnoty výstupního signálu. Vzhledem k tomu, že jazyk C# zpracovává naprogramovaný algoritmus, kdežto jazyk VHDL pracuje spojitě a především paralelně, musí se zde zavést určitá časová synchronizace pomocí časovače, který bude obdobou diskrétního signálu. Pro potřeby simulátoru toto nevadí, ale při zpracování v reálném čase by toto řešení mohlo vytvářet neurčité či zakázané stavy. Z tohoto důvodu je zde vytvořena metoda, nazvaná jako mapping(), která je vždy volána při modifikaci atributu (obdoba process). Jakmile je změněn vstupní signál, je volána metoda mapping, která zajistí přiřazení hodnoty dané proměnné metodou set. Díky metodě set jsou pak spuštěny všechny procesy svázané s touto proměnnou. Tímto postupem je docílena podobné funkcionalita spojitosti jako je v původním VHDL kódu.

7.4 Kompilátor RISC-V procesoru

Pro účely simulace bylo zapotřebí zdrojový kód zkompilovat. Nabízely se dvě cesty, a to buď použití již existujících kompilátorů, nebo druhý způsob, a to naprogramování vlastního kompilátoru. Pro účel této diplomové práce byla zvolena druhá cesta. Tato možnost byla zvolena především pro lepší pochopení instrukční sady. Tento jednoduchý kompilátor parsuje vstupní instrukci pomocí dvou fází. První fáze detekuje instrukci a na základě typu této instrukce jsou parsovány zbylé atributy.

Pro tento účel byla vytvořena třída Instr_code, která obsahuje všechny instrukce z instrukční sady RV32I rozdělené do kategorií dle typu instrukce. Pro tento účel byly použity komponenty Dictionary, které slouží nejen jako seznam instrukcí daných kategorií, ale obsahují i další potřebné hodnoty jako např. hodnotu funct3.

```
addi x1,x0,0x05
addi x1,x0,-0x05
addi x1,x0,5
addi x1,x0,-5
addi x1,x0,0xfffffffb
```

Obrázek 17 - Ukázka způsobu zadání čísla (imm)

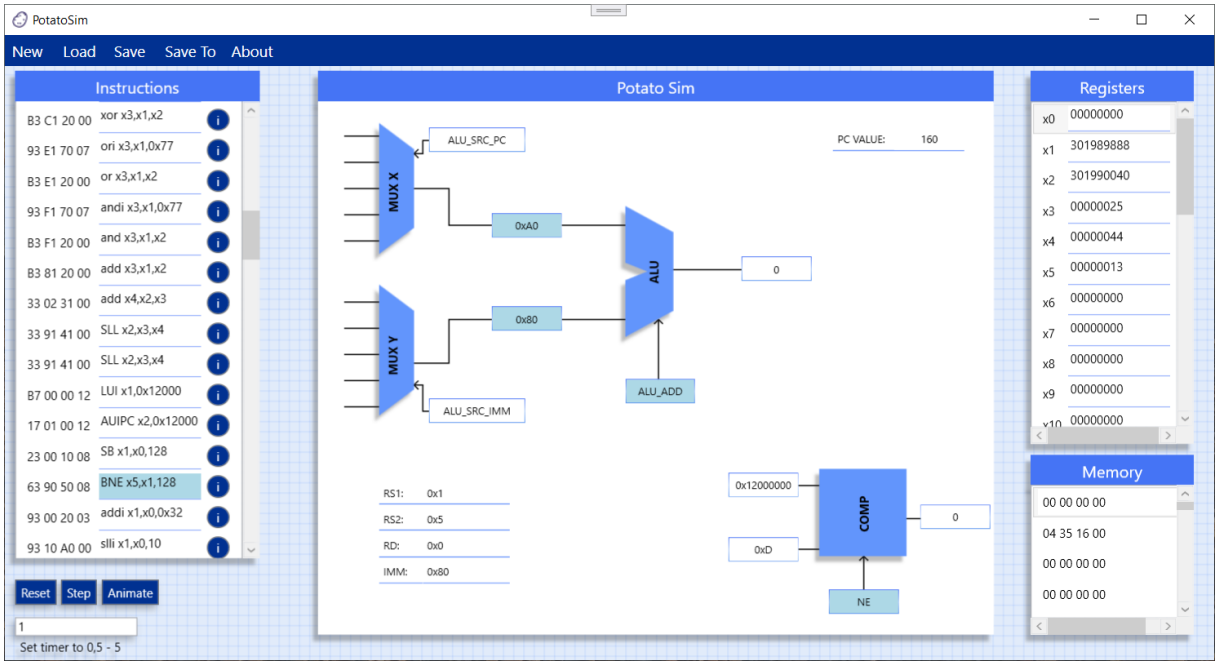
8 POPIS GUI SIMULÁTORU

Tato část práce se věnuje popisu samotné vyvíjené aplikace. Je zde popsáno rozložení grafického prostředí a také je zde popsána práce se samotnou aplikací. Další součástí této kapitoly je popis chování a základních pravidel pro správnou práci s aplikací. A dále jsou zde popsány funkce a grafické rozložení některých dalších oken použitých v aplikaci.

8.1 Popis rozložení aplikace

V této části je popsáno samotné rozložení výsledné aplikace. Rozložení aplikace je velmi jednoduché a přehledné, v horní části aplikace se nachází lišta okna s hlavním menu. Pod tímto menu je poté samotné okno aplikace, které lze pomyslně rozdělit na tři části.

