FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Simulátor 32-bitového procesoru podporující instrukční sadu MIPS
Student:	Petr Nešpůrek
Vedoucí:	Ing. Michal Štepanovský, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce zimního semestru 2020/21

Pokyny pro vypracování

Cílem práce je realizovat software pro podporu výuky počítačově orientovaných předmětů, zejména BI-APS. Úkolem studenta je napsat simulátor znázorňující průběh vykonávání instrukcí v jednoduchém jedno-cyklovém procesoru.

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.
3. Implementujte simulátor navržený v předchozím bodě ve vybraném programovacím jazyce.
4. Vzhled a funkcionalitu simulátoru přizpůsobte potřebám předmětu BI-APS.
5. Vypracujte dokumentaci a manuál k navrženému simulátoru.

Jednotlivé požadavky a postupy konzultujte s vedoucím práce.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 20. února 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Simulátor 32bitového procesoru podporující instrukční sadu MIPS

Petr Nešpůrek

Katedra teoretické informatiky
Vedoucí práce: Ing. Michal Štepanovský, Ph.D.

12. května 2019

Poděkování

Na tomto místě bych chtěl poděkovat Ing. Michalu Štepanovskému, Ph.D. za vedení práce, konzultace a rady, které přispěly k vypracování této práce.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 12. května 2019 ……

Odkaz na tuto práci Nešpůrek, Petr. *Simulátor 32bitového procesoru podporující instrukční sadu MIPS*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Hlavním cílem této práce je implementovat simulátor procesoru, který podporuje instrukční sadu MIPS32. Implementaci vyřeším jako dva programy – jádro simulátoru napsané v jazyce Verilog a obalující grafická aplikace v jazyce Java s použitím grafického frameworku JavaFX. Výsledkem práce je aplikace, která přečte a spustí program napsaný ve strojovém jazyce architektury MIPS32 a zobrazí historii běhu programu. Ukazuje hodnoty v jednotlivých vodičích a modulech procesoru, datovou paměť, instrukční paměť a simulaci skryté paměti (cache). Tento simulátor bude v budoucnu použit pro výuku předmětu BI-APS (Architektury počítačových systémů), použitá instrukční sada je proto upravena pro potřeby tohoto předmětu.

Klíčová slova simulátor procesoru, instrukční sada MIPS32, cache, Verilog, Java

Abstract

Goal of this work is to implement a processor simulator with MIPS32 instruction set. I will implement this as two programs – a simulator core written in Verilog hardware definition language and a GUI application written in Java with usage of graphic framework JavaFX. Result of this work is an application that reads and runs a program written in MIPS32 machine code and shows history of the program run. It displays values in processor modules and wires, data memory, instruction memory and cache simulation. This simulator will be used in BI-APS (Architectures of Computer Systems) course, instruction set is modified for needs of this course.

Keywords processor simulator, MIPS32 instruction set, cache, Verilog, Java

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V dnešní době jsou počítače nezbytnou součástí života a společnosti. Pro mnoho lidí se ovšem činnost počítačů může jevit jako magie. Před svým studiem jsem také nevěděl, jak něco takového může vůbec fungovat, ale nyní už vím, že na tom nic záhadného není. Chtěl bych tedy přiblížit, co se v jádře takového počítače odehrává.

Hlavním mozkem každého počítače je procesor, což je zařízení, které je zodpovědné za čtení a vykonávání strojových instrukcí – základních jednotek počítačových programů. Je to integrovaný obvod, který obsahuje řídící jednotku (řadič), počítací (aritmeticko-logickou) jednotku, počítadlo instrukcí (program counter) a paměťové registry. Řídící jednotka ovládá ostatní součásti procesoru, zajišťuje načítání instrukcí z paměti a jejich spravné vykonání. Strojové instrukce se skládají ze samotného pojmenování instrukce (mnemonika) a určitého počtu operandů. Množinu instrukcí, které daný procesor dokáže zpracovat, určuje architektura procesoru (instrukční sada).

V této práci implementuji simulátor jednoduchého jednocyklového 32bitového procesoru postaveném na instrukční sadě MIPS. Tento simulátor dokáže interpretovat program zapsaný ve strojovém kódu a zároveň znázorňuje průběh vykonávání jednotlivých instrukcí – umožňuje zobrazit hodnoty v jednotlivých komponentách, registrech, paměti a vodičích, a přiblížit tak uživateli, co se uvnitř pracujícího procesoru děje.

Kapitola 1

Cíle práce

Cílem práce je implementovat simulátor znázorňující průběh vykonávání strojových instrukcí v jednoduchém jednocyklovém procesoru:

1. Identifikovat výhody a nevýhody volně dostupných simulátorů procesoru podporujícího instrukční sadu MIPS.
2. Navrhnout ideové schéma simulátoru s důrazem na oddělení simulace a vizualizace.
3. Implementovat simulátor navržený v předchozím bodě ve vybraném programovacím jazyce.
4. Vzhled a funkcionalitu simulátoru přizpůsobit potřebám předmětu BI-APS.
5. Vypracovat dokumentaci a manuál k navrženému simulátoru.

Kapitola 2

Teorie

2.1 Procesor a architektura

Procesor (CPU, central processor unit) je základní součást počítače. Je zodpovědný za čtení a zpracování strojových instrukcí. Skládá se ze dvou hlavních komponent: [1]

ALU – aritmeticko-logická jednotka, provádí výpočty a logické operace;

Řadič – řídící jednotka, směruje data a operace uvnitř procesoru.

Kromě výše uvedených obsahuje pomocné komponenty, jako jsou registry a programový čítač. Procesor obecně vykonává instrukce následujícím způsobem:

1. Procesor přečte instrukci z paměti[1] z adresy dané programovým čítačem.
2. Řadič dekóduje instrukci.
3. Řadič nasměruje vstupy pro ALU (z registrů nebo instrukce).
4. ALU provede početní nebo logickou operaci.
5. Výsledek se zapíše do paměti[2] nebo registru.
6. Zvýší se programový čítač na další instrukci[3].

Architektura procesoru (ISA, instruction set architecture) je abstraktní model popisující provozní principy procesoru. Definuje například seznam podporovaných strojových instrukcí, seznam registrů, nativní datové typy, paměťovou architekturu, obsluhu přerušení nebo vstupy a výstupy.[2]

[1] Instrukční paměť nebo spojená hlavní paměť v závislosti na paměťové architektuře
[2] Datová paměť nebo spojená hlavní paměť v závislosti na paměťové architektuře
[3] V případě skokové instrukce se změní úplně

2.2 Architektura MIPS32

MIPS32 je architektura procesorových instrukcí založená na instrukcích s pevnou délkou. Používá load/store model, což znamená, že instrukce operují pouze s hodnotami v registrech, nikoli s hodnotami v paměti. K paměti se přistupuje pouze instrukcemi, které uloží hodnotu z paměti do registru a obráceně (load a store instrukce). Tím se přispívá k nižšímu počtu přístupů do paměti a zjednodušuje se sada instrukcí.[3, str. 21]

2.2.1 Instrukce

Všechny instrukce mají délku 32 bitů a v paměti jsou vždy zarovnané. Bit 0 v instrukci je první zprava, Bit 31 je první zleva. Hlavní kód operace (`opcode`) je zakódován v bitech 26–31.[3, str. 48]

Instrukce se dělí do tří typů: R, I a J. Instrukce typu R specifikují tři registry a hodnoty `funct` a `shamt`. Instrukce typu I specifikují dva registry a 16bitovou konstantu. Instrukce typu J specifikují 26bitovou adresu skoku.[4, A-174] Formát instrukcí je zobrazen v tabulce 2.1.

typ	formát (bity 31–0)					
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (5)
I	opcode (6)	rs (5)	rt (5)	imm (16)		
J	opcode (6)	addr (26)				

Tabulka 2.1: Formát instrukcí

Zkratky v tabulce znamenají:

opcode 6bitový primární kód operace

rs 5bitová specifikace zdrojového registru

rd 5bitová specifikace cílového registru

rt 5bitová specifikace zdrojového/cílového registru

shamt shift amount, 5bitová míra posunutí, pomocný parametr

funct 6bitová specifikace funkce

imm 16bitová konstanta pro znaménkové operandy a offsety

addr 26bitová adresa skokových instrukcí, následně posunutá o dva bity doleva (specifikuje celkem 28 bitů, 2 nejnižší bity vždy nulové)

MIPS32 definuje seznam procesorových instrukcí. Pro cíle této práce je sada redukovaná na několik základních instrukcí odpovídající zadání semestrální práce z předmětu BI-APS.[5] Seznam těchto instrukcí ukazuje tabulka 2.2.

Poznámky k instrukcím:

— Písmena d, s a t odkazují na hodnoty uložené v těchto registrech.

— PC značí programový čítač – adresu současné instrukce.

— addu.qb a addu_s.qb jsou operace sčítající vektorově po jednotlivých bytech. Druhá zmíněná instrukce aplikuje saturované sčítání.

— Instrukce `jr` používá pro zjednodušení implementace jiný kód operace než originální MIPS32 architektura[6, str. 151].

— Instrukce `jal` má lehce jiný účinek než originální MIPS32 architektura (originálně $31 = PC+8)[6, str. 143].

instr	význam	kódování
add	d = s + t	`000000 sssss ttttt ddddd 00000 100000`
sub	d = s – t	`000000 sssss ttttt ddddd 00000 100010`
and	d = s & t	`000000 sssss ttttt ddddd 00000 100100`
or	d = s | t	`000000 sssss ttttt ddddd 00000 100101`
slt	d = (s<t) ? 1 : 0	`000000 sssss ttttt ddddd 00000 101010`
sllv	d = t<<s	`000000 sssss ttttt ddddd 00000 000100`
srlv	d = t>>s	`000000 sssss ttttt ddddd 00000 000110`
srav	d = (signed)t>>s	`000000 sssss ttttt ddddd 00000 000111`
addu.qb	$d_{31:24} = s_{31:24} + t_{31:24}$ $d_{23:16} = s_{23:16} + t_{23:16}$ $d_{15:8} = s_{15:8} + t_{15:8}$ $d_{7:0} = s_{7:0} + t_{7:0}$	`011111 sssss ttttt ddddd 00000 010000`
addu_s.qb	$d_{31:24} = s_{31:24} +_{sat} t_{31:24}$ $d_{23:16} = s_{23:16} +_{sat} t_{23:16}$ $d_{15:8} = s_{15:8} +_{sat} t_{15:8}$ $d_{7:0} = s_{7:0} +_{sat} t_{7:0}$	`011111 sssss ttttt ddddd 00100 010000`
addi	t = s + imm	`001000 sssss ttttt iiiiiiiiiiiiiiii`
lw	t = mem[s + imm]	`100011 sssss ttttt iiiiiiiiiiiiiiii`
sw	mem[s + imm] = t	`101011 sssss ttttt iiiiiiiiiiiiiiii`
beq	s==t → PC = PC + 4 + (imm << 2)	`000100 sssss ttttt iiiiiiiiiiiiiiii`
j	PC = (PC&0xf0000000) | (adr<<2)	`000010 jjjjjjjjjjjjjjjjjjjjjjjjjj`
jal	PC = (PC&0xf0000000) | (adr<<2) $31 = PC+4	`000011 jjjjjjjjjjjjjjjjjjjjjjjjjj`
jr	PC = s	`000111 sssss 00000 00000 00000 001000`

Tabulka 2.2: Podporovaná instrukční sada

2.2.2 Registry

Architektura pracuje s 32 univerzálními (general-purpose) 32bitovými registry. Registr 0 má vždy hodnotu 0. Pokud je registr 0 použit jako cíl zápisu výsledku nějaké operace, tak se tento výsledek ignoruje. Registr 31 je použit jako návratová adresa, zapisují do něj skokové instrukce se zápisem návratu (`jal`). Jinak ovšem může být použit jako jakýkoliv jiný registr. Ostatní registry mohou být použity univerzálně.[3, str. 40]

MIPS32 také zahrnuje speciální registry: PC, HI, LO. PC je programový čítač, je v něm uložena adresa současné instrukce. HI a LO jsou pomocné registry pro operace násobení a dělení.[3, str. 40] Násobení a dělení ovšem v není v naší redukované sadě podporované, takže tyto registry nejsou potřeba.

