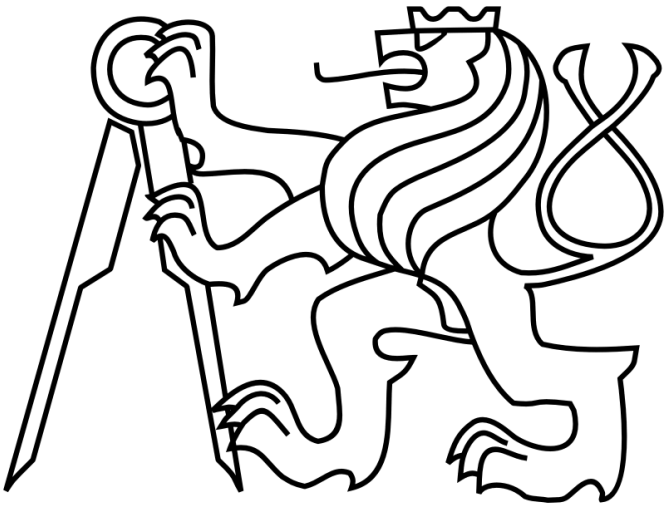
České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky



BAKALÁŘSKÁ PRÁCE

Simulátor procesoru podorujícího MIPS ISA

Autor: Adam Svoboda

Vedoucí práce: Ing. Michal Štepanovský, Ph.D.

Praha, 2016

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 27.5.2016 Adam Svoboda

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Adam Svoboda**

Studijní program: Kybernetika a robotika
Obor: Systémy a řízení

Název tématu: **Simulátor procesoru podorujíciho MIPS ISA**

Pokyny pro vypracování:

1. Identifikujte výhody a nevýhody volně dostupných simulátorů procesoru podporujícího instrukční sadu MIPS.
2. Navrhněte ideové schéma simulátoru s důrazem na oddělení simulace a vizualizace. Zaměřte se i na práci s instrukční a datovou cache.
3. Implementujte simulátor navržený v předchozím bodě ve vybraném programovacím jazyce.
4. Vzhled a funkcionalitu simulátoru přizpůsobte potřebám předmětu A0B36APO Architektury počítačů.
5. Vypracujte dokumentaci a manuál k navrženému simulátoru.
6. Jednotlivé požadavky a postupy konzultujte s vedoucím práce.

Seznam odborné literatury:

[1] Hennesy, J. L., Patterson, D. A.: Computer Architecture: A Quantitative Approach, Third Edition, San Francisco, Morgan Kaufmann Publishers, Inc., 2002
[2] Hyvl, D.: Simulátor MIPS procesoru, Bakalářská práce, Katedra řídicí techniky, FEL ČVUT v Praze, 2016.

Vedoucí: Ing. Michal Štepanovský, Ph.D.

Platnost zadání: do konce letního semestru 2016/2017

L.S.

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 24. 2. 2016

Poděkování

Chtěl bych poděkovat Ing. Michalu Štepanovskému, Ph.D. za odborné vedení, za pomoc a rady při zpracování této práce.

Název práce: Simulátor procesoru podorujíciho MIPS ISA

Autor: Adam Svoboda

Katedra (ústav): Katedra řídicí techniky

Vedoucí bakalářské práce: Ing. Michal Štepanovský, Ph.D.

e-mail vedoucího: stepami9@fel.cvut.cz

Abstrakt Hlavním cílem této práce je návrh a implementace simulátoru MIPS procesoru. Vedlejším cílem je zhodnocení a porovnání současných simulátorů MIPS procesoru, za účelem identifikace jejich výhod a nevýhod a zapracování těchto poznatků do simulátoru. Tento simulátor může být využit pro pedagogické účely v předmětu A0B36APO, jehož potřebám je přizpůsobeno grafické uživatelské rozhraní. Bude sloužit jako názorná pomůcka při studiu, díky které budou moci studenti lépe pochopit principy chodu procesoru.

Klíčová slova: MIPS, assembler, pipeline, grafické uživatelské rozhraní

Title: MIPS ISA processor simulator

Author: Adam Svoboda

Department: Department of control engineering

Supervisor: Ing. Michal Štepanovský, Ph.D.

Supervisor's e-mail address: stepami9@fel.cvut.cz

Abstract The main objective of this thesis is to design and implement MIPS processor simulator. The secondary objective is to evaluate and compare the existing MIPS processor simulators, in order to identify their advantages and disadvantages and involve this knowledge in the project. This simulator can be used for educational purposes of the course A0B36APO. Its graphical user interface is adapted to the needs of the course. It will serve as a visual utility during the studies. It will help students understand the principles of the processor better.

Keywords: MIPS, assembler, pipeline, graphical user interface

Obsah

1 Kapitola 1

Úvod

Procesor MIPS je v principu jedním z nejjednodušších a nejnázornějších procesorů na světě. Je to procesor typu RISC, což znamená, že jeho instrukční sada obsahuje malý počet instrukcí, které představují jednoduché operace. Zkratka MIPS pochází z anglického názvu Microprocessor without Interlocked Pipeline Stages, což v překladu znamená „procesor bez pozastavování (uzamykání) vnitřích částí pipeline“. Pipeline představuje složitější způsob vykonávání instrukcí, při kterém se vykonává více instrukcí najednou, přičemž každá tato instrukce využívá v každém okamžiku jinou část procesoru. Pokud se o správné rozvrhování instrukcí do procesoru musí starat překladač nebo programátor (tak aby procesor vykonal program dle sémantiky programu), jedná se o tzv. non-interlocked pipeline. V tomto případě procesor nedokáže vykonat správně jisté sekvence na sobě závislých instrukcí. Pokud procesor sám detekuje tyto hazardy, které vnitřně řeší pozastavením frond-end částí pipeline - zavádění tzv. bubbles, jedná se o interlocked pipeline. Dnešní MIPS procesory běžně využívají interlocked pipeline, a proto původní akronym přestává z tohoto pohledu platit a spíše tedy označuje pouze použitou instrukční sadu.

Procesory MIPS našly využití zejména v počítačích společnosti SGI, kde se pomocí nich podařilo dosáhnout vysokého výkonu vykreslování 3D scén. Dále se uplatnily v herních konzolích jako jsou PlayStation 2, PlayStation Portable a Nintendo 64[8].

Pro studijní účely byl zvolen pro jeho přehlednou a názornou funkčnost, tedy stejnou délku všech instrukcí a vykonání každé instrukce v jednom cyklu.

V první části této práce se zaměříme na zhodnocení a porovnání současných simulátorů MIPS ISA, jako jsou MARS, SPIM, MipsIt a další. Shrneme jejich výhody a nevýhody a podle těchto poznatků se pokusíme navrhnout a realizovat vlastní simulátor, který bude později použit v předmětu A0B36APO[3]. Simulátor by měl být schopen načíst soubor se zapsaným programem ve formátu assembleru a nasimulovat běh tohoto programu. Pro výukové účely je nutné vizualizovat obsah všech registrů, vnější paměti, sledovat plnění instrukční a datové cache a průchod jednotlivých instrukcí procesorem. Instrukční sada není pro tyto účely potřeba celá, bude stačit pouze omezená množina instrukcí.

Kapitola 2

2 Popis procesoru

V této kapitole se pokusím popsat princip fungování procesoru. Nejprve popíši instrukční sadu, kterou budeme v simulátoru implementovat, a pak vysvětlím funkce jednotlivých komponent, ze kterých se procesor skládá.

2.1 Instrukční set

Instrukce procesoru MIPS mají všechny shodnou délku 32b. Dělíme je do tří typů: R, I, J. Všechny tyto typy mají společný význam pouze pro prvních šest bitů, které označují tzv. op-code, pomocí kterého se dekóduje konkrétní instrukce. Význam ostatních bitů závisí na typu instrukce (Tabulky 2.1, 2.3 a 2.5)[4].

Pro využití v předmětu A0B36APO bude stačit redukovaný instrukční set: add, and, or, sll, slt, sub, lui, ori, addi, lw, sw, beq, bne, la, nop

2.1.1 Instrukce typu R

31 - 26	25 - 21	20 - 16	15 - 10	10 - 6	5 - 0
opcode	rs	rt	rd	shamt	funct

Tabulka 2.1: Formát instrukce typu R

Instrukce typu R jsou instrukce aritmetických a logických operací, které pracují s hodnotami v registrech. Vstupem jsou hodnoty ze dvou registrů, po provedení dané operace se výsledek uloží do třetího registru. Zdrojové registry jsou rs a rt, výsledný registr je rd. Bity 10 - 6 (shamt [1]) slouží k uložení informace o případném bitovém posunu. Všechny tyto instrukce mají opcode roven nule, liší se hodnotou funkčního znaku (funct).

[1] „shift amount“

Instrukce	Funct	Popis funkce
Add	100000	rd = rs + rt
Sub	100010	rd = rs - rt
And	100100	rd = rs & rt
Or	100101	rd = rs | rt
Slt	101010	rd = 1 pokud rs < rt, jinak rd = 0
Sll	000000	rd = (rt << shamt)

Tabulka 2.2: Přehled instrukcí typu R

2.1.2 Instrukce typu I

31 - 26	25 - 21	20 - 16	15 - 0
opcode	rs	rt	imm

Tabulka 2.3: Formát instrukce typu I

Instrukce typu I jsou instrukce aritmetických a logických operací, které pracují s jednou hodnotou z registru a druhou hodnotu nese sama instrukce jako 16-bitový přímý operand uložený ve dvojkovém doplňku. Výsledek operace uloží do druhého registru. Zdrojový registr je rs, výsledný registr je rt.

Mezi instrukce typu I řadíme i instrukce, které manipulují s datovou pamětí: lw a sw. Instrukce lw načítá data z adresy, která se vypočítá jako součet hodnoty v registru rs a přímého operandu. Data jsou uložena do registru rt. Instrukce sw pracuje v podstatě stejně, avšak slouží pro uložení dat.

Dále mezi instrukce řadíme skokové instrukce beq a bne, které v případě naplnění podmínky provedou skok o určitý počet instrukcí uložený v hodnotě imm.

Instrukce	Funct	Popis funkce
Addi	001000	rt = rs + imm
Lui	001111	rt = (imm << 16)
Ori	001101	rt = rs | imm
Lw	100011	rt = MEM[rs + imm]
Sw	101011	MEM[rs + imm] = rt
Beq	000100	PC += (imm <<) pokud rs == rt
Bne	000101	PC += (imm <<) pokud rs != rt

Tabulka 2.4: Přehled instrukcí typu I

2.1.3 Instrukce typu J

31 - 26	25 - 0
opcode	target

Tabulka 2.5: Formát instrukce typu J

Instrukce typu J realizují skok v instrukční paměti na specifickou adresu. Tento simulátor žádnou takovouto instrukci nepodporuje, aby se zachovala jednoduchost procesoru. Skoky lze přesto realizovat pomocí instrukcí bne a beq, které svým formátem patří do instrukcí typu I.