Vlevo se nachází blok, ve kterém se zobrazují zapsané instrukce a pod tímto blokem se pak nachází jedno pole a tři tlačítka sloužící k ovládání simulace. Prostřední blok pak zobrazuje muxy, ALU a komparátor, v této části se nachází informace o jednotlivých instrukcích a výpočtech prováděných nad těmito instrukcemi. Informace zobrazované v tomto prostředním bloku se tak vždy mění v průběhu simulace. Pravá část okna poté obsahuje dva pod sebou umístěné bloky, přičemž první z těchto bloků slouží pro výpis registrů a druhý spodní blok zobrazuje datovou paměť. Také informace zobrazované v těchto dvou blocích jsou měněny v průběhu samotné simulace programu.

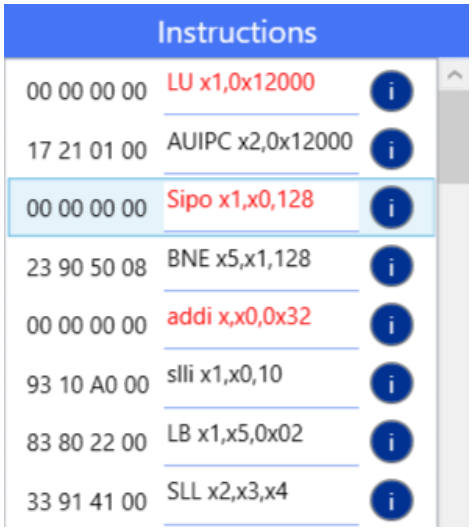


Obrázek 18 - Hlavní okno aplikace

8.2 Popis práce s aplikací

Tato část práce se zabývá popisem základní práce s aplikací a vysvětlením fungování aplikace. Po načtení hlavního okna aplikace je pole s instrukcemi prázdné, uživatel má tedy dvě možnosti, buď do pole zapsat instrukce ručně nebo si načíst instrukce ze souboru. Pro načtení instrukcí ze souboru je v horním menu k dispozici tlačítko Load. Po kliknutí na toto tlačítko si uživatel může zvolit *.asm soubor, ze kterého se budou instrukce načítat. Pokud by uživatel preferoval ruční zapsání instrukcí do pole, nabízí aplikace samozřejmě i tuto možnost.

Přičemž instrukce vložené do pole jsou vždy validovány, ať už dojde k vložení instrukcí jakýmkoliv ze dvou výše uvedených způsobů. V kódu je pro validaci vytvořena metoda IsInstructionComplete, která se nachází ve třídě Instr_code, tato metoda kontroluje, zda je instrukce správného typu a zda na základě typu obsahuje správný počet parametrů. Pokud instrukce projde validací je zobrazena normálně v poli, pokud však validací neprojde, je daná instrukce označena červenou barvou, aby si byl uživatel vědom, že je daná instrukce zapsána špatně.

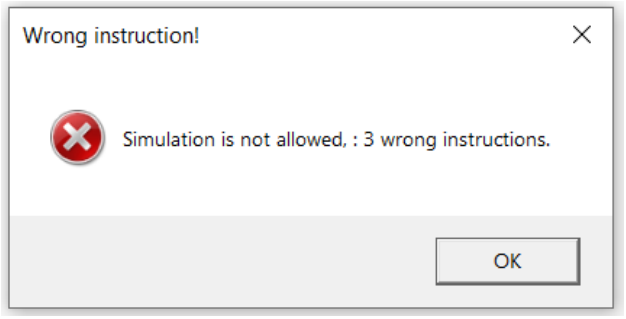


Obrázek 19 - Ukázka špatného zápisu instrukcí

Pokud instrukce projde validací, je zároveň ihned vypočítána hexadecimální hodnota dané instrukce. Tato hodnota se zobrazuje v instrukční paměti, která se nachází hned vedle pole s instrukcemi. Veškeré instrukce jsou zároveň ukládány do kolekce, která je definována při načtení hlavního okna. Tato kolekce se pak dále používá i při simulaci programu. Dále je vytvářena i kolekce pro instrukční paměť, kam se na příslušné pozice ukládají hexadecimální hodnoty daných instrukcí. Instrukce je pak také možné uložit do *.asm souboru, aby mohly být při novém otevření aplikace znovu jednoduše načteny. K ukládání instrukcí slouží tlačítka Save a Save to, která se nachází stejně jako již zmíněné tlačítko Load v horním menu aplikace. Obě

tato tlačítka vyvolají ukládací okno a v kódu je následně zavolána třída FileController, která provede uložení instrukcí a vytvoření souboru.

Ve chvíli, kdy uživatel dokončí zápis instrukcí, je možné přejít k samotné simulaci. Pro ovládání simulace slouží již zmíněná tlačítka a pole, které se nachází pod blokem pro zápis instrukcí. Konkrétně se jedná o tlačítka Reset, Step a Animate a o pole pro zadávání času pro animaci. Tlačítko Animate spouští animaci, přičemž nastaví časovač na hodnotu zadanou v uvedeném poli. Časovač určuje, jak dlouhou dobu bude trvat přechod mezi jednotlivými kroky animace. Ještě před samotným zahájením animace jsou zkontrolovány zadané instrukce, pokud je některá z instrukcí zadaná špatně, tak není povoleno spuštění animace a uživateli je zobrazeno informační okno, které mu oznamuje, že jsou některé z instrukcí zadány špatně. Předpokládá se, že by k tomuto nemělo docházet, protože jak již bylo zmíněno, instrukce ve špatném formátu jsou označeny červeně. Pokud by ale uživatel i přesto zkusil spustit simulaci, bude upozorněn právě zmíněným vyskakovacím oknem a simulace se nespustí.