Ačkoli se dají registry použit univerzálně, existuje užívací konvence[7, str. 9], která rozděluje registry dle účelu:

— registr 0 ($zero) – nulový registr

— registr 1 ($at) – registr pro pomocné proměnné asembleru

— registry 2–3 ($v0–$v1) – návratové hodnoty

— registry 4–7 ($a0–$a3) – parametry volaných subrutin

— registry 8–15, 24–25 ($t0–$t9) – dočasné hodnoty

— registry 16–23 ($s0–$s7) – uložené hodnoty

— registry 26–27 ($k0–$k1) – registry pro kernel

— registr 28 ($gp) – ukazatel adresy globálních proměnných

— registr 29 ($sp) – ukazatel na vrchol zásobníku

— registr 30 ($fp) – rámcový ukazatel

— registr 31 ($ra) – návratová adresa ze subrutiny

2.3 Cache

Cache je pomocná skrytá paměť, do které se dočasně ukládají data z paměti, ke kterým se přistupuje, popřípadě data z okolních adres. Je to paměť zpravidla řádově menší než hlavní paměť, ale její přístupový čas je mnohem kratší. Využitím cache se zrychluje práce procesoru, protože se snižuje počet přístupů do hlavní paměti.[8, str. 17, 19]

2.3.1 Principy lokality

Cache využívá principy časové a prostorové lokality. Princip časové lokality znamená, že k datům, ke kterým se přistupovalo nedávno, se bude brzy přistupovat znovu. Příkladem je procházení dat v cyklu nebo rekurzivní volání stejných instrukcí.[8, str. 16]

Princip prostorové lokality znamená, že k datům, které se nacházejí poblíž dat, které jsou právě používané, se bude brzy přistupovat také. Příkladem je sekvenční přístup k datovým polím či sekvenční přístup k instrukcím programu.[8, str. 16]

2.3.2 Pojmy

Cache se týkají tyto pojmy: [8, str. 17]

Cache block je souvislý úsek hlavní paměti, který se do cache přenáší během jedné paměťové transakce. Číslo bloku v hlavní paměti se spočítá vydělením adresy bloku velikostí bloku.

Cache hit je přístup k datům nebo instrukcím, které jsou již v cache a jsou platná.

Cache miss je přístup k datům nebo instrukcím, které v cache nejsou a musí se přenést.

Hit rate je poměr počtu cache hitů k celkovému počtu přístupů.

2.3.3 Mapování

Mapování je způsob, jakým se adresám bloků v hlavní paměti přiřazuje místo v cache. Dle způsobu mapování se rozlišují tři typy cache: [8, str. 27]

— přímo mapovaná,

— částečně asociativní,

— plně asociativní.

V přímo mapované cache se umístění počítá jako n mod B, kde n je číslo bloku a B je počet setů (řádků) v cache. Pro uložení libovolného bloku hlavní paměti je určen pouze jediný blok cache. Protože víc bloků z hlavní paměti se mapuje na stejnou pozici v cache, musí se z důvodu identifikace bloku do cache zároveň ukládat číslo bloku z hlavní paměti, část původní adresy. To se označuje jako tag.[8, str. 20]

Částečně asociativní cache obsahuje několik cest (sloupců). Počet cest se označuje jako stupeň asociativity. Blokům je přiřazen set (řádek) funkcí n mod S, kde n je číslo bloku a S je počet setů. V rámci vybraného setu může být blok vložen do libovolné cesty.[8, str. 24]

Plně asociativní cache obsahuje jediný set, stupeň asociativity se rovná celkovému počtu bloků v cache. Číslo bloku nemá vliv na umístění bloku v cache. U všech bloků je tagem celé číslo bloku.[8, str. 26]

2.3.4 Formát zápisu v cache

Blok uložený v cache mívá tyto informace: [8, str. 32]

Validity bit indikuje, zda je obsah bloku vůbec platný.

Dirty bit indikuje, že v cache je jiná hodnota než v hlavní paměti.

Tag je index odpovídajícího bloku v hlavní paměti (adresa hlavní paměti vydělená délkou bloku).

Data je vlastní nakopírovaný blok z příslušné adresy hlavní paměti.

2.3.5 Strategie nahrazení

Když se vyskytne cache miss, je třeba natáhnout blok z hlavní paměti do cache. V takovém případě je třeba vybrat cestu, do které se požadovaný blok umístí. Obecně se cache nejdříve pokusí umístit blok do cesty, kde v požadovaném setu nejsou platná data. Pokud taková cesta neexistuje, cestu zvolí strategie nahrazení (replacement policy). [8, str. 35] Strategie nahrazení může být:

Náhodná (random) – zvolí se libovolná cesta.

LRU (Least recently used) – zvolí se nejdéle nepoužitý blok.

LFU (Least frequently used) – zvolí se nejméně používaný blok.

2.4 Verilog

Verilog je formální jazyk pro popis hardwaru a elektronických obvodů (Hardware definition language). Je popsán standardem IEEE Std 1364-2001.[9]

Verilog umožňuje modulární popis – obvod složený z jednotlivých modulů, které jsou popsány zvlášť. Moduly mají vstupní a výstupní rozhraní a přes tato rozhraní jsou mezi sebou propojené.

Obvod popsaný jazykem Verilog může být syntetizován do hardwaru nebo simulován. Při simulaci může být celý průběh zapsán do souboru VCD (value change dump, výpis změn hodnot).

2.4.1 Value change dump

Value change dump je formát souboru, který obsahuje informace o změnách hodnot uvnitř modulů a proměnných v průběhu simulace hardwaru. Je popsán spolu s jazykem Verilog standardem IEEE Std 1364-2001.[9]

Jednotlivé tokeny v souboru VCD jsou oddělené bílými znaky, klíčová slova začínají znakem $. Klíčovým slovem obecně začíná sekce, která je následně ukončena klíčovým slovem `$end`.[10]

Soubor VCD obsahuje:

— hlavičku,

— sekci definic,

— sekci změn

Hlavička obsahuje informace o simulátoru (`$version`), časové jednotce (`$timescale`) a datu (`$date`) simulace. Tyto údaje pro simulátor nebudou významné.

Sekce definic obsahuje definice proměnných (modulů a spojů). Moduly jsou zde definovány jako hierarchické rámce (`$scope`), rámec se ukončuje klíčovým slovem `$upscope`. Příklad definice modulu je:

```
$scope module name_of_module $end
<definice proměnných>
$upscope $end
```

V rámci modulů jsou definované proměnné, u kterých se definuje druh (`wire` nebo `reg`), bitová velikost, název rozhraní a identifikátor[4] spoje. Příklad definice proměnné typu `reg` o velikosti tři bity v rozhraní `inp` s přiřazeným identifikátorem `x)`:

```
$var reg 3 inp x) $end
```

Sekce definic je ukončena klíčovým slovem `$enddefinitions`.[10]

Sekce změn začíná po konci sekci definic. Změny zde patří do časových rámců. Časový rámec se uvozuje časovou značkou začínající znakem `#`. Například značka `#6` znamená, že všechny následující změny, dokud se nedojde k další časové značce, se udály v čase 6. Počáteční nastavení hodnot je zapsáno na počátku uvnitř `$dumpvars`. Změny 1bitových proměnných jsou zaznamenané jako hodnota následovaná identifikátorem (bez mezery), například `1?"` Změny vícebitových hodnot jsou zaznamenené jako písmeno `b` následované binárním zápisem hodnoty (bez mezery), mezerou a identifikátorem, například `b010 x)`[10]

[4] unikátní identifikátor skládající se z tisknutelných ASCII znaků

Kapitola 3

Analýza současných řešení

Zkoumal jsem současně dostupné MIPS simulátory, ve kterých jsem zkoušel spustit program napsaný v jazyce symbolických instrukcí, který by měl do paměti zapsat Fibonacciho posloupnost. Výše zmíněný program můžeme vidět na obrázku 3.1

```
main:
  addi  $t1,$zero,0
  addi  $t2,$zero,1
  addi  $t3,$sp,-8
cyc:
  sw    $t1,0($t3)
  addi  $t3,$t3,-4
  add   $t4,$t2,$t1
  addi  $t1,$t2,0
  addi  $t2,$t4,0
  j     cyc
  nop
```

Obrázek 3.1: Generátor Fibonacciho posloupnosti

Program se nejdříve inicializuje – zapíše první dvě hodnoty posloupnosti (0 a 1) do registrů `t1` a `t2`. Do registru `t3` zapíše hodnotu `$sp - 8`, což je adresa 2 slova před ukazatelem zásobníku. Registry `t1` a `t2` se používají pro ukládání současných hodnot posloupnosti, do registru `t3` se zapisuje adresa paměti, do které se má hodnota posloupnosti zapsat.

Potom se spustí cyklus, který začíná zapsáním hodnoty z posloupnosti do paměti. Adresa zápisu (`t3`) se pak posune o 4. Dále se spočítá následující hodnota posloupnosti sečtením hodnot v registrech `t1` a `t2`. Ta se uloží do pomocného registru `t4`. Nakonec se obsah registru `t1` přesune do registru `t2` a

nová hodnota v `t4` se uloží do `t1`. Program pak skočí na začátek cyklu. Tento cyklus se opakuje do nekonečna.

3.1 WeMips

WeMips je webová aplikace dostupná online[11]. Zobrazuje obsah registrů rozdělených podle účelu a obsah zásobníku (ukazuje se paměť mezi nejnižší zapsanou adresou a frame pointerem). Obsah paměti je zapsán po jednotlivých bytech jako čísla (0–255).

Umožňuje krokování po jednotlivých instrukcích a lze zde přímo editovat hodnoty v registrech a paměti za běhu. Stack pointer je nastaven automaticky. V případě zapsání daleko před stack pointer ovšem celá aplikace spadne. To jsem zjistil, když jsem se pokoušel začít zapisovat od adresy 0.

Na obrázku 3.2 je zobrazení paměti v simulátoru WeMips. Nahoře v obrázku můžeme vidět číslo 377 ($144 + 233$) zapsané po bytech 1 a 121 ($377 = 1 \times 256 + 121$).

386651045:	1
386651046:	121
386651047:	0
386651048:	0
386651049:	0
386651050:	233
386651051:	0
386651052:	0
386651053:	0
386651054:	144
386651055:	0
386651056:	0
386651057:	0
386651058:	89
386651059:	0

Obrázek 3.2: Zobrazení paměti ve WeMips

+ Jednoduché použití
+ Možná editace hodnot
+ Možnost krokování
− Paměť pouze po bytech
− Obtížné vyhledávání v paměti

3.2 VisualMips

VisualMips je webová aplikace dostupná online[12]. Umožňuje zobrazení paměti, registrů a programového čítače. Registry – kromě nultého, který má vždy hodnotu 0 – jsou zde obecné a rovnocenné. Instrukce musí být zapsány velkými písmeny, jako parametry instrukcí jsou zde jen číselné údaje (ne symbolické adresy a dolary). Například místo `addi $t3,$sp,-8` zde bylo třeba zapsat `ADDI 9,29,-8`. Nefungovaly mi zde skokové instrukce, takže jsem nemohl vytvořit cyklus.

+ Jednoduchost
+ Možná editace hodnot
+ Možnost krokování

– Nefunkčnost skokových instrukcí
– Nezvyklé identifikátory

3.3 Spim

Spim (QtSpim)[13] je multiplatformní simulátor. Neumožňuje přímo psát kód, instrukce se musí nahrát z externího souboru. Nahraný soubor se symbolickými instrukcemi se vloží doprostřed přednastavené „šablony“ – není zde možné psát přímo v programu, není možné mít čistý program bez této „šablony“.

Aplikace při načtení mého programu padala, dokud jsem nezměnil poslední `nop` na `addi $zero,$zero,0` Při krokování se to chovalo rozumně, po plném spuštění programu nastala chyba `ArithmeticOverflow`, což bylo očekávané vzhledem k povaze tohoto cyklu a stálému přičítání. V zobrazení registrů byly dobře vidět měnící se hodnoty. V zobrazení paměti se ovšem nedaly zobrazit rozsahy mimo User data segment, User stack a Kernel data segment.

+ Možnost krokování
+ Zobrazení hodnot registrů
+ Dobrý formát paměti
+ Ukazuje kompletní program

– Nemožnost editace
– Nezobrazuje paměť mimo určité rozsahy

3.4 MARS

MARS[14] je pokročilé multiplatformní vývojové prostředí, ve kterém lze psát i spouštět programy v jazyce symbolických instrukcí. Je zde možnost zobrazit (a upravit) obsah paměti a registrů.