2.1.4 Pseudoinstrukce

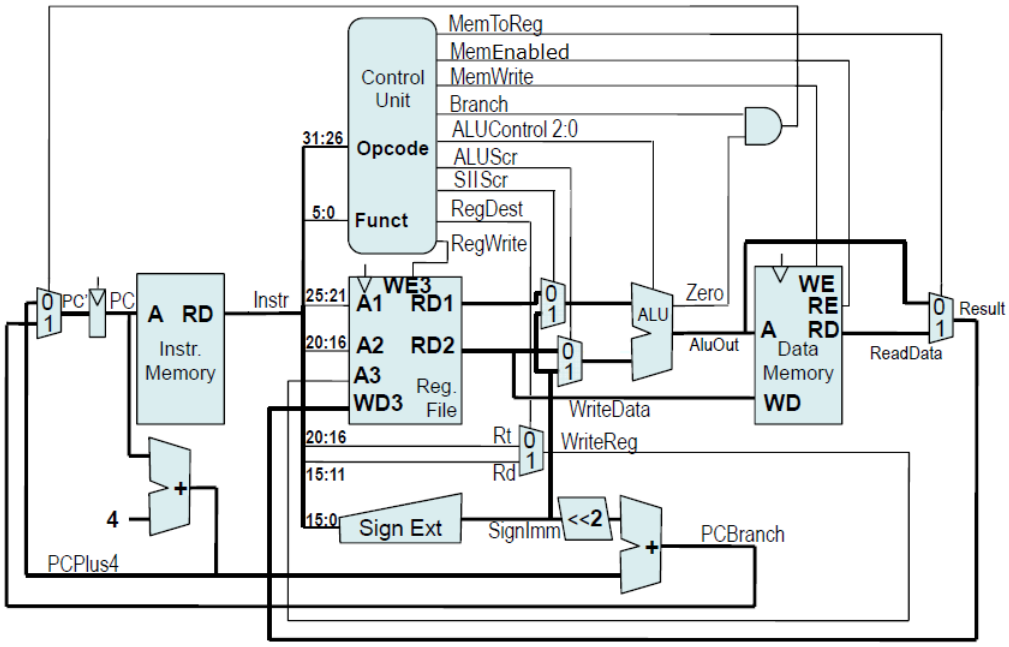
Pseudoinstrukce jsou instrukce, které nelze vykonat v jednom cyklu procesoru. V našem simulátoru je podporována pouze jedna pseudoinstrukce: la, která slouží k načtení konkrétní adresy dat uložených překladačem. Instrukce se rozloží na instrukci lui a ori, přičemž lui načte horních 16 bitů adresy a ori dolních 16 bitů.

la rt, address	⇒	lui rt, high(address)
		ori rt, low(address)

2.2 Komponenty

Jednotlivé komponenty, ze kterých se procesor skládá, jsou vidět ve schematickém zapojení na obrázku 2.1.

S náběžnou hranou hodin se překlápí program counter (PC) a na sběrnici je vystavena adresa následující instrukce. Komponenta instrukční paměti načte strojový kód instrukce, který postupuje do instrukčního registru, kde se interpretuje jeho obsah. Řídící jednotka přijímá operační a funkční kód, soubor registrů přijímá adresy zdrojových registrů a přímý operand postupuje přes komponentu znaménkového rozšíření. Řídící jednotka nastaví multiplexory podle popisu dané instrukce, data z registrů a přímý operand postupují do aritmeticko-logické jednotky, kde se provede zvolená operace. Dále je vyhodnocen případný podmíněný skok a mohou být vyčtena data z paměti. Výsledek operace je pak s příchodem nové náběžné hrany hodin zapsán do registrů nebo paměti a program counter se opět překlápí na další instrukci[1].



Obrázek 2.1: Schéma jednoduchého jednocyklového procesoru[3]

2.2.1 Registry

Registr je úložiště dat přímo v procesoru. Přístup k datům uložených v registrech je mnohem rychlejší než příštup k datům, která jsou uložena v paměti počítače, proto se v nich ukládají pracovní hodnoty. Procesor MIPS má 32 registrů, které jsou 32-bitové. Registr 0 je nastaven tak, aby vždy obsahoval hodnotu 0. Ostatní pracovní registry nejsou nijak omezené. Dalším specifickým registrem je tzv. program counter, ve kterém je uložena adresa instrukce, která se vykonává. Strojový kód vykonávané instrukce je uložen v instrukčním registru, který zajišťuje čtení jednotlivých částí strojového kódu instrukce (Tabulky 2.1, 2.3, 2.5).

Komponenta, která zajišťuje přístup k pracovním registrům má dva 5-bitové vstupy pro výběr registrů ke čtení, jeden 5-bitový vstup pro výběr registru k zápisu, jeden 32-bitový vstup pro zapisovanou hodnotu, dva 32-bitové výstupy pro čtení ze zvolených registrů a jeden řídicí vstup, který ovlivňuje zda proběhne zápis. Zápis probíhá vždy s náběžnou hranou hodinového impulzu, čtení probíhá nezávisle na hodinovém impulzu.

2.2.2 Aritmeticko-logická jednotka

Aritmeticko-logická jednotka[2] je jedna ze stěžejních komponent procesoru. Vykonává aritmetické a logické oprace, čímž tvoří podstatu funkce celého procesoru. Má dva 32-bitové vstupy

[2] zkratkou ALU

pro operandy, jeden 32-bitový výstup pro výsledek operace, jeden 1-bitový příznak, který indikuje, zda je na výstupu ALU nula[3], a jeden 3-bitový vstup pro řídicí signál, kterým dává řídicí jednotka aritmeticko-logické jednotce informaci, kterou operaci má provést. Jednotlivé operace (Tabulka 2.6) přímo souvisí s funkčním kódem instrukcí typu R (Tabulka 2.2).

Operace	Řídicí signál	Popis
And	000	bitový součin vstupních operandů
Or	001	bitový součet vstupních operandů
Add	010	aritmetický součet vstupních operandů
Sll	011	logický bitový posuv prvního operandu vlevo o hodnotu druhého operandu
Lui	100	logický bitový posuv vlevo o 16 bitů
Sub	101	aritmetický rozdíl vstupních operandů
Slt	110	porovnání operandů, pokud je první menší než druhý, výsledek je 1, jinak 0

Tabulka 2.6: Popis aritmetický a logických operací

2.2.3 Řídící jednotka

Řídící jednotka je jakýmsi mozkem procesoru. Dekóduje vykonávané instrukce pomocí jejich operačního a funkčního kódu (tabulky 2.2 a 2.4) a nastavuje řídicí signály, kterými ovlivňuje interpretaci strojového kódu instrukce, jednotlivé multiplexory a další komponenty tak, aby každá instrukce prošla procesorem správně. Jednotlivé řídicí signály jsou uvedeny v tabulce 2.7 a nastavení jejich hodnot se řídí tabulkou 2.8. Další řídicí signály souvisí s organizováním pipeline a budou vysvětleny později.

Název	Význam
MemToReg	vybírá, zda se do registrů bude zapisovat hodnota z paměti nebo výsledek ALU
MemWrite	určuje, zda se v následujícím hodinovém cyklu bude zapisovat do paměti
BranchBeq	detekce skokové instrukce beq
BranchBne	detekce skokové instrukce bne
ALUControl	3-bitová hodnota, nastavuje operaci ALU
SllSrc	vybírá první operand ALU
AluSrc	vybírá druhý operand ALU
RegDest	vybírá regsitr pro zápis výsledku operace
RegWrite	určuje, zda se v následujícím hodinovém cyklu bude zapisovat do registrů

Tabulka 2.7: Řídicí signály

[3] tuto hodnotu využívá řídící jednotka při vyhodnocování podmíněných skokových instrukcí

Instrukce	MR	MW	Beq	Bne	ALU	SS	AS	RD	RW
Add	0	0	0	0	010	0	0	1	1
And	0	0	0	0	000	0	0	1	1
Or	0	0	0	0	001	0	0	1	1
Sll	0	0	0	0	011	1	0	1	1
Slt	0	0	0	0	110	0	0	1	1
Sub	0	0	0	0	101	0	0	1	1
Lui	0	0	0	0	100	0	0	0	1
Addi	0	0	0	0	010	0	1	0	1
Lw	1	0	0	0	010	0	1	0	1
Sw	0	1	0	0	010	0	1	0	0
Beq	0	0	1	0	101	0	0	0	0
Bne	0	0	0	1	101	0	0	0	0
Ori	0	0	0	0	001	0	1	0	1

Tabulka 2.8: Nastavení řídích signálů

2.2.4 Multiplexor

Multiplexor je komponenta, která má více vstupů, jeden výstup a jeden řídicí vstup. Na výstup je přepnut jeden ze vstupů, vybraný pomocí řídicího signálu.

2.2.5 Znaménkové rozšíření

Instrukce typu I v sobě nesou 16-bitový přímý operand. Procesor pracuje avšak s 32-bitovými čísly, takže je potřeba tento operand znaménkově rozšířit, tedy pokud je záporný, doplnit zleva jedničkami, pokud je kladný, nulami.

2.2.6 Bitový posuv

Skokové instrukce Beq a Bne v sobě nesou informaci o kolik instrukcí se bude skákat. Jelikož jsou instrukce v paměti slovně zarovnány, je adresa sousední instrukce vždy o čtyři vyšší. Bitový posun o dvě místa vlevo v podstatě znamená násobení čtyřmi. Tato komponenta tedy získá na vstupu přímý operand skokové instrukce a na výstup dává hodnotu, kterou je potřeba pro provedení daného skoku přičíst k program counteru.

2.2.7 Logická hradla

Pro vyhodnocení skokových instrukcí potřebujeme 1-bitová logická hradla and, or a negaci. Ke skoku dojde pokud máme instrukci bne a zároveň nejsou operandy shodné, nebo pokud máme instrukci beq a zároveň jsou operandy shodné.

2.2.8 Paměť

Navržený simulátor nepodporuje virutální pamět, veškeré adresy jsou fyzické. Hlavní pamět o velikosti 4KB je rozdělena na dvě části - pamět programu (dolní 2KB) a pamět dat (horní 2KB). Pro názornost je v simulátoru tato pamět viualizována dvěma samostatnými komponentami (instrukční a datová pamět), aby bylo na první pohled zřejmé, že se jedná o výběr instrukce z paměti nebo o přístup k datům. Nicméně nevhodný programátorský přístup může zapříčinit, že program counter (PC) nabyde hodnot náležicích paměti dat, a tak může procesor nesprávně interpretovat data jako instrukce. Takové chování je v souladu s principem von Neumannovy architektury.

Instrukční paměť

Do instrukční paměti jsou překladačem assembleru nahrány instrukce programu v pořadí, ve kterém budou vykonávány.

Komponenta instrukční paměti vybírá strojový kód instrukce, která bude vykonávána, na základě adresy, kterou získá z program counteru. Přístup do paměti není přímý, ale obvykle probíhá přes instrukční cache. Pamět je slovně zarovnána, tedy adresa následující instrukce je vždy o čtyři vyšší.

Datová paměť

Datová pamět slouží k uložení dat, na která už není místo v registrech. Může být naplněna překladačem assembleru.

Komponenta datové paměti funguje podobně, jako komponenta instrukční paměti, tedy na základě vstupní adresy dodává na výstup příslušná data. Do datové paměti lze však data i ukládat, což je řízeno řídicím signálem MemWrite. Komponenta má tedy jak vstup pro adresu, tak vstup pro ukládaná data. Zápis do paměti probíhá podobně jako u registrů s náběžnou hranou hodinového cyklu, čtení pak kdykoli během hodinového cyklu. K paměti se opět přistupuje skrze cache a i tato pamět je slovně zarovnána.

2.2.9 Hazard unit

Hazard unit je komponenta, která má význam pouze u procesoru vykonávajích instrukce zřetězeně. Při zřetězeném vykonávání totiž může docházet k různým hazardům. V principu rozdělujeme hazardy do dvou skupin: datové a řídicí.

Datový hazard vznikne, pokud instrukci, která upravuje hodnotu v registru X, následuje instrukce, která čte hodnotu z regsitru X. Hodnota, kterou druhá instrukce čte, není platná, protože zápis nové hodnoty se děje později než toto čtení.

Řídicí hazard souvisí s podmíněnými skokovými instrukcemi. Při náčítání následujících instrukcí není známo, zda se bude provádět skok. To znamená, že pokud se skok provede, instrukce načtené po dané skokové instrukci je pořeba z procesoru vymazat.

Toto se dá vyřešit vhodnou skladbou instrukcí za sebou, ale to nelze vždy provést. Proto je v procesoru hazard unit, která dokáže tyto hazardy identifikovat a vyřešit. Datový hazard se řeší přeposláním hodnoty na místo, kde je potřeba, pokud jde o data, která již v procesoru jsou, nebo pozastavením, pokud jde o čtení dat z paměti. Toto je fyzicky zvládnuto pomocí čtyř multiplexorů a oddělovacích registrů. Řídicí hazard se nijak efektivně řešit nedá, procesorovou pipeline je třeba vyprázdnit.