Obrázek 20 - Alert window

Pokud jsou zadané instrukce ve správném formátu, dojde po stisknutí tlačítka Animate ke spuštění animace. Po kliknutí na tlačítko se zároveň změní text tlačítka na Stop, aby bylo uživateli jasné, že má možnost animaci kdykoliv pomocí tlačítka pozastavit. Zároveň v kódu dochází k zapnutí časovače, který byl nastaven na hodnotu zadanou v příslušném poli. Samotná simulace je pak rozdělena do šesti kroků, přičemž tyto kroky jsou prováděny postupně pro každou z instrukcí. Kromě těchto šesti kroků je zde ještě krok nultý, který nastává v případě, že simulace končí, tedy když už nejsou k dispozici žádné další instrukce ke zpracování. Po skončení simulace je uživatel informován o tomto konci opět pomocí jednoduchého vyskakovacího okna. Zároveň je v tomto nultém kroku zastaven časovač a pomocné atributy pro simulaci jsou nastaveny na výchozí hodnoty.

V prvním kroku animace je zavolána metoda SimulationStep1. Na začátku této metody je stejně jako i ve všech ostatních krocích zavolána metoda SetLabelsBackgroundToDefault, která jak napovídá její název nastavuje pozadí veškerých textových polí na výchozí hodnotu. Tato metoda je volána z toho důvodu, že při průběhu simulace jsou postupně barevně označovány

hodnoty, se kterými je v daném kroku simulace pracováno, při každém dalším kroku je pak zapotřebí hodnoty označené v předchozích stavech odznačit. V metodě pro první krok simulace pak dále dochází k načtení příslušné instrukce z kolekce instrukcí. Poté je načtená instrukce dekódována a je tak umožněno její další zpracování. Zároveň se v tomto kroku animace barevně zvýrazní instrukce, se kterou se zde pracuje, aby uživatel věděl, pro jakou instrukci jsou v dalších krocích simulace zobrazovány výpočty.

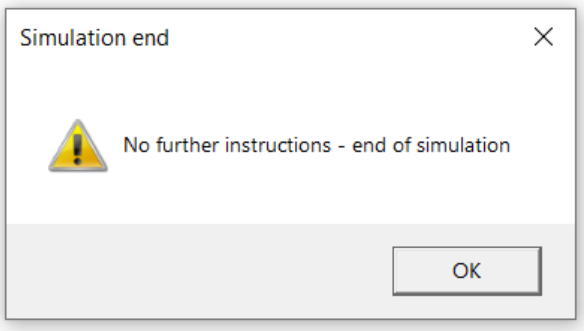
Následuje druhý krok simulace, pro který je obdobně jako u kroku prvního vytvořena metoda, tentokrát jde o metodu s názvem SimulationStep2. Tato metoda pracuje se třídou Pp_alu_control_unit, přičemž této třídě nastaví hodnoty ze třídy pro dekódování instrukce, ke kterému došlo v předchozím kroku simulace. Zároveň jsou zde nastaveny i hodnoty pro třídy Pp_alu_mux_X, Pp_alu_mux_Y a Pp_comparator. V neposlední řadě jsou pak zdrojové hodnoty vstupující do těchto tříd zobrazeny a barevně zvýrazněny v textových polích. Po tomto nastavení se přechází na další krok.

I pro třetí krok je v programu vytvořena metoda, tentokrát se jmenuje SimulationStep3. V této metodě se pracuje opět se třídami Pp_alu_mux_X, Pp_alu_mux_Y, Pp_comparator a nově také se třídou Pp_alu, která slouží pro zobrazení ALU. Této třídě jsou nastaveny vstupní hodnoty a je vyvolána metoda calculate, která na základě vstupních hodnot vypočte hodnotu výstupní. V metodě pro třetí simulační krok jsou poté ještě nastaveny a barevně zvýrazněny hodnoty, které vstupují do ALU a případně i hodnota operace vstupující do komparátoru.

Čtvrtý krok simulace je v kódu vyjádřen metodou SimulationStep4, přičemž tato metoda pouze zobrazuje a barevně zvýrazňuje výstupní hodnotu ALU a i případně výstupní hodnotu komparátoru, které byly vypočteny již v předchozím kroku simulace.

Předposlední pátý krok, pro který je zde metoda SimulationStep5, má za úkol provést zápis hodnot do registrů popřípadě do paměti. Zároveň je v tomto kroku řešena inkrementace čítače PC, popřípadě jeho modifikace, pokud se jedná o instrukce skoku.

Poslední krok simulace, pro který je zde vytvořena metoda SimulationStep6, zvyšuje číslo načítané instrukce a zkontroluje, zda jsou k dispozici ještě nějaké další instrukce k načtení. Pokud jsou k dispozici další instrukce, je nastaven krok simulace opět na jedničku a celý proces se opakuje pro další načtenou instrukci. Pokud již nejsou k dispozici žádné další instrukce k načtení, je simulační krok nastaven na nulu a uživatel je informován o konci simulace.