Obsahuje mnoho doplňkových nástrojů, například konvertor čísel v plovoucí řádové řádce nebo animovaný „rentgen“ procesoru. Je zde i simulátor paměťové cache, ve kterém ovšem nelze zobrazit samotný obsah cache. MARS

také dokáže exportovat zkompilovaný program do souboru v hexadecimálním formátu. Toho jsem využil při vytváření vstupu pro vlastní simulátor.

Při krokování mého programu se vše jevilo správně. Po plném spuštění programu i zde dle očekávání nastala chyba `ArithmeticOverflow`.

+ Velmi pokročilé
+ Možnost krokování
+ Editor JSI
+ Možná editace hodnot
+ Export do hex formátu
+ Celá paměť
+ Mnoho doplňků

− Může působit složitě

Kapitola 4

Implementace

Implementace simulátoru se skládá ze dvou aplikací:

— Simulační jádro napsané v jazyce Verilog;

— Grafická aplikace napsaná v jazyce Java.

Tyto dvě aplikace mezi sebou interagují. Simulační jádro je zkompilované do formátu vvp – simulační formát softwaru iverilog. Grafická aplikace spouští simulační jádro s informací o umístění vstupních souborů. Aplikace je dokumentovaná pomocí nástroje Javadoc.

4.1 Simulační jádro

Simulační jádro je napsané v jazyce Verilog. Jsou ze popsány jednotlivé součásti procesoru a informace o tom, jak jsou propojené. Jako základ simulačního jádra byla použita šablona ze zadání 1. semestrální práce předmětu BI-APS[5], upravená pro potřeby této práce.

Simulační modul (na obrázku 4.1) v sobě obsahuje vrchní modul (`top`) a ovládá jeho procesorové hodiny a reset. Na počátku simulace nastaví výstup do souboru `output.vcd`. Přečte parametr simulace `+cyc`, který udává délku simulace (počet vykonaných instrukcí). Tento počet se vynásobí dvěmi, protože každá instrukce má ve skutečnosti dva cykly (jeden cyklus se zapnutými hodinami a jeden s vypnutými).

V prvních dvou cyklech simulace je zapnut reset, od třetího cyklu se vypne. Potom se periodicky každý cyklus zapínají či vypínají procesorové hodiny. Simulace nakonec skončí po $2 \times (n + 1)$ cyklech, kde n je vstupní parametr `+cyc`.

Vrchní modul `top` (na obrázku 4.2) zajišťuje propojení mezi samotným procesorem, datovou pamětí a instrukční pamětí. Jako vstup má procesorové hodiny a reset.

```
module simulation();
  reg clk;
  reg reset;

  integer cycles;
  integer res;

  top simulated_system(clk, reset);

  initial begin
    $dumpfile("output.vcd");
    $dumpvars;
    res = $value$plusargs("cyc=%d", cycles);
    cycles = cycles * 2;
    reset<=1;
    # 2;
    reset<=0;
    # cycles;
    $finish;
  end

  always begin
    clk<=1; # 1; clk<=0; # 1;
  end
endmodule
```

Obrázek 4.1: Simulační modul ve Verilogu

```
module top (input clk, reset);
  wire [31:0] data_to_mem, address_to_mem;
  wire write_enable;
  wire [31:0] pc, instruction, data_from_mem;

  inst_mem imem(pc[31:2], instruction);
  data_mem dmem(clk, write_enable,
      address_to_mem, data_to_mem, data_from_mem);
  processor CPU(clk, reset, pc, instruction, write_enable,
      address_to_mem, data_to_mem, data_from_mem);
endmodule
```

Obrázek 4.2: Vrchní modul simulačního systému ve Verilogu

```
module inst_mem (input [29:0] address,
  output [31:0] rd);

  reg [31:0] RAM[65535:0];
  reg [8*1024:0] memfile;
  integer res;
  integer i;

  initial begin
    for (i=0; i<65536; i=i+1)
      begin
        RAM[i] = 0;
      end
    res = $value$plusargs("insf=%s", memfile);
    $readmemh (memfile, RAM, 0, 65535);
  end
  assign rd=RAM[address];
endmodule
```

Obrázek 4.3: Instrukční paměť ve Verilogu

4.1.1 Instrukční paměť

Modul `inst_mem` (na obrázku 4.3) reprezentuje instrukční paměť. Jako vstup přijímá index instrukce (*adresa*/4), výstupem je 32bitové instrukční slovo. Samotná data jsou uložena v proměnné `RAM`, vejde se tam až 65536 32bitových slov.

Paměť se na počátku simulace vynuluje, následně se přečte adresa hexadecimálního souboru instrukcí z parametru `+insf` a zapíše se do proměnné `memfile`. Potom se načte soubor instrukcí do proměnné `RAM`.

4.1.2 Datová paměť

Modul `data_mem` (na obrázku 4.4) reprezentuje datovou paměť. Jako vstup přijímá adresu, procesorové hodiny, data k zapsání a příznak zápisu. Výstupem je 32bitové datové slovo. Samotná data jsou uložena v proměnné `RAM`, vejde se tam až 65536 32bitových slov.

Paměť se na počátku simulace vynuluje, následně se přečte adresa hexadecimálního souboru počátečních dat z parametru `+datf` a zapíše se do proměnné `memfile`. Potom se načte soubor dat do proměnné `RAM`.

Do datové paměti se mohou v průběhu simulace zapsat data. Děje se tak na náběžné hraně procesorových hodin – zapíšou se data ze zápisového vstupu, pokud je zapnutý příznak zápisu.

```
module data_mem (input clk, we,
  input  [31:0] address, wd,
  output [31:0] rd);

  reg [31:0] RAM[65535:0];
  reg [8*1024:0] memfile;
  integer res;
  integer i;

  initial begin
    for (i=0; i<65536; i=i+1)
      begin
        RAM[i] = 0;
      end
    res = $value$plusargs("datf=%s", memfile);
    $readmemh (memfile, RAM, 0, 65535);
  end

  assign rd=RAM[address[31:2]];

  always @ (posedge clk)
    if (we)
       RAM[address[31:2]]<=wd;
endmodule
```

Obrázek 4.4: Datová paměť ve Verilogu

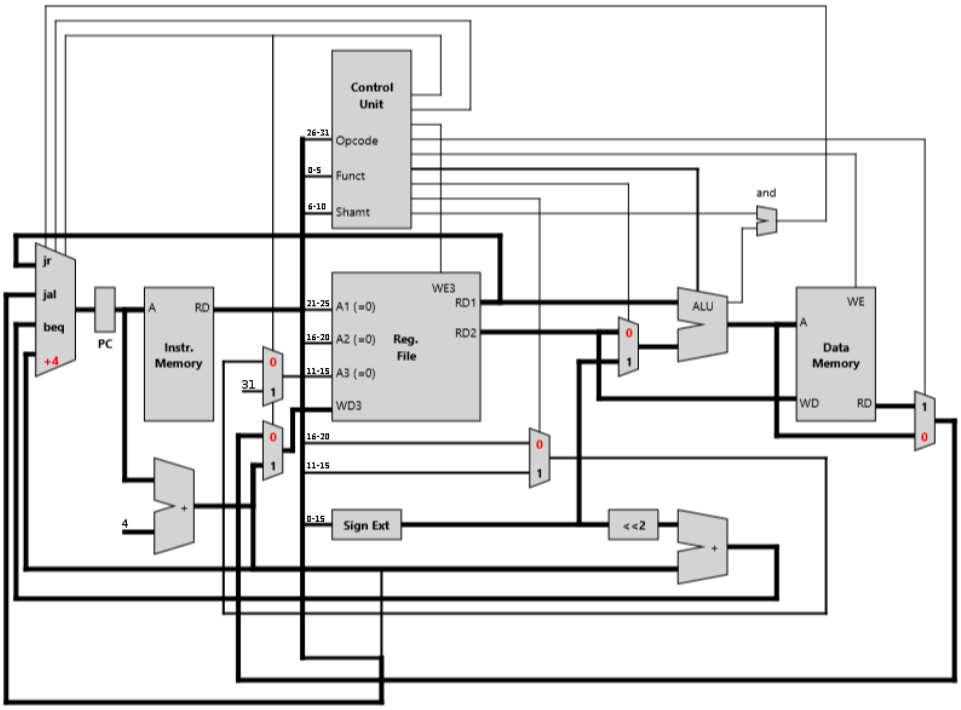
4.1.3 Procesor

Modul `processor` popisuje propojení jednotlivých procesorových modulů. Jsou propojené dle schématu na obrázku 4.6. (Toto schéma obsahuje i datovou a instrukční paměť, i když nejsou přímo součástí procesoru). Zápis těchto propojení je na obrázku 4.5.

```
controlunit cunit(Instr[31:26], Instr[5:0], Instr[10:6],
  AluControl, RegWrite, RegDst, AluSrc, Branch,
  MemWrite, MemToReg, PCSrcJal, PCSrcJr);
mux32_4 mux1(SrcA, PCJal, PCBranch,
  PCPlus4, PCSrcJr, PCSrcJal, PCSrcBeq, PCIn);
resregister32 pc(PCIn, clk, reset, PCOut);
adder32 plus4(PCOut, four, c0, PCPlus4, c1);
mux5_2 mux2(WriteReg, stackc, PCSrcJal, A3Wire);
mux32_2 mux3(Result, PCPlus4, PCSrcJal, WD3Wire);
registerfile32 regs(Instr[25:21], Instr[20:16], A3Wire,
  RegWrite, clk, WD3Wire, SrcA, WriteData, reset);
extension32 ext(Instr[15:0], SignImm);
mux5_2 mux4(Instr[20:16], Instr[15:11], RegDst, WriteReg);
mux32_2 mux5(WriteData, SignImm, AluSrc, SrcB);
alu32 alu(SrcA, SrcB, AluControl,
  AluOut, Zero, carry, overflow);
mux32_2 mux6(AluOut, ReadData, MemToReg, Result);
multiply4 mult(SignImm, Times4);
adder32 addtobranch(Times4, PCPlus4, c0, PCBranch, c2);

assign PC = PCOut;  // vstup do instrukční paměti
assign Instr = instruction;  // výstup z instrukční paměti
assign four = 4;  // délka instrukčních slov
assign stackc = 31;  // index registru z návratovou adresou
assign c0 = 0;  // carry alu
assign PCSrcBeq = Zero & Branch;
assign PCJal = (PCPlus4 & 32'hF000_0000) +
  4 * (Instr & 32'h03FF_FFFF);
assign WE = MemWrite;  // vstup do datové paměti
assign ReadData = data_from_mem;  // výstup z datové paměti
assign data_to_mem = WriteData;  // vstup do datové paměti
assign address_to_mem = AluOut;  // vstup do datové paměti
```

Obrázek 4.5: Zapojení modulů ve Verilogu



Obrázek 4.6: Schéma procesoru

4.1.4 Řadič

Řadič (Control unit) je modul, který řídí tok dat v procesoru. Je implementován jako kombinační obvod. Vstupy řadiče jsou:

opcode kód operace, bity 26–31 z instrukce;

funct typ ALU operace, bity 0–5 z instrukce;

shamt bity 6–10 z instrukce.

Výstupy řadiče jsou:

regwrite zda se zapisuje do registru;

regdst cílový registr pro zápis: pokud je 1, zapisuje se do registru na adrese dané bity 11–15, jinak 16–20;

alusrc zdroj druhého operandu ALU: pokud je 1, je jím hodnota z bitů 0–15, jinak je zdrojem hodnota druhého[1] registru;

branch zda má programový čítač přičíst konstantu v bitech 0–15, pokud výsledek operace ALU je nula;

memwrite zda se zapisuje do datové paměti;

memtoreg zda je zdrojem zapisovaných dat do registru datová paměť (nebo výsledek z ALU);

jump zda se má programový čítač změnit na konstantu v bitech 0–15;

jr zda se má programový čítač změnit na hodnotu v prvním[1] registru;

alucontrol 4bitový výstup – operace, kterou má vykonat ALU.

Výstupy řadiče kromě `alucontrol` závisí pouze na kódu operace. Tato logika je zobrazena v tabulce 4.1. Výstup `alucontrol` závisí na hodnotách `funct` a `shamt` a na mezivýsledku `aluop` (také v tabulce 4.1). Tento výstup se dělí podle `aluop` mezivýsledku:

- `aluop` je 0 → `alucontrol` je 0010 (sčítání);
- `aluop` je 1 → `alucontrol` je 0110 (odčítání);
- `aluop` je 2 → `alucontrol` závisí na `funct`, dle tabulky 4.2;
- `aluop` je 3 → `alucontrol` závisí na `funct` a `shamt`, zde přijímá pouze `funct` 010000:

[1] Prvním se myslí registr daný adresou ve vstupu A1 (bity 21–25 z instrukce) v registrovém poli. Druhým je myšlený registr daný adresou ve vstupu A2 (bity 16–20 z instrukce).