Kapitola 3

3 Implementace procesoru

Pro implementaci procesoru jsem zvolil jazyk Java z důvodu snadné přenositelnosti na různé platformy. V rámci objektového návrhu jsem se nechal inspirovat návrhovým vzorem MVC[1] a celý projekt rozdělil do tří balíčků[2]:

- *mips.procesor*, která zajišťuje výpočetní část simulátoru
- *mips.viewer*, která obsahuje grafické uživatelské rozhraní
- *mips.controller*, která obě části propojuje.

Balíček *mips.viewer*, ve kterém jsou implementovány ovládací a zobrazovací prvky simulátoru, si probereme samostatně v kapitole 5, protože tyto funkce budou přímo záviset na závěrech z kapitoly 4. Zbývající dva balíčky si popíšeme zde.

3.1 Mips.processor

Tento balíček tvoří jakési jádro simulátoru. Obsahuje několik veřejných a několik neveřejných tříd popsaných v souborech v tabulce 3.1:

Soubor	Popis
Components.java	definuje třídy komponent, tyto třídy nejsou veřejné
Instructions.java	definuje třidu instrukcí a dekodér instrukcí
Parser.java	definuje objekt, který načítá soubor v assembleru
Processor.java	abstraktní třída definující základní atributy a funkce procesoru
SimpleProcessor.java	třída, která implementuje nezřetězený procesor, dědí z Processor
PipelineProcessor.java	třída, která implementuje zřetězený procesor, dědí z Processor

Tabulka 3.1: Source package: mips.processor

[1] model-view-controller
[2] source packages

3.1.1 Components.java

Výpočetní část procesoru je vystavěna podle obrázku 3.1 a komponenty zastávají své funkce tak, jak jsme si je popsali v kapitole 2. Objekty jednotlivých komponent jsou mezi sebou propojeny sběrnicemi, které jsou implementovány také pomocí objektů[2].

Třída symbolizující sběrnici se jmenuje *Wire* a uchovává hodnotu *data (int)* a *name (String)*. Nabízí dvě jednoduché metody *void setData(int data)* a *int getData()*, pomocí kterých probíhá výměna dat po sběrnici.

Všechny komponenty pak mají pro každý vstup nebo výstup jeden objekt typu *Wire*, kterým jsou vzájemně propojeny. Pro implementaci funkce jednotlivých komponent jsem vytvořil dvě rozhraní[3]:

- *Updateable*, definující metodu *void update()*, která charakterizuje funkci komponenty v hodinovém cyklu
- *Preupdateable*, definující metodu *void preupdate()*, která charakterizuje fuknci komponenty v náběžné hraně hodinového cyklu.

Většina komponent implementuje pouze jedno rozhraní (vetšinou *Updateable*), některé komponenty ale implementují obě rozhraní najednou. Pokud má tedy komponenta např. dva vstupy a jeden výstup, po zavolání metody *update()* si pomocí funkcí *getData()* na svých vstupních sběrnicích přečte vstupní data, provede svou charakteristickou funkci, kterou v procesoru zastává, a zapíše výsledek na výstupní sběrnici pomocí metody *setData()*.

Register

Tato třída je velmi jednoduchá. Implementuje rozhraní *Preupdateable* a obsahuje pouze jednu výstupní a jednu vstupní sběrnici. Po zavolání metody *preupdate()* jednoduše přepíše vstupní hodnotu na výstup.

Tento objekt tedy funguje jako jakýsi oddělovač hodnot. V jednocyklovém procesoru ho využijeme pouze jako *program counter*. Ve zřetězeném se pomocí něj oddělují jednolivé části pipeline. Pro tyto účely jsou zde zděděny další variatny registrů: *ControlledRegister*, *ControlledRegisterClr* a *ControlledRegisterEnClr*, které mají zapojeny ještě řídicí signály z hazard unit. Tyto registry tedy mohou být pozastaveny nebo vymazány.

Pracovní registry mají na obsluhu vlastní komponentu *RegisterFile*, které stačí uchovávat jejich hodnoty v poli objektů typu *Wire*. Tato komponenta implementuje obě rozhraní, protože podporuje zápis i čtení. Zápis je řízen řídicím vodičem *RegWrite*. Pokud je na tomto vodiči hodnota 1, komponenta v metodě *preupdate()* zapisuje do zvoleného registru vstupní hodnotu. Čtení probíhá v každém cyklu, metodou *update()* se vyčtou příslušné registry na výstupní sběrnice.

[3] java interface

Polední komponentou, která připadá v úvahu, je instrukční registr, který z důvodu mnoha výstupů tvoří vlastní třídu *InstructionRegister*. Vstupní hodnotou je strojový kód instrukce, která je potřeba rozdělit na jednotlivé významové úseky bitů podle tabulky 3.2 (příp. tabulek 2.1, 2.3, 2.5). Toto se děje pomocí jednoduché maskovací funkce *int getBits(int number, int startBit, int endBit)*. Výstupem instrukčního registru jsou všechny významové úseky všech typů instrukcí najednou[4]:

operační kód	31 - 26
zdrojový registr	25 - 21
zdrojový/cílový registr	20 - 16
cílový registr	15 - 11
hodnota posunu	10 - 6
funkční kód	5 - 0
přímý operand	15 - 0

Tabulka 3.2: Význam jednotlivých úseků strojového kódu všech typů instrukcí

SignExtension

Tato komponenta přebírá 16-bitové číslo, uložené ve dvojkovém doplňku, a jejím výstupem má být stejná hodnota, uložená jako 32-bitové číslo. Je tedy potřeba vstupní hodnotu znaménkově rozšířit, což znamená pro kladná čísla doplnit zleva nulami, pro záporná čísla doplnit zleva jedničkami. Znaménkový bit se vyčte pomocí masky 0x00008000, kladné číslo pak doplní Java automaticky nulami, záporné je třeba logicky sečíst s hodnotou 0xFFFF0000.

Multiplexor

Multiplexor zapisuje na svůj výstup jednu ze vstupních hodnot v závislosti na hodnotě řídicího signálu. V nezřetězeném procesoru využijeme pouze dvouvstupový multiplexor, ve zřetězeném i třívstupový.

ALU

Aritmeticko-logická jednotka pracuje se dvěma vstupy, jedním řídicím vstupem, jedním normálním výstupem a jedním výstupem *zero*. V metodě *update()* je použit jednoduchý *switch* na hodnotu přijatou z řídicí sběrnice. Podle této hodnoty se provede příslušná operace, tak jak bylo popsáno v tabulce 2.6.

U této komponenty lze velmi dobře využít dědičnosti a implementovat pomocí ní další logické a aritmetické obvody, které se v návrhu objevují: bitový posun, sčítání, logické and, or a ekvivalence.

ControlUnit

Řídící jednotka má dvě vstupní hodnoty: operační a funkční kód instrukce. Výstupy z řídící jednotky jsou všechny řídicí vodiče v procesoru, např. multiplexory, ALU atd. Pro dekódování instrukce využívá objekt *InstructionDecoder*, který podle přijatého operačního a funkčního kódu vrací objekt dekódované instrukce, jenž v sobě uchovává hodnoty jednotlivých řídicích vodičů v procesoru. Řídící jednotka tedy pouze nastaví tyto hodnoty na příslušné vodiče.

Memory

Třída Memory symbolizuje externí pamět. Je implementována jako pole datových typů byte s maximálním počtem 4096 prvků, tedy pamět má velikost 4 KB. Jednotlivá slova se ukládají po vzoru big-endianu. Pamět je společná pro data i pro instrukce, jediné oddělení spočívá v rozdílných bázových adresách, které lze nastavit v konstruktoru.

Komponenta pracující s pamětí má jako vstup 32-bitovou adresu a musí umět uložit/vyčíst 32-bitové slovo. Vzhledem k tomu, že je pamět slovně zarovnána, je třeba přistupovat vždy k začátku slova. Adresa je tedy nejprve přepočtena na začátek slova (pokud ukazuje do nějaké jiné části) a pak jsou uloženy/vyčteny čtyři následující bajty.

Pro načítání instrukcí stačí komponenta, která nabízí jen čtení, na což potřebuje vstupní adresu a nastavuje výstupní slovo. Do datové paměti lze i zapisovat, tedy potřebuje jeden vstup pro ukládané slovo navíc. Čtení z obou pamětí probíhá metodou *update()*, zápis do datové paměti metodou *preupdate()*.

Tyto komponenty nepřistupují k paměti přímo, ale skrze objekt typu *Cache*, který představuje mezipamět. Aby byla u cache volně nastavitelná velikost bloku, počet setů a stupeň asociativity, vytvořil jsem pro ni pyramidový model sestávající ze tříd: *Row, SimpleCache, Cache.*

Row představuje jeden řádek v jednoduché jednocestně associované cache, který má atributy *tag, valid bit, dirty bit, data[]*.

SimpleCache představuje jednocestně asociovanou cache a obsahuje tedy pole řádků (objektů typu *Row*). Tento objekt je také jediný, který má přístup přímo do paměti a obsluhuje finální čtení a zápis.

Cache pak představuje obecnou n-cestně asociovanou cache (obsahuje pole objektů typu *SimpleCache*).

Dále možné nastavit, zda se data zapsaná do cache propisují do paměti rovnou, nebo až při nutnosti načíst místo zapsaných dat nová data. Pro vybrání bloku, který bude nahrazen není implementován žádný vyšší algoritmus, výběr probíhá náhodně.

HazardUnit

HazardUnit se uplatní pouze v zřetězeném procesoru. Má několik vstupů z různých sběrnic v procesoru, protože potřebuje porovnávat adresované registry z různých stupňů pipeline a vy-

hodnocovat podmínky skokových instrukcí. Lze ji v konstruktoru vypnout, pokud nemá do běhu procesoru nijak zasahovat.

Řízení chodu procesoru probíhá pomocí několika multiplexorů, které jsou umístěny před *ALU* a před *Ekvivalencí*, která vyhodnocuje skokové instrukce. Dále je *HazardUnit* schopná pozastavit nebo vymazat první tři části pipeline z důvodu uskutečněného podmíněného skoku, při kterém jsou v procesoru nahrány neplatné instrukce, které se nemají provádět.

Logika přeposílání: k přeposlání dojde, pokud se bude zapisovat do registru, ze kterého se v následující instrukci bude číst.

Logika pozastavení: k pozastavení dojde, pokud se budou zapisovat data z paměti do registru, ze kterého se v následující instrukci bude číst

Logika vyprázdnění: k vyprázdnění dojde, pokud je vyhodnocena podmínka skokové instrukce jako platná

3.1.2 Instructions.java

V této třídě je definována jakási knihovna instrukcí v podobě *ArrayListu*, ve kterém jsou uložené objekty typu *Instruction*. Každý tento objekt má identifikační atributy (name, tag, opcode, funct) a hodnoty jednotlivých řídicích signálů pro *ControlUnit*. Pro dekódování je zde implementován objekt *InstructionDecoder*, který poskytuje funkce na dekódování:
Instruction getInstrByOpcodeAndFunct(int opcode, int funct),
Instruction getInstrByOpcode(int opcode),
Instruction getInstrByTag(String tag).
První dvě jsou využívány řídící jednotkou, tu poslední využívá parser, který překládá assembler do strojového kódu.

3.1.3 Parser.java

Tato třída se stará o přečtení vstupního souboru s programem v assembleru a naplnění instrukční paměti strojovými kódy instrukcí.

Čtení je rozdělenu do dvou částí. V první části se načtou definice registrů a datová pamět se naplní daty zapsanými v části *.data*. Ve druhé části se načte sekce *.text*, která obsahuje instrukce i s navěštími, která mohou být použita ve skokových instrukcích. Parser otevře přijatý soubor a prochází jej řádek po řádku. Jednotlivé řádky jsou rozděleny na slova pomocí funkce split(" "), která rozdělí string podle mezer. Další postup se řídí podle prvního slova v řádku.