Obrázek 21 - Upozornění na konec simulace

Celou simulaci lze spustit pomocí tlačítka Animate, jak již bylo zmíněno. Po spuštění animace je možné tím stejným tlačítkem simulaci pozastavit, pokud je to zapotřebí. Pokud dojde k pozastavení simulace, tak po jejím opětovném spuštění se bude pokračovat přesně tam, kde se předtím skončilo. Zároveň je tu tlačítko Step, které slouží ke krokování simulace. Krokování probíhá po již zmíněných krocích, jedná se v podstatě o stejný proces jako při animaci, jediným rozdílem je, že v tomto případě si přechod mezi jednotlivými kroky simulace ovládá sám uživatel. Jinak je však průběh úplně totožný jako při animaci. Zároveň je zde k dispozici možnost přechodu mezi krokováním a animací. Pokud je spuštěna animace a uživatel klikne na tlačítko Step je animace pozastavena a od aktuálního kroku může uživatel simulaci dále krokovat ručně. Stejně tak je možný i opačný přechod od krokování k animaci, opět to funguje tím způsobem, že je animace zapnuta od aktuálního kroku. Posledním tlačítkem pro ovládání animace je pak tlačítko Reset. Toto tlačítko ukončuje celou simulaci, nastavuje veškeré pomocné atributy do výchozího stavu a vrací tak uživatele na úplný začátek před spuštění simulace.

V pravé části okna, kde se nachází bloky pro registry a pro datovou paměť je pak uživateli dovoleno vyplnit před začátkem animace hodnoty do registrů, s těmito hodnotami se pak bude v rámci simulace pracovat. V průběhu simulace je pak do těchto bloků postupně zapisováno.

8.3 Základní pravidla

Tato část ve stručnosti shrnuje základní pravidla pro práci s aplikací. Veškerá pravidla už byla zmíněna v popisu chování aplikace, ale pro lepší přehlednost jsou ještě znovu shrnuta v této části práce. Základním pravidlem je, že se pracuje s instrukcemi z instrukční sady RV32I, jakékoliv jiné instrukce nebudou považovány za validní. Dalším pravidlem je, že se instrukce mohou načítat pouze ze souborů typu *.asm, protože se jedná o program v assembleru, opět nebudou akceptovány jakékoliv jiné typy souborů. Stejně tak ukládání instrukcí je vždy do souboru tohoto typu. Co se týče simulace, tak základním pravidlem pro její spuštění je, že všechny zapsané instrukce musí být validní. Pokud nebudou zapsány žádné instrukce, tak lze

aplikaci spustit, ale ihned je přesměrována do nultého kroku a uživatel je informován o jejím konci. Dále platí, že lze zapisovat do registrů, ale v tomto případě aplikace uživateli povolí zapsání pouze validních hodnot. Při ovládání simulace je možné spuštění animace, její pozastavení, krokování nebo resetování.

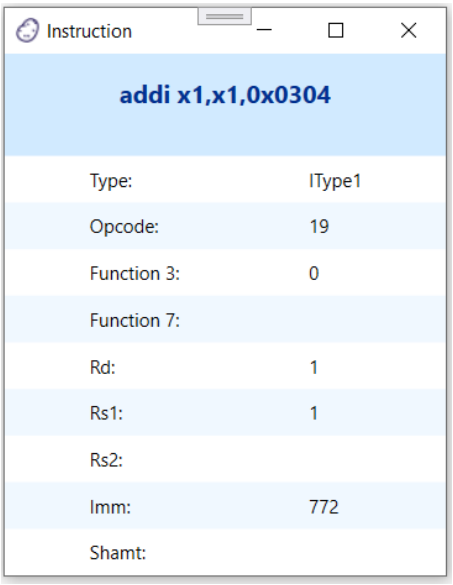
8.4 Další okna aplikace

Nyní jsou popsána a ukázána další okna, která byla použita v rámci aplikace. Konkrétně se jedná se o okno, které slouží pro ukládání instrukcí a o okno, které zobrazuje podrobné informace o instrukci.

8.4.1 Zobrazení informací o instrukci

K vyvolání tohoto okna dochází při kliknutí na informační tlačítko, které se nachází vedle každé instrukce. Po kliknutí na toto tlačítko je hodnota z textového pole pro instrukci předána do konstruktoru tohoto okna. Pomocí této hodnoty je zjištěno, zda se jedná či nejedná o správný formát instrukce.

V případě, že instrukce není ve správném formátu, dojde k zobrazení okna, přičemž je nahoře červeným textem napsána informace, že se jedná o instrukci ve špatném formátu. Pokud je instrukce správného formátu, tak jsou na základě typu instrukce vyplněny jednotlivé informace o instrukci jako je kategorie instrukce, její opcode a hodnoty registrů rd, rs1, popřípadě rs2. Pro každý typ instrukce jsou vždy vyplněny příslušné hodnoty. Zároveň je nahoře v okně zobrazena i samotná instrukce, aby uživatel přesně věděl, k jaké instrukci zobrazované hodnoty patří.