– `shamt` je 00000 → `alucontrol` je 1000 (vektorové sčítání).

– `shamt` je 00100 → `alucontrol` je 1001 (saturované vektorové sčítání).

opcode	rw	rd	alusrc	br	mw	mr	jmp	jr	aluop
000000 (R typ)	1	1	0	0	0	0	0	0	2
100011 (lw)	1	0	1	0	0	1	0	0	0
101011 (sw)	0	0	1	0	1	0	0	0	0
000100 (beq)	0	0	0	1	0	0	0	0	1
001000 (addi)	1	0	1	0	0	0	0	0	0
000010 (j)	0	0	0	0	0	0	1	0	0
000011 (jal)	1	0	0	0	0	0	1	0	0
000111 (jr)	0	0	0	0	0	0	0	1	0
011111 (adduq)	1	1	0	0	0	0	0	0	3

Tabulka 4.1: Kombinační logika řadiče

funct	alucontrol
100000	0010 (sčítání)
100010	0110 (odčítání)
100100	0000 (konjunkce)
100101	0001 (disjunkce)
101010	0111 (menší než)
000100	1100 (bitový posun doleva)
000110	1110 (bitový posun doprava)
000111	1111 (aritmetický bitový posun doprava)

Tabulka 4.2: ALU operace dle funct

4.1.5 Aritmeticko-logická jednotka

ALU je modul, který provádí početní a logické operace. Vstupem do ALU jsou dva 32bitové operandy a 4bitový typ operace. První operand je vždy z prvního registru, druhý operand je buď z druhého registru nebo z konstanty v instrukci (rozhoduje výstup `alusrc` z řadiče). O typu operace rozhoduje řadič výstupem `alucontrol`. Typy podporovaných operací jsou v tabulce 4.3 Výstupem je 32bitový výsledek operace a 1bitový indikátor nulového výsledku.

alucontrol	operace
0010	sčítání
0110	odčítání
0000	konjunkce (po bitech)
0001	disjunkce (po bitech)
0111	menší než
1000	vektorový součet (po bytech)
1001	saturovaný vektorový součet (po bytech)
1100	logický bitový posun doleva
1110	logický bitový posun doprava
1111	aritmetický bitový posun doprava

Tabulka 4.3: Operace podporované ALU

4.1.6 Registrové pole

Registrové pole (`registerfile32`) je modul, který obsahuje 32 32bitových registrů. Popsán je na obrázku 4.7. Vstupy jsou 5bitové adresy `a1`– `a3`, 32bitová data k zapsání, 1bitový příznak zápisu, procesorové hodiny a reset. Výstupy jsou data z registrů (`rd1` a `rd2`).

Registry jsou na začátku inicializovány na hodnotu 0. Pokud je příznak zápisu zapnutý, lze zapisovat do registru na adrese dané vstupem `a3`. Výjimkou je registr 0, do kterého nelze zapisovat. Ten má vždy hodnotu 0. Zápis se děje při náběžné straně procesorových hodin.

Z registrového pole se dá ve výstupech `rd1` a `rd2` přečíst obsah registrů na adresách, které jsou dané vstupy `a1` a `a2`.

4.1.7 Ostatní moduly

Programový čítač je implementován jako samostatný registr. Je inicializovaný na nulu. Při náběžné hraně procesorových hodin se jeho hodnota přepíše výstupem z čítačového multiplexoru `mux1`. Jeho výstup je adresním vstupem v instrukční paměti.

Sčítačky jsou moduly, které sečtou dva vstupní operandy a výsledek pošlou na výstup. Jsou zde dvě sčítačky. První přičítá 4 (délka instrukčního slova)

```
module registerfile32(input [4:0] a1, a2, a3,
    input we3,
    input clk,
    input [31:0] wd3,
    output [31:0] rd1, rd2,
    input reset);

  // deklaruje 32 32bitových registrů
  reg [31:0] val [31:0];

  // inicializace na nulu
  integer i;
  initial begin
    for (i=0;i<=31;i=i+1)
      val[i] = 0;
  end

  // zápis do registrů, pokud se nejedná o registr 0
  always @ (posedge clk, posedge reset)
    if (reset)
      for (i=0;i<=31;i=i+1)
        val[i] = 0;
    else if (we3 & a3 != 0)
      val[a3] <= wd3;

  // přiřazení výstupů
  assign rd1 = val[a1];
  assign rd2 = val[a2];
endmodule
```

Obrázek 4.7: Registrové pole

k programovému čítači. Druhá sčítačka (větvící) přičítá konstantu s rozšířením znaménka k adrese následující instrukce pro případ větvení běhu programu (instrukce `beq`).

Multiplexory jsou moduly řízené řadičem, které na výstup přiřazují jeden ze vstupů. Je zde celkem 6 multiplexorů:

mux1 4vstupový, vstup programového čítače;

mux2 adresa registru pro zápis – 31 nebo výstup multiplexoru `mux4`;

mux3 data pro zápis do registru – adresa následující instrukce nebo výstup multiplexoru `mux6`;

mux4 vstup do `mux2` – instrukční bity 11–15 nebo 16–20;

mux5 druhý operand ALU – obsah druhého registru nebo výstup znaménkového rozšiřovače;

mux6 vstup do `mux3` – výstup z datové paměti nebo výsledek ALU.

Rozšiřovač znaménka (Sign extension) je modul který přijímá 16bitovou hodnotu a mění ji na 32bitovou hodnotu se stejným znaménkem – do nových bitů zkopíruje hodnotu nejvýznamnějšího bitu. Jeho výstup je možným druhým operandem (přes `mux5`) aritmeticko-logické jednotky a vstupem větvící sčítačky.

4.2 Hlavní třída aplikace

Hlavní třída grafické aplikace je třída `MipsSimulator`. Obsahuje záznam o běhu programu, historii datových struktur (pamětí a registrů) a zajišťuje komunikaci se simulačním jádrem.

4.2.1 Simulace

Simulaci zajišťuje metoda `simulate` (na obrázku 4.8). Tato metoda:

1. Inicializuje pole registrů, datovou paměť a instrukční paměť.
2. Spustí simulační jádro.
3. Přečte výstup simulace.
4. Zapíše historii změn do registrů, datové paměti a instrukční paměti.

Pokud vše proběhne bez problému, metoda vrátí `true`. Jestliže se něco nezdaří, metoda se ukončí a vrátí `false`.

```
public boolean simulate() {
  regs = new RegisterFile();
  imem = new InstructionMemory(65536);
  dmem = new DataMemory(65536);

  if (!runvvp())
    return false;

  if (!readRecord())
    return false;

  if (!regs.init(rec, lastFrame()))
    return false;

  imem.load(getProgramPath());
  dmem.load(getDataPath());
  dmem.readChanges(rec, lastFrame());

  return true;
}
```

Obrázek 4.8: Třída MipsSimulator - Simulace

4.2.1.1 Spuštění simulačního jádra

Třída má ve své členské proměnné uloženou adresu interpreteru vvp. Při spuštění simulace se zkontroluje, zda adresa není prázdná. Název souboru zde musí začínat řetězcem „vvp“ jako kontrola toho, že byla vybrána správná aplikace. Tato kontrola samozřejmě nevyřeší vše, slouží spíše k zabránění spuštění nesprávné aplikace omylem.

Následně se spustí interpreter s parametry simulace:

— umístění zkompilovaného simulačního jádra (`proc.vvp`, v současném adresáři nebo v adresáři s JAR archivem);

— umístění souboru s počáteční datovou pamětí;

— umístění souboru s programem (instrukční pamětí);

— počet instrukcí (jak dlouho má simulace běžet).

Potom se čeká, než se simulace dokončí. Pokud simulace proběhne v pořádku, interpreter skončí s návratovým kódem 0.

```
// kontrola, jestli uživatel omylem nevybral špatnou aplikaci
if (fName.length() < 3 ||
    !fName.substring(0, 3).toLowerCase().equals("vvp")) {
  return false;
}

// spuštění interpreteru
Process vvpProc = new ProcessBuilder(
  vvpPath,
  interpreter.getCanonicalPath(),
  String.format("+datf=%s", dataPath),
  String.format("+insf=%s", programPath),
  String.format("+cyc=%d", instrCount)).start();

// čekání na ukončení
int retCode = vvpProc.waitFor();
if (retCode != 0) return false;
```

Obrázek 4.9: Třída MipsSimulator - Spuštění simulace

4.2.1.2 Čtení výstupu simulace

Je třeba přečíst soubor VCD, který vytvořila předchozí simulace. Ten by se měl nacházet v současném adresáři pod názvem „output.vcd“. Před přečtením se čeká jednu sekundu, protože výstup by ještě nemusel být přístupný. Pokud se nepodaří soubor přečíst, tak se pokusí přečíst znovu (maximálně 10krát, vždy s 1s prodlevou). Soubor se analyzuje třídou `RecordReader`, která z něj vytvoří `ProgramRecord`.

4.2.2 Přístup k datům

`MipsSimulator` umožňuje přístup k uloženým záznamům simulace prostřednictvým svých metod:

getRecord vrátí záznam běhu programu.

getRegVal získá hodnotu reigstru daného indexu v daném čase.

getInstruction získá instrukci na dané adrese.

getData získá data z datové paměti na dané adrese v daném čase.

lastFrame zjistí index posledního cyklu v simulaci.

```
public class ProgramRecord {
  private HashMap<String, ProcessorModule> modules;
  private HashMap<String, Wire> wires;

  public ProgramRecord() {
    this.modules = new HashMap<>();
    this.wires = new HashMap<>();
  }

  public void addModule(ProcessorModule module) {
    modules.put(module.getName(), module);
  }

  public void addWire(Wire wire) {
    wires.put(wire.getIdentifier(), wire);
  }

  public ProcessorModule getModule(String name) {
    if (!modules.containsKey(name)) return null;
    return modules.get(name);
  }

  public Wire getWire(String identifier) {
    if (!wires.containsKey(identifier)) return null;
    return wires.get(identifier);
  }
}
```

Obrázek 4.10: Třída ProgramRecord

4.3 Záznam běhu programu

Pro záznam běhu simulace programu slouží třída `ProgramRecord`. Třída obsahuje procesorové moduly a spoje, které jsou v něm uloženy v asociativních strukturách. Moduly jsou identifikovány jejich interními názvy. Spoje jsou identifikovány jednoduchými unikátními identifikátory, které jim přidělí interpreter Verilogu.

4.3.1 Záznam změny hodnoty

Změny hodnoty ve spojích v rámci běhu programu jsou zaznamenány v objektech třídy `ValueChangeSnapshot`. Tyto objekty obsahují hodnotu uloženou ve

```
private TreeMap<Integer, ValueChangeSnapshot> valueHistory;

public void addChangeSnapshot(ValueChangeSnapshot snapshot,
      int time) {
  valueHistory.put(time, snapshot);
}

public ValueChangeSnapshot getValueSnapshot(int time) {
  int lastChangeTime = valueHistory.floorKey(time);
  return valueHistory.get(lastChangeTime);
}
```

Obrázek 4.11: Třída Wire – zápis změn hodnot a přístup k hodnotám

formě `java.util.BitSetu` – vektoru bitů s proměnlivou velikostí. Snapshot může být vytvořen z:

— booleanu;

— čísla;

— řetězce jedniček a nul.

Ze snapshotu lze zpátky získat jednotlivé bity nebo celé číslo.

4.3.2 Spoj

Spoj (`Wire`) je třída, která reprezentuje spoje v procesoru. Každý `Wire` má svůj arbitrární kompaktní identifikátor, který vytvoří interpreter verilogu a nachází se ve VCD souboru. Dále obsahuje bitovou velikost spoje. Hlavní vlastnost spoje je historie změn hodnot. Ta je uložena v kolekci `TreeMap`.

Přístup k hodnotám je v kódu na obrázku 4.11. Metodou `addChangeSnapshot` se do objektu spoje přidá hodnota v daném čase. Ta se uloží do asociativní mapy. Čas změny je zde klíčem. Metoda `getValueSnapshot` se používá k zjištění hodnoty spoje v požadovaném čase. Využije se zde `floorKey` – metoda `TreeMapy`, která vrátí nejbližší nižší klíč, tedy čas poslední změny. Přístupem do mapy přes čas poslední změny získáme současnou hodnotu.