Při prvním průchodu se interpretují pouze řádky, která začínají *#define* nebo jsou v sekci *.data*.

Ve druhém průchodu se podle prvního slova na řádku buď vytvoří návěští nebo načte instrukce podle svého tagu.

Překlad definic registrů a načtení datových položek

Definice registrů se předpokládají ve tvaru: *#define XX $Y*. Tuto formulaci interpretuje překladač tak, že registr číslo Y je v textu označován značkou XX. Např.: *#define s0 $2*.

Definice dat se předpokládá ve tvaru: *var: .word A, B, C...* Toto se interpretuje jako pole hodnot, které se ukládají do datové paměti, a adresa začáku pole je zapamatována překladačem pod aliasem *var*. Tento alias může být později použit v instrukci *la*. Např.: *pole: .word 1, 2, 3.*

Načtení instrukcí

Definice instrukce závisí na typu instrukce.

Instrukce typu R se předpokládá ve tvaru: *tag rs, rt, rd*, kde registry mohou být zapsány pomocí symbolu dolaru nebo pomocí definovaných značek. Např.: *add $2, s0, $3.*
Pokud se jedná o instrukci *sll*, poslední registr je nahrazen konstantou. Např.: *sll $1, $2, 2.*

Instrukce typu I se předpokládá ve tvaru: *tag rs, rt, imm*, kde registry mohou být zapsány stejně jako u typu R, přímý operand může být v decimální nebo hexadecimální soustavě. Např.: *addi s1, $0, 10*.
Pokud se jedná o skokovou instrukci *beq* nebo *bne*, přímý operand je nahrazen návěštím cílové instrukce. Např.: *bne t1, t2, loop*.
Pokud se jedná o instrukci *lw* nebo *sw*, je očekáván formát: *lw rt, offset(rs)*, kde offset může být zapsán formálně stejně jako přímý operand. Např.: *lw s0, 4($5).* Závorku je nutné psát i pokud je offset roven nule.

Instrukce *la* se předpokládá intuitivně ve tvaru: *la rs, var*. Např.: *la s0, pole*.

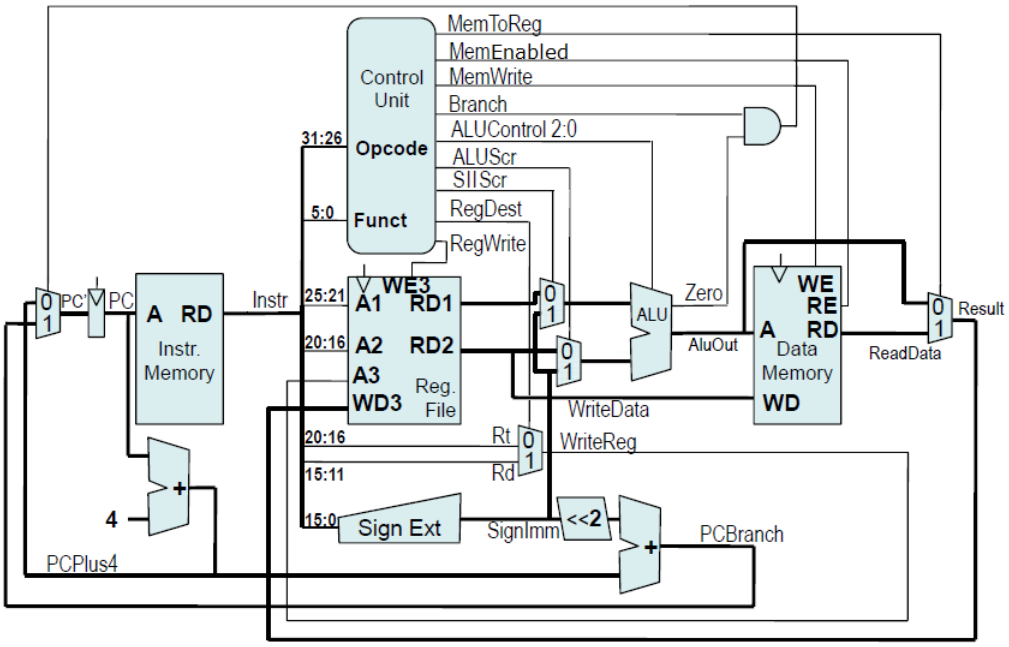
3.1.4 Processor.java

Processor je abstraktní třída, která v sobě definuje základní atributy, jenž budou společné pro oba její potomky, jimiž budou jednocyklový a zřetězený procesor.

Definuje základní komponenty, paměti, jejich bázové adresy, cache a abstraktní metodu *void clock()*, která provede jeden cyklus procesoru. Dále si tento objekt inicializuje parser a načte vstupní soubor. Tuto třídu budeme také využívat k pozorování procesoru skrze grafické uživatelské prostředí. Pro tyto potřeby jsou zde metody pro čtení dat ze základních komponent.

3.1.5 SimpleProcessor.java

Nyní se dostáváme ke stavbě samotného procesoru. Tato třída dědí základní atributy ze třídy *Processor* a velké množství atributů, zvláště všechny atributy typu *Wire*, si zde definuje sama. V konstruktoru pak inicializujeme všechny sběrnice a komponenty tak, abychom získali funkční jednocyklový procesor z obrázku 3.1.

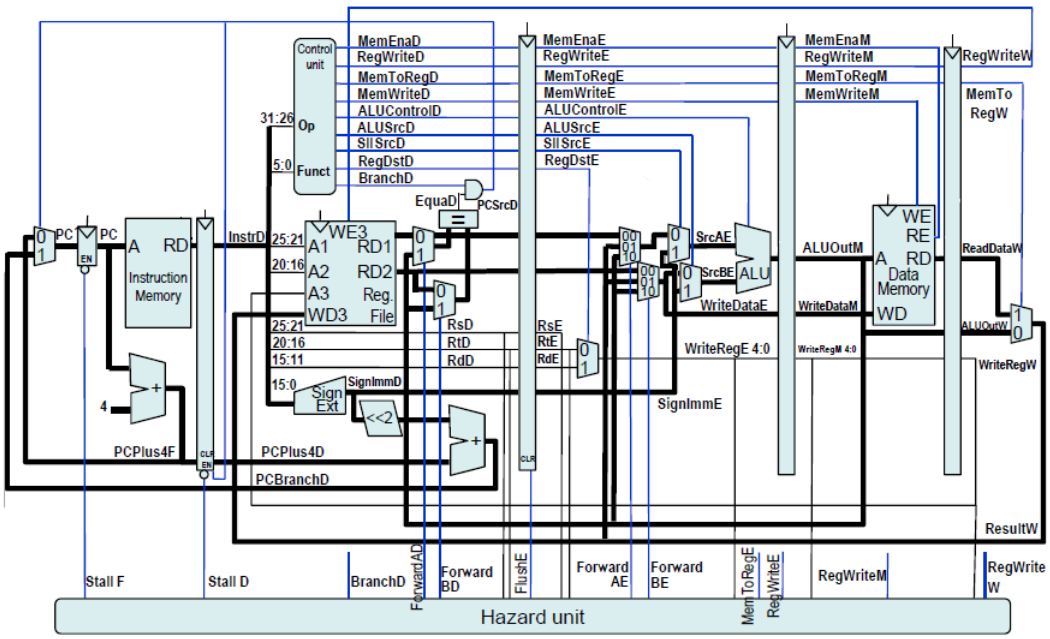


Obrázek 3.1: Schématický obrázek propojení komponent jedno-cyklového procesoru[3]

Zásadní pro funkci procesoru je implementace metody *void clock()*, která charakterizuje, v jakém pořadí budou jednotlivé komponenty aktualizovat své výstupní hodnoty. V jednocyklovém procesoru je nejdůležitejší oddělit od sebe náběžnou hranu hodin a aktivní dobu hodin. S náběžnou hranou je třeba aktualizovat programový čítač, pamět registrů a datovou pamět. Zbytek komponent lze aktualizovat postupně, jak se propagují hodnoty procesorem, tedy zleva doprava.

3.1.6 PipelineProcessor

Zřetězený procesor je v principu podobně vystavěn jako jednocyklový procesor. Obsahuje mnohem více sběrnic a více dalších komponent, které jeho popis zesložiťují. Při popisu se držíme obrázku 3.2.



Obrázek 3.2: Schématický obrázek propojení komponent zřetězeného procesoru[3]

Pro funkci zřetězeného procesoru jsou kromě komponent reagujících na náběžnou hranu důležité registry, které oddělují jednotlivé stupně pipeline. Funkci *void clock()* zde můžeme rozdělit do dvou oddělených funkcí *void compute()* a *void pipeline()*, které vykonáme postupně.

Funkce *compute()* aktualizuje všechny komponenty, vypadá tedy podobně jako funkce *clock()* u jednocyklového procesoru s tím rozdílem, že nakonec přidává aktualizaci *HazardUnit*. Funkce *pipeline()* pak překlopí vypočtené hodnoty do dalšího stupně pipeline a teprve tím končí jeden cyklus. Překlápění se děje aktualizací všech oddělujících registrů.

3.2 Mips.controller

V tomto balíčku se nachází jediná spustitelná třída s názvem *MIPSSimulator* a jedna řídící třída s názvem *Controller*.

Třída *Controller* propojuje všechny balíčky dohromady a tvoří kostru programu. Udržuje si referenci na *Simulator* (grafická část programu) i na *Processor* (výpočetní část programu). Tyto objekty mají referenci pouze na *Controller*, čímž lze dosáhnout vyšší přehlednosti a bezpečnosti programu. Všechny požadavky uživatele, které pocházejí z uživatelského rozhraní, jsou tedy obslouženy skrze tuto třídu.

Program tedy po spuštění ve třídě *MIPSSimulator* vytvoří objekt typu *Controller*, čímž se rozeběhne celý program.

Kapitola 4

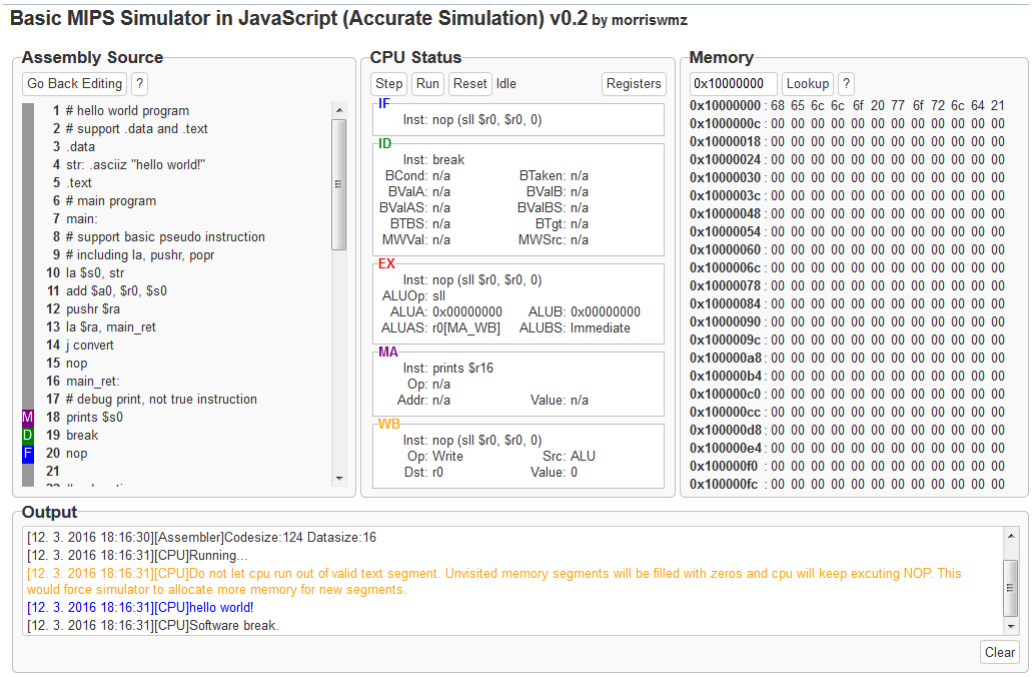
4 Srovnání současných simulátorů

Abychom byli shopni navrhnout uživatelsky co možná nejlepší simulátor, je třeba se porozhlédnout po již vytvořených simulátorech a zhodnotit jejich klady a zápory, ze kterých pak budeme vycházet při vlastním návrhu.