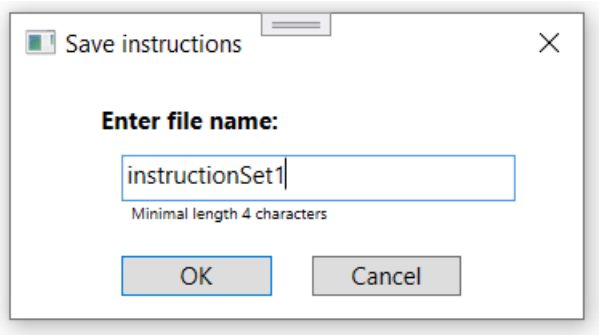


Obrázek 22 - Okno s podrobnostmi o instrukci

8.4.2 Ukládání instrukcí

Toto okno je vyvoláno při kliknutí na tlačítko Save, které se nachází v horním menu. Okno obsahuje pouze pole pro zadání názvu souboru a dvě tlačítka, tlačítko pro potvrzení a tlačítko pro zrušení ukládání. Tlačítko pro potvrzení ukládání je zpřístupněné pouze v případě, že je zadaný název souboru delší než čtyři znaky. Informace o této podmínce je v okně také zobrazena. Po stisknutí tlačítka Ok aplikace provede uložení instrukcí, které jsou zapsány v poli s instrukcemi do souboru typu *.asm.

Všechny tyto soubory se automaticky ukládají do adresáře workspace. Přičemž v případě, že se uživatel pokusí uložit soubor s instrukcemi, který by měl stejné jméno jako některý již existující soubor v tomto adresáři, tak bude o této skutečnosti informován pomocí chybové hlášky ve vyskakovacím okně, kde bude mít na výběr buď již existující soubor přepsat, nebo zvolit pro ukládaný soubor jiné jméno.

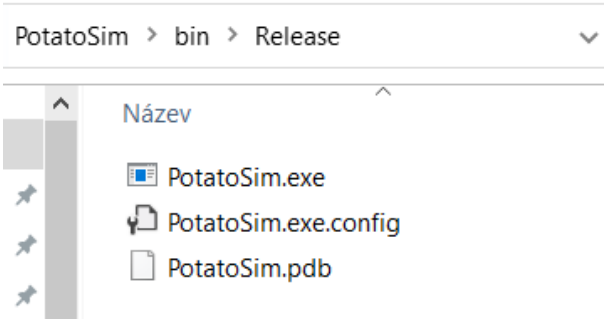


Obrázek 23 - Okno pro ukládání instrukcí

9 POPIS INSTALACE A UKÁZKOVÉ ÚLOHY

Tato část práce popisuje instalaci a zprovoznění vyvíjené aplikace a zároveň jsou zde uvedeny ukázkové úlohy. Prvním krokem instalace je stažení všech potřebných souborů, tedy samotné vyvíjené aplikace. Samotnou aplikaci PotatoSim není potřeba nijak instalovat, stačí ji pouze stáhnout a po stažení ji umístit do libovolného adresáře.

Velmi důležité ale je zachovat stejnou adresářovou strukturu jakou má aplikace po jejím stažení. Jedná se především o adresáře bin, workspace, images, structs, instructions a potato. Pokud by tyto adresáře byly odstraněny nebo přesunuty, aplikace by nefungovala správně. Spouštěcí soubor aplikace je pak umístěn ve složce bin, konkrétně v podsložce release. Ze zde umístěného exe souboru si lze vytvořit zástupce, kterého je pak možné umístit do libovolné složky. To je, co se týče samotného stažení a zprovoznění aplikace, vše.



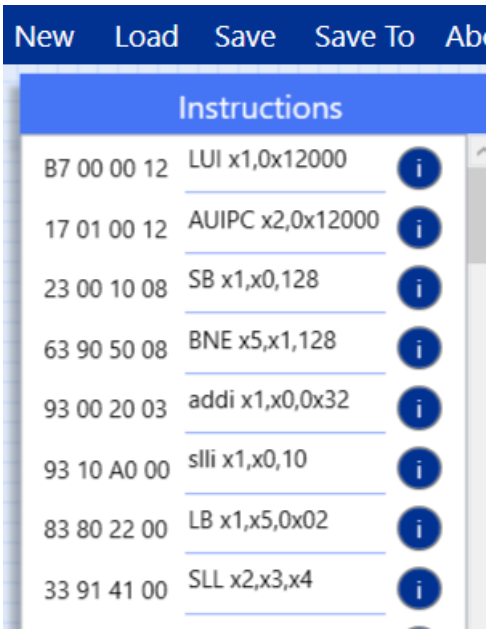
Obrázek 24 - Umístění exe souboru

Co se týče ukázkových souborů s instrukcemi, tak se v adresářové struktuře nachází složka workspace, do této složky jsou ukládány veškeré soubory s instrukcemi, jak již bylo popsáno výše v této práci. Zároveň se v této složce nachází i tři ukázkové soubory instrukcí, které si je možné v aplikaci otevřít a otestovat si pomocí nich simulaci bez nutnosti znalosti zápisu instrukcí.

Tyto tři ukázky se nachází v již zmíněné složce, v podsložkách s názvy ukazka01, ukazka02, ukazka03. Každá z těchto složek pak obsahuje specifický seznam instrukcí, které je možné si nahrát do aplikace. Veškeré tyto soubory jsou typu asm (assembler source). Ukázky by celkově měly pokrýt celou instrukční sadu, aby si uživatel mohl vyzkoušet, jak funguje simulátor pro jednotlivé typy instrukcí.