4.3.3 Modul procesoru

Modul procesoru (`ProcessorModule`) je třída, která reprezentuje jednotlivé součástky procesoru. Obsahuje jméno, které slouží jako identifikátor v rámci procesoru, a kolekci (`HashMap`) spojů, které jsou v rámci modulu identifikovány názvem rozhraní v modulu (narozdíl od `ProgramRecordu`, který má spoje označeny jejich interními identifikátory).

```
private HashMap<String, Wire> wires;

public void addWire(String socketName, Wire wire) {
  wires.put(socketName, wire);
}

public Wire getWire(String socketName) {
  if (!wires.containsKey(socketName)) return null;
  return wires.get(socketName);
}
```

Obrázek 4.12: Třída ProcessorModule – přístup ke spojům

4.4 Čtečka záznamu

Čtečka záznamu (`RecordReader`, na obrázku 4.13) je třída, která zajišťuje přečtení a zpracování souboru VCD (Value change dump), což je výstup simulace, kterou provádí interpreter Verilogu. Z toho čtečka vygeneruje `ProgramRecord`.

Vstupním bodem čtečky je statická metoda `readRecordFromStream`, která vytvoří instanci čtečky, spustí čtení ze streamu, který získá jako parametr – zavolá syntaktický parser – a vrátí záznam, který získá. V případě, že ve zpracování nastane chyba, vyhodí výjimku `BadSyntaxException`.

4.4.1 Token

Token je datová třída, která reprezentuje samostatné významové „slovo“ ve VCD souboru. Token může být:

prostý token bez symbolického významu – hodnota;

symbol token označený znakem dolaru;

EOF speciální token označující konec vstupu.

Prosté tokeny a symboly se tvoří přes `public` konstruktor se `String` parametrem. Token označující konec vstupu se získá statickou metodou `EOF`, která vytvoří token `private` konstruktorem.

Token má následující vlastnosti:

symbol boolean, který říká, zda je daný token symbolem;

eof boolean, který říká, zda je daný token EOF;

raw celý textový obsah tokenu;

```
public class RecordReader {

  private InputStream stream;

  private RecordReader(InputStream is) {
    stream = is;
  }

  public static ProgramRecord readRecordFromStream
      (InputStream is)
      throws BadSyntaxException {
    RecordReader rr = new RecordReader(is);
    return rr.readProgramRecord();
  }

  private ProgramRecord readProgramRecord()
      throws BadSyntaxException {
    return new SyntaxParser(stream).parse();
  }
}
```

Obrázek 4.13: Třída RecordReader – čtečka VCD záznamu

symbolname název symbolu – text bez dolaru, `null` pokud token není symbolem.

4.4.2 Lexikální analyzátor

Lexikální analyzátor (`TokenParser`, na obrázku 4.16) je třída, která čte vstup z VCD souboru, ke kterému v konstruktoru dostane stream, a tento vstup rozděluje na jednotlivé `Token`y.

Parser používá pro čtení třídu `java.util.Scanner`, jehož metoda `next` získává řetězec ze vstupu, dokud nenalezne bílý znak, čímž je vhodný pro toto použití – VCD soubor má tokeny rozdělené bílými znaky.

`TokenParser` má metodu `getNextToken`, která získá ze streamu následující token a posune `Scanner` dál. Pokud je již celý vstup přečtený, metoda vrátí EOF token.

4.4.3 Syntaktický parser

`SyntaxParser` je třída, která zajišťuje syntaktické zpracování VCD záznamu. Je volána metodou `readProgramRecord` ze třídy `RecordReader`. Získává jed-

```
// public konstruktor vytváří prostý token nebo symbol
public Token(String content) {
  if (content == null || content.length() == 0)
    throw new BadSyntaxException(...);

  this.raw = content;

  // pokud začíná dolarem, je to symbol
  if (content.charAt(0) == '$')  {
    symbol = true;
    symbolName = content.substring(1);
  } else {
    symbol = false;
    symbolName = null;
  }
}
```

Obrázek 4.14: Třída Token – konstruktor

```
// private konstruktor vytváří EOF token
private Token() {
  eof = true;
  symbol = false;
}

public static Token EOF() {
  return new Token();
}
```

Obrázek 4.15: Třída Token – konstruktor EOF tokenu

```
class TokenParser {
  private Scanner sc;

  public TokenParser(InputStream stream) {
    sc = new Scanner(stream);
  }

  public Token getNextToken()
  {
    if (!sc.hasNext())
      return Token.EOF();

    Token t;
    t = new Token(sc.next());
    return t;
  }
}
```

Obrázek 4.16: Třída TokenParser

notlivé `Token`y od lexikálního analyzátoru (`TokenParser`) a skládá z nich záznam `ProgramRecord`. Pokud se parsování nepodaří (syntaktická chyba), parser se ukončí výjimkou `BadSyntaxException`.

4.4.3.1 Získávání tokenů

Syntaktický parser má metodu `fetchNextToken`, která získá následující token z lexikálního analyzátoru a uloží ho do členské proměnné. Dále jsou zde kontrolní metody `consumeExpectedToken` a `consumeExpectedSymbol`. Tyto metody zkontrolují, zda současný token odpovídá očekávanému tokenu, popřípadě symbolu. Pokud srovnání uspěje, přečte se a uloží se další token. Pokud neuspěje, parser se ukončí výjimkou `BadSyntaxException`.

4.4.3.2 Čtení VCD souboru

Syntaktický parser provádí syntaktickou analýzu shora dolů (top-down). Průběh analýzy je následující:

1. Vytvoří se nová instance `ProgramRecord`u.
2. Přečte se první token ze vstupu.
3. Analyzuje se hlavička VCD souboru (`parseHeader`).

```
private TokenParser _tokener;
private Token _currentToken;

private void fetchNextToken() {
  _currentToken = _tokener.getNextToken();
}

private void consumeExpectedSymbol(String symbol) {
  if (!_currentToken.hasSymbol())
    throw new BadSyntaxException(...);
  if (!_currentToken.getSymbolName().equals(symbol))
    throw new BadSyntaxException(...);
  _currentToken = _tokener.getNextToken();
}

private void consumeExpectedToken(String content) {
  if (!_currentToken.getRaw().equals(content))
    throw new BadSyntaxException(...);

  _currentToken = _tokener.getNextToken();
}
```

Obrázek 4.17: Třída SyntaxParser – získávání a kontrola Tokenů

4. Analyzuje se sekce definic (`parseDefinitions`).
5. Analyzuje se sekce změn hodnot (`parseChanges`).
6. Vrátí se hotový `ProgramRecord`.

Hlavička VCD souboru neobsahuje žádné informace, které by byly pro zkoumání běhu programu důležité. Analyzuje se pouze z důvodu syntaktické správnosti, nic z hlavičky se do `ProgramRecord`u nezapisuje.

4.4.3.3 Čtení sekce definic

Sekce definic souboru VCD obsahuje definice jednotlivých modulů procesoru. Samotné moduly jsou vždy uvnitř scopu. Parser zde tedy hledá klíčový symbol `scope`, za nímž musí následovat token `module`. Obsah modulu se pak načte funkcí `parseModule` a modul se přidá do záznamu programu.

Tato sekce se analyzuje v cyklu, který se ukončí, když parser dojde k symbolu `enddefinitions`, který značí konec sekce definic. Parser poté přečte symbol `end`, který zde musí být, a vystoupí ze sekce definic. V případě, že se nalezne neočekávaný symbol, parser se ukončí výjimkou `BadSyntaxException`.

```
private ProgramRecord parseVcd() {
  _record = new ProgramRecord();
  fetchNextToken();

  parseHeader();
  parseDefinitions();
  parseChanges();

  return _record;
}
```

Obrázek 4.18: Třída SyntaxParser – analýza vnější struktury VCD

Totéž se stane v případě, že v definicích bude token, který není symbolem (prostý nebo EOF).

4.4.3.4 Čtení definice modulu

Funkce `parseModule` (na obrázku 4.20) vytvoří nový `ProcessorModule` s názvem podle obsahu prvního tokenu. V rámci čtení modulu může parser zpracovat symboly:

upscope přeskočí se spolu s ukončovacím symbolem `end`, protože simulátor scopování modulů řešit nepotřebuje (podmoduly se nečtou).

var přečte definici spoje, který je zde zapojený – zavolá funkci `parseWire`.

Analyzuje se zápis modulu, dokud se nedojde k symbolu, který sem již nepatří – v ten moment se vystoupí z funkce a vrátí se vytvořený modul.

4.4.3.5 Čtení definice proměnné

Funkce `parseWire` nejprve z prvního tokenu přečte typ proměnné:

reg interní proměnná modulu;

wire spoj mezi moduly;

integer interní číselná proměnná, ignoruje se, do modulu se nezapisuje.

Z dalších tokenů se přečte:

1. bitová velikost spoje;
2. interní identifikátor spoje;
3. název rozhraní modulu, kde je spoj zapojen.

```
private void parseDefinitions() {
  defCycle: while(true)
  {
    if (!_currentToken.hasSymbol())
      throw new BadSyntaxException(...);
    switch (_currentToken.getSymbolName()) {
      case "scope":
        consumeExpectedSymbol("scope");
        consumeExpectedToken("module");
        ProcessorModule pm = parseModule();
        _record.addModule(pm);
        break;
      case "enddefinitions":
        consumeExpectedSymbol("enddefinitions");
        consumeExpectedSymbol("end");
        break defCycle;
      default:
        throw new BadSyntaxException(...);
    }
  }
}
```

Obrázek 4.19: Třída SyntaxParser – analýza sekce definic

Následovat musí ukončovací symbol `end`. Spoj se přidá do kolekce spojů daného modulu. Zapíše se také do kolekce spojů celého záznamu, pokud zde již není.

4.4.3.6 Čtení sekce změn

Na počátku čtení změn nastaví funkce `parseChanges` (na obrázku 4.21) čítač času na nulu. Potom čte změny, dokud nedojde na konec souboru (token EOF). Pro zjednodušení ignoruje symboly `dumpvars` a `end` – tyto symboly zde nijak význam obsahu nemění.

Pokud parser přečte počátek časové značky (znak `#`), očekává dále číselnou hodnotu času. Nastaví čítač času na tuto hodnotu. Pokud se zde číselná hodnota nenachází, následuje výjimka `BadSyntaxException`.

Když se zde nenachází časová značka ani ignorovaný symbol, zavolá parser funkci `parseSingleChange`, která přečte změnu hodnoty ve spoji označeném interním identifikátorem, z této změny vytvoří `ValueChangeSnapshot` a vloží ho do spoje.

```
private ProcessorModule parseModule() {
  ProcessorModule pm =
    new ProcessorModule(_currentToken.getRaw());
  fetchNextToken();
  consumeExpectedSymbol("end");

  while(true)
  {
    if (!_currentToken.hasSymbol())
      throw new BadSyntaxException(...);

    switch (_currentToken.getSymbolName()) {
      case "upscope":
        consumeExpectedSymbol("upscope");
        consumeExpectedSymbol("end");
        break;
      case "var":
        consumeExpectedSymbol("var");
        parseWire(pm);
        consumeExpectedSymbol("end");
        break;
      default:
        return pm;
    }
  }
}
```

Obrázek 4.20: Třída SyntaxParser – analýza definice modulu

4.5 Registr

`Register` je jednoduchá třída, která slouží k ukládání historie hodnoty jednotlivých registrů během simulace programu. Používá k ukládání historie hodnot kolekci `TreeMap`, jednotlivé hodnoty jsou uložené ve formě objektů třídy `ValueChangeSnapshot`. Při vytvoření instance registru se do času -1 zapíše jako implicitní inicializace hodnota 0.

Třída `Register` také obsahuje statickou metodu `getRegisterName` pro získání konvenčního názvu MIPS registru podle zadaného indexu. Ta se používá disassemblerem a tabulkou hodnot v registrech.

```
private void parseChanges() {
  timeCounter = 0;

  while(!_currentToken.isEof())
  {
    if (_currentToken.hasSymbol() &&
        (_currentToken.getSymbolName().equals("dumpvars") ||
        _currentToken.getSymbolName().equals("end"))) {
      fetchNextToken();
      continue;
    }
    if (_currentToken.getRaw().charAt(0) == '#') {
      try
      {
        timeCounter = Integer.parseInt
          (_currentToken.getRaw().substring(1));
      }
      catch (Exception ex)
      {
        throw new BadSyntaxException(...);
      }
      fetchNextToken();
      continue;
    }
    parseSingleChange();
  }
}
```

Obrázek 4.21: Třída SyntaxParser – analýza sekce změn

4.6 Registrové pole

`RegisterFile` je třída, která obsahuje jednotlivé registry a zajišťuje zapsání do jejich historie. Při inicializaci vytvoří 32 instancí třídy `Register`. Poté přečte záznam běhu programu (`ProgramRecord`), ze kterého získá informace o historii modulu registrového pole („regs“). Obsah registrů se obecně nenachází v souboru VCD, takže se musí dopočítat podle přístupu k registrovému poli.