4.1 Basic MIPS Simulator in JavaScript

Tento simulátor se nachází na adrese:
`http://morriswmz.jit.su/static/simple-mips-pipelined.html`.
Jde o velmi jednoduchý simulátor napsaný v javascriptu. Implementovaný procesor obsahuje pětistupňovou pipeline bez ošetřených hazardů, tedy nepodporuje přeposílání, pozastavení ani vyprázdnění.

4.1.1 Uživatelské rozhraní



Obrázek 4.1: Basic MIPS Simulator

Simulátor obsahuje prostý editor assembleru, přehled registrů, přehled paměti a výstupní konzoli. Editor nezvýrazňuje syntax, takže pro přímé psaní assembleru není moc praktický. Při vykonávání instrukcí se obarví řádky, které jsou momentálně vykonávány. Přehled registrů lze překliknout na přehled pipeline s vypsanými vykonávanými instrukcemi a řídicími signály. Hodnoty lze zobrazit hexadecimálně nebo decimálně. Pamět je vypsána v přehledné tabulce a lze v ní i vyhledávat přímo podle adresy. Výstupní konzole informuje o stavu procesoru a případně slouží jako výstup simulovaného programu.

Simulace lze spustit celá, nebo procházet krok po kroku. Po skončení ji lze restartovat.

4.1.2 Výhody a nevýhody

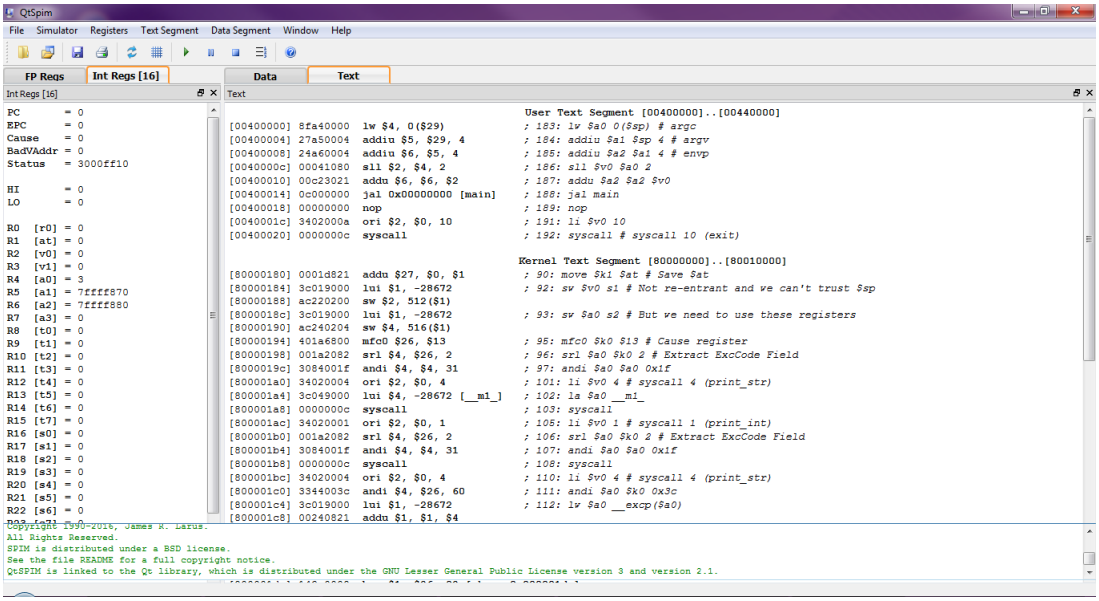
Mezi výhody bych uvedl možnost zobrazení procházejících instrukcí v pipeline, krokování simulace a zvýrazňování vykonávaných řádků v kódu. Další výhodou může být jeho dostupnost na internetu.

Hlavní nevýhodou je vykonávání instrukcí v pipeline bez ošetřených hazardů. Dalším nedostatkem je chybějící dokumentace se seznamem podporovaných instrukcí.

4.2 QtSPIM

SPIM (MIPS pozpátku) je soběstačný simulátor MIPS procesoru, který podporuje téměř kompletní instrukční sadu. Byl navržen jak pro studijní, tak pro vývojové účely. Je napsán v jazyce C a jeho nejnovější verze, QtSPIM, je napsána v jazyce C++ a využívá QT knihovnu pro grafické uživatelské rozhraní, díky čemuž může běžet na všech třech hlavních platformách[1] ve stejné verzi[6].

4.2.1 Uživatelské rozhraní



Obrázek 4.2: QtSPIM

Tento simulátor na rozdíl od předchozího neobsahuje žádný editor assembleru a soubory *.s se do něj musí importovat. Pokud dojde při překladu k chybě, je potřeba soubor opravit v externím editoru.

Simulátor má čistě textovou podobu, takže nabízí klasický textový náhled do registrů a do datové a instrukční paměti. Tato oddělení lze otevřít do samostatných oken, což dovoluje uživateli nastavit si rozhraní jak potřebuje. Data v registrech lze zobrazit ve třech formátech: decimálním, hexadecimálním a binárním a lze je při simulaci manuálně měnit. Nástroje pro simulaci nalezneme jak v nabídce File, tak i jako tlačítka přímo v okně. Simulovat můžeme celý program najednou, nebo postupovat klasicky po krocích. Dále můžeme využít možnost debugování s breakpointy. Po simulaci lze simulátor restartovat do původní polohy.

[1] Windows, Mac OS X a Linux

4.2.2 Výhody a nevýhody

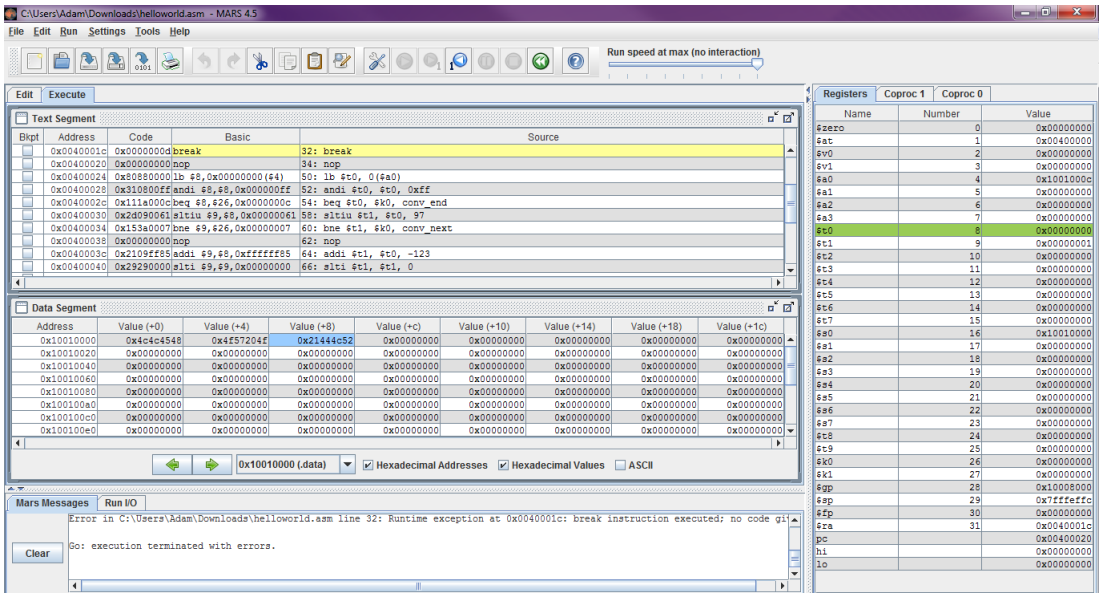
Hlavní výhodou je dle mého názoru podpora téměř kompletní instruikční sady MIPS, díky čemuž lze tento simulátor využívat i pro vývojové účely. S tím souvisí i možnost breakpointů a manuálního měnění obsahu registrů.

Nevýhodou je absence editoru assembleru a hlavně jakéhokoliv grafického znázornění chodu procesoru.

4.3 Mars

MARS[2] je jedním z nejpoužívanějších simulátorů obzvlásť pro výukové účely. Byl navržen na univerzitě v Missouri a byl implementován podle publikace Computer Organization and Design s ohledem na potřeby profesorů a studentů. Je napsán v Javě, takže je stejně jako QtSPIM nezávislý na platformě, ale nemá bohužel implementovanou pipeline[3]. Podporuje většinu instrukcí a pseudoninstrukcí z MIPS ISA[4] a i některá systémová volání, přičemž jsou všechny tyto funkce podrobně zdokumentovány v nápovědě přímo v simulátoru[7].

4.3.1 Uživatelské rozhraní



Obrázek 4.3: MARS

Simulátor MARS má velmi bohaté grafické uživatelské rozhraní. Obsahuje velmi propracovaný editor assembleru a dovoluje nastavit si simulační okno podle svých představ. Kromě tradičních

[2] Mips Assembly and Runtime Simulator
[3] zřetězené vykonávání instrukcí
[4] instrukční set

nástrojů týkajících se samotné simulace (krokování, breakpointy) nabízí ještě řadu rozšiřujících nástrojů, mezi které patří například prediktor skokových instrukcí, cachování datové paměti a pedagogicky velmi povedený nástroj MIPS X - Ray, který dovoluje pozorovat průchod instrukce procesorem krok po kroku a jednotlivé kroky vykresluje na schematickém obrázku procesoru.

4.3.2 Výhody a nevýhody

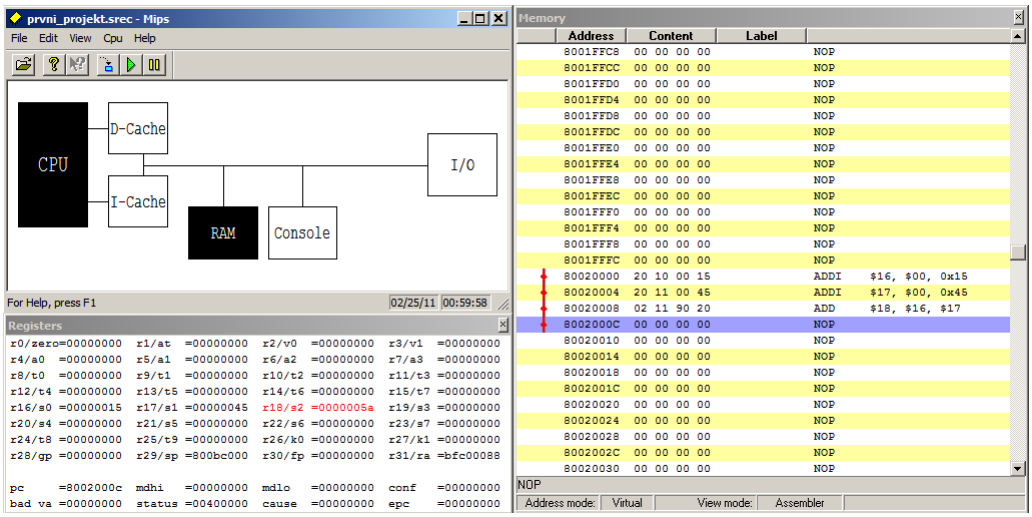
Výhod nalezneme u tohoto simulátoru opravdu mnoho. V první řadě je to velmi praktický editor assembleru, který má implementováno obarvování textu a automatické doplňování. Dále má velmi dobře ovladatelnou simulaci. Dovoluje spustit simulaci zvolenou rychlostí, např. 1 instrukce za sekundu. Nabízí kromě klasického krokování i krokování zpět. Velkým plusem jsou hlavně již zmíněné doplňující a rozšiřující nástroje v čele s MIPS X - Ray, který je velmi názornou pomůckou k výuce.