Nahrání instrukcí do aplikace je pak možné po spuštění aplikace kliknutím na tlačítko Load. Poté už uživateli pouze stačí najít adresář workspace a vybrat si jeden ze zmíněných adresářů. Poté ve vybraném adresáři zvolí soubor asm a v aplikaci se do pole s instrukcemi nahrají příslušné instrukce z tohoto souboru. Všechny připravené ukázky by měly obsahovat pouze

validní instrukce, tudíž po nahrání instrukcí do aplikace by již nic nemělo bránit spuštění samotné simulace.



Obrázek 25 - Ukázková sada instrukcí

ZÁVĚR

Cílem DP bylo vytvoření funkčního simulátoru pro RISC-V procesor Potato. Vytvořená aplikace byla nazvána PotatoSim a je navržena, aby uživatelům posloužila jako pomůcka pro obeznámení se s funkčností jádra procesoru.

Tento procesor je napsán v jazyce VHDL, a proto bylo nutné navrhnout a vytvořit postupy převodu VHDL kódu do jazyka C#. Tato část práce byla nejsložitější, neboť bylo zapotřebí seznámit se se syntaxí jazyka VHDL. Procesor Potato je naštěstí rozdělen na několik logicky členěných modulů, proto toto rozdělení bylo aplikováno i v jazyce C#. Simulátor nyní simuluje jádro tohoto procesoru.

Aplikace plně splňuje veškeré stanovené cíle, zároveň je také graficky povedená a působí hezkým a moderním dojmem. Celý simulátor je graficky znázorněn v jednom hlavním okně, které je rozděleno do několika logicky uspořádaných částí, díky tomu aplikace působí přehledně a přívětivě. Aplikace uživateli dovoluje zadávat instrukce, vyplňovat registry, a především pak spouštět samotnou simulaci. Během simulace pak uživatel může pozorovat, jak se instrukce načítají, ukládají do paměti, dekódují a poté se provádějí příslušné operace nad těmito instrukcemi a výsledky těchto operací se zapisují do registrů. Uživateli je také umožněno vyzkoušet si zapsání vlastních instrukcí nebo jejich načtení ze souboru. Jedním z cílů práce bylo vytvoření několika vzorových ukázek instrukcí, určených především pro uživatele, kteří nemají s psaním instrukcí žádné předchozí zkušenosti. Tyto ukázky jsou tedy součástí výsledné práce. Dále pak uživatel může v aplikaci spustit simulaci, a to buď jako celkovou animaci, nebo si může simulaci procházet postupně po jednotlivých krocích.

V budoucnu je poté jedním z cílů vytvořit rozšíření jádra procesoru například o sběrnici Wishbone, popřípadě o další externí komponenty, například UART. Diplomová práce zároveň nastínila možnost způsobu převodu VHDL jazyka do jazyka C#. Položila tak základy pro převod existujícího VHDL kódu a mohla by být základním prvkem pro zautomatizování těchto převodů, ať již z VHDL do C# nebo i z C# do VHDL. Zvláště druhá možnost by mohla být velmi zajímavou variantou, neboť MATLAB a jazyky C/C++, Python již umožňují podobné konverze.

POUŽITÁ LITERATURA

[1] STALLINGS, William. *Computer Organization and Architecture: Designing for Performance*. 10. Pearson, 2015. ISBN 978-0-13-410161-3.

[2] TANENBAUM, Andrew a Herbert BOS. *Moder operating systems*. 4. Pearson, 2015. ISBN 978-0-13-359162-0.

[3] PATTERSON, David a John HENNESSY. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. 5. Morgan Kaufmann, 2013. ISBN 978-0124077263.

[4] BURTON, D. P. a A. L. DEXTER. *Microprocessor Systems Handbook*. 1. Irsko: Analog Devices, 1977. ISBN 0-916550-04-4.

[5] *Architektura procesorů* [online]. 2016 [cit. 2023-04-04]. Dostupné z: https://docplayer.cz/6689460-Architektura-procesoru.html

[6] *RISC vs. CISC Architecture: Which is Better?* [online]. 2019 [cit. 2023-04-04]. Dostupné z: https://www.per-international.com/news-and-insights/risc-vs-cisc-architecture-which-is-better

[7] *Potato Processor* [online]. 2018 [cit. 2023-04-04]. Dostupné z: https://opencores.org/projects/potato

[8] WATERMAN, Andrew a Krste ASANOVIĆ. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA* [online]. Document Version 20191213. San Francisco: RISC-V Foundation, 13.12.2019 [cit. 2023-04-14]. Dostupné z: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[9] HO, Steven. *Great Ideas in Computer Architecture: RISC-V Instruction Formats* [online]. Berkeley: University of California, 2018 [cit. 2023-04-14]. Dostupné z: https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture7.pdf

[10] ENGHEIM, Erik. *RISC-V Instruction-Set Cheatsheet* [online]. 2022 [cit. 2023-04-14]. Dostupné z: https://itnext.io/risc-v-instruction-set-cheatsheet-70961b4bbe8

[11] AGGARWAL, Anshul. *Introduction to Visual Studio* [online]. 2022 [cit. 2023-04-25]. Dostupné z: https://www.geeksforgeeks.org/introduction-to-visual-studio/

[12] *Visual Studio 2022* [online]. Microsoft, 2022 [cit. 2023-04-25]. Dostupné z: https://visualstudio.microsoft.com/cs/vs/