Zápis je zobrazen na obrázku 4.22. Iteruje se přes celou historii simulace, a pokud je v daném čase nastaveno „we3“ (write enable) rozhraní v registrovém poli, tak se přečte adresa registru z rozhraní „a3“ a do historie příslušného registru se zapíše datová hodnota z rozhraní „wd3“.

```
ProcessorModule rfMod = rec.getModule("regs");
Wire enableWire = rfMod.getWire("we3");
Wire dataWire = rfMod.getWire("wd3");
Wire indexWire = rfMod.getWire("a3");
for (int i = 0; i <= lastFrame; i++) {
  if (enableWire.getValueSnapshot(i).getBit(0)) {
    int index = (int) indexWire.getValueSnapshot(i).getLong();
    regs[index].addChangeSnapshot
        (dataWire.getValueSnapshot(i), i);
  }
}
```

Obrázek 4.22: Třída RegisterFile – zápis změn do registrů

4.7 Instrukční paměť

Instrukční paměť je reprezentována třídou `InstructionMemory`. Využívá toho, že je instrukční paměť oddělená od datové paměti a že je jen pro čtení. Nemusí si tedy pamatovat žádnou historii. Obsahuje jen jednoduché pole instrukcí ve formě čísel `long`, které se načte po simulaci z paměťového souboru.

4.8 Datová paměť

Datová paměť je reprezentována třídou `DataMemory`. Ta si musí pamatovat celou historii hodnot, protože hodnoty se v čase běhu mohou měnit. Pro ukládání historie používá objekty `Register`.

Počáteční stav paměti se, stejně jako u instrukční paměti, načte po simulaci z paměťového souboru. Následné změny v paměti se vypočítají podle přístupu k modulu datové paměti (na obrázku 4.23), podobně jako se počítají změny v registrech.

4.9 Načítač paměťových souborů

Načítač `MemoryFileReader` je třída, která se stará o načítání souborů s počáteční pamětí v hexadecimálním formátu. Vstupní metodou je zde statická metoda `readMemoryFile`, která vnitřně vytvoří instanci třídy a zavolá na ní metodu `read`. Vstupem je `string` s názvem souboru, výstupem je pole paměťových slov přečtených z daného souboru. V případě chyby nebo nenalezení souboru se vrátí `null`.

Načítač funguje jako stavový automat, který ze vstupu odstraní komentáře a znaky, které nejsou hexadecimálními číslicemi. Výstup z automatu je řetězec hexadecimálních čísel oddělených mezerami. Tento řetězec se potom rozdělí

```
ProcessorModule rfMod = rec.getModule("dmem");
Wire enableWire = rfMod.getWire("we");
Wire dataWire = rfMod.getWire("wd");
Wire indexWire = rfMod.getWire("address");
for (int i = 0; i <= lastFrame; i++) {
  if (enableWire.getValueSnapshot(i).getBit(0)) {
    int index = (int) indexWire.getValueSnapshot(i)
        .getLong() / 4;  // prvky v paměti jsou po 4B slovech,
                         // adresa se tedy dělí čtyřmi.
    data[index].addChangeSnapshot
        (dataWire.getValueSnapshot(i), i);
  }
}
```

Obrázek 4.23: Třída DataMemory – zápis změn do paměti

podle mezer na pole řetězců. Tyto řetězce se nakonec zkonvertují na čísla metodou `Long.parseLong` a pole těchto čísel se vrátí jako výstup.

4.10 Cache

Cache je simulovaná zvlášť ze simulačního okna až po simulaci samotného programu. Inicalizuje se s parametry, které udávají:

— bitovou délku počtu slov;

— bitovou délku počtu řádků;

— stupeň asociativity cache;

— čtecí objekt;

— nahrazovací strategii.

Cache po vytvoření obsahuje n cest `CacheWay`, kde n je stupeň asociativity. Po inicializaci se objektu `Cache` pošlou události přístupu metodou `accessEvent`, Tato metoda má jako parametry čas přístupu, adresu přístupu a příznak zápisu. Metoda `accessEvent`:

1. Spočítá, ve kterém řádku cache by data měla být.

2. Projde jednotlivé cesty, zjistí, zda se data již v cache nachází – pokud ano, zapíše se hit count, jinak miss.

3. Pokud nebyl hit, vybere se cesta pro zapsání – volný blok, pokud zde nějaký je, nebo se vybere blok podle nahrazovací strategie.

4. V případě missu nebo zapisovací události se do vypočítaného řádku a vybrané cesty zapíše blok paměti. Data bloku se získají pomocí čtecího objektu.

5. Do cesty cache se zaznamená informace o přístupu.

4.10.1 Datový blok

Datový blok (třída `CacheRow`) je datová třída obsahující:

— validitu bloku,

— příznak změny,

— tag,

— data.

Také nabízí statickou metodu `empty`, která generuje prázdný (invalid) blok.

4.10.2 Čtecí objekt

Čtecí objekt paměti implementuje rozhraní `IMemReader`, které deklaruje metodu `readData`, která má pro cache číst data z daného času a dané adresy. Existuje implementace, která čte data z datové paměti, a implementace, která čte z instrukční paměti.

4.10.3 Cesta

`CacheWay` je třída, která představuje jednu cestu (sloupec) v simulaci cache. Obsahuje pole historií datových bloků (`TreeMap[]` s `Integer` klíči a `CacheRow` hodnotami). Umožňuje získání bloku z libovolného řádku a času metodou `getRow`. Změny a přepsání bloků se provádí metodou `putRow`. `CacheWay` dále nabízí metody:

hasAccess zjistí, zda byl v požadovaný čas přístup do daného bloku.

isHit zjistí, zda byl v době požadavku daný blok již v této cestě.

lastUse zjistí poslední čas přístupu do daného bloku před daným časem (pro LRU strategii).

useFrequency zjistí počet přístupů do daného bloku před daným časem (pro LFU strategii).

4.11 Disassembler

Disassembler je pomocná bezestavová třída, která překládá strojový kód do jazyka symbolických instrukcí. Je implementována jako singleton. Používá se v okně pro zobrazení instrukční paměti.

Hlavní metoda této třídy je metoda `getInstructionString` (na obrázku 4.24), která přijímá kód instrukce ve formě čísla `long` a vrací `String` zápisu instrukce. Metoda se větví podle kódu operace (`opCode`). Do výstupního řetězce zapíše odpovídající instrukci a její parametry. V případě, že se kód operace rovná nule, jedná se o instrukci typu R a konverze se deleguje na metodu `rOperationString` (na obrázku 4.25). Ta navrátí „nop“, pokud je instrukce celá prázdná. V jiném případě složí název instrukce (podle `funct`) a názvy tří účastněných registrů.

Třída `Disassembler` dále vnitřně využívá tyto pomocné privátní metody:

opCode bity 26–31 z instrukce;

regS bity 21–25 z instrukce;

regT bity 16–20 z instrukce;

regDest bity 11–15 z instrukce;

immediateShort bity 0–15 z instrukce, jako číslo se znaménkem;

immediateLong bity 0–25 z instrukce;

funct bity 0–5 z instrukce;

shamt bity 6–10 z instrukce;

regName konvenční název registru daného indexu.

4.12 Konfigurační soubor

Aplikace si zapisuje naposledy použité parametry simulace do konfiguračního souboru, aby se nemusely znovu zadávat při každém spuštění. Pro přístup ke konfiguračnímu souboru slouží singletonová třída `ConfigurationFile`.

Konfigurační soubor tato třída hledá pod názvem `config.ini` v adresáři `.MIPSSimConfig/`, který by se měl nacházet v domovském adresáři uživatele. V případě, že soubor či adresář neexistuje, pokusí se ho vytvořit. Konfigurační soubor má následující formát:

```
vvp=<adresa interpreteru verilogu (vvp)>
prog=<adresa souboru s programem (strojovým kódem)>
data=<adresa souboru s počátečním obsahem datové paměti>
icnt=<počet instrukcí v simulaci>
```

```
switch (opCode(icode)) {
  case 0b000000:  // Instrukce typu R
    return rOperationString(icode);
  case 0b100011:  // lw
    return String.format("lw %s, %d(%s)",
      regName(regT(icode)),
      immediateShort(icode),
      regName(regS(icode)));
  case 0b101011:  // sw
    return String.format("sw %s, %d(%s)",
      regName(regT(icode)),
      immediateShort(icode),
      regName(regS(icode)));
  case 0b000100:  // beq
    return String.format("beq %s, %s, %d",
      regName(regS(icode)),
      regName(regT(icode)),
      immediateShort(icode));
  case 0b001000:  // addi
    return String.format("addi %s, %s, %d",
      regName(regT(icode)),
      regName(regS(icode)),
      immediateShort(icode));
  case 0b000011:  // jal
    return String.format("jal 0x%04X",
      immediateLong(icode) << 2);
  case 0b000010:  // j
    return String.format("j 0x%04X",
      immediateLong(icode) << 2);
  case 0b000111:  // jr
    return String.format("jr %s",
      regName(regS(icode)));
  case 0b011111:  // addu[_s}.qb
    return String.format("%s %s, %s, %s",
      shamt(icode) == 0 ? "addu.qb" : "addu_s.qb",
      regName(regDest(icode)),
      regName(regS(icode)),
      regName(regT(icode)));
}
```

Obrázek 4.24: Třída Disassembler – metoda getInstructionString

```
private String rOperationString(long icode) {
  if (icode == 0)
    return "nop";

  return String.format("%s %s, %s, %s",
    rOperationName(icode),
    regName(regDest(icode)),
    regName(regS(icode)),
    regName(regT(icode))
    );
}
```

Obrázek 4.25: Třída Disassembler – metoda rOperationString

`ConfigurationFile` má metodu pro zápis konfigurace a několik metod pro čtení hodnot z konfigurace. Pro zápis slouží metoda `writeConfig`. Ta má jako vstup hodnoty, které má zapsat. Metoda se nejprve pokusí vytvořit soubor, pokud ještě neexistuje. Pokud neexistuje, a ani se ho nepodaří vytvořit, tak se metoda ukončí a vrátí `false`. Do souboru se následně zapíší konfigurační údaje.

Pro čtění konfigurace slouží metody, které získávají jednotlivé údaje z konfigurace:

- `getVvpPath()`
- `getProgPath()`
- `getDataPath()`
- `getInstrCount()`

Používají k tomu pomocnou privátní metodu `getVal` (na obrázku 4.26). Ta prochází konfigurační soubor řádek po řádku a porovnává klíče na začátku s požadovaným údajem, který je předán jako parametr funkce. V případě, že se požadovaný klíč nenajde nebo pokud soubor neexistuje, vrací se `null`. Ten se v případě textových údajů později interpretuje jako prázdný řetězec, v případě informace o počtu instrukcí se to implicitně interpretuje jako 1000 instrukcí.

```
private String getVal(String valName) {
  File f = new File(getFileName());
  if (!f.exists()) return null;

  try (BufferedReader br =
      new BufferedReader(new FileReader(f))){
    String line;
    while ((line = br.readLine()) != null) {
      if (line.startsWith(valName + "=")) {
        return line.substring(valName.length()+1).trim();
      }
    }
  }
  catch (IOException ex) {
    return null;
  }
  return null;
}
```

Obrázek 4.26: Třída ConfigurationFire – metoda getVal

4.13 Grafické rozhraní

Aplikace používá grafický framework JavaFX. Jednotlivá okna a formuláře jsou popsány ve FXML souborech a mají přidružené controllery – třídy, které mají členské proměnné spárované s prvky v okně. Controllery všech oken kromě hlavního okna dědí ze třídy `WindowBase`, která umožňuje vytvoření okna, základní operace s oknem (zavření, znovuotevření) a obnovení obsahu.

4.13.1 Hlavní okno

Hlavní okno aplikace se dělí na tři části – levý panel, pravý panel a schéma procesoru.