Jediným nedostatkem je absenece zřetězeného vykonávání instrukcí, které je ale podstatnou částí funkcionality a principu chodu MIPS procesoru.

4.4 MipsIt

MipsIt byl stejně jako MARS vyvinut přímo pro výuku a implementován podle publikace Computer Organization and Design. Podporuje většinu instrukcí MIPS ISA a momentálně je využíván jako pomůcka při výuce v předmětu A0B36APO. Obsahuje praktický editor a několik simulátorů[5].

4.4.1 Uživatelské rozhraní



Obrázek 4.4: MipsIt

V editoru je možné zapsat program v assembleru nebo v C. Velmi praktické je stejně jako v editoru MARSu zvýrazňování syntaxe a psaní komentářů. Editor funguje také jako překladač a kompilátor pro simulátory.

MipsIt obsahuje tři oddělené simulátory. Jeden pro jednoduchý nezřetězený procesor, jeden pro jednoduchý zřetězený procesor a jeden úplně zřetězený. Formálně vypadají okna všech tří simulátorů stejně. Podporují cachování obou pamětí včetně nastavení parametrů cache. Dále lze zobrazit přímo paměťová úložiště a vstupně výstupní sběrnici, která je znázorněna spínači a LED diodami.

Simulace lze opět spustit celá nebo lze postupovat po jednotlivých krocích. V zřetězených simulátorech lze zobrazit schéma procesoru a sledovat změny hodnot na jednotlivých komponentách.

4.4.2 Výhody a nevýhody

Mezi výhody můžeme zařadit podobně jako u MARSu praktický editor a schématický nákres procesoru, který pomáhá k pochopení principu fungování procesoru. Velmi dobře působí i nastavitelné cache.

Nevýhodou může být absence schématického znázornění nezřetězeného procesoru a trochu nepohodlná práce s kompilátorem.

4.5 Shrnutí

V této kapitole jsme se pokusili jednoduše popsat čtyři vybrané simulátory procesoru MIPS a zhodnotit jejich klady a zápory. Nejlepé byly hodnoceny simulátory MARS a MipsIt, protože nabízejí názorné schematické obrázky procesoru, které jsou při výuce velkou výhodou.

Výsledkem této kapitoly je následující seznam vlastností dobrého simulátoru:

- zobrazení registrů, pipeline, pamětí
- možnost přímého vstupu do registrů a pamětí
- debug s breakpointy, krokování vpřed i vzad, nastavení rychlosti zpracovávání instrukcí
- zvýrazňování zpracovávaných instrukcí a jejich pozic v pipeline
- vizualizace průchodu instrukce procesorem
- prediktor skokových instrukcí
- cachování pamětí
- editor se zvýrazňováním syntaxe

Kapitola 5
5 Výsledný simulátor

V předchozí kapitole jsme si ujasnili, jaké vlastnosti by měl dobrý simulátor mít. Vzhledem k tomu, že simulátor je určen výhraně k výukovým účelům, hlavní důraz by měl být kladen na přehlednost a názornost simulace. V rozsahu bakalářské práce není možné vytvořit simulátor, který by splňoval všechny body zmíněné v závěru 4. kapitoly, a proto vybereme jen ty nejdůležitější.

K implementaci využijeme knihovnu *javax.swing*, která vyhovuje technickým i estetickým požadavkům.

5.1 Balíček Mips.viewer

Nejprve si stručně popíšeme jednotlivé třídy, které tvoří GUI[1] programu.

Název třídy	Popis
Simulator.java	základní kámen celého GUI, zajišťuje komunikaci se zbytkem programu
formSimulator.java	hlavní okno simulátoru s hlavními ovládacími prvky
formProcessor.java	okno, pro nastavení parametrů simulovaného procesoru
viewProcessor.java	interface definující metody pro vykreslování chodu dat procesorem
viewSimpleProcessor.java	okno, pro názorné vykreslování chodu dat jednocyklovým procesorem
viewPipelineProcessor.java	okno, pro názorné vykreslování chodu dat zřetězeným procesorem
viewCache.java	okno, pro vykreslení obsahu cache pamětí

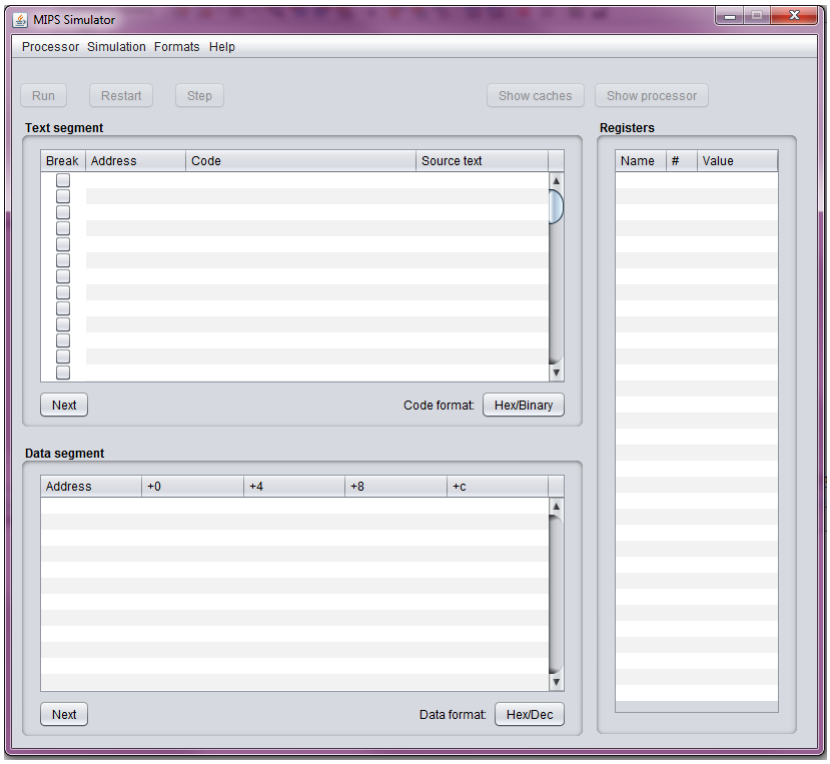
Tabulka 5.1: Přehled balíčku Mips.viewer

Třída *Simulator* se stará o řízení běhu celého GUI. Na začátku vytváří nové vlákno, ve kterém uživatelské rozhraní běží, a při běhu programu otevírá nová okna a nastavuje jejich vlastnosti v závislosti na volbách uživatele. Pro přístup do výpočetní části programu využívá třídu *Controller*, skrze kterou řídí spouštění a parametrizaci simulace.

Ostatní třídy (kromě viewProcessor, což je pouze interface) už zastávají v programu specifickou grafickou funkci, takže si je popíšeme z pohledu uživatelského použití celého simulátoru.

[1] graphical user interface

5.2 Hlavní okno



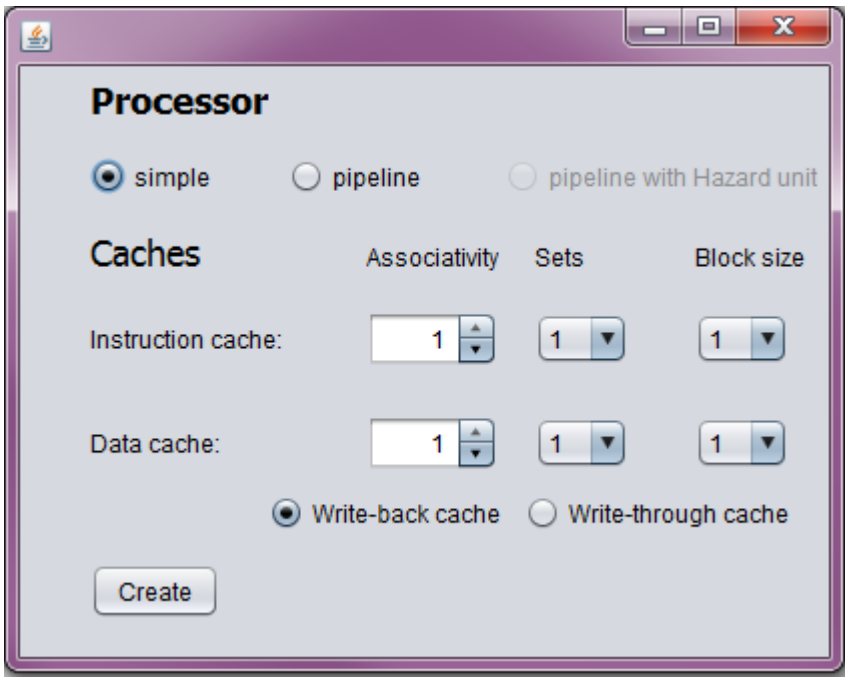
Obrázek 5.1: Hlavní okno simulátoru

Po spuštění simulátoru se nám zobrazí hlavní okno (obrázek 5.1). Okno je inspirováno přímo simulátorem MARS.

Kromě hlavního menu v záhlaví obsahuje tři tabulky a několik tlačítek pro ovládání simulace a vzhledu. V hlavním menu se po spuštění velmi snadno intuitivně zorientujeme, protože nabídky, které nejsou momentálně k dispozici jsou zablokované.

5.2.1 Vytvoření procesoru

Nový procesour můžeme vytvořit kliknutím na možnost *Processor - N*ew, která otevře příslušné dialogové okno (obrázek 5.2).



Obrázek 5.2: Nový procesor

V první řadě si můžeme vybrat, zda chceme simulovat na jednocyklovém (simple) nebo zřetězeném (pipeline) procesoru. Zřetězený procesor má ještě dvě varianty: bez aktivní hazard unit a s aktivní hazard unit (vypnuté nebo zapnuté řešení datových a řídicích hazardů). Dalšími parametry jsou parametry instrukční a datové cache. Kromě různého stupně asociativity, různého počtu setů a různé velikosti bloku lze také nastavit, zda má být cache typu write-back nebo write-through (zda se do paměti propisuje pouze když je to nutné, nebo přímo při zápisu do cache). Tlačítkem *Create* nastavení potvrdíme.

5.2.2 Načtení souboru

Nyní můžeme postoupit ke kroku dvě: otevření simulace. V menu *Simulation* vybereme *New*, což nám otevře dialogové okno pro výběr vstupního souboru assembleru. Vybereme soubor, který chceme simulovat a volbu potvrdíme. Nyní proběhne parsování souboru, při kterém může dojít k chybě, která se nám zobrazí v chybovém informačním okně. Pokud k tomu dojde, je třeba podle chyby opravit zdrojový soubor. Informace o formátu zdrojového souboru jsou v menu v nabídce *Help* pod možností *Assembler file*.

5.2.3 Přehledy

Pokud proběhlo parsování v pořádku, tabulky v hlavním okně se nám naplní daty. Tabulky na levé straně nabízejí přímý náhled do paměti, přičemž ta horní zobrazuje programovou část paměti (dolní 2KB) a spodní datovou část paměti (horní 2KB). Programová tabulka nabízí kromě adresy

instrukce, jejího strojového kódu a zdrojové řádky ještě možnost nastavení tzv. breakpointu, před kterým se spuštěná simulace zastaví. Tabulky mají pouze 64 řádků, což dovoluje zobrazit pouze první polovinu paměti. Náhledu na druhou část lze dosáhnout kliknutím na tlačítko *Next*. Na pravé straně pak najdeme tlačíko, pomocí kterého lze změnit formát zobrazovaných čísel. V tabulce instrukcí lze měnit zobrazení strojového kódu instrukcí mezi hexadecimálním a binárním a v tabulce dat lze měnit zobrazení hodnot mezi hexadecimálním a decimálním. Vpravo je pak tabulka zobrazující registry. Formát dat registrů lze také změnit, avšak pouze v nabídce hlavního menu pod názvem *Format*.

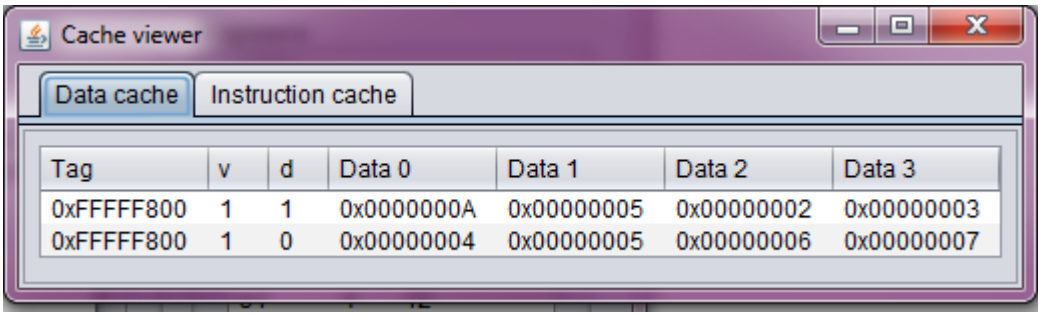
5.2.4 Simulace

Simulaci můžeme ovládat v hlavním menu nebo přímo tlačíky v horní části okna. Tlačíko *Run* spustí simulaci, která se zastaví buď na zvoleném breakpointu nebo po 100 vykonaných instrukcích. Pokud chceme simulací postupovat krok po kroku, využijeme tlačítko *Step*. Právě vykonávaná instrukce se pak v tabulce instrukčního segmentu paměti zvýrazňuje. Po skončení simulace ji můžeme vrátit opět do základního stavu tlačítkem *Restart*. Simulaci můžeme kdykoliv pomocí volby *New* změnit. Stejně tak můžeme pomocí volby *Processor - Edit* změnit parametry procesoru.

5.3 Vedlejší okna

Pomocí tlačítek *Show caches* a *Show processor* můžeme otevřít vedlejší okna s rozšiřujícími vizualizacemi.

5.3.1 Cache



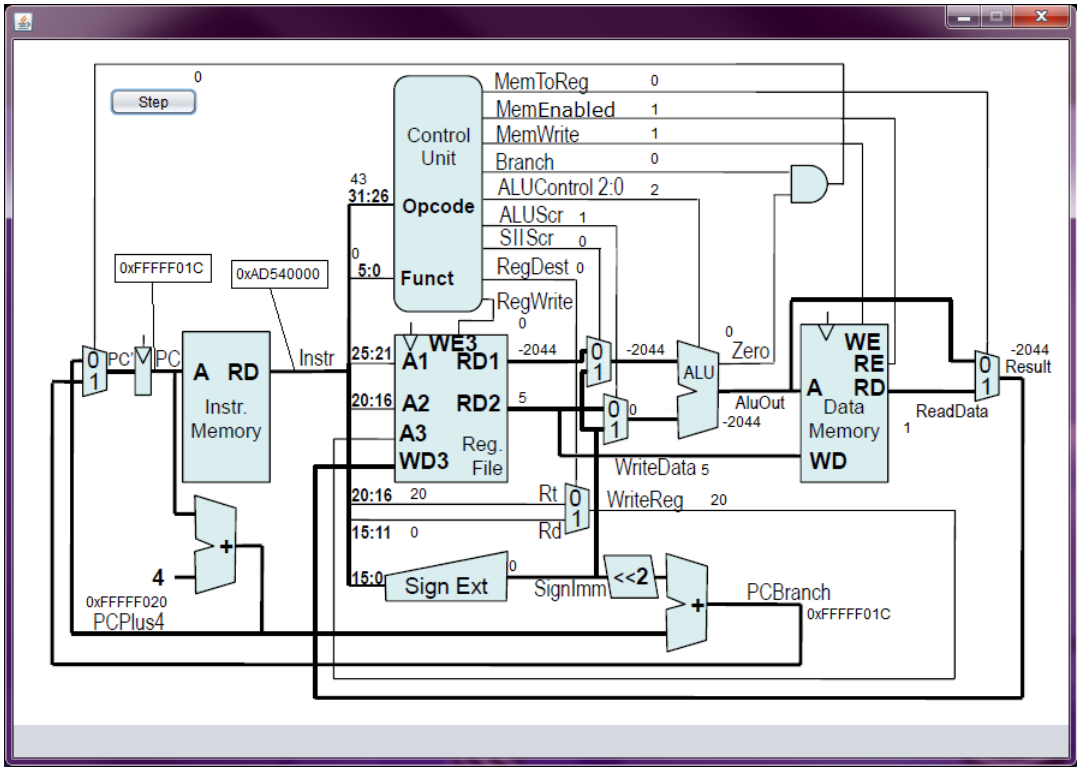
Obrázek 5.3: Náhled na cache paměti

V tomto okně (obrázek 5.3) vidíme obsah cache pamětí. Každá cache má vlastní panel, mezi kterými můžeme přepínat. Cache jsou vizualizovány klasicky pomocí tabulek. Obsahují hodnotu *tag*, příznaky *valid* a *dirty* a pak samotná *data*. Cache se automaticky aktualizují při běhu simulace.

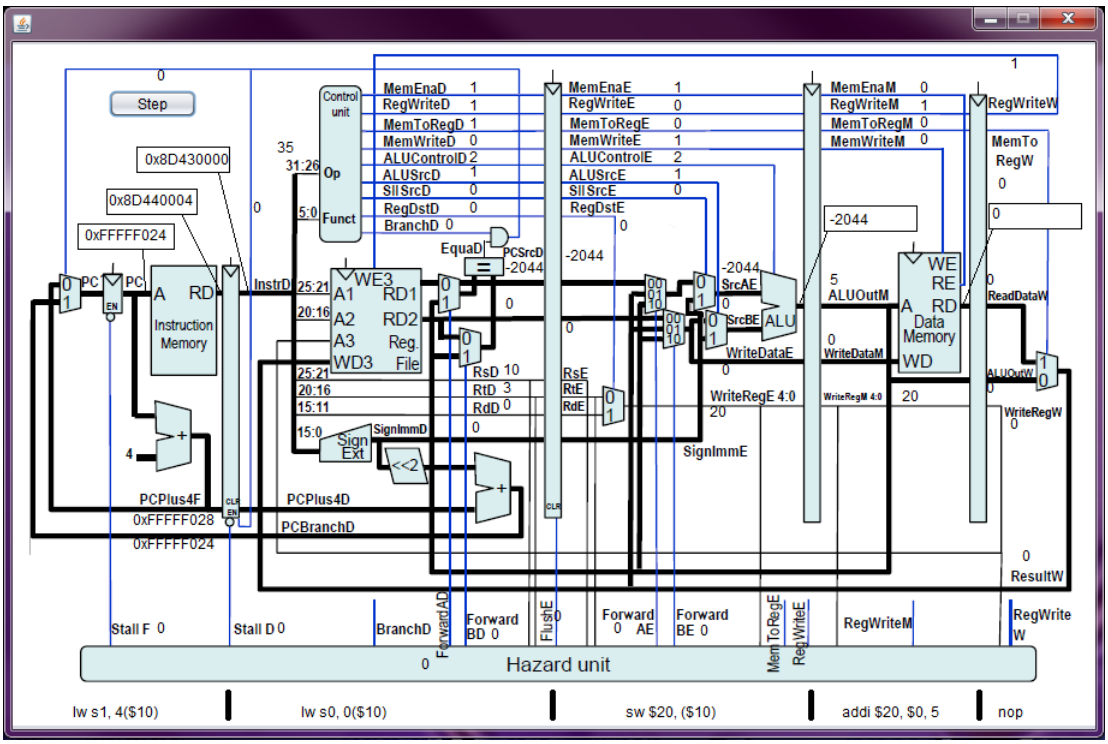
5.3.2 Procesor

Toto okno nabízí asi nejnázornější pohled na simulaci. Jsou zde přímo ve schématu procesoru vypsané hodnoty, které se nacházejí na sběrnicích. Okno se opět aktualizuje automaticky při běhu simulace. Pro každý typ procesoru se liší, protože každý typ má jinak zapojené jednotlivé komponenty.

V levé horní části se nachází tlačítko *Step*, které slouží k pohodlnému posouvání simulace přímo v tomto okně. Zřetězený procesor má navíc ve spodní části uvedené zdrojové texty instrukcí, které se nacházejí v jednotlivých částech pipeline.



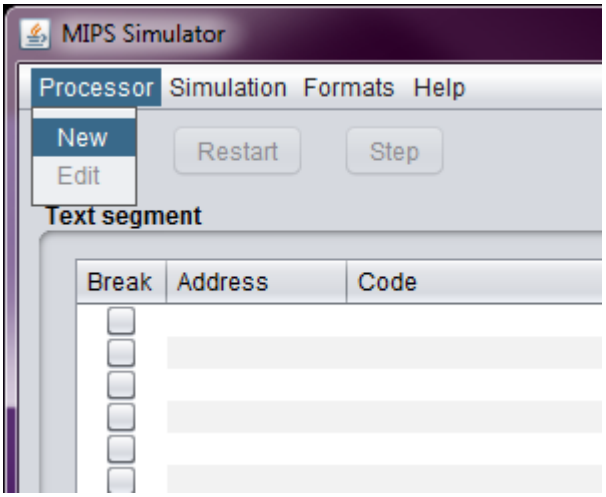
Obrázek 5.4: Simulační schéma jednocyklového procesoru



Obrázek 5.5: Simulační schéma zřetězeného procesoru

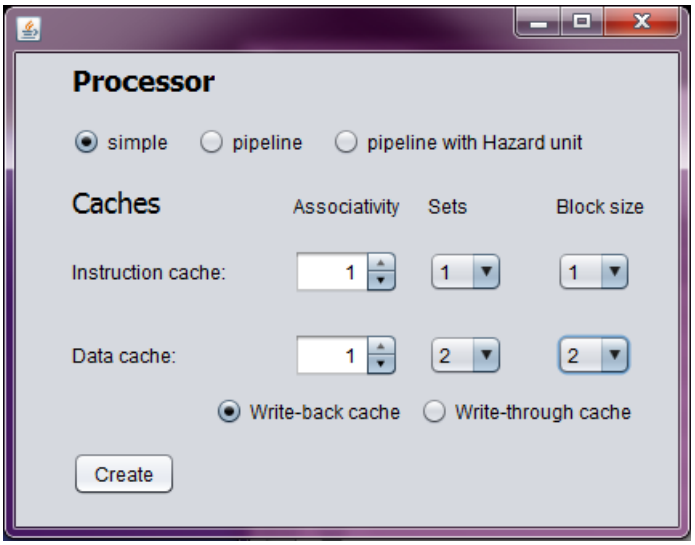
5.4 Příklad simulace

Prvním krokem je vytvoření nového procesoru, na kterém budeme program simulovat. V hlavní nabídce vybereme *Processor - New*.



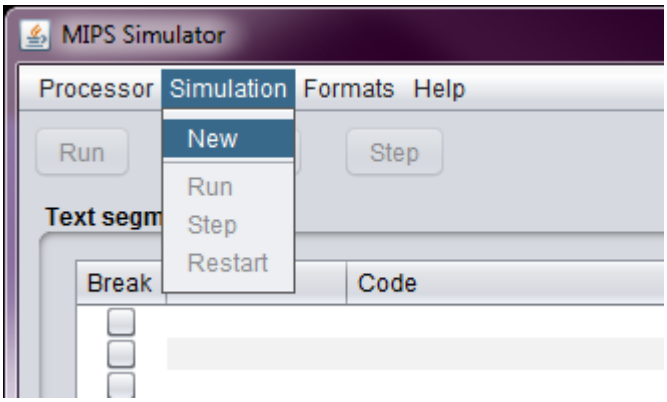
Obrázek 5.6: Vytvoření nového procesoru

Nastavíme si jednocyklový procesor a nějaké rozumné cache paměti.