[13] C# – Overview. *Tutorialspoint* [online]. 2017 [cit. 2023-04-25]. Dostupné z: https://www.tutorialspoint.com/csharp/csharp_overview.htm#

[14] GANESH, Gnana Arun. C# - Overview. *C# Corner* [online]. 2022 [cit. 2023-04-25]. Dostupné z: https://www.c-sharpcorner.com/article/C-Sharp-and-its-features/

[15] WPF - Overview. *Tutorialspoint* [online]. 2018 [cit. 2023-04-25]. Dostupné z: https://www.tutorialspoint.com/wpf/wpf_overview.htm

[16] ČÁPKA, David. Úvod do WPF. *ITnetwork* [online]. 2017 [cit. 2023-04-25]. Dostupné z: https://www.itnetwork.cz/csharp/wpf/c-sharp-tutorial-wpf-uvod-a-prvni-formularova-aplikace

[17] PEDAMKAR, Priya. Winforms vs WPF. *EDUCBA* [online]. 2019 [cit. 2023-04-25]. Dostupné z: https://www.educba.com/winforms-vs-wpf/

[18] PINKER, Jiří a Martin POUPA. *Číslicové systémy a jazyk VHDL*. 1. Praha: BEN – technická literatura, 2006. ISBN 80-7300-198-5.

[19] CADENCE PCB SOLUTIONS. Hardware Description Languages: VHDL vs Verilog, and Their Functional Uses. *CADENCE* [online]. 2020 [cit. 2023-04-28]. Dostupné z: https://www.educba.com/winforms-vs-wpf/

[20] VHDL Tutorial. *Javapoint* [online]. 2021 [cit. 2023-04-28]. Dostupné z: https://www.javatpoint.com/vhdl

[21] Basic Elements of VHDL. *FPGAKey* [online]. 2020 [cit. 2023-04-28]. Dostupné z: https://www.fpgakey.com/tutorial/chapter72

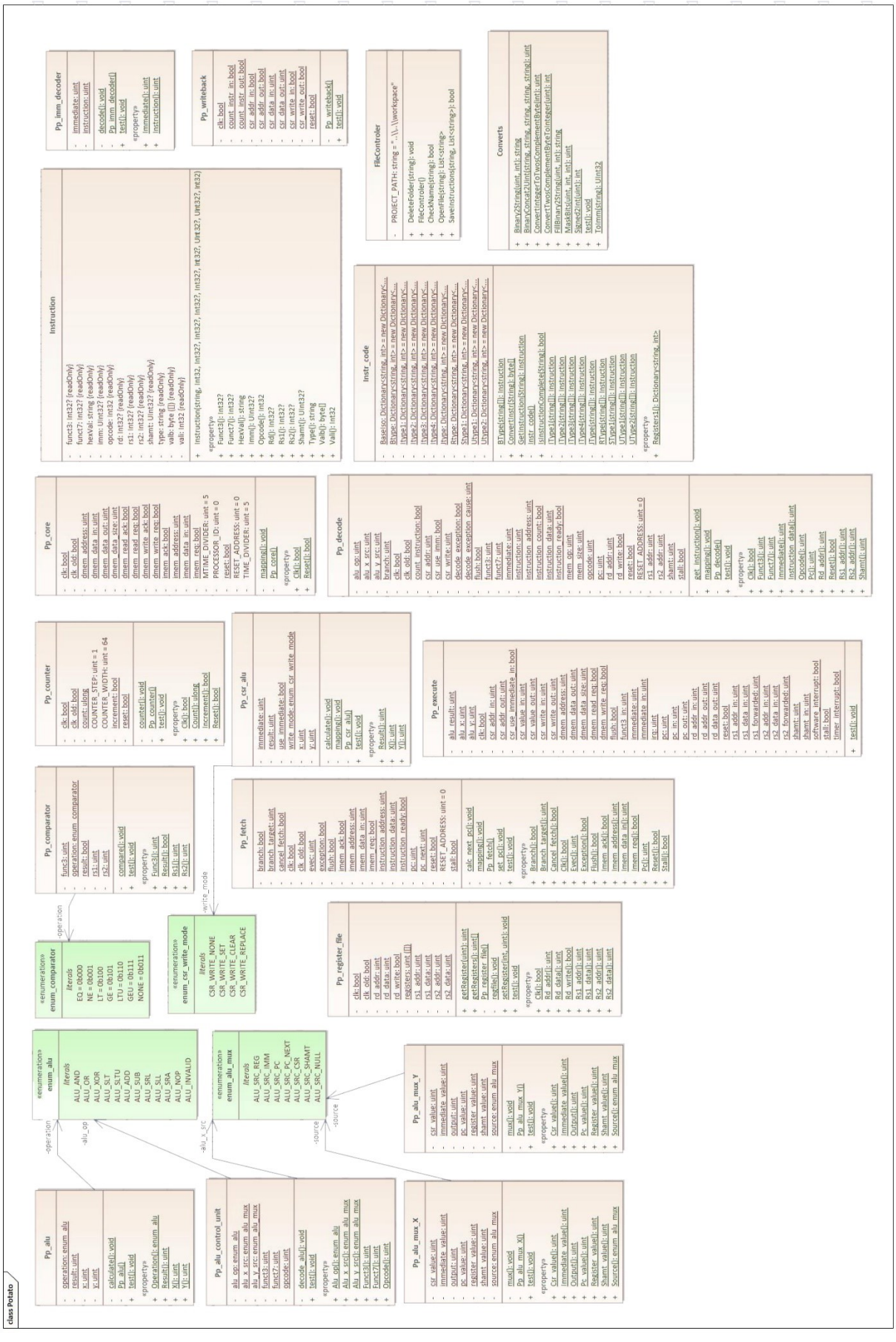
[22] VHDL Syntax Web-Course. *Surf-VHDL* [online]. 2018 [cit. 2023-04-28]. Dostupné z: https://surf-vhdl.com/vhdl-syntax-web-course-surf-vhdl