4.13.1.1 Levý panel

Levý panel hlavního okna obsahuje tabulku, která zobrazuje hodnoty registrů. Tyto hodnoty se získávají z hlavní třídy metodou `getRegVal`. V této tabulce je také hodnota procesorových hodin, resetu a programového čítače.

Pod tabulkou registrů se nachází tlačítko pro zobrazení instrukční paměti a tlačítko pro zobrazení datové paměti.

4.13.1.2 Pravý panel

V pravém panelu hlavního okna se nahoře nachází tlačítka pro navigaci v historii běhu programu. Jsou zde tři tlačítka na každé straně:

— předchozí/následující cyklus,

— předchozí/následující instrukce (2 cykly, na sudý cyklus),

— přesun na začátek/konec simulace.

Pod navigací se nachází údaje o vybraném modulu (nebo spoji) ve schématu procesoru – jméno modulu a tabulka hodnot na rozhraních modulu. Informace jsou získány ze záznamu běhu programu (`ProgramRecord`) v hlavní třídě. Hodnoty se dají zobrazit v binární, dekadické nebo hexadecimální formě. Kořen se mění přepínacími tlačítky nad tabulkou.

Pod tabulkou hodnot modulu je tlačítko, které otevírá simulační okno, které spouští simulaci.

4.13.1.3 Schéma procesoru

Uprostřed hlavního okna se nachází schéma procesoru, ve kterém jsou zobrazeny jednotlivé moduly a spoje. Spoje, které v daném čase jsou nulové, jsou zobrazeny černě, ostatní červeně. Moduly i spoje se dají vybrat kliknutím myši, tím se informace o nich zobrazí v pravém panelu okna.

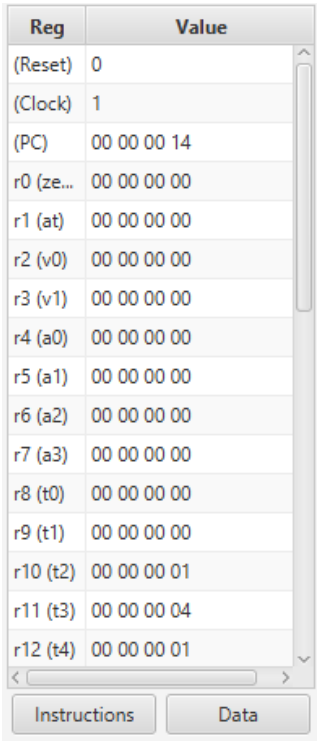
Veškeré informace o modulech kromě samotných hodnot – umístění ve schématu, typ, velikost, název a názvy rozhraní – poskytuje pomocná třída `PartDisplayManager`. Poskytnuté moduly jsou po načtení uloženy v objektech třídy `ModuleDisplay`, popřípadě v jejích podtřídách. Vykreslují se na plátno následujícím způsobem:

ModuleDisplay se vykresluje jako obdélník, vypíše se navíc text podle konkrétního typu modulu,

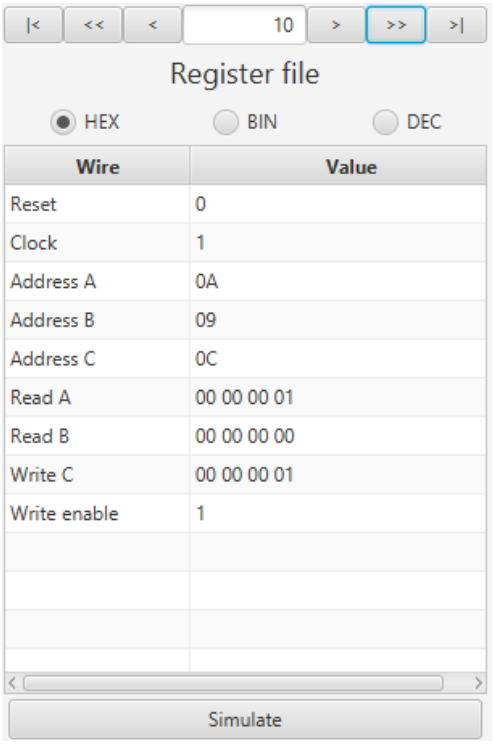
MultiplexorDisplay se vykresluje jako lichoběžník orientovaný doprava, vykreslí se vstupy, aktivní vstup se označí červenou barvou.

OperatorDisplay se vykresluje jako vyříznutý lichoběžník.

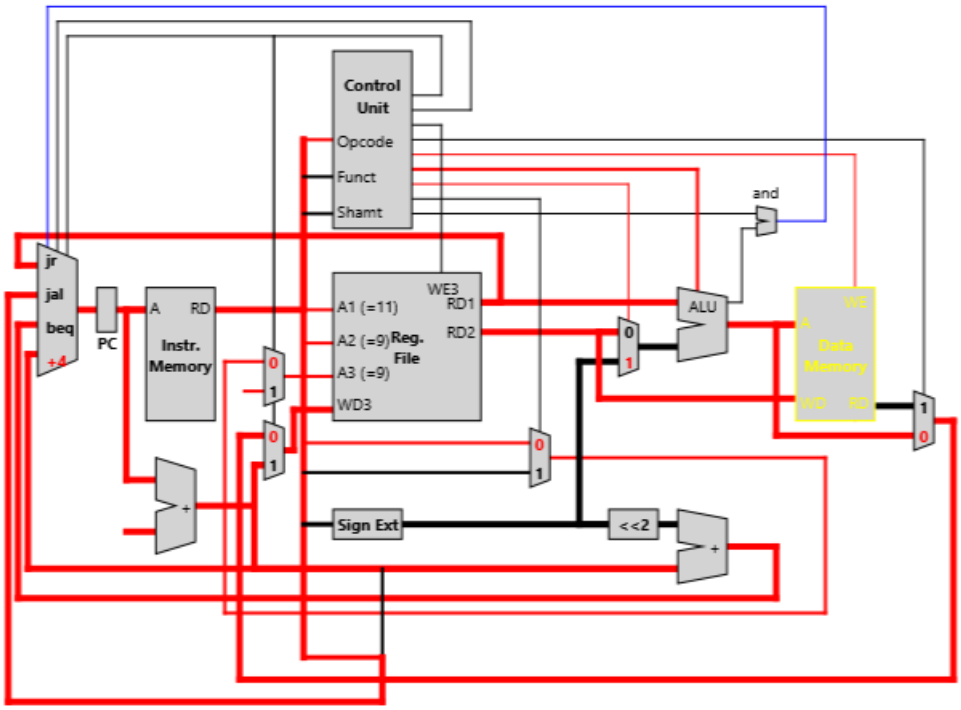
Informace o spojích (pozice uzlů ve schématu; modul, ze kterého se čte jejich hodnota) jsou uloženy v objektech třídy `WireDisplay`. `ModuleDisplay` a `WireDisplay` dědí ze třídy `ProcessorPartDisplay`. Hlavní okno má seznam objektů `ProcessorPartDisplay` a při vykreslování zavolá na všechny tyto objekty metodu `draw` s odkazem na kreslicí plátno.



(a) levý panel



(b) pravý panel



(c) schéma procesoru (datová paměť vybrána)

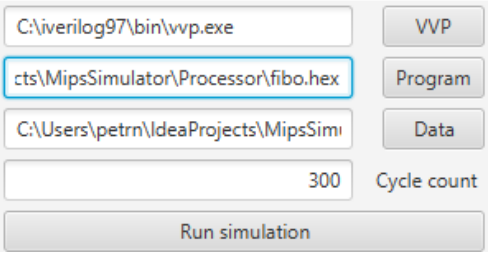
Obrázek 4.27: Hlavní okno

4.13.2 Simulační okno

Simulační okno `RunOptions` je okno, do kterého se zadávají parametry simulace – umístění interpreteru vvp, umístění souborů s datovou a instrukční pamětí a délka simulace. Umístění lze zadat ručně nebo lze tlačítkem zobrazit File Chooser.

Simulace se poté spouští tlačítkem „Run Simulation“. Při stisku tohoto tlačítka se hlavní třídě aplikace `MipsSimulator` odešlou zadané údaje a ta potom spustí metodu `simulate`. V případě neúspěchu se zobrazí chybová hláška, ta se čte z objektu `MipsSimulator`. V případě úspěchu se parametry uloží do konfiguračního souboru, okno se zavře a veškeré zobrazení se obnoví.

Simulační okno se otevírá tlačítkem „Simulate“ v dolní části pravého panelu hlavního okna.



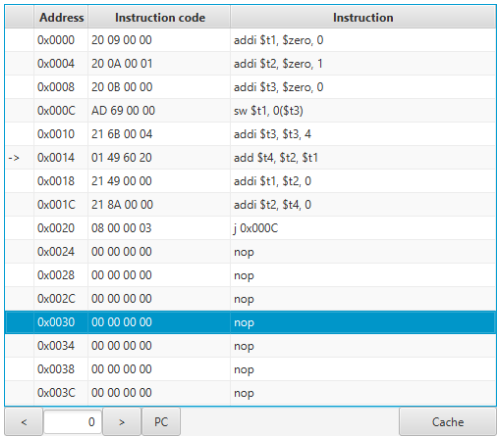
Obrázek 4.28: Simulační okno

4.13.3 Zobrazení instrukční paměti

Okno instrukční paměti se zobrazí stiskem tlačítka „Instructions“ v levém panelu hlavního okna. Instrukce jsou načítány z objektu `InstructionMemory` v objektu hlavní třídy metodou `getInstruction`. V tomto okně je mřížka se čtyřmi sloupci, které zobrazují:

1. ukazatel na současnou instrukci,
2. adresy instrukcí,
3. hexadecimální kód instrukce,
4. instrukci převedenou do jazyka symbolických instrukcí (pomocí třídy `Disassembler`).

Tato mřížka zobrazuje 16 instrukcí na stránku. Mezi stránkami lze přepínat tlačítky s šipkami pod mřížkou, popřípadě přepsáním čísla stránky. Tlačítkem „PC“ se přeskočí na stránku se současnou instrukcí.



	Address	Instruction code	Instruction
	0x0000	20 09 00 00	addi $t1, $zero, 0
	0x0004	20 0A 00 01	addi $t2, $zero, 1
	0x0008	20 0B 00 00	addi $t3, $zero, 0
	0x000C	AD 69 00 00	sw $t1, 0($t3)
	0x0010	21 6B 00 04	addi $t3, $t3, 4
->	0x0014	01 49 60 20	add $t4, $t2, $t1
	0x0018	21 49 00 00	addi $t1, $t2, 0
	0x001C	21 8A 00 00	addi $t2, $t4, 0
	0x0020	08 00 00 03	j 0x000C
	0x0024	00 00 00 00	nop
	0x0028	00 00 00 00	nop
	0x002C	00 00 00 00	nop
	0x0030	00 00 00 00	nop
	0x0034	00 00 00 00	nop
	0x0038	00 00 00 00	nop
	0x003C	00 00 00 00	nop

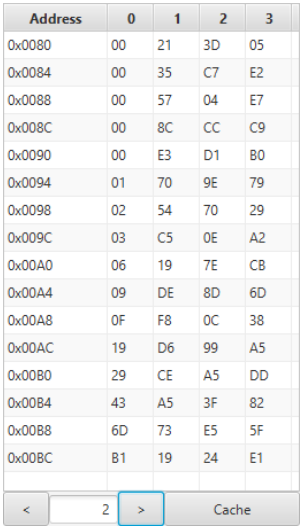
< 0 > PC Cache

Obrázek 4.29: Okno instrukční paměti

4.13.4 Zobrazení datové paměti

Okno datové paměti se zobrazí stiskem tlačítka „Data“ v levém panelu hlavního okna. Data jsou načítána z objektu `DataMemory` v objektu hlavní třídy metodou `getData`.

Zobrazují se spolu s adresami v tabulce po jednom datovém slově (4B) na řádek. Tabulka má 16 řádků na stránku. Mezi stránkami lze přepínat tlačítky s šipkami pod mřížkou, popřípadě přepsáním čísla stránky.