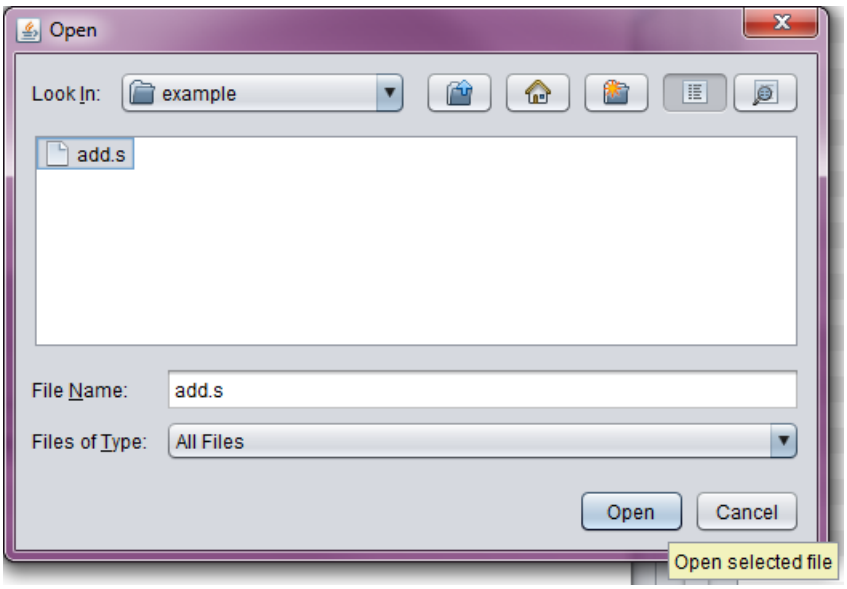
Obrázek 5.7: Nastavení procesoru

V dalším kroku si vytvoříme novou simulaci. V hlavní nabídce vybereme *Simulation* - *New*.



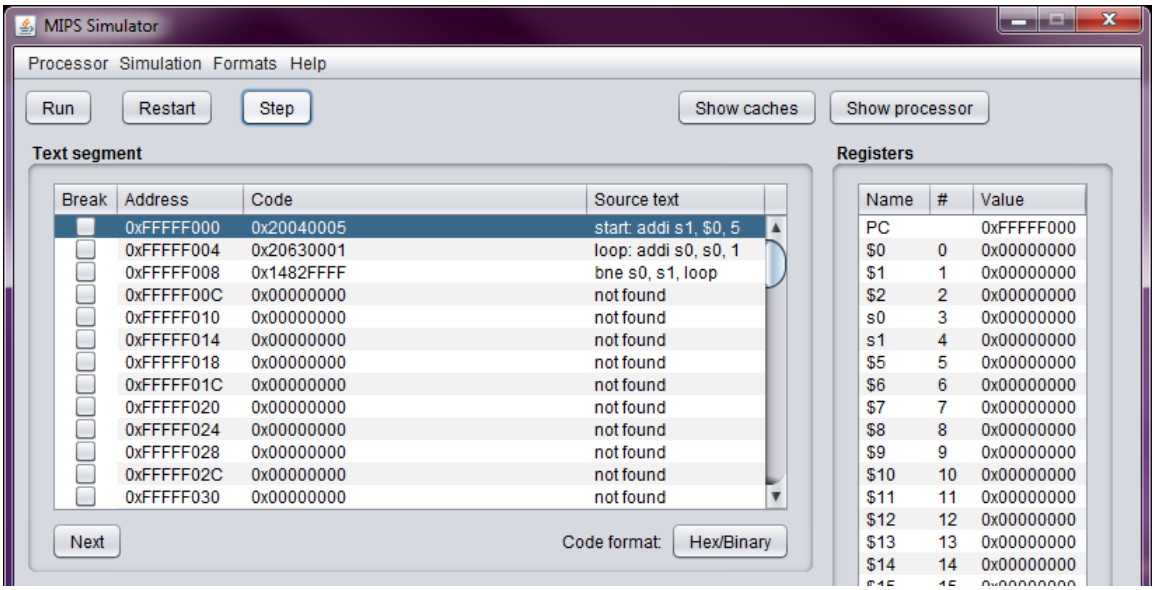
Obrázek 5.8: Nová simulace

V dialogovém okně vybereme soubor s programem, který chceme simulovat.



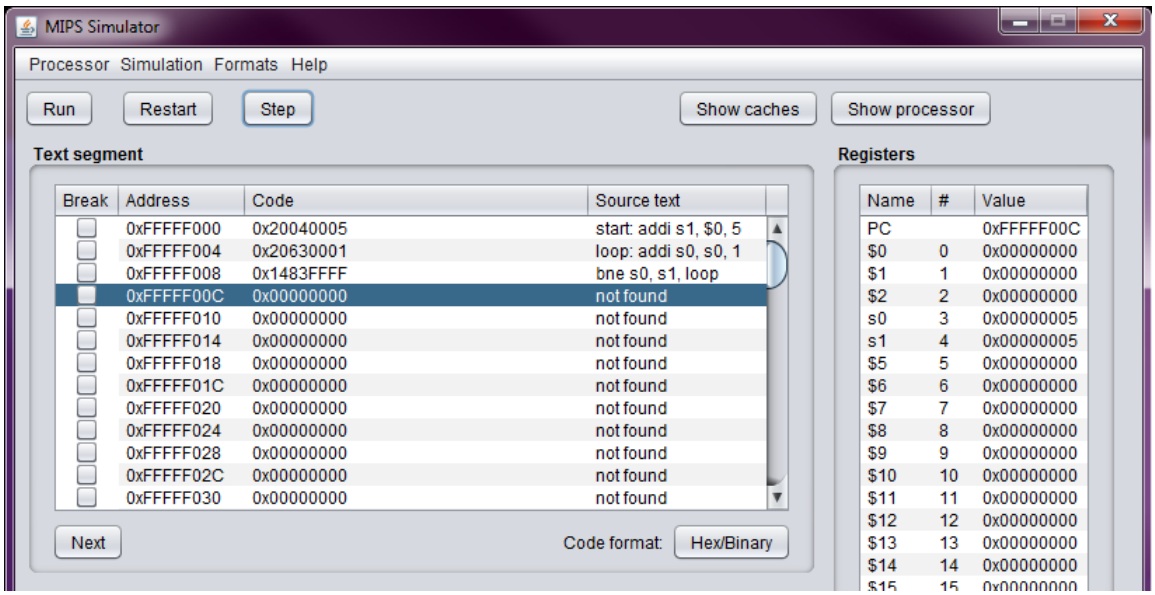
Obrázek 5.9: Výběr souboru

V simulačním okně si můžeme ověřit, že se nám program nahrál do paměti procesoru. Můj ukázkový program představuje jednoduchý cyklus počítající od jedné do pěti.



Obrázek 5.10: Simulace

Simulaci provádíme klikáním na tlačítko *Step*, přičemž můžeme sledovat, jak se mění hodnoty v registrech. Pro simulování složitejších programů můžeme využít tlačítko *Run* v kombinaci s breakpointy.



Obrázek 5.11: Konec simulace

Program counter již nyní ukazuje do prázdné paměti, simulace je tedy u konce. Pro opakování stiskneme tlačítko *Restart*, čímž simulaci vrátíme do základního stavu.

Při simulování složitějších programů si pro názornější ukázku můžeme nechat zobrazit náhled procesoru tlačítkem *Show processor* nebo náhled cache pamětí tlačítkem *Show caches*.

Kapitola 6
Shrnutí

Cílem této práce bylo navrhnout a implementovat simulátor MIPS procesoru s grafickým uživatelským rozhraním. Návrh uživatelského rozhraní se měl opírat o porovnání dostupných simulátorů a identifikování jejich výhod a nevýhod. Při práci měl být kladen důraz na názornost a přehlednost vytvářeného simulátoru, který by měl být později používán v předmětu A0B36APO.

Mezi zkoumanými simulátory je z mého pohledu nejlepší simulátor MARS, který nabízí mnoho přehledných grafických nástrojů, které mohou pomoct k pochopení základních funkčních principů procesoru. Naopak simulátor SPIM, který je pouze v textové formě, mi nepřipadal pro výukové učely vhodný. Jediným nedostatkem MARSu byla absence možnosti zřetězeného vykonávání instrukcí, které tvoří podstatnou část látky probírané v předmětu APO. Z těchto důvodů jsem se rozhodl navrhnout svůj simulátor po vzoru MARSu, avšak s doplněnou možností zřetězeného vykonávání.

Simulátor byl podle zadání navržen tak, aby byla oddělena výpočetní a vizualizační část programu. Dále byla pozorně pečlivě zpracována práce s cache pamětí. Práce byla průběžně konzultována s vedoucím a požadavky byly v průběhu upravovány a doplňovány. Navržený simulátor obsahuje přehled registrů a paměti, vizualizaci cache pamětí a průchodu instrukce procesorem.

Pro implementaci byl zvolen jazyk Java pro jeho platformní nezávislost a jednoduchou realizací GUI.

Některé požadavky bohužel nebylo možné v této práci zvládnout, simulátor lze dále rozšiřovat. Nepodařilo se zcela odladit chod zřetězeného procesoru s aktivní hazard unit, takže tato možnost je v odevzdávané verzi zablokována. Jedním z chybějících nástrojů je praktický editor se zvýrazněním syntaxe a našeptáváním. Dále zde není implementován žádný algoritmus vyprázdňování cache ani žádný prediktor skokových instrukcí.

I přes tyto nedostatky jsem s vytvořeným simulátorem spokojený a doufám, že bude použitelný při výuce v předmětu APO.

Literatura

[1] Hennesy, J. L., Patterson, D. A.: *Computer Architecture: A Quantitative Approach, Third Edition,* San Francisco, Morgan Kaufmann Publishers, Inc., 2002

[2] Hyvl, D.: *Simulátor MIPS procesoru*, Bakalářská práce, Katedra řídicí techniky, FEL ČVUT v Praze, 2016

[3] předmět A0B36APO, ČVUT, 2016
`https://cw.fel.cvut.cz/wiki/courses/a0b36apo/start`

[4] Gerry Kane, Joe Heinrich: *Mips Risc Architecture*, Prentice-Hall Inc., New Jersey 1992

[5] M. Brorsson, MipsIt - A Simulation and Development Environment Using Animation for Computer Archicture Education, 2002
`https://www.ncsu.edu/wcae/ISCA2002/submissions/brorsson.pdf`

[6] James Larus, SPIM: A MIPS32 Simulator, 2011
`http://http://spimsimulator.sourceforge.net/`

[7] Ken Vollmar, MARS (MIPS Assembler and Runtime Simulator), 2016
`http://courses.missouristate.edu/KenVollmar/MARS/`

[8] Wikipedia, MIPS (architektura), 2016
`https://cs.wikipedia.org/wiki/MIPS_%28architektura%29`

Přílohy

7 Kapitola 7

MIPS	Microprocessor without Interlocked Pipeline Stages
ISA	Instruction Set Architecture
SPIM	MIPS pozpátku
MARS	Mips Assembly and Runtime Simulator
GUI	Graphical User Interface
ALU	Arithmetic Logic Unit
APO	Architektura počítačů
RISC	Reduced Instruction Set Computer
MVC	Model-View-Controller

Tabulka 7.1: Seznam použitých zkratek

8 Kapitola 8

MIPSSimulator	složka se spustitelným simulátorem MIPSSimulator.jar
bakalářská práce	složka s textem bakalářské práce včetně zdrojových souborů

Tabulka 8.1: Obsah CD