[23] PREW, Robert. Battle Over the FPGA: VHDL vs Verilog! Who is the True Champ?. *Digilent* [online]. 2023 [cit. 2023-04-28]. Dostupné z: https://digilent.com/blog/battle-over-the-fpga-vhdl-vs-verilog-who-is-the-true-champ/

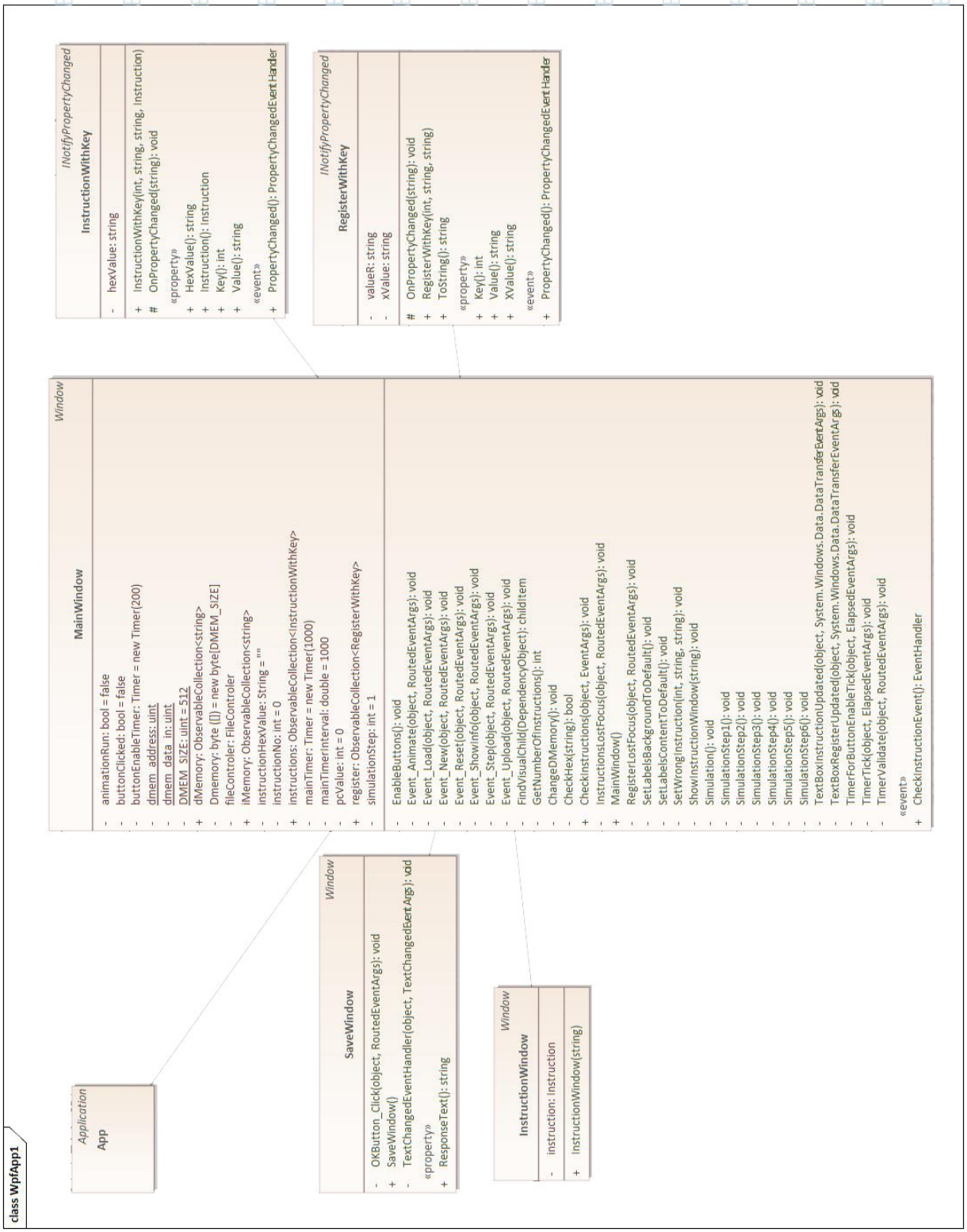
[24] BRISC-V Toolbox. *BRISC-V* [online]. 2020 [cit. 2023-04-29]. Dostupné z: https://ascslab.org/research/briscv/index.html

PŘÍLOHY

PŘÍLOHA A – DIAGRAM TŘÍD POPISUJÍCÍCH PROCESOR



PŘÍLOHA B – DIAGRAM GRAFICKÝCH TŘÍD



PŘÍLOHA C – ZDROJOVÉ KÓDY A DOKUMENTY

K práci jsou přiloženy veškeré zdrojové kódy aplikace.