Address	0	1	2	3
0x0080	00	21	3D	05
0x0084	00	35	C7	E2
0x0088	00	57	04	E7
0x008C	00	8C	CC	C9
0x0090	00	E3	D1	B0
0x0094	01	70	9E	79
0x0098	02	54	70	29
0x009C	03	C5	0E	A2
0x00A0	06	19	7E	CB
0x00A4	09	DE	8D	6D
0x00A8	0F	F8	0C	38
0x00AC	19	D6	99	A5
0x00B0	29	CE	A5	DD
0x00B4	43	A5	3F	82
0x00B8	6D	73	E5	5F
0x00BC	B1	19	24	E1

< 2 > Cache

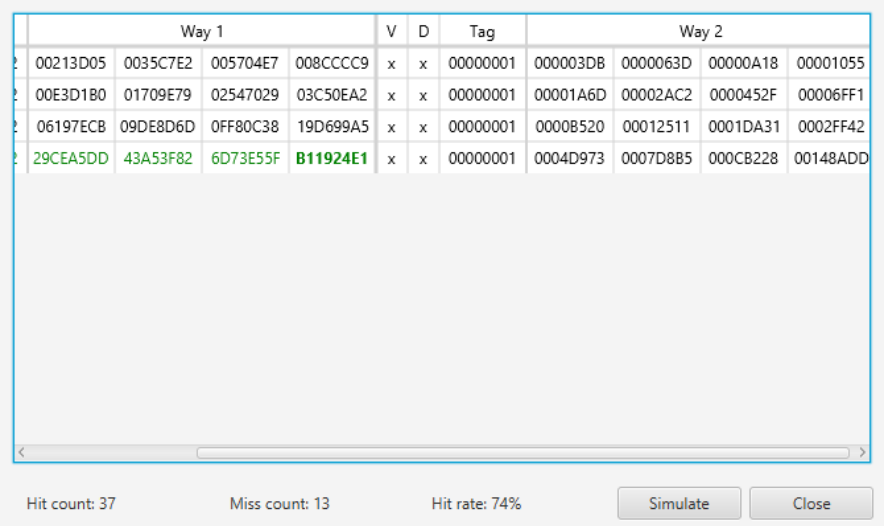
Obrázek 4.30: Okno datové paměti

4.13.5 Zobrazení cache

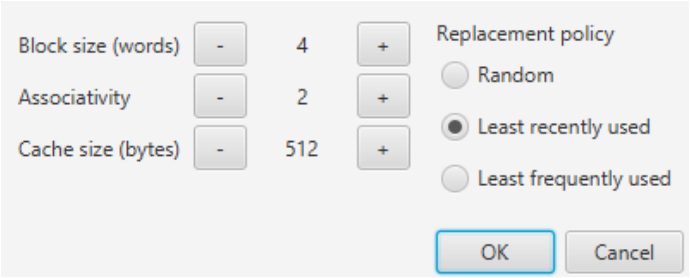
Okno cache se otevírá tlačítkem „Cache“ v dolní části okna datové nebo instrukční paměti. Při vytvoření získá čtecí objekt, který cache používá ke čtení obsahu konkrétní paměti (čte z hlavní třídy buď metodou `getData` nebo metodou `getInstruction`).

Hlavní částí tohoto okna je tabulka, ve které se cache zobrazuje. Obsah cache se načítá z objektu `Cache`, který se vytvoří po nastavení parametrů v okně možností `CacheOptions`.

To se zobrazí po stisknutí tlačítka „Simualate“ v okně cache. Lze zde nastavit parametry cache – velikost cache, velikost bloku, stupeň asociativity a strategii nahrazování. Po vyplnění se vytvoří objekt `Cache` se zadanými parametry a čtecím objektem.



Obrázek 4.31: Okno cache



Obrázek 4.32: Možnosti cache

Kapitola 5

Testování

Implementace simulátoru byla testována uživatelsky a automatickými testy.

5.1 Automatické testování

Pro automatické testy byl použit framework JUnit. Takto byly testovány:

— třídy záznamu běhu programu,

— čtečka VCD záznamu,

— čtečka paměťových souborů,

— simulace cache.

5.1.1 Testování záznamu běhu programu

V záznamu běhu programu se testuje vytvoření modulu, přidání modulu a přístup k němu pomocí jeho jména, přidání spoje a přístup k němu pomocí jeho unikátního identifikátoru.

V testech záznamů změny hodnoty se provádí vytvoření z booleanu, integeru a stringu. Testuje se pak jeho obsah po bitech. Také se zde testuje změna bitů.

U spoje se testuje, zda se správně nastaví a přečte velikost spoje a identifikátor, dále se u něj testuje přístup k jeho hodnotě v daném čase po přidání několika změn hodnot. Testují se časy, ve kterých jsou změny, i časy, ve kterých nejsou změny.

U modulů se testuje vytvoření – shoda jména vytvořeného modulu se zadaným jménem. Dále se zde testuje přidání několika spojů a přístup k nim pomocí názvu rozhraní.

5.1.2 Testování čtečky VCD záznamu

V testech tokenů se nejprve vytvoří jednoduchý prostý token a jednoduchý symbolový token. Následně se zde testuje obsah tokenu a symbolový obsah.

Test tokenového parseru spočívá ve vytvoření jednoduchého textového řetězce a spuštění parseru nad tímto řetězcem v cyklu, který skončí, když parser získá EOF token. Poté se kontroluje počet a obsah tokenů, které parser získal.

Celý syntaktický parser se testuje třemi testy, které parseru dají validní uměle vytvořený VCD soubor:

1. test prochází pouze hlavičku VCD souboru.
2. test zahrnuje čtení definic modulů a následnou kontrolu obsahu záznamu.
3. test čte celý VCD soubor. Kontrolují se zde hodnoty v časech v jednotlivých spojích.

5.1.3 Testování čtečky paměťových souborů

Test spouští čtečku nad souborem s následujícím obsahem:

```
12_34_56_78 //comment
ab_Cd /*
666
*/
eFD /1//2
12_34_57_8F /* 6 * 5 **/
3597c
```

Kontroluje se výsledek – počet hodnot a samotné hodnoty, porovnává se s očekávanými hodnotami.

5.1.4 Testování simulace cache

Simulace cache má pět testů:

1. test vytvoří cache, porovná její vlastnosti se zadanými parametry a zkontroluje, že bloky nejsou validní.
2. test vloží jednu přístupovou událost do cache, otestuje vlastnosti bloku, do kterého přístup patří.
3. test vkládá několik přístupů do cache a zároveň postupně kontroluje hit count, miss count a hit rate.
4. test vytvoří cache se stupněm asociativity 1. Vloží do ní blok, potom vloží další blok do stejného setu (přepíše). Zkontroluje, že na stejném místě je tento nový blok.

5. test vytvoří cache se stupněm asociativity 2. Vloží do ní blok, potom vloží další blok do stejného setu. Zkontroluje, že v příslušném setu jsou v obou cestách validní bloky.

5.2 Uživatelské testování

Aplikaci jsem testoval uživatelsky. Samotnou funkcionalitu aplikace jsem testoval na programu generujícím Fibonacciho posloupnost (na obrázku 5.1). Program se nepatrně liší od programu použitém k analýze existujících aplikací – zapisuje do paměti od adresy 0 směrem výše.

```
addi  $t1,$zero,0
addi  $t2,$zero,1
addi  $t3,$zero,0
sw    $t1,0($t3)
addi  $t3,$t3,4
add   $t4,$t2,$t1
addi  $t1,$t2,0
addi  $t2,$t4,0
j     3
```

(a) jazyk symbolických instrukcí

```
20090000
200a0001
200b0000
ad690000
216b0004
01496020
21490000
218a0000
08000003
```

(b) hexadecimální forma

Obrázek 5.1: Generátor Fibonacciho posloupnosti 2

Byla zde testována samotná simulace a následné procházení historie běhu programu, hodnoty v registrech a jednotlivých modulech. Dále byl sledován obsah instrukční paměti a zápisy do datové paměti během historie přes jejich zobrazovací okna. Nakonec byla testována cache a její zobrazení. Byla spouštěna s různými parametry, pokaždé se chovala dle očekávání.

Toto testování bylo z většiny prováděno na operačním systému Windows, ale některé prvky bylo třeba testovat i na jiném systému. Aplikace tedy byla testovéna také na Linuxovém systému, kde bylo třeba vyzkoušet, zda správně funguje chování, které externě komunikuje se systémem – přistup ke konfiguračnímu souboru (adresace uživatelského adresáře) a spouštění externího programu (interpreter vvp).

Závěr

Vytvořil jsem program, který simuluje chování procesoru využívající architekturu MIPS32 lehce modifikovanou pro účely předmětu BI-APS. Tento program čte vstupy ze souborů strojového kódu v hexadecimálním formátu.

Schéma programu bylo navrženo jako tři celky. Prvním je jádro simulace, které provádí samotnou procesorovou simulaci. Dalším je obalující modul komunikující s jádrem, který dopočítává pomocné a doplňkové údaje. Posledním je grafické rozhraní, které zobrazuje údaje a hodnoty získané od předchozích celků.

Obalující modul a grafické rozhraní je implementováno dohromady v jedné aplikaci napsané v jazyce Java. Jádro simulace bylo vytvořeno v jazyce pro popis hardwaru Verilog a je volané z obalujícího modulu. Funkcionalita programu byla následně otestována automaticky i uživatelsky na různých počítačích a různých operačních systémech.

Všechny cíle práce byly splněné. Práce by se dále dala rozšířit například překladačem z jazyka symbolických instrukcí a editorem vstupního kódu.

Literatura

[1] Computer hope: What is CPU (Central processor unit)? 2019, [online], 2019-05-06. Dostupné z: https://www.computerhope.com/jargon/c/cpu.htm

[2] Computer hope: What is an instruction set? 2018, [online], 2019-05-06. Dostupné z: https://www.computerhope.com/jargon/i/instset.htm

[3] Wave Computing Inc.: MIPS® Architecture For Programmers. Volume I-A: Introduction to the MIPS32® Architecture, revize 6.01. 2014, [online], 2019-04-18. Dostupné z: https://www.mips.com/?do-download=introduction-to-the-mips32-architecture-v6-01

[4] Price, C.: MIPS IV Instruction Set, vydání 3.2. 1995, [online], 2019-05-06. Dostupné z: https://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf

[5] Štepanovský, M.: Zadání semestrálního projektu č.1: Jednocyklový procesor. 2018, [online]. Dostupné z: https://courses.fit.cvut.cz/BI-APS/tutorials/05/semester_project_cz.html

[6] Wave Computing Inc.: MIPS® Architecture For Programmers. Volume II-A: The MIPS32® InstructionSet, revize 5.04. 2013, [online], 2019-05-05. Dostupné z: https://www.mips.com/?do-download=the-mips32-instruction-set-v5-04

[7] Silicon Graphics, Inc.: MIPSpro™ N32 ABI Handbook. 1999, [online], 2019-05-05. Dostupné z: https://www.linux-mips.org/pub/linux/mips/doc/ABI/MIPS-N32-ABI-Handbook.pdf

[8] Štepanovský, M.; Tvrdík, P.: Architektura počítačových systémů (BI-APS), Přednáška č.6, Paměťová hierarchie I. 2018, [prezentace]. Dostupné z: https://courses.fit.cvut.cz/BI-APS/media/lectures/BI-APS-Prednaska06-Cache.pdf

[9] IEEE 1364-2001: IEEE Standard Verilog Hardware Description Language. Standard, 2001, doi:10.1109/IEEESTD.2001.93352.

[10] BeyondTTL.com: Value Change Dump VCD. [online], 2019-04-18. Dostupné z: https://web.archive.org/web/20120323132708/http://www.beyondttl.com/vcd.php

[11] Wooley, E.: WeMips. [webapp], 2019-04-19. Dostupné z: https://rivoire.cs.sonoma.edu/cs351/wemips

[12] Jiantastic: Visual MIPS. [webapp], 2019-04-19. Dostupné z: https://visualmips.github.io

[13] Larus, J. R.: Spim, A MIPS32 Simulator. [software], 2019-04-16. Dostupné z: http://spimsimulator.sourceforge.net

[14] Sanderson, P.; Vollmar, K.: MARS, MIPS Assembler and Runtime Simulator. [software], 2019-04-16. Dostupné z: http://courses.missouristate.edu/KenVollmar/mars

Příloha A

Seznam použitých zkratek

ALU Aritmeticko-logická jednotka

CPU Centrální procesorová jednotka

EOF Konec souboru

GUI Graphical user interface

IEEE Institut pro elektrotechnické a elektronické inženýrství

ISA Architektura instrukční sady

PC Programový čítač

RAM Paměť s přímým přístupem

VCD Value change dump, výpis změn hodnot

Příloha B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `bin` ..... adresář se spustitelnou formou implementace
- `man` ..... manuál k užití
  - `man.pdf` ..... manuál k užití ve formátu PDF
- `doc` ..... dokumentace kódu
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF