FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Simulátor 32-bitového procesoru podporující instrukční sadu MIPS
Student:	Ondřej Marek
Vedoucí:	Ing. Michal Štepanovský, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce zimního semestru 2019/20

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 jednocyklovém 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 12. března 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Simulátor 32-bitového procesoru podporující instrukční sadu MIPS

Ondřej Marek

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

28. června 2018

Poděkování

Děkuji panu Ing. Michalu Štepanovskému, Ph.D za rady a konzultace při vývoji této práce. Dále bych rád poděkoval svým přátelům a rodině za jejich povzbuzování.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) 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 28. června 2018 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2018 Ondřej Marek. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Marek, Ondřej. *Simulátor 32-bitového procesoru podporující instrukční sadu MIPS*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato práce se zabývá analýzou, návrhem a implementací systému, který umožňuje simulovat vykonávání programů na vybrané mikroarchitektuře. Řešeršní část práce se zabývá srovnáním již existujích řešení a popisuje technologie a termíny použité dále v práci, praktická část navazuje představením návrhu aplikace, popisem jeho částí a postupu použitém při implementaci. Zvolený problém byl vyřešen pomocí integrace programu s hardware simulátorem Icarus Verilog. Byl vytvořen systém, který v základu podporuje zjednodušenou mikroarchitekturu MIPS32 s možností nakonfigurování dalších mikroarchitektur. Přínos práce spočívá v možnosti pro studenty počítačově orientovaných předmětů díky vizualizaci snadněji pochopit strukturu a činnost procesorů. V příloze na USB disku lze nalézt program, zdrojové kódy a manuál k použití.

Klíčová slova simulátor procesoru, MIPS 32-bit, vizualizace, výukový program pro BI-APS, Verilog, grafické uživatelské rozhraní

Abstract

This thesis deals with analysis, design and implementation of a system which allows to simulate running programs for target microarchitecture. Research part of the thesis deals with comparing of existing solutions and describes

technologies and terms used further in the thesis, the practical part follows the presentation of the program design, description of its parts and methods used during implementation. The chosen problem was solved by integration of the program with hardware simulator Icarus Verilog. There was created a system which supports simplified microarchitecture MIPS32 in the base with the possibility to configure further microarchitectures. The benefit of this thesis is that students of computer-driven subjects will understand the structure and operation of processors thanks to visualisation. In attachment on USB stick there is a program, source codes and a manual.

Keywords processor simulator, MIPS 32-bit, visualisation, educational program for BI-APS, Verilog, graphical user interface

Obsah

Úvod **1**

1 Cíl práce **3**

2 Analýza **5**
2.1 MipsIt 5
2.2 MARS 8
2.3 Shrnutí 10

3 Definice **13**
3.1 MIPS32 mikroarchitektura 13
3.2 Icarus Verilog 15

4 Návrh **17**
4.1 Architektura 17
4.2 Stavový diagram 18
4.3 Uživatelské rozhraní 20

5 Realizace **23**
5.1 Uživatelské rozhraní 23
5.2 Prezenční vrstva 23
5.3 Načítání mikroarchitektur 26
5.4 Spuštění VVP 30
5.5 Parsování VCD 31
5.6 Výpočet cache 33
5.7 Logování a chybové hlášky 35
5.8 Testování 37

Závěr **39**

Literatura **41**

A Seznam použitých zkratek **43**

B Obsah přiloženého USB **45**

C Dokumentace **47**

Seznam obrázků

2.1 Vývojové prostředí MipsIt . . . 6
2.2 Simulátor MipsIt . . . 6
2.3 MipsIt registry . . . 7
2.4 MipsIt ukázka cache . . . 7
2.5 MipsIt ukázka paměti . . . 8
2.6 MARS editor . . . 9
2.7 MARS simulátor . . . 10

4.1 Architektura . . . 18
4.2 Stavový diagram . . . 19
4.3 Návrh uživatelského rozhraní . . . 21

5.1 Tlačítko další krok . . . 23
5.2 GUI logger . . . 35

Seznam tabulek

3.1 Podporované instrukce 15

5.1 Ukázka programu 27
5.2 Ukázka datové paměti 28

Úvod

Zcela jistě jednou z klíčových součástí všech dnešních počítačů je proces. Co vlastně takový procesor dělá? Z čeho se skládá? Bylo by šikovné, kdyby existovala pomůcka, která nám pomůže pochopit, jak jeho jednotlivé části fungují. Kterak jaká část procesoru dopodrobna způsobuje tu a kterou činnost. Právě z toho důvodu jsem si toto téma vybral, abych pomohl budoucím studentům nejen na Fakultě informačních technologií při ČVUT pochopit danou problematiku. Jedním z nejjednodušších procesorů je MIPS s 32-bitovou architekturou a je vhodné pro něj vytvořit řešení, které by dokázalo pro školní účely vhodně vizualizovat skutečnou práci procesoru i s hodnotami v paměti či v cache procesoru.

Za cíl své práce jsem si zvolil implementovat program s grafickým uživatelským rozhraním, který vyhoví těmto požadavkům. Pro uživatele má být snadno pochopitelný, vidí různé části systému v celkovém obraze. Dalším hlavním požadavkem má být dostupnost, program bude možné spustit na libovolném operačním systému. Nakonec má být pro pokročilejší uživatele plusem i to, že program bude konfigurovatelný za pomoci XML a při použití schématu v jazyce Verilog lze nakonfigurovat jinou mikroarchitekturu. Vše je samozřejmě zdokumentované.

V rešeršní části bakalářské práce se zabývám analýzou existujících řešení, ve které popisuji instalaci a používání vybraných programů. Diskutuji o tom, proč dané programy nejsou vhodným řešením a co je třeba udělat lépe. V rešeršní části se věnuji i definicím technologií, na které se odkazuji v praktické části práce.

Praktická část se zabývá návrhem a postupem při implementaci. Popisuje dané kroky, které jsem udělal a taky těžkosti, se kterými jsem se setkal a způsob, jakým jsem je řešil.

Kapitola 1

Cíl práce

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

1. Identifkace výhod a nevýhod volně dostupných simulátorů procesoru podporujícího instrukční sadu MIPS.
2. Návrh ideového schématu simulátoru s důrazem na oddělení simulace a vizualizace.
3. Implementace simulátoru navrženého v předchozím bodě ve vybraném programovacím jazyce.
4. Přizpůsobení vzhledu a funkcionality simulátoru potřebám předmětu BI-APS.
5. Vypracování dokumentace a manuálu k navrženému simulátoru.

Kapitola 2

Analýza

Tato kapitola si dává za cíl zanalyzovat a porovnat existující řešení.

2.1 MipsIt

V současné době se při výuce předmětu používá program MipsIt, který byl vyvinut pro potřeby KTH, Royal Institute of Technology ve Švédsku.

„MipsIt systém se skládá z vývojového prostředí, z hardwarové platformy a sérií simulátorů. Téma tohoto dokumentu je hlavně simulace animace pro cache paměť a pipelanovou simulaci, ale pro úplnost popíši i ostatní části. Veškerý software pro MipsIt systém je určen pro platformu Windows (95-XP) jako hostující počítač.“ [1](vlastní překlad)

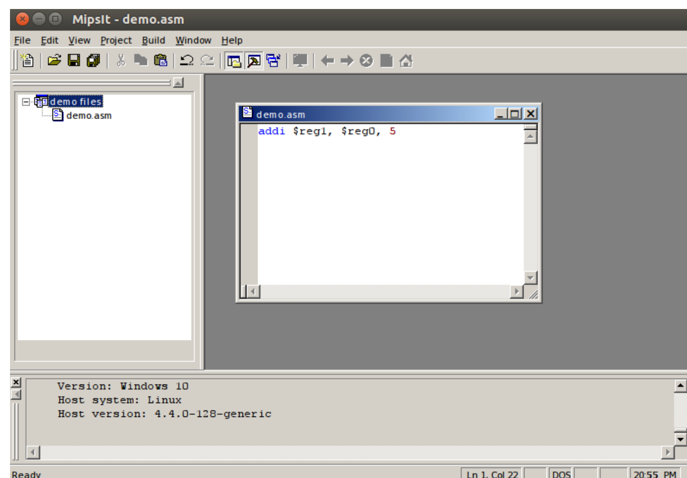
Program MipsIt není třeba instalovat. Program je zabalen do ZIP souboru, stačí jej rozbalit. Program je k sehnání např. zde [2].

2.1.1 Použití MipsIt

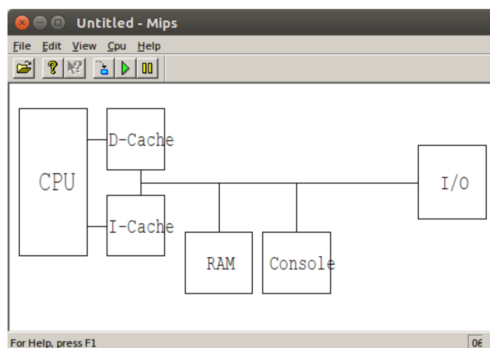
Jednou z částí programu MipsIt je vývojové studio (na obrázku 2.1). Dle [1] bylo vývojové studio inspirováno vzhledem programu Visual Studio od firmy Microsoft. Dále platí, že jsou v základu podporovány programy psané v C jazyku nebo v assembleru pro MIPS architekturu. Kompilaci kódu zajišťuje přibalený GCC kompilátor, který převede zdrojové kódy do binární podoby pro architekturu MIPS. Binární podobu programů lze spustit buď na skutečném hardwaru anebo v simulátoru.

Při spuštění simulátoru naběhne úvodní obrazovka (obrázek 2.2). Na obrázku je nakresleno hrubé schéma přítomných komponent. Po dvojím poklikání na danou komponentu se otevře detail. Komponenty jsou tyto:

CPU je komponenta, která ukazuje obsah registrů. Příklad je na obrázku 2.3.



Obrázek 2.1: Vývojové prostředí MipsIt



Obrázek 2.2: Úvodní obrazovka simulátoru MipsIt

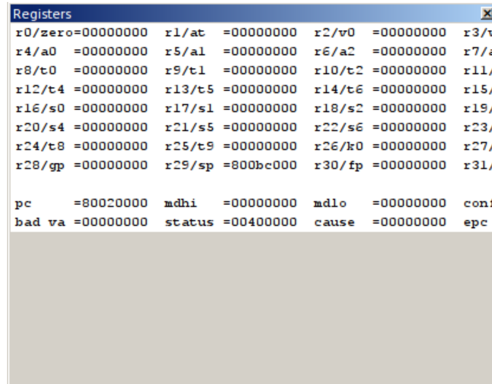
D-Cache a I-Cache jsou komponenty, které vizualizují práci datové, respektive instrukční cache (na obrázku 2.4). Vizualizuje vstupní adresu, návratovou hodnotu, datové tabulky a statistiku.

Console je komponenta, která vizualizuje obsah terminálu.

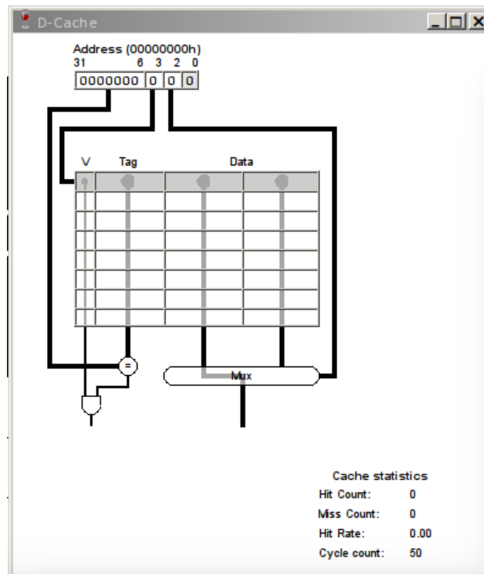
I/O je komponenta, která vizualizuje jednoduché I/O rozhraní. Je zde vizualizováno 8 vstupních přepínačů a 8 výstupních světel.

2.1.2 Výhody a nevýhody MipsIt

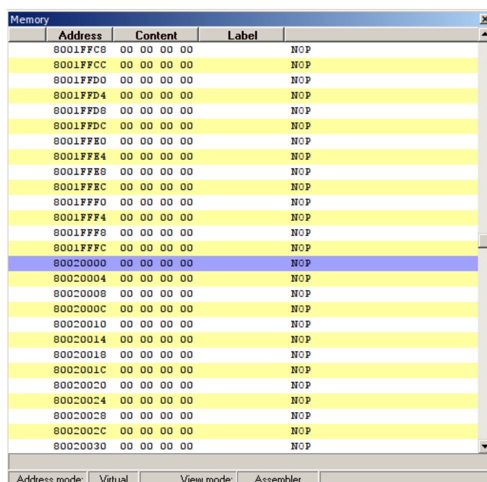
Mezi výhody použití programu MipsIt patří možnost implementovat program, který lze přímo simulovat. I vizualizační stránka programu, zejména cache části, je na dobré úrovni. Podle [1] lze simulovat i procesory z proudovým zpracováním instrukcí.



Obrázek 2.3: Obsah registrů simulovaného programu v MipsIt



Obrázek 2.4: Obsah cache simulovaného programu v MipsIt



Obrázek 2.5: Obsah paměti simulovaného programu v MipsIt

Mezi největší nevýhodu patří především zastaralost programu, který je dle [1] určen pro operační systém verze 95 - XP. Bohužel nefunguje správně na nejmodernějším operačním systému Windows, který je v současnosti Windows 10. Zároveň tento program není nativně podporován na dalších operačních systémech, ať již linux nebo macOS.

Uvedu i další nevýhody, kterým jsou drobné chyby při vykreslování obrazovky. Všiml jsem si, že ve vývojovém prostředí se nemusí vždy vykreslit sloupec (na obrázku 2.1 vlevo) se seznamem souborů v projektu. Setkal jsem se i s případem, že se nenačetl nový obsah RAM paměti.

Vizualizace RAM paměti je uživatelsky nepřívětivá. Uživatel si sice pomocí posouvátka může prohlédnout obsah celé paměti, avšak není jednoduchá cesta, jak se vrátit na původní místo, případně se podívat na určitou adresu.

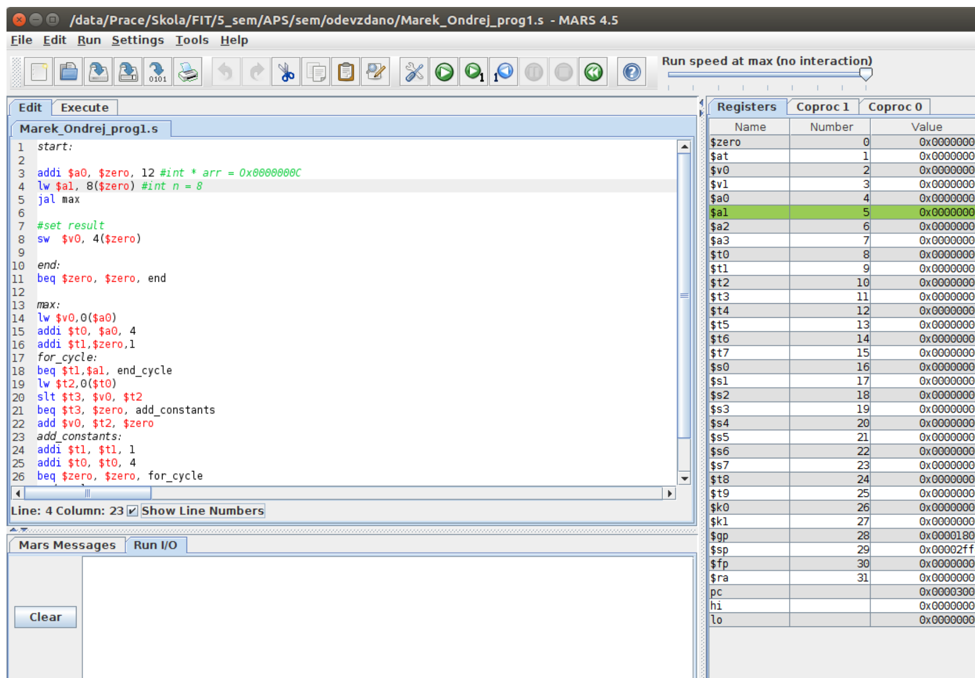
2.2 MARS

MARS (MIPS Assembler and Runtime Simulator) je interaktivní a odlehčené vývojové prostředí pro programování v MIPS assembler jazyce. Tento program je určen zejména ke vzdělávacím účelům. [3]

Program MARS není nutno instalovat. Dostačuje stáhnout si z webových stránek [3] spustitelný soubor typu JAR, který se dá spustit pomocí dvojitého poklepání ihned po stáhnutí. Jediným požadavkem je, že v systému musí být nainstalován Java Runtime Enviroment verze 5 nebo vyšší.

2.2.1 Použití simulátoru MARS

MARS umí editovat a spouštět pouze programy, jejichž zdrojový kód je napsaný v assembleru. Tento program má dva režimy:



Obrázek 2.6: Editor assembleru v simulátoru MARS

režim editace je režim, ve kterém se upravuje zdrojový soubor (obrázek 2.6). Editor při psaní zobrazuje nápovědu se seznamem dostupných instrukcí. Pro sestavení programu je potřeba kliknout na Run ->Assemble v kontextovém menu, nebo stiknout klávesu F3.

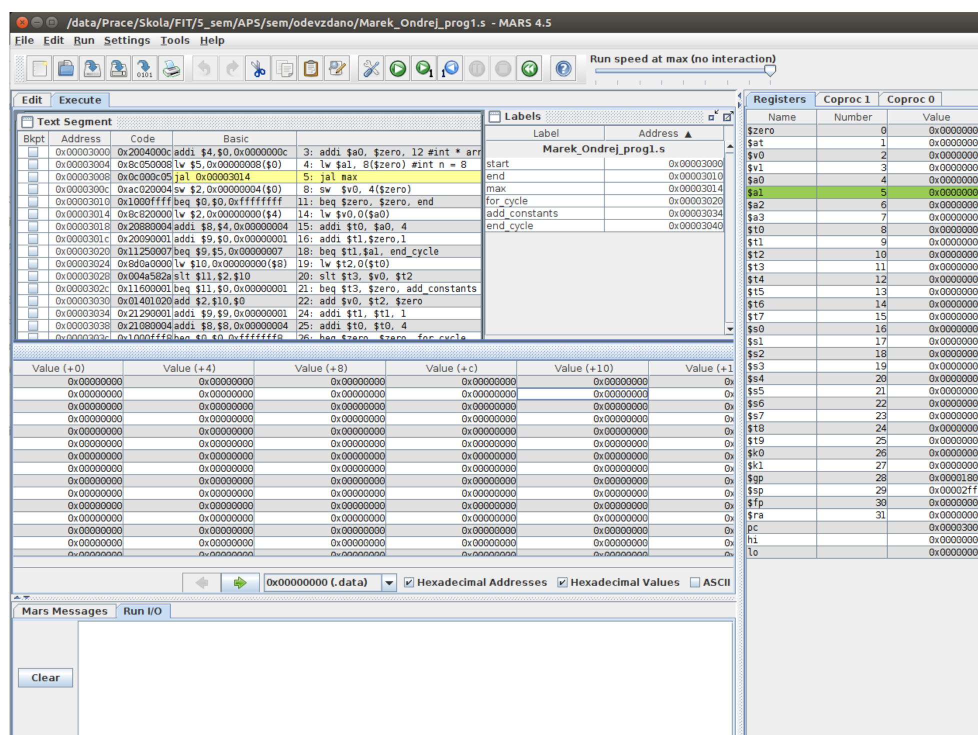
režim simulace je režim, ve které probíhá simulace samotného programu (obrázek 2.7). Jsou k dispozici tabulky hodnot registrů, datové paměti, instrukční paměti. Průběh programu se ovládá pomocí tlačítek v horním panelu nebo pomocí klávesových zkratek.

Dále dle [3] tento program umí:

- editovat hodnoty registrů a paměti za běhu,
- zobrazit hodnoty v šestnáctkové a desítkové soustavě,
- zobrazit registry pro výpočet float hodnot,
- dávkové zpracování instrukcí nebo po krocích. . .

2.2.2 Výhody a nevýhody simulátoru MARS

Editor assembleru v simulátoru MARS je přehledný. V programu lze nakonfigurovat, na jaké adrese začíná datová paměť (případně instrukční). Další



Obrázek 2.7: Simulace programu v simulátoru MARS

výhodou jsou přehledné a uživatelsky přívětivé tabulky, které přehledně zobrazují obsah pamětí. Jsou použité barvy ke zvýraznění změněných hodnot.

Velikou výhodou tohoto programu je přenositelnost. Program běží v Java Virtual Machine, proto je možné jej použít na různých operačních systémech.

Mezi nevýhody programu patří nedostatečná cache vizualizace. Program umí spočítat statistiku cache paměti, avšak nezobrazí její obsah. V programu chybí vizulizace jakýchkoli schémat, uživatel si hůře představí, jak jsou jednotlivé části simulátoru ve skutečném hardwaru propojené.

2.3 Shrnutí

Po analýze těchto programů jsem došel k závěru, že je vhodné vytvořit program, který napravuje nedostatky uvedených řešení. Tento budoucí simulátor se zároveň inspiruje u dobrých stránek těchto programů.

Z programu MipsIt jsem se inspiroval především u způsobu, jakým je řešena vizualizace cache. V MipsIt je přehledné a důkladné schéma procesoru s proudovým zpracováním instrukcí, které však chybí pro jednoduchý procesor. V novém simulátoru toto schéma chybět nebude.

Simulátor MARS mě zaujal z hlediska přehlednosti tabulek, které se mají

pro uživatele přínosnou informační hodnotu a ve kterých se uživatel dobře naviguje. Zároveň se inspiruji i o snadné přenositelnosti, která je způsobena tím, že program je napsán v jazyce Java.

Kapitola 3

Definice

Cílem této kapitoly je blíže přiblížit čtenáři opakující se pojmy použité v této práci.

3.1 MIPS32 mikroarchitektura

Mikroarchitektura MIPS32 je standard, který využívá mnoho elektronických zařízení. Poskytuje robustní instrukční set, škálovatelný od 32 do 64 bitů. MIPS32 je kompatibilní s MIPS64 mikroarchitekturou. Jsou podporovány privilegované instrukce, virtualizace a další, které umožňují využití v dnešních počítačových systémech. [4]

Mikroarchitektura MIPS32 je založena na instrukcích s fixní délkou, které jsou pravidelně zakódované, a používá load/store data model. Aritmetické a logické instrukce používají tří-operandový formát. Dostupnost 32 registrů umožňuje kompilátorům vygenerovat efektivní kód, který drží data s nejčastějším přístupem v registrech. [4]

3.1.1 Podporované MIPS instrukce

V rámci simulátoru FitMips je podporována pouze podmnožina instrukčního setu MIPS. Množina byla vybrána podle existujícího zadání semestrální práce, ve které studenti implementují vlastní MIPS procesor, a to v předmětu BI-APS. Zadání semestrální práce a tabulka je k nalédnutí na stránkách eduxu [5].

Instrukce	Syntaxe	Operace	Kódování	Poznámka
`add`	`add d, s, t`	`d = s + t;`	`0000 00ss ssst tttt dddd d000 0010 0000`	

sub	sub d, s, t	d = s - t;	0000 00ss ssst tttt dddd d000 0010 0010	
and	and d, s, t	d = s & t;	0000 00ss ssst tttt dddd d000 0010 0100	
or	or d, s, t	d = s I t;	0000 00ss ssst tttt dddd d000 0010 0101	
slt	slt d, s, t	d = (s<t) ? 1 : 0;	0000 00ss ssst tttt dddd d000 0010 1010	
addi	addi d, s, imm	d = s + imm;	0010 00ss ssst tttt iiii iiii iiii iiii	
lw	lw t, offset(s)	t = MEM[s + offset];	1000 11ss ssst tttt iiii iiii iiii iiii	
sw	sw t, offset(s)	MEM[s + offset] = t;	1010 11ss ssst tttt iiii iiii iiii iiii	
beq	beq s, t, offset	if s==t then PC = PC + 4 + (offset ≪ 2); else PC = PC + 4;	0001 00ss ssst tttt iiii iiii iiii iiii	
jal	jal target	$31=PC+4; PC=(PC & 0xf0000000) I (target ≪ 2);	0000 11ii iiii iiii iiii iiii iiii iiii	neodpovídá původnímu MIPS32
jr	jr s	PC = s;	0001 11ss sss0 0000 0000 0000 0000 1000	nodpovídá původnímu MIPS32, opcode má jinou hodnotu

`addu.qb`	`addu.qb d, s, t`	`d31:24 = s31:24 + t31:24; atd.`	`0111 11ss ssst tttt dddd d000 0001 0000`	sčítání dvou vektorů bytových hodnot bez znaménka
`addu_s.qb`	`addu_s.qb d, s, t`	`d31:24 = sat(s31:24 + t31:24); atd.`	`0111 11ss ssst tttt dddd d001 0001 0000`	se saturací
`sllv`	`sllv d, t, s`	`d = t ≪ s;`	`0000 00ss ssst tttt dddd dxxx xx00 0100`	
`srlv`	`srlv d, t, s`	`d = (unsigned)t ≫ s;`	`0000 00ss ssst tttt dddd d000 0000 0110`	
`srav`	`srav d, t, s`	`d = (signed)t ≫ s;`	`0000 00ss ssst tttt dddd d000 0000 0111`	vsunut znaménkový bit

Tabulka 3.1: Podporované instrukce MIPS

3.2 Icarus Verilog

„Icarus Verilog je simulační a syntetizační nástroj pro jazyk Verilog. Funguje jako kompilátor, který kompiluje zdrojový kód napsaný ve Verilogu (IEEE-1364) do cílového formátu. Pro dávkové zpracování kompilátor generuje mezikód zvaný *vvp assembly*. Tento mezikód je vykonán pomocí „vvp“ příkazu. Při syntéze vytváří popisy konektivity v požadovaném formátu.“ [6] (vlatní překlad)

3.2.1 VVP

„vvp je runtime engine, který vykonává kód, který je defaultně generován programem Icarus Verilog. Výstup z příkazu iverilog není spustitelný na žádné platformě. Místo toho je spuštěn program vvp k vykonání výstupního kódu.“ [7] (vlastní překlad)

3.2.2 VCD

Dle [8] je VCD (Value Change Dump) je souborový formát specifikovaný v IEEE-1364 standardu. Soubory uložené ve VCD formátu obsahují:

- datum a čas vzniku souboru,

- číslo verze,
- direktivu $timescale,
- specifikaci signálů,
- direktivu $dumpvars, která definuje, jaké proměnné jsou uloženy.

KAPITOLA **4**

Návrh

V následující kapitole popisuji, jakým způsobem jsem navrhnul simulátor, který jsem pojmenoval FitMips. Ukáži, jaká je celková architektura programu FitMips a též stavový diagram programu. Není zapomenuto ani na návrh uživatelského rozhraní.

4.1 Architektura

Celá tato sekce pojednává o architektuře FitMipsu. Ta je vyobrazena na ilustraci 4.1. Skládá se z těchto částí:

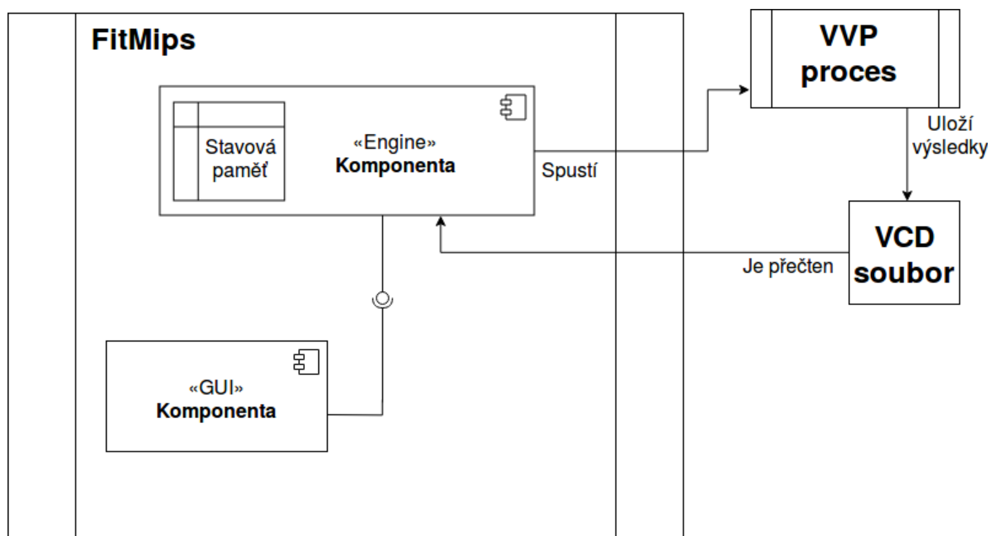
FitMips je konteiner pro celou aplikaci, která běží jako samostaný proces v rámci Java Virtual Machine.

GUI komponenta je část aplikace, která se stará o vykreslování výsledků a interakci s uživatelem. Je využit typ architektury MVC, kde v této komponentě je přítomna prezenční vrstva a controller. Controller komunikuje s Engine komponentou pomocí API.

Engine komponenta se stará spouštění VVP procesů a parsování jejich výsledků, které převede do interní podoby. Dále je zodpovědna za výpočet pamětí cache. Veškeré vypočítané výsledky si ukládá do vnitřní paměti, kde ukládá stavy simulovaného programu pro každý tik počítačových hodin. Tyto stavy pak na požádání posílá přes API GUI vrstvě. Engine komponenta je na GUI nezávislá.

VVP proces je externí proces programu Icarus Verilog, který dostane na vstupu simulovaný program a popis procesoru (zkompilovaný popis procesoru v jazyce Verilog) a vygeneruje výstupní soubor typu VCD.

VCD soubor je výstupní soubor VVP procesu. Obsahuje počáteční hodnoty všech registrů, pamětí a jejich spojení a následně jejich změny v čase. Tento soubor je následně zpracován a využit v Engine komponentě.



Obrázek 4.1: Architektura programu FitMips

4.2 Stavový diagram

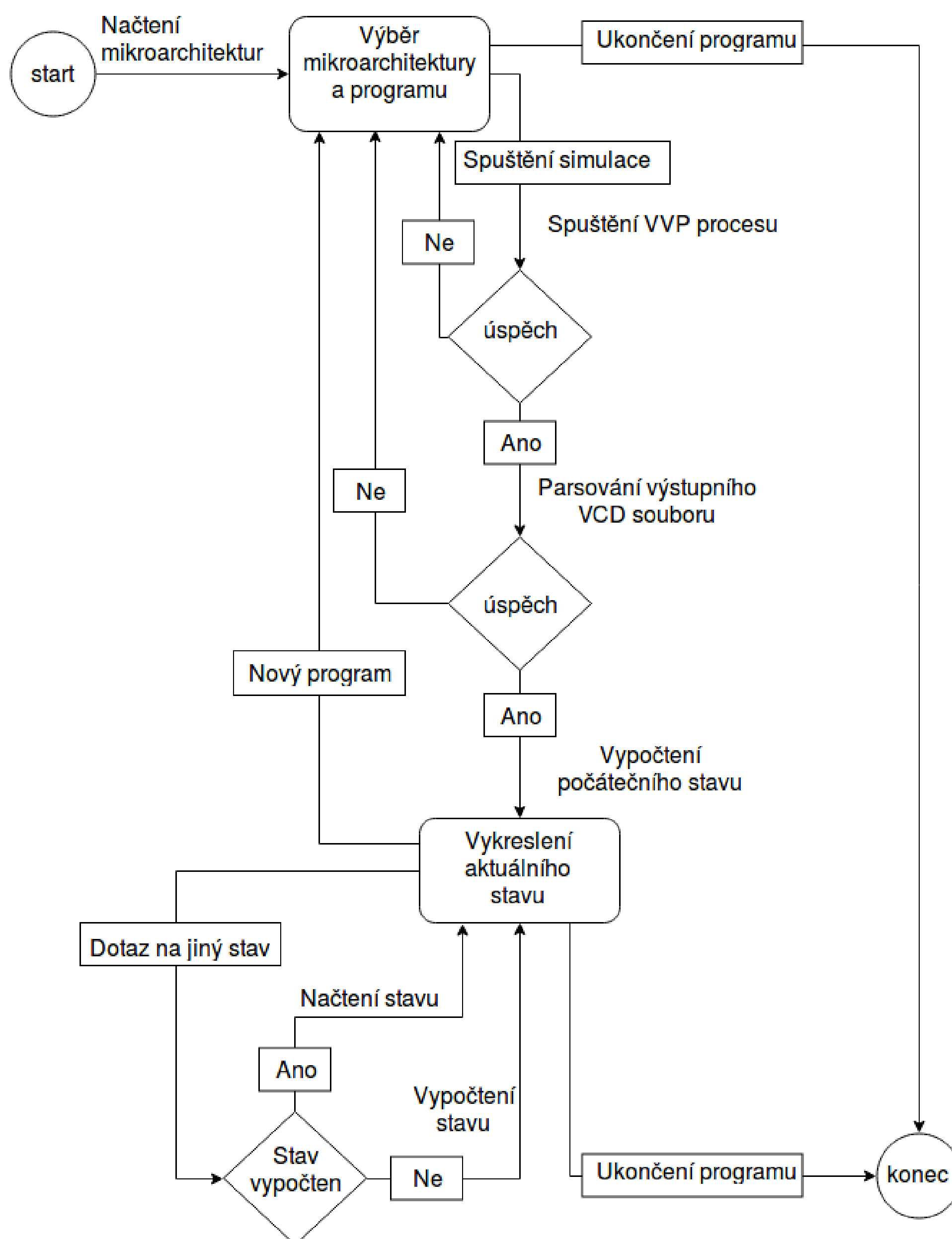
V této sekci se zabýváme popisem stavů, které nastávají v aplikaci FitMips. Celý diagram se nalézá na obrázku 4.2. Následuje popis jednotlivých stavů a jejich přechodů.

Start při startu aplikace dochází ke skenování adresáře (buď definovaného pomocí proměnné prostředí „SCAN_DIR“ případně aktuální pracovní složky) a načítání dostupných mikroarchitektur a ukázkových programů z dané složky. Více informací lze nalézt v sekci 5.3.

Výběr mikroarchitektury a programu je úvodní stav, do kterého se uživatel dostane po startu aplikace. Zde si uživatel vybere, jaký program s jakou pamětí a dalším nastavením chce spustit. Po výběru dojde ke spuštění procesu, který spočítá stav vybrané mikroarchitektury v jednotlivých ticích hodin počítačových hodin a který uloží výsledky do souboru. Tento soubor je zpracován a vykreslí se počáteční stav. Pokud nějaká část tohoto procesu selže, vyskočí chybová hláška a uživatel může znova vyzkoušet spustit jiný program nebo vybrat jinou cílovou mikroarchitekturu.

Vykreslení aktuálního stavu je stav, ve kterém uživatel se vyskytuje nejvíce času. Simulovaný program je spuštěn a nachází se v určitém stavu, ve kterém sledujeme obsah registrů, paměti a cache. Uživatel může pomocí tlačítek na ovládacím panelu měnit stav programu. Při každém dotazu se program zeptá Engine komponenty, zda je požadovaný stav

Stavový diagram FitMips



Obrázek 4.2: Stavový diagram programu FitMips

již vypočítán. Na základě toho je tento stav buď načten přímo načte nebo se mezitím vypočítá a uloží. Nový stav je znova vykreslen. Pomocí menu je možné spustit jiný program a tím pádem přejít do předhozího stavu uvedeného v tomto seznamu.

4.3 Uživatelské rozhraní

Návrh uživatelské rozhraní, které je zobrazeno na obrázku 4.3 je rozděleno na části, které jsou níže popsány.

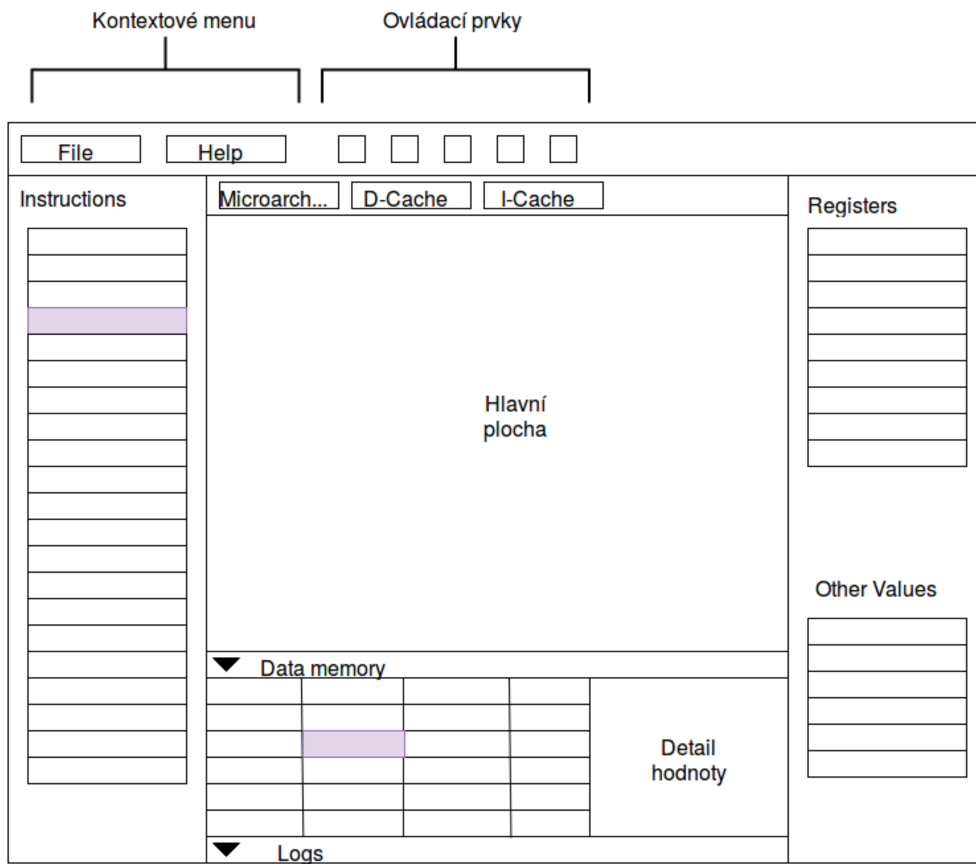
horní část zde se nachází kontextové menu a ovládací prvky. V kontextovém menu uživatel si může nechat zobrazit nový dialog pro spuštění nového programu nebo si zobrazit informace o tomto programu. Nachází se zde též ovládací prvky, které slouží k ovládání simulovaného programu. Jsou to tlačítka reset, backward step (krok zpět), next step (krok vpřed), run (běh bez zastavení), stop (pozastaví běžící program).

levá část zde uživatel vidí obsah instrukční paměti. Právě prováděná instrukce je podbarvena.

pravá část je rozdělena na dvě části. V hořejší uživatel vidí obsah registrů, v té dolní obsah vybranných proměnných z daného schématu mikroarchitektury.

prostřední část v prostřední části je možné přepínat záložky, které mají rozdílný obsah. Tyto záložky jsou „Microarchitecture“, „Data cache“ a „Instruction cache“. V záložce „Microarchitecture“ uživatel vidí schéma procesoru, v záložkách „Instruction cache“ a „Data cache“ jsou přítomny vizualizace datové, respektive instrukční cache.

dolní část má dvě záložky. V první je znázorněn pomocí tabulky obsah datové paměti. Pokud uživatel klikne na buňku, zobrazí se mu v oblasti označené jako „Detail hodnoty“ veškeré možné informace o této buňce, zejména interpretaci hodnoty v různých číselných základech. Ve druhé záložce jsou přítomny aplikační logy.



Obrázek 4.3: Návrh uživatelského rozhraní programu FitMips

Kapitola 5

Realizace

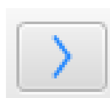
Účelem této kapitoly je poskytnout čtenáři přehled, jakým způsobem byly implementovány jednotlivé části aplikace. V této kapitole se nachází komentované výňatky kódu v jazyce Java, ukázkových programů a XML souborů. Tato kapitola není uživatelský manuál. Uživatelský manuál je k dispozici jako příloha této bakalářské práce.

5.1 Uživatelské rozhraní

Uživatelské rozhraní je implementováno za pomocí frameworku JavaFX. Byl použita architectura MVC, následovat budou ukázky kódů jednotlivých částí a to na příkladu tlačítka „Next step“ (viz. obrázek 5.1), který je zodpovědný za načtení dalšího stavu programu.

5.2 Prezenční vrstva

Prezenční vrstva aplikace je z valné části nakonfigurovaná pomocí FXML souborů, v menší míře je i generována přímo v javě (to může být příklad obsahu datové paměti, kde se obsah generuje i na základě velikosti použité architektury. Následuje konfigurace zmíněného tlačítka.



Obrázek 5.1: Tlačítko další krok

```
<Button fx:id="nextButton" disable="true" mnemonicParsing="false"
    onAction="#nextStep">
   <graphic>
      <ImageView fitHeight="16.0" fitWidth="16.0" pickOnBounds="true"
          preserveRatio="true">
         <image>
            <Image url="@../icons/make_step.png" />
         </image>
      </ImageView>
   </graphic>
   <tooltip>
      <Tooltip text="Next step" />
   </tooltip>
</Button>
```

Jak si můžete všimnout, FXML element „Button“ zastřešuje celý tento prvek. Nejdůležitější atributy jsou „fx:id“, pomocí kterého se dá tento prvek v controlleru jednoznačně identifikovat a „onAction“, který naproti tomu identifikuje akci, která se spustí při zmáčknutí tlačítka.

Následují další elementy a to „ImageView“, který definuje cestu k použitému obrázku a „Tooltip“, který zobrazí malou nápovědu při přejetí myší.

5.2.1 Controller vrstva

Controller vrstva zajišťuje interakci mezi prezenční vrstvou a modelem. Na dalším příkladu ukáži, jak snadné je zaintegrovat tlačítko do controlleru (třída **ProcessorInstanceController**). Stačí přidat anotaci FXML, pojmenovat proměnnou stejně jako „fx:id“ atribut FXML souboru.

```
@FXML
private Button nextButton;
```

Následuje ukázka event handleru ze stejné třídy.

```
@FXML
private void nextStep(ActionEvent event) {
   ISimulation simulation = allData.getSimulation();

   try {
      synchronized (lock) {
         SchemaInHistoryPointModel model = simulation.nextStep();
         redraw(model, false);
      }
   } catch (Exception e) {
      GuiUtils.processThrowable(e);
   }
}
```

Rozeberme si následují řádky:

ISimulation je API, které se využívá při komunikaci mezi GUI komponentou a Engine komponentou. Toto API je detailněji popsané v další sekci.

try blok se přes API dotazuje na další stav programu. Jakmile dostane model tohoto stavu, vykreslí jej. Tento blok je synchronizovaný, aby vždy docházelo pouze k jednomu volání přes API v konkrétním časovém okamžiku.

catch blok v případě nečekané chyby se provede kód, který tuto chybu zpracuje. Více viz. sekce o logování a chybových hláškách 5.7.

5.2.2 API

Modelová vrstva se volá přes API (třída **ISimulation**). Ukázka tohoto API je k nahlédnutí níže.

```
public interface ISimulation {

   IWireResolver getResolver();

   SchemaInHistoryPointModel start() throws Exception;

   SchemaInHistoryPointModel restart();

   SchemaInHistoryPointModel nextStep() throws Exception;

   SchemaInHistoryPointModel getCurrentStep();

   SchemaInHistoryPointModel previousStep() throws Exception;

   CacheProperties getCacheProperties(String cacheId);
}
```

Následuje popsaný seznam možných volání:

getResolver vrací tzv. „wire resolver“. Tento objekt umí vrátit runtime objekt hodnoty, která se mění v čase pomocí cesty. Tato cesta k hodnotě je stejná jako ve Verilog modelu pro danou mikroarchitekturu

start spustí simulaci a vrátí model počátečního stavu

restart vrátí simulaci na začátek a vrátí model počátečního stavu

nextStep posune simulaci o jeden krok dopředu a vrátí model tohoto stavu

getCurrentStep nemění simulaci, pouze vrátí aktuální model stavu

previousStep vrátí simulaci o krok vzad a jeho stav

getCacheProperties poskytne informace o cache paměti referencované pomocí id

5.3 Načítání mikroarchitektur

Při startu programu FitMips dochází k načtení dostupných konfigurací mikroarchitektur a ukázkových demo programů. Vždy se načte mikroarchitektura pro zjednodušený 32-bitový MIPS procesor a programy na výpočet maxima v poli a na vypočtení dvojkového logaritmu desetinného čísla, kde výsledkem je celé číslo.

Dále probíhá skenování. Skenuje se vždy jedna z těchto dvou složek:

1. složka definovaná pomocí proměnné prostředí „SCAN_DIR“
2. aktuální pracovní adresář, pokud není proměnná „SCAN_DIR“ definována

Pokud se ve skenované složce nachází soubor typu HEX, přidá se do seznamu známých HEX souborů (které mohou být interpretovány jako obsah paměti při spuštění simulovaného programu). HEX soubory musí pro správné fungování obsahovat alespoň tolik slov, jaká je velikost cílové paměti ve slovech.

Dále se více zkoumají soubory typu XML. Pokud FitMips při skenování na tento soubor narazí, pokusí se jej interpretovat jako konfiguraci mikroarchitektury. Pokud se vydaří skenovaný soubor interpretovat, načte se do seznamu známých mikroarchitektur.

5.3.1 Ukázkový program

Jeden ze dvou ukázkových programů, které jsou obsaženy v programu FitMips, je algoritmus pro hledání maxima v poli. Instrukce programů jsou zakódovány do HEX souborů. Ukázkový program se nechází v tabulce 5.1.

Ukázkový program zavolá funkci max s adresou pole a s počtem prvků v poli (uloženo na adrese 0x8). Jakmile funkce projde celé pole, vrátí maximum, které se uloží do paměti na adresu 0x4. Poté se program cyklí v nekonečné smyčce.

Ukázka je zkrácená. Zdrojový soubor obsahuje na každém řádku hodnotu ve sloupečku „Kódování“. Další sloupečky jsou přítomny jako řádková poznámka, která začíná dvěma lomítky. Dále jsou přítomny prázdná slova, která jsou v ukázce vynechána.

Tabulka 5.1: Ukázková instrukční paměť pro algoritmus nalezení maxima v poli

Kódování	Instrukce	Poznámka
2004000c	addi $a0, $zero, 12	int * arr = 0x0000000C
8c050008	lw $a1, 8($zero)	int n = * 0x00000008
0c000005	jal max	go to max function
ac020004	sw $v0, 4($zero)	store result into memory
1000ffff	beq $zero, $zero, end	end cycle, stop here
8c820000	lw $v0, 0($a0)	function max begins here, res = *arr;
20880004	addi $t0, $a0, 4	int * ptr = arr + 1, address for second element
20090001	addi $t1, $zero, 1	int i = 1
11250007	beq $t1, $a1, end_cycle	begins for cycle, if i == n, then end cycle
11250007	beq $t1, $a1, end_cycle	begins for cycle, if i == n, then end cycle
8d0a0000	lw $t2,0($t0)	int number = *ptr
004a582a	slt $t2,0($t0)	tell if actual number is smaller than result
11600001	beq $t2, $zero,add_constants	if previous result false, then skip next step
01401020	add $v0, $t2, $zero	result = number
21290001	addi $t1, $t1, 1	i++
21080004	addi $t0, $t0, 4	ptr++ (next address, it means +4 bytes)
1000fff8	beq $zero, $zero, for_cycle	go to the beginning of the cycle
1fe00008	jr $ra	end cycle here, jump back to main
00000000	NOOP	

Tabulka 5.2: Ukázková datová paměť pro algoritmus nalezení maxima v poli

Adresa	Kódování	Poznámka
0x00	00000000	hodnota je vždy nulová
0x04	00000000	adresa rezervovaná pro výsledek
0x08	00000003	velikost pole, hodnota je 3
0x0C	fffffffb	první prvek pole, hodnota je -5
0x0C	00000109	druhý prvek pole, hodnota je 265
0x10	00000012	třetí prvek pole, hodnota je 18

5.3.2 Ukázková datová paměť

Pro ukázkový program hledání maximálního prvku v pole je ukázkový obsah paměti, který můžeme nahrát do datové paměti při spuštění. Protože ukázková mikroarchitektura je 32-bitová, mají všechna slova uvedená ve sloupečku kódování velikost 4 byty. Ukázkový obsah paměti se nalézá v tabulce 5.2.

Skutečný soubor je uložen v HEX souboru, obsahuje pouze obsah sloupečku „Kódování“ a dále obsahuje další prázdná slova, která jsou

5.3.3 Ukázková XML konfigurace

Následuje ukázková konfigurace mikroarchitektury. Kompletní příklad, jak nakonfigurovat novou mikroarchitekturu naleznete v přiložené dokumentaci.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<ArchitectureConfiguration>
    <name>Mips32 simplified</name>
    <description>Simplified design of Mips 32-bit architecture.

These instructions are supported:
Arithmetic &amp; logical:
add, sub, and, or, slt, addi, addu.qb, addu_s.qb

Memory:
lw, sw

Branching &amp; jumping:
beq, jal, jr

Important note:
Instruction jr does not match exactly the original Mips
    specification. The opcode differs.
Encoding: 0001 11ss sss0 0000 0000 0000 0000 1000</description>
    <engineConfiguration...>
    <guiConfiguration...>
</ArchitectureConfiguration>
```

Konfigurace mikroarchitektury má tyto elementy:

name je název mikroarchitektury, jakým se zobrazí uživateli

description je popisek mikroarchitektury, který se zobrazí vedle jméma

engineConfiguration konfigurace Engine komponenty. Není zde rozvinuta, ukázka následuje v další sekci

guiConfiguration konfigurace GUI komponenty. Ukázka následuje v sekci 5.3.3.2

5.3.3.1 XML konfigurace Engine komponenty

Následuje ukázka konfigurace, která je použita k nastavení Engine komponenty. Kompletní průvodce nastavením se nachází v dokumentaci.

```xml
<engineConfiguration>
    <archSize>32</archSize>
    <simulationConfiguration>
        <tickLength>2</tickLength>
        <vvpFile>resource:defaults/architecture.vvp</vvpFile>
        <vcdOutputFile>test.vcd</vcdOutputFile>
        <memoryConfigs...>
    </simulationConfiguration>
    <cacheConfiguration>
        <cacheConfiguration...>
        <cacheConfiguration...>
    </cacheConfiguration>
    <instructionConfiguration...>
    <regNameConfiguration>
        <registers>32</registers>
        <nameMap...>
    </regNameConfiguration>
</engineConfiguration>
```

Element „engineConfiguration“ má tyto další položky:

archSize značí počet bitů, které architektura má.

simulationConfiguration je nastavení, které je využito ke spuštění VVP procesu a k parsování jeho výstupu.

cacheConfiguration nadefinuje cache procesoru a jejich chování

instructionConfiguration je nastavení, pomocí kterého je možné parsovat instrukce daného procesoru do čitelné podoby.

regNameConfiguration nastavuje aliasy pro různé registry.

5.3.3.2 XML konfigurace GUI komponenty

Následuje zkrácená ukázka konfigurace GUI komponenty. Kompletní návod k nastavení se nachází v přiložené dokumentaci.

```
<guiConfiguration>
    <modelItems...>
    <pcCounter...>
    <registerConfiguration...>
    <dataMemory...>
    <instructionMemory...>
    <otherValues...>
</guiConfiguration>
```

V této části XML souboru se nachází tyto položky:

modelItems v tomto elementu se nachází nejvíce položek. Tento element definuje schéma mikroarchitektury.

pcCounter je reference na hodnotu PC čítače. Referencuje se pomocí cesty, která je stejná jako ve Verilog schématu dané mikroarchitektury

registerConfiguration obsahuje číslo (počet registrů) a referenci na počátek pole hodnot registrů ve schématu.

dataMemory obsahuje číslo (velikost datové paměti ve slovech) a referenci na počátek datového pole ve schématu

instructionMemory obsahuje číslo (velikost instrukční paměti ve slovech) a referenci na počátek instrukčního pole ve schématu

otherValues obsahuje seznam dvojic popisek + reference hodnot, které se budou vypisovat ve zvláštní tabulce

5.4 Spuštění VVP

Samotná simulace programu probíhá v externím procesu VVP (runtime programu Icarus Verilog). Tento program následně vygeneruje výstupní VCD soubor, který program přečte a zpracuje. O spuštění procesu se stará třída **SimulationRunner**. Nejdůležitější metoda této třídy je metoda „run“, která je uvedena pod tímto textem.

```
public VcdModel run() throws IOException, InterruptedException,
    SyntaxErrorException, URISyntaxException {
   log.info("Running runtime...");
   prepareEnvironment();

   File processOutput = tempDir.resolve(OUTPUT).toFile();
```

```
    List<String> args = buildCommand();
    ProcessBuilder pb = new ProcessBuilder(args);
    pb.redirectErrorStream(true);
    pb.directory(tempDir.toFile());
    pb.redirectOutput(processOutput);

    log.info("Executing system command: {}", args);
    Process process = pb.start();
    int exitValue = process.waitFor();
    logProcess(processOutput);

    if (exitValue != 0) {
       throw new IllegalStateException("Simulation process has ended
           with an invalid exit value: " + exitValue);
    }

    Path outputFile = tempDir.resolve(vcdOutputFile);
    log.info("Parsing output vcd file: {}", outputFile);

    VcdModel model = VcdReader.readModel(new
        FileInputStream(tempDir.resolve(vcdOutputFile).toFile()));

    log.info("Vcd file has been parsed successfully.");
    return model;
}
```

Průběh této metody lze popsat v těchto krocích:

1. Nejprve probíhá příprava prostředí. Vytvoří se dočasná složka na hostujícím operačním systému, do které se nakopírujou potřebné soubory

2. V druhém kroce se připraví spuštění procesu. Do argumentů se předají jména souborů, které budou využity jako obsah pamětí

3. Program se spustí a počká se na výsledek. Výstup z procesu za zaloguje

4. Nakonec se přečte výstupní soubor, který se naparsuje do vnitřního modelu

5.5 Parsování VCD

K parsování VCD souboru dochází poté, co úspěšně doběhne externí VVP proces. VCD je výstupní soubor VVP procesu, který obsahuje deklarace proměnných a změnu hodnot v čase. Tento soubor je potřeba převést do vnitřního modelu FitMipsu. Tento problém řeší třída **VcdReader**. Ukázka veřejné metody „readModel“ je k vidění pod tímto odstavcem.

```
public static VcdModel readModel(InputStream is) throws IOException,
    SyntaxErrorException {
   VcdModel model = new VcdModel();
   Module mainModule = new Module();
   Map<String, Wire> allWires = new HashMap<>();
   //main loop
   try (WordReader reader = new WordReader(is)) {
      while (true) {
         String word = reader.nextWord();
         if (word == null) {
            break;
         }

         if (!isCommand(word)) {
            throw new SyntaxErrorException(reader.getLineNumber(),
                "Command expected.");
         }

         Command cmd = parseCommand(word);
         int line = reader.getLineNumber();
         switch (cmd) {
            case DATE: model.setDate(readDate(reader)); break;
            case VERSION: model.setVersion(readStringInfo(reader));
                break;
            case TIMESCALE:
                model.setTimescale(readStringInfo(reader)); break;
            case SCOPE:
               Module scopeModule = parseScopeModule(reader,
                   mainModule);
               readModule(reader, scopeModule, allWires);
               break;
            case ENDDEFINITIONS:
               readHistory(reader, allWires);
               break;
            case UNKNOWN:
               log.warn("Unknown command {} on line {}.", word, line);
               readCommand(reader, x -> {});
               break;
            default: throw new SyntaxErrorException(line, "Command " +
                word + " not expected.");
         }

      }
   }

   model.setMainModule(mainModule);
   model.setAllWires(allWires);
   return model;
}
```

V ukázce se nachází cyklus, ve se parsují jednotlivé části modelu. Na nejvyšší úrovni se očekávají tokeny, které jsou příkazy. Příkazy ve VCD formátu vždy začínají znakem „$“. Níže se nachází popis způsobu, jakým se parsují očekávané příkazy na nejvyšší úrovni.

DATE je příkaz, za kterým následuje datum. Datum se naparsuje pomocí metody „parseDate“.

VERSION je příkaz, za kterým následuje verze Verilogu, která se naparsuje pomocí metody „readStringInfo“.

TIMESCALE je příkaz, za kterým následuje časový údaj (obvykle „1s“). Parsuje se opět pomocí metody „readStringInfo“.

SCOPE je příkaz, za kterým následuje model proměnných. Deklarace modelu je zpracována v metodě „parseScopeModule“. Definice modelu a jeho proměnných se rekurzivně parsuje v metodě „readModule“ – model může být zanořený.

ENDDEFINITIONS je příkaz, který označuje místo, kde končí definice modelu a kde začíná výpis hodnot v různých časech. V čase 0 jsou vypsány počáteční hodnoty všech proměnných, v dalších časech jsou hodnoty vypsány pouze pokud jsou změněny.

5.6 Výpočet cache

Výpočet cache paměti se narozdíl od výpočtu registrů, datové paměti a dalších hodnot přítomných v modelu mikroarchitektury nepočítá v externím procesu, ale za běhu programu FitMips, když se počítá aktuální stav systému v daném časovém okamžiku.

O výpočet se stará třída **CacheRuntime**. Zvenčí je volaná metoda „computeNextState“, která má za argumenty vypočítaný předchozí stav a hodnoty proměnných v daném časovém okamžiku.

```
public CacheModelInHistory computeNextState(CacheModelInHistory
    previous, Map<String, SchemaInHistoryPointModel.WireValueInTime>
    values) {
  CacheModelInHistory current = new CacheModelInHistory();

  WireValue writeIn = writeToMemValueProvider.getValue(values);

  WireValue valueOut = readFromMemValueProvider.getValue(values);
  current.setAddressIn(addressValueProvider.getValue(values));
  current.setValueOut(valueOut);
  current.setWriteIn(writeIn);
```

```
    boolean readFromMem =
        isReadFromMemValueProvider.getValue(values).toBoolean();
    boolean writeToMem =
        isWriteToMemValueProvider.getValue(values).toBoolean();

    if (readFromMem) {
        performAction(current, previous, values,
            CacheModelInHistory.ECacheActionType.READ_HIT,
             CacheModelInHistory.ECacheActionType.READ_MISS, (row,
                 cell) -> {});
    } else if (writeToMem) {
        performAction(current, previous, values,
            CacheModelInHistory.ECacheActionType.WRITE_HIT,
             CacheModelInHistory.ECacheActionType.WRITE_MISS, (row,
                 cell) -> row.getBlock()[cell] = writeIn);
    } else {
        doNothing(current, previous);
    }

    current.setPreviousClkAction(previous.getAction());
    previous.setFollowingClkAction(current.getAction());

    return current;
}
```

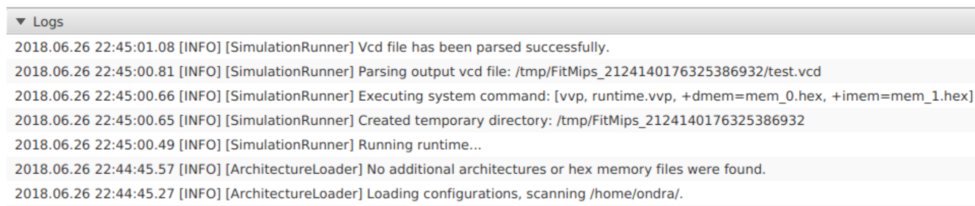
Metoda „computeNextState“ se dá rozdělit na tyto tři části:

1. V prvé části se zjišťují vstupní a výstupní hodnoty paměti, kterou se program snaží simulovat. Také si zjišťuje, zda dochází ke čtení nebo zápisu.

2. V druhé části se přepočítává obsah cache paměti a statistiku, pokud dochází ke čtení nebo zápisu. Tuto funkcionalitu zajišťuje volaná metoda „performAction“, která má tyto argumenty:

    a) **current** aktuálně počítaný stav,
    b) **previous** předchozí stav,
    c) **values** aktuální hodnoty proměnných,
    d) **hitActionType** enum, který označuje HIT pro danou operaci (čtení nebo zápis),
    e) **missActionType** enum, který označuje MISS pro danou operaci čtení nebo zápisu,
    f) **rowModifier** je lambda výraz, který se uplatní pouze při zápisu hodnoty do řádku.

    Pokud nedojde ke čtení ani zápisu, statistika a obsah cache paměti se pouze kopíruje.



Obrázek 5.2: Logger přítomný v programu

Ať už dojde ke čtení, zápisu nebo ani jednomu z těchto uvedených, nedochází ke zbytečnému kopírování stejných hodnot do nového stavu. Na řádky či tabulky, které se nezmění, se pouze předává reference a nedochází tudíž k plýtvání paměti.

3. Propojí se reference. Předchozí stav dostane referenci na aktuální stav a naopak.

5.7 Logování a chybové hlášky

K logování se používá logovací framework SLF4j. Výhodou tohoto frameworku je použití jednotného API bez ohledu na implementaci loggeru.

Pro účely simulátoru FitMips se ukázalo být vhodné implementovat vlastní implementaci kompozitního loggeru, který zobrazuje logy uživateli (viz. obrázek 5.2) a zároveň vypisuje vše na standartní výstup i s detaily, které se běžně nezobrazují.

Tohoto bylo docíleno implementací třídy s názvem **org.slf4j.impl.StaticLoggerBinder**. Jak je vidět níže, tato třída, která implementuje interface **LoggerFactoryBinder**, vytváří tzv. „logger factory“, která se stará o vytváření loggerů pro jednotlivé třídy.

```
package org.slf4j.impl;

import org.slf4j.ILoggerFactory;
import org.slf4j.spi.LoggerFactoryBinder;

public class StaticLoggerBinder implements LoggerFactoryBinder {

   private static final StaticLoggerBinder SINGLETON = new
       StaticLoggerBinder();
   private static final String LOGGER_FACTORY_STRING =
       GuiLoggerFactory.class.getName();

   public static String REQUESTED_API_VERSION = "1.7.25";

   public static final StaticLoggerBinder getSingleton() {
      return SINGLETON;
```

```
    }

    private ILoggerFactory loggerFactory;

    private StaticLoggerBinder() {
        loggerFactory = new GuiLoggerFactory();
    }

    @Override
    public ILoggerFactory getLoggerFactory() {
        return loggerFactory;
    }

    @Override
    public String getLoggerFactoryClassStr() {
        return LOGGER_FACTORY_STRING;
    }
}
```

FitMips implementace „logger factory“ vytvoří takzv. „record holder“, která udržuje až 50 nejnovějších zpráv ve své paměti. Tuto instanci „record holderu“ poskytne nově vytvořeným instancím loggerů a i samotnému grafickému rozhraní.

Na následující ukázce ze třídy **GuiLogger** je vidět, jak se vytvoří text ve formátu „ *«čas» [«úroveň hlášky loggeru»][«jednoduchý název třídy»] «zpráva»*“. Časový formát je „ *yyyy.MM.dd HH:mm:ss.SS*“.

Po vytvoření zprávy se text vytiskne na standartní výstup a pokud tato hláška není úrovně „DEBUG“, předá se do „record holderu“. Metoda „Platform.runLater“ se zde volá proto, aby zprávu zalogovalo vlákno, které se stará o běh GUI a nedocházelo k problémům ve vykreslování.

```
private void log(String logLvl, String msg) {
    LocalDateTime date = LocalDateTime.now();

    StringBuilder sb = new StringBuilder();
    sb.append(TIME_FORMAT.format(date)).append("
        [").append(logLvl).append("] [").append(name).append("]
        ").append(msg);

    String logRow = sb.toString();

    System.out.println(logRow);

    if (!DEBUG_LEVEL.equals(logLvl)) {
        Platform.runLater(() -> recordHolder.log(logRow));
    }
}
```

5.7.0.1 Chybové hlášky

Pokud dojde k chybě za běhu programu, následující kód ze třídy **GuiUtils** se postará o to, aby byla chyba řádně zalogována. Následně dojde k vytvoření varovného okna s popisem chyby, které se zobrazí uživateli.

```
public static void processThrowable(Throwable t) {
   log.error(ERROR_MSG, t);

   StringWriter sw = new StringWriter();
   t.printStackTrace(new PrintWriter(sw));

   Alert alert = new Alert(Alert.AlertType.ERROR);
   alert.setTitle("An error occurred");
   alert.setHeaderText(ERROR_MSG);
   alert.setContentText(t.getMessage());

   TextArea excArea = new TextArea(sw.toString());
   alert.getDialogPane().setExpandableContent(excArea);
   alert.showAndWait();
}
```

5.8 Testování

Cílem následující sekce je poskytnout čtenáři přehled, jakým byly testovány různé části simulátoru FitMips.

5.8.1 Uživatelské rozhraní

Testování uživatelského rozhraní bylo provedeno zběžným proklikáním i za pomoci několika dobrovolníků. Ovládací prvky jsou funkční.

Došlo i na drobné připomínky. Například můj vedoucí, pan Ing. Michal Štepanovský, Ph.D, mi doporučil, ať není vizualizace obsahu instrukční a datové paměti oddělena od vizualizace architektury. Tento návrh jsem vzal v úvahu, následkem čehož jsem změnil přeuspořádání zmíněných částí uživatelského rozhraní.

5.8.2 Integrace FitMipsu s VVP

Parser VCD souborů a počítání stavů simulovaného programu bylo testováno při vývoji. Kontrola správnosti výsledků probíhala porovnáním s výsledky programu GTKWave, který taky umí parsovat VCD soubory a umí sledovat vývoj hodnot proměnných v čase.

Správnost funkčnosti cache byla testována manuálně. Byly zkoušeny různé parametry nastavení (k dispozici jsou asociativita, počet řádků v tabulce a

velikost bloku ve slovech). V případě nefunkčního zobrazení nebo špatných hodnot byl problém vyřešen.

Samotné spouštění procesu VVP přes FitMips bylo vyzkoušeno na operačních systémech Linux distribuce Ubuntu 16.04 a Windows 10. Na operačním systému Windows 10 se zjistilo, že při spouštění docházelo k deadlocku a program VVP nebyl ukončen. Tento problém byl způsoben nesprávným nebo chybějícím odchytávání standardního a chybového výstupu. K deadlockům již nedochází, problém byl odstraněn.

5.8.3 Mikroarchitektura

Samotný popis mikroarchitektury ve Verilogu jsem vyvinul již v předmětu BI-APS jako semestrální práci v zimním semestru školního roku 2017/2018. Součástí semestrální práce bylo i napsat programy k výpočtu nejvyšší hodnoty v poli a ke spočítání dvojkového logaritmu reprezentovaného jako celé číslo z desetinného čísla. Tyto programy jsou k dispozici jako ukázkové.

Testování proběhlo spolu s vývojem procesoru a ukázkových programů. Po odevzdání semestrální práce byla funkčnost mikroarchitektury zkontrolována mým nynějším vedoucím Ing. Michalem Štepanovským, Ph.D. Byl nalezen pouze jediný drobný problém a to počítání špatných výsledků v operaci aritmetického posunu v krajních případech. Tento problém je opraven.

Zdrojové kódy popisu ve Verilogu se od té doby prakticky nezměnily. Výjimkou jsou drobné úpravy, které řeší načítání vstupních souborů a ukládání datových polí do výstupního souboru. Tyto úpravy však nemají vliv na sémantiku popisu procesoru a korektnost těchto dílčích změn je zajištěna tím, že integrace mezi FitMips a VVP je ozkoušena.

Závěr

Cílem práce bylo vytvořit simulátor, který umožňuje snadno pochopit práci procesoru MIPS 32-bitové architektury. V první části práce jsem se zabýval analýzou již existujících řešení, ze kterých jsem čerpal inspiraci pro návrh vlastního řešení, které nedostatky stávajících řešení odstraňuje. Popsal jsem důležité pojmy a technologie, které práce využívá. V hlavní části jsem se zabýval návrhem a postupem, který jsem využil při implementaci simulátoru.

Podařilo se mi splnit všechny cíle práce. Můj program FitMips má jednoduché a snadně pochopitelné funkční grafické prostředí. Vykresluje schéma procesoru MIPS v 32-bitové architektuře. Lze se pohybovat v simulačním čase, kdy se vykonává program a na schématu je krásně vidět, kde se mění jaké hodnoty. Tento simulátor i umí ukazovat hodnoty registrů, pamětí a též cache. Program lze spustit na libovolném operačním systému.

Pro více zdatné uživatele zde existuje možnost nakonfigurovat si vlastní mikroarchitekturu. To se dá docílit přidáním XML souboru do složky, kde je program umístěn, spolu se spustitelným VVD souborem, který Icarus Verilog (program na simulaci hardwaru) používá k simulaci. Vše je zdokumentované. Dokumentaci lze nahlédnout v příloze této práce.

Rád bych nastínil vizi, jakým směrem se může vývoj aplikace nadále vyvíjet. Průměrný uživatel by ocenil, kdyby se zpřívětilo vytváření programů bez nutnosti psát instrukce zakódované v šestnáctkové soustavě.

Konfigurace architektury by šla zjednodušit. Hlavním problémem je to, že při konfiguraci nové architektury se musí komplexně popisovat obrázkové schéma. Musí se u každého drátu nastavovat začátek, konec i jednotlivé klouby toho, jak bude na výsledné obrazovce vypadat. Podobně je to i s moduly, popisky atd. Uživatel by ocenil, kdyby se všechny tyto položky nemusely konfigurovat, ideálně by se použil popis schématu ve Verilogu a aplikace by si vygenerovala schéma sama.

Vylepšit by se dala i grafika. Dají se přidat různé barvy a loga, aby vypadala aplikace lépe (ta teď využívá systémové barvy a grafiku, vlastní se používá jen v případě nejvyšší nutnosti). Do této sekce bych přidal i vylepšení

flexibility využití již existujících UI komponent (jako třeba instrukční paměť, registry, cache a další), čímž mám na mysli možnost výběru, kde tyto komponenty na obrazovce budou viditelné nebo dokonce i vymyslet a nakonfigurovat vlastní.

Literatura

[1] Brorsson, M.: MipsIt: a simulation and development environment using animation for computer architecture education. 05 2002, doi:10.1145/1275462.1275479.

[2] 3. Processor organization, instruction set. Poslední modifikace 2016-03-17, [cit. 2018-06-27]. Dostupné z: `https://cw.fel.cvut.cz/old/courses/a0b36apo/en/tutorials/03/start`

[3] University, M. S.: MARS (MIPS Assembler and Runtime Simulator). 2014, [cit. 2018-06-28]. Dostupné z: `http://courses.missouristate.edu/KenVollmar/mars/index.htm`

[4] MIPS: MIPS32 Architecture. [cit. 2018-06-28]. Dostupné z: `https://www.mips.com/products/architectures/mips32-2/`

[5] Ing. Michal Štepanovský, P.: 4. Seminar - Basic computer components II, Single cycle CPU. 2017, [cit. 2018-06-28]. Dostupné z: `https://edux.fit.cvut.cz/courses/BI-APS/tutorials/04/start`

[6] Williams, S.: Icarus Verilog. [cit. 2018-06-28]. Dostupné z: `http://iverilog.icarus.com/home`

[7] Williams, S.: *vvp - Icarus Verilog vvp runtime engine*. Manuál pro příkaz vvp v operačním systému Linux.

[8] VCD-Format Files. [cit. 2018-06-28]. Dostupné z: `ftp://ece.buap.mx/pub/Secretaria_Academica/SDC/Active_HDL_4.2_Student_Version_Installer/Doc/avhdl/avh00229.htm`

Příloha A

Seznam použitých zkratek

API Application interface

GUI Graphical user interface

UI User interface

XML Extensible markup language

MVC Model-View-Controller (typ architektury softwaru)

VVP Icarus Verilor vvp runtime engine

VCD Value Change Dump (výstup z VVP)

Příloha B

Obsah přiloženého USB

- readme.txt .... stručný popis obsahu USB
- build
  - FitMips .... adresář s buildem
    - demo .... adresář obsahující demo konfiguraci a demo programy
    - lib .... adresář se závislostmi
    - FitMips-1_0_0.jar .... spustitelný soubor s implementací
    - documentation.pdf .... text dokumentace ve formátu PDF
  - FitMips-1_0_0.zip .... zip soubor s obsahem programu
- src
  - impl .... zdrojové kódy implementace
  - verilog .... zdrojové kódy popisu MIPS32 mikroarchitektury
  - thesis .... zdrojová forma práce ve formátu LaTeX
  - documentation.odt .... zdrojová forma dokumentace ve formátu ODT
- thesis.pdf .... text práce ve formátu PDF
- documentation.pdf .... text dokumentace ve formátu PDF

Příloha C

Dokumentace

Následuje příloha s dokumentací.

Dokumentace a manuál k použití programu FitMips

Simulátor procesoru

Obsah

1 Úvod a instalace....3
1.1 Systémové požadavky....3
1.2 Instalace....4
1.3 Spuštění programu....4
1.3.1 Spuštění přes příkazovou řádku....4
2 Použití....5
2.1 Výběr mikroarchitektury....5
2.1.1 Nastavení paměti....6
2.1.2 Nastavení cache....6
2.1.10 Spuštění simulace....7
3 Ovládání aplikace....8
3.1 Menu a ovládací tlačítka....8
3.1.1 Menu....8
3.1.2 Ovládací tlačítka....9
3.2 Panely....9
3.2.1 Mikroarchitektura....9
3.2.2 Cache....10
3.3 Instrukční paměť....11
3.4 Datová paměť....12
3.5 Registry....13
3.6 Tabulka zajímavých hodnot....14
3.7 Logování....14
4 Vývoj vlastních programů ke spuštění....16
5 Vývoj vlastní mikroarchitektury....18
5.1 Jak na popis hardwaru ve Verilogu....18
5.1.1 Vytvoření spustitelného skriptu....19
5.2 Konfigurace architektury v XML....20
5.2.2 Konfigurace enginu....20
5.2.26 Konfigurace GUI....26
6 Řešení problémů....32
6.1 Nejde spustit program na mikroarchitektuře....32
6.1.1 Není známá cesta k VVP....32
6.1.2 Neodpovídající verze....32
6.2 Program nenajde dodatečnou mikroarchitekturu....33
6.2.1 Špatně nastavená složka....33
6.2.2 Nevalidní XML soubor....33
6.3 Vlastní nová mikroarchitektura nejde spustit....33

1 Úvod a instalace

Program FitMips je na platformě nezávislý program určený k simulaci a spouštění programů, zejména pro architekturu MIPS. FitMips je upraven pro potřeby výuky počítačově orientovaných předmětů, zejména BI-APS při Fakultě informačních technologií ČVUT v Praze.

1.1 Systémové požadavky

FitMips je multiplatformní program, lze jej spustit na libovolné platformě. Program byl testován na operačních systémech Windows 10 a Ubuntu 16.04. Program zabírá na disku cca 3 MB. Paměťová náročnost za běhu je kolem 400 MiB.

K tomu, aby šlo program spustit a používat, je potřeba, aby na systému byly nainstalovány tyto programy:

- Java Runtime Environment (jre) verze 8 a vyšší
- Icarus Verilog verze 0.9.7

Tip: K jednoduchému spuštění simulátoru doporučuji přidat binární spustitelné soubory do systémové cesty na vašem OS. Tento příkaz na příkazové řádce zjistí, zda je potřebný program přítomen a vypíše jeho verzi.

```
vvp -V
```

Toto je očekávaný výstup:

```
Icarus Verilog runtime version 0.9.7  (v0_9_7)

Copyright 1998-2010 Stephen Williams

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License along
  with this program; if not, write to the Free Software Foundation, Inc.,
  51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
```

1.2 Instalace

Instalace simulátoru je jednoduchá. Stačí rozbalit ZIP soubor FitMips-1_0_0.zip na libovolné místo. Vytvoří se složka FitMips s následujícím obsahem:

- **lib** – složka obsahující knihovny a závislosti.
- **demo** – složka obsahující konfiguraci architektury a demo programy, které jsou skutečně součástí programu. Změnou těchto souborů nedojde ke změně chovaní programu.
- **FitMips-1_0_0.jar** – spustitelný program.
- **documentation.pdf** – tento soubor.

1.3 Spuštění programu

Obvykle postačí dvojí poklepání na spustitelný program, případně lze pravým tlačítkem vybrat, že chceme spustit program pomocí Java 8 Runtime (například Oracle Java 8 Runtime nebo OpenJDK Java 8 Runtime).

1.3.1 Spuštění přes příkazovou řádku

Spuštění přes příkazovou řádku se může hodit například v případech, kdy je potřeba mít dodatečnou kontrolu nad spuštěným programem a nastavit konfigurační parametry. Hodí se i v případě, kdy nemáme zajištěno, že Icarus Verilog je viditelný a je třeba specifikovat cestu k tomuto programu. Tyto parametry se nastaví jako proměnné prostředí.

Další výhoda spouštění přes příkazovou řádku je ta, že na pozadí poběží **logování** programu, což se může hodit při případných problémech.

Parametr **VVP_PATH** nastaví cestu ke spustitelnému VVP souboru. Pozor, je potřeba zadat celou cestu, nejenom složku, ve které se VVP nachází!

Parametr **SCAN_DIR** nastaví složku, která se proskenuje při startu aplikace, kde může najít konfigurace spustitelných architektur ve formátu XML, stejně tak může najít HEX soubory, které se uloží do seznamu známých souborů s obsahem paměti.

Program se spustí pomocí příkazu.

```
java -jar FitMips-1_0_0.jar org.cvut.fit.simulator.gui.MainApp
```

Ukázka s konfigurací pro operační systém Linux a Mac OS:

```
export VVP_PATH=/home/user/iverilog/bin/vvp
export SCAN_DIR=/data/FitMips
java -jar FitMips-1_0_0.jar org.cvut.fit.simulator.gui.MainApp
```

Obdobně pro Windows:

```
set VVP_PATH="C:\iverilog\bin\vvp"
set SCAN_DIR="D:\FitMips"
java -jar FitMips-1_0_0.jar org.cvut.fit.simulator.gui.MainApp
```

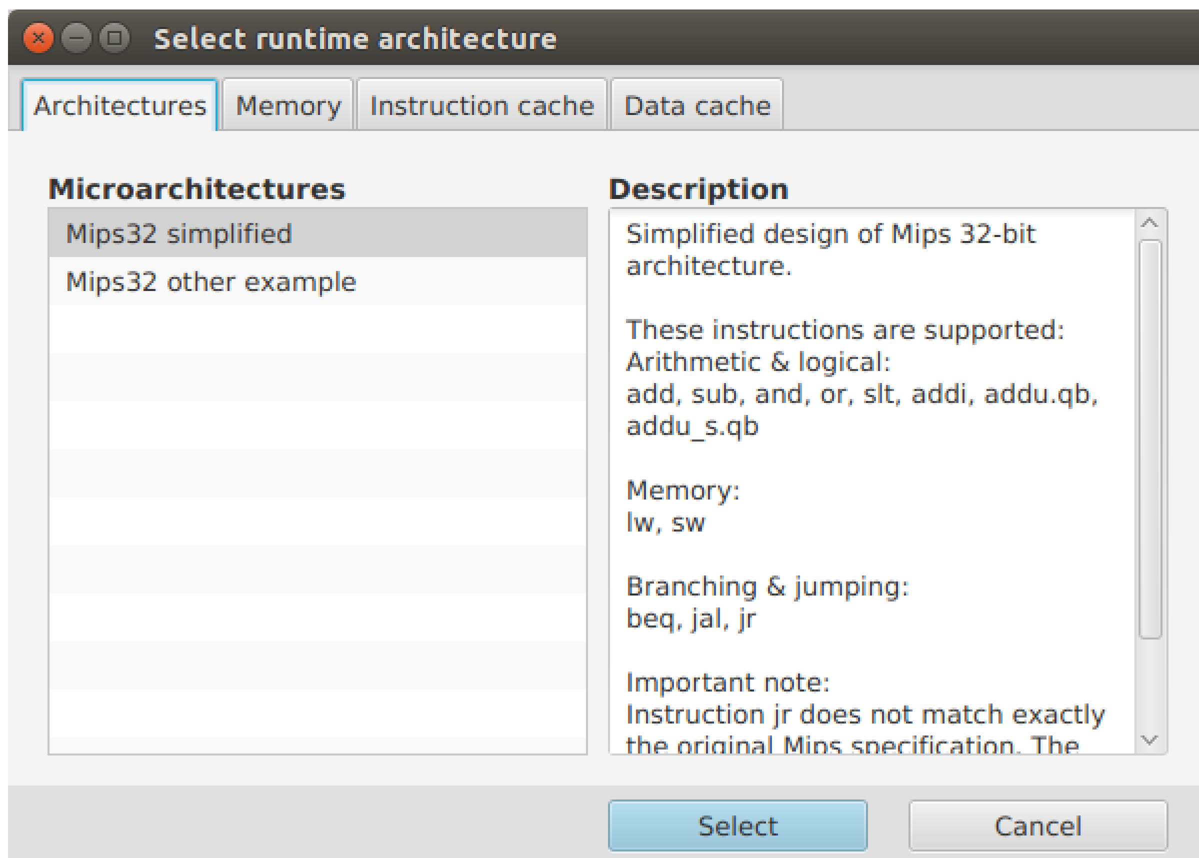
2 Použití

V následujících sekcích popíši jednotlivé části aplikace.

2.1 Výběr mikroarchitektury

Po startu programu vyskočí okno s výběrem cílové architektury. Mikroarchitektura se vybere pomocí kliknutí na název vlevo. Další viditelné panely mají konfigurační charakter. Tato konfigurace bude rozebraná dále v textu.

Pouze architektura s názvem „Mips32 simplified" je standardní součástí programu. Ostatní architektury lze načíst přidáním konfiguračních souborů do pracovní složky programu (může a nemusí být stejná složka, ve které je program). Pro lepší specifikaci konfigurační složky se podívejte do kapitoly Spuštění přes příkazovou řádku.



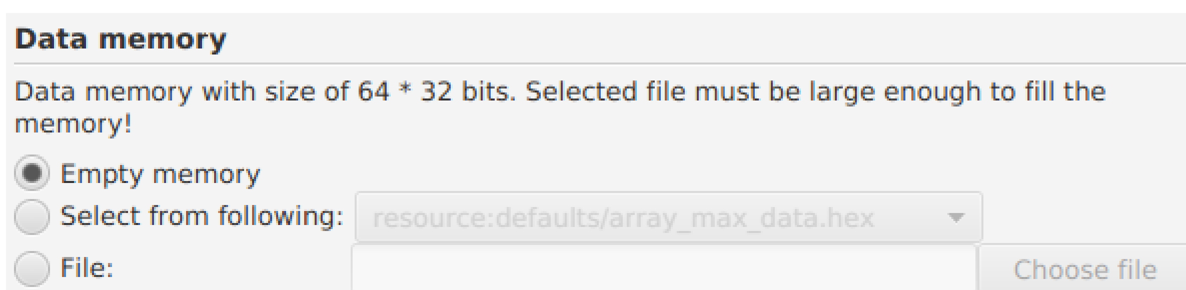
Ilustrace 1: Výběr cílové architektury

2.1.1 Nastavení paměti

Panel „Memory“ obsahuje dva identické formuláře, viz. Ilustrace 2: Nastavení obsahu paměti. Jeden nastavuje obsah datové paměti a druhý obsah instrukční paměti, jinak jsou identické. Tyto formuláře nahrávají obsah paměti v HEX souborech (tedy v textové reprezentaci). Je nutné, aby byly tyto soubory dostatečně velké na to, aby obsáhly celou paměť.

Obsah paměti lze nastavit různými způsoby:

- **Empty memory** – prázdná paměť. Vygeneruje se prázdný obsah paměti.
- **Select from following** – vybere z uvedeného seznamu. V seznamu jsou obsaženy demo paměti (program na výpočet maxima z pole a na výpočet druhého logaritmu zaokrouhleného desetinného čísla), dále pak všechny nalezené HEX soubory v konfigurační složce (opět viz. Spuštění přes příkazovou řádku).
- **File –** umožňuje vybrat soubor kdekoli na disku.



Ilustrace 2: Nastavení obsahu paměti

2.1.2 Nastavení cache

Oba panely „Instruction cache“ a „Data cache“ obsahují stejnou konfiguraci, znázorněnou na obrázku Ilustrace 3: Nastavení cache.

Nastavit cache lze pomocí těchto parametrů:

- **Associativity** – udává úroveň asociace, rozsah 0 – 3, (0 – přímo assoociovaná cache, 3 – cache s 8 asociovanými tabulkami).
- **Rows** – nastavuje počet řádků v jedné tabulce, rozsah od 2 až do 32, validní hodnoty jsou mocniny 2.
- **Block size (words)** – nastaví rozsah, kolik slov bude obsahovat jeden blok cache (tedy jeden řádek).
- **Replacement strategy** – strategie nahrazování platných bloků. Tato strategie je náhodná, nelze měnit.
- **Type –** typ cache, výchozí typ je write through. Nelze změnit.

Ilustrace 3: Nastavení cache

2.1.3 Spuštění simulace

Simulace se spustí stisknutím tlačítka „Select“, viz. Ilustrace 1: Výběr cílové architektury. Tímto se spustí série těchto akcí:

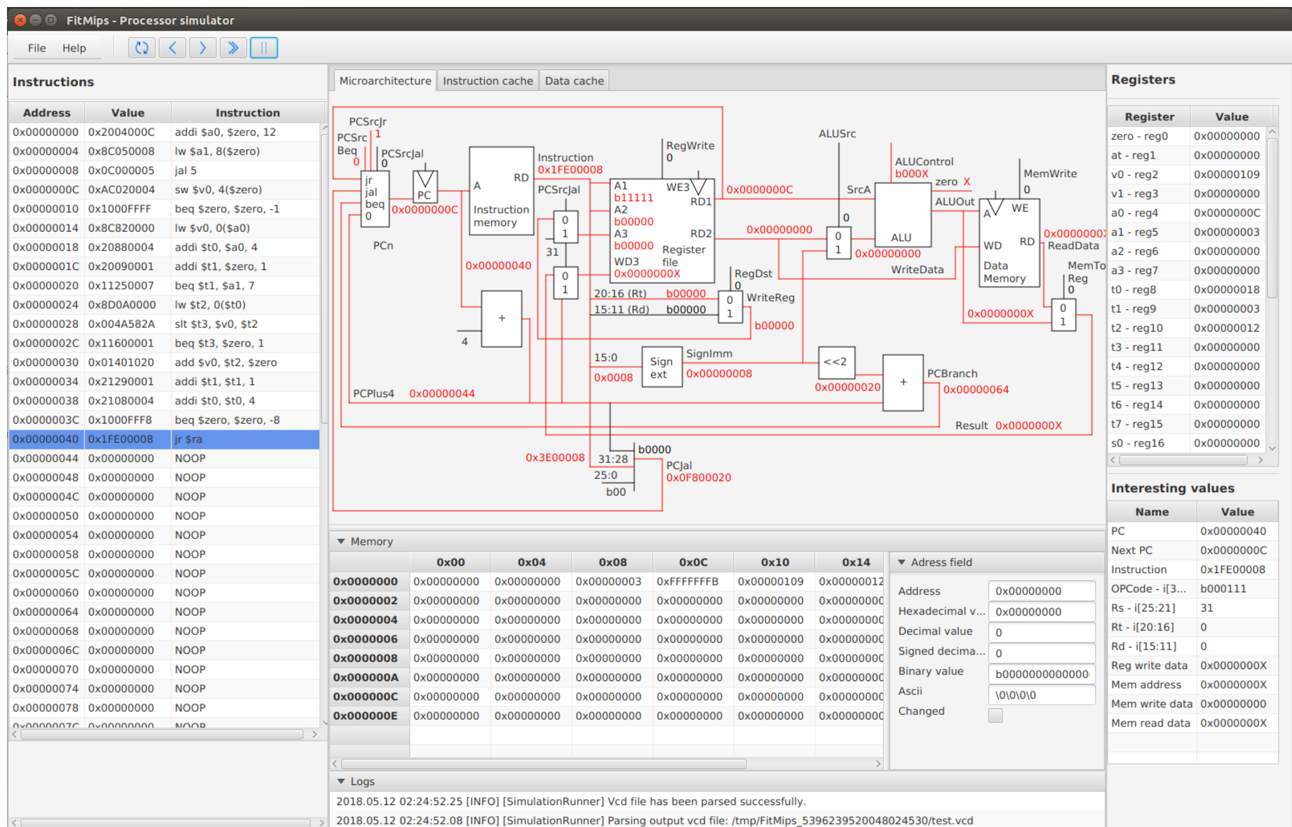
1. Vytvoří se dočasná složka, do které se nakopírují všechny potřebné soubory.
2. Spustí program VVP s parametry, který vygeneruje soubor s výsledky (má příponu .vcd)
3. Soubor s výsledky se naparsuje a uloží se do vnitřní paměťové reprezentace.
4. Vypočítá se stav systému při startu, výsledky se předají obrazové části aplikace.

Tyto kroky jsou logovány (viz. Logování). Při problémech lze zjistit dodatečné informace z logu v příkazové řádce nebo se lze podívat do dočasné složky, kde je uložen výstup programu.

3 Ovládání aplikace

Tato kapitola popisuje organizaci aplikace a použití jednotlivých částí.

Následující obrázek (Ilustrace 4: Přehled aplikace) znázorňuje hlavní přehled celé aplikace:



Ilustrace 4: Přehled aplikace

3.1 Menu a ovládací tlačítka

Menu a ovládací tlačítka (viz. Ilustrace 5: Menu a ovládací tlačítka) jsou umístěné v hořejší části aplikace.

Ilustrace 5: Menu a ovládací tlačítka

3.1.1 Menu

Menu obsahuje dvě položky:

- **File** → **New session** – znovu spustí dialogové okno k výběru cílové architektury spolu se vší konfigurací (viz. Výběr mikroarchitektury).
- **Help** → **About** – Zobrazí informace o tomto programu.

3.1.2 Ovládací tlačítka

Tlačítka jsou zde popsána v pořadí, v jakém jsou zobrazena v panelu.

1. Vrátí se na začátek simulace.
2. Vrátí simulaci o krok vzad.
3. Přejde v simulaci o krok vpřed.
4. Simulace poběží sama.
5. Zastaví běžící simulaci.

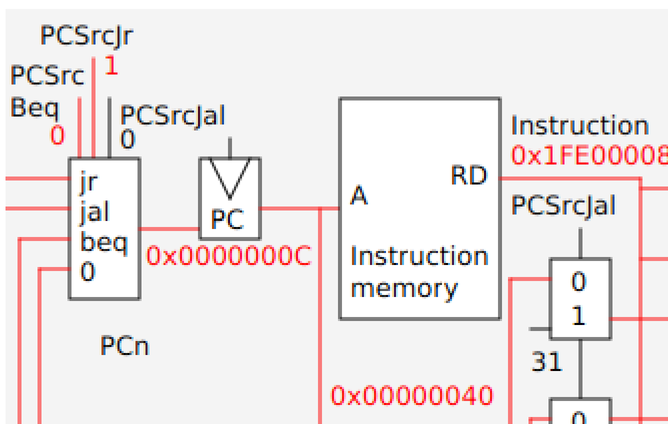
3.2 Panely

V hlavní části aplikace se nachází přepínací panely. Tyto jsou dvojího typu. V prvním typu je vykresleno schéma aplikace, druhý obsahuje samotnou cache.

3.2.1 Mikroarchitektura

Tento panel je vykreslen jako první. Obsahuje schéma procesoru, tedy různé moduly a jejich propojení. Zajímavou funkcionalitou je i vyznačení spojení, ve kterých se změnila hodnota oproti předchozímu tiku hodin v průběhu času.

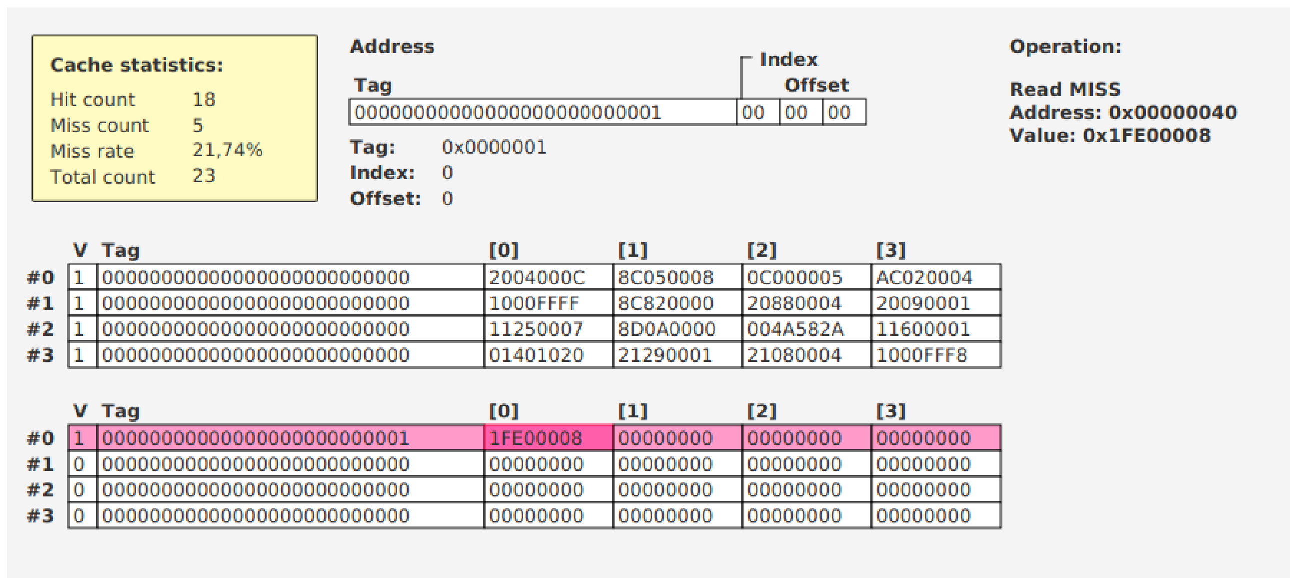
Dále lze vidět v schématu i aktuální hodnoty těchto spojení. Hodnoty můžou být zobrazeny v různých základech. Namátkou hodnoty začínající „0x“ jsou vypsány v šestnáctkové soustavě, hodnoty začínající na „b“ jsou vypsány ve dvojkové soustavě a hodnoty bez prefixu jsou obvykle v desítkové soustavě. Písmeno „X“ značí neznámou hodnotu.



Ilustrace 6: Ukázka části schématu

3.2.2 Cache

Dva panely jsou vyhrazeny cache. Jeden je pro instrukční paměť, druhý pro datovou paměť. Celkové rozložení lze vidět na obrázku (Ilustrace 7: Instrukční cache).



Ilustrace 7: Instrukční cache

Toto rozložení lze rozdělit celkem na 4 oblasti:

- **Statistika** – nachází se ve žlutém rámečku. Počítá hit count, miss count, miss rate a celkový počet přístupů do cache.
- **Adresa** – nachází se v hořejší části uprostřed. Adresa je graficky rozdělena na čtyři části: tag, index, offset a nepojmenovaná oblast. Index označuje číslo řádku v tabulce, offset označuje číslo bloku v řádku. Poslední oblast označuje část adresy, která je při zápisu a čtení vždy nulová.
- **Operace** – nachází se v hořejší části vpravo. Vždy je uveden název operace, v případě čtení nebo zápisu jsou přítomny i informace o právě použité adrese a přečtené/zapsané hodnotě. Možné názvy operací jsou: Read HIT, Read MISS, Write HIT, Write MISS, NOOP. NOOP značí, že neprobíhá zápis ani čtení do paměti.
- **Tabulky s daty** – zabírají většinu panelu. Počet tabulek se určí podle vzorečku $n=2^a$, kde proměnná n je počet tabulek a proměnná a je úroveň asociace. Počet řádků a buněk v bloku je opět nakonfigurován při startu aplikace. Každý řádek obsahuje validity bit, tag a blok dat. Řádek se označí růžovou barvou, pokud dojde při čtení či zápisu k výpadku cache. Pokud k výpadku nedojde, řádek bude označen bledě modrou barvou.

3.3 Instrukční paměť

Obsah instrukční paměti najdeme v levém panelu aplikace (viz. Ilustrace 8: Obsah instrukční paměti). Obsah instrukční paměti je vizualizován v tabulce, kde na každém řádku je zobrazena právě jedna instrukce. Právě prováděná instrukce je zvýrazněna modrou barvou.

Každý řádek je rozdělen do tří sloupců:

- **Address** – určuje adresu instrukce, zobrazena v šestnáctkové soustavě.
- **Value** – obsah paměti na dané adrese, zobrazena v šestnáctkové soustavě.
- **Instruction** – obsah paměti interpretován jako instrukce procesoru.

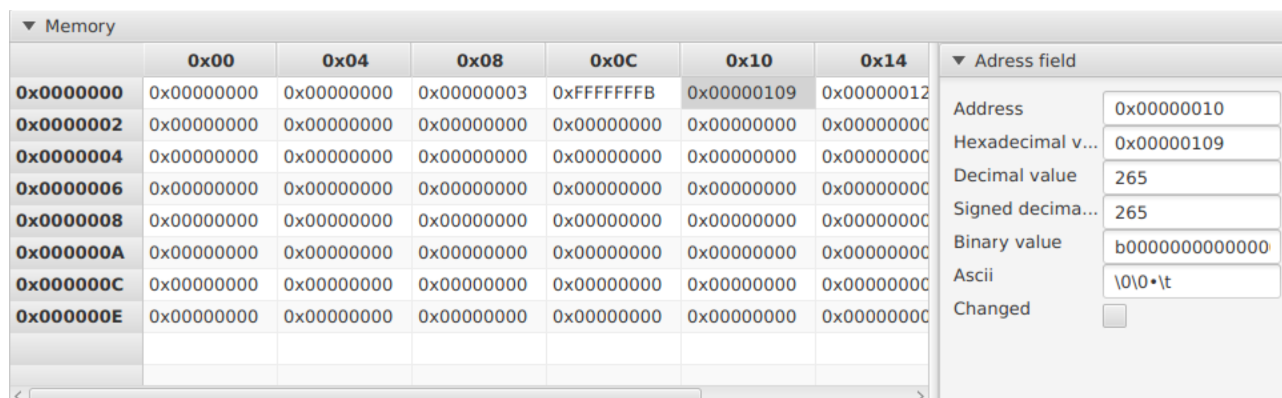
Instructions

Address	Value	Instruction
0x00000000	0x2004000C	addi $a0, $zero, 12
0x00000004	0x8C050008	lw $a1, 8($zero)
0x00000008	0x0C000005	jal 5
0x0000000C	0xAC020004	sw $v0, 4($zero)
0x00000010	0x1000FFFF	beq $zero, $zero, -1
0x00000014	0x8C820000	lw $v0, 0($a0)
0x00000018	0x20880004	addi $t0, $a0, 4
0x0000001C	0x20090001	addi $t1, $zero, 1
0x00000020	0x11250007	beq $t1, $a1, 7
0x00000024	0x8D0A0000	lw $t2, 0($t0)
0x00000028	0x004A582A	slt $t3, $v0, $t2
0x0000002C	0x11600001	beq $t3, $zero, 1
0x00000030	0x01401020	add $v0, $t2, $zero
0x00000034	0x21290001	addi $t1, $t1, 1
0x00000038	0x21080004	addi $t0, $t0, 4
0x0000003C	0x1000FFF8	beq $zero, $zero, -8
0x00000040	0x1FE00008	jr $ra
0x00000044	0x00000000	NOOP
0x00000048	0x00000000	NOOP

Ilustrace 8: Obsah instrukční paměti

3.4 Datová paměť

Obsah datové paměti lze nalézt v tabulce, která se nachází uprostřed v dolní obrazové části programu. Při kliknutí na slovo „Memory“ (v obrázku Ilustrace 9: Ukázka datové paměti se nachází v horním řádku) se celý obsah skryje. Datová paměť se skládá z levé a pravé části.



Ilustrace 9: Ukázka datové paměti

V levé části je k dispozici tabulka s obsahem paměti. Tabulka je organizována po řádcích, tedy dvě sousedící slova se nachází vedle sebe. Pro rychlejší orientaci se v prvním sloupečku nachází prefix adresy, v prvním řádku je specifikována vzdálenost od tohoto prefixu. Tato tabulka je interaktivní – při kliknutí na hodnotu v tabulce se zobrazí její detail v pravé části.

V pravé části se nachází detailní vyobrazení právě vybrané hodnoty. Detail obsahuje seznam těchto hodnot:

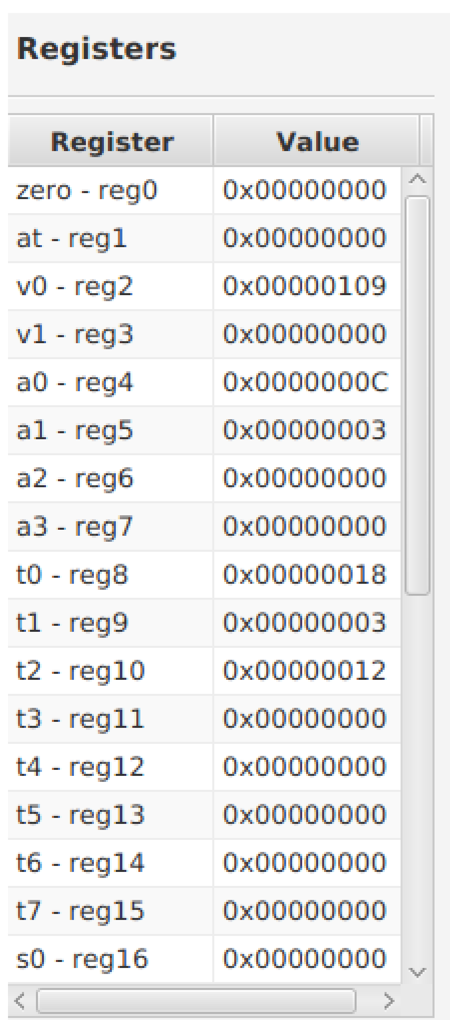
- **Adress** – přesná adresa vybrané hodnoty, v šestnáctkové soustavě.
- **Hexadecimal value** – hodnota adresy v šestnáctkové soustavě.
- **Decimal value** – hodnota této adresy interpretovaná v desítkové soustavě.
- **Signed decimal value** – hodnota této adresy interpretována v desítkové soustavě i se znaménkem (negativní číslo je kódováno jako dvojkový doplněk).
- **Binary value** – hodnota této adresy interpretovaná ve dvojkové soustavě.
- **Ascii** – Hodnota této adresy jako text v ASCII (co byte, to jeden znak). Pokud je hodnota bytu v ASCII neznámá, vypíše se místo znaku puntík.
- **Changed** – značí, jestli se hodnota změnila v aktuálním tiku hodin.

Tip: Pokud přejedete myší nad hodnotou políčka v pravé části datové paměti, zobrazí se popisek s celým obsahem. To se může hodit zejména u binárních hodnot, kdy celá část obvykle není viditelná.

3.5 Registry

Pohled na obsah registrů procesoru najdeme v pravé horní části aplikace (náhled viz. Ilustrace 10: Náhled na obsah registrů). Náhled do obsahu registrů je opět tabulka a ta má dva sloupečky.

- **Register –** názvy jednotlivých registrů. Každý registr má dva názvy oddělené pomlčkou. První název je konvenční pro danou mikroarchitekturu, druhý označuje pořadí registru.
- **Value –** aktuální hodnoty registrů, zobrazené v šestnáctkové soustavě.



Registers

Register	Value
zero - reg0	0x00000000
at - reg1	0x00000000
v0 - reg2	0x00000109
v1 - reg3	0x00000000
a0 - reg4	0x0000000C
a1 - reg5	0x00000003
a2 - reg6	0x00000000
a3 - reg7	0x00000000
t0 - reg8	0x00000018
t1 - reg9	0x00000003
t2 - reg10	0x00000012
t3 - reg11	0x00000000
t4 - reg12	0x00000000
t5 - reg13	0x00000000
t6 - reg14	0x00000000
t7 - reg15	0x00000000
s0 - reg16	0x00000000

Ilustrace 10: Náhled na obsah registrů

3.6 Tabulka zajímavých hodnot

Pro větší pohodlí aplikace zobrazuje v tabulce některé zajímavé hodnoty (které jsou zároveň přítomny ve schématu mikroarchitektury), aby byly lehce přístupné bez hledání ve schématu. Náhled se nachází v pravé části aplikace pod registry.

Tabulka má dva sloupce:

- **Name** – jméno proměnné,
- **Value** – hodnota proměnné. Každá hodnota může být zobrazená v různých základech (viz. náhled Ilustrace 11: Ukázka zajímavých hodnot), nejčastěji v těchto třech:

1. **V šestnáctkové soustavě** (hodnota začíná na 0x)
2. **V desítkové soustavě** (bez prefixu)
3. **Ve dvojkové soustavě** (hodnota začíná na písmeno b)

Interesting values

Name	Value
PC	0x00000040
Next PC	0x0000000C
Instruction	0x1FE00008
OPCode - i[3...	b000111
Rs - i[25:21]	31
Rt - i[20:16]	0
Rd - i[15:11]	0
Reg write data	0x0000000X
Mem address	0x0000000X
Mem write data	0x00000000
Mem read data	0x0000000X

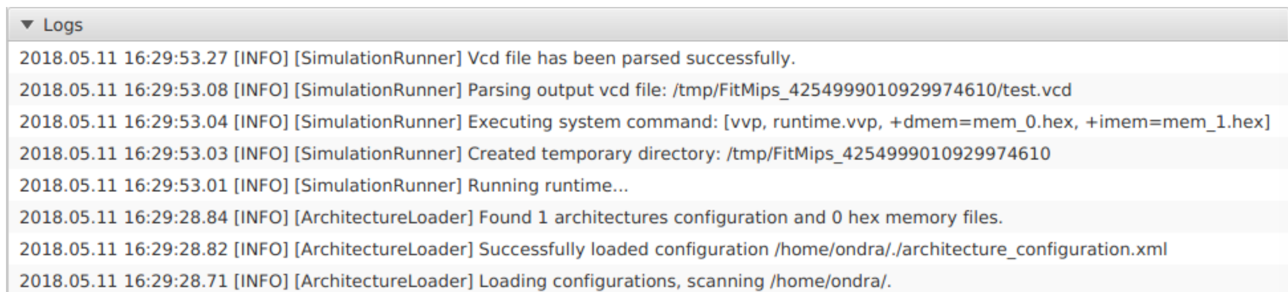
Ilustrace 11: Ukázka zajímavých hodnot

3.7 Logování

Aplikace loguje své výstupy na dvě místa – na standardní výstup a zároveň je ukazuje přímo na obrazovce v obrazové části programu. Logování se v aplikaci nachází v prostřední části dole, pod datovou pamětí. Aby bylo logování lépe vidět, je nutné skrýt vizualizaci obsahu datové paměti.

Každý řádek logu obsahuje informace o tom, kdy byl log pořízen, jak moc je důležitý (využívají se úrovně DEBUG, INFO, WARN a ERROR) a obsah sdělení. Narozdíl od standardního výstupu nejsou však hlášky typu DEBUG zobrazeny v aplikaci.

Na tomto obrázku (viz. Ilustrace 12: Logování) je zobrazen ukázkový log. Čtyři nejstarší logy ukazují, jakým způsobem proběhlo načtení dodatečné konfigurace. Zbytek ukazuje spuštění externího programu k vygenerování výsledků a jeho následného zpracování.



Ilustrace 12: Logování

4 Vývoj vlastních programů ke spuštění

Pro vývoj vlastního programu pro cílovou mikroarchitekturu je nutno zakódovat instrukce do textového souboru s příponou HEX v šestnáctkové soustavě. Každý řádek souboru obsahuje právě jednu zakódovanou instrukci a instrukcí musí být minimálně tolik, jaká je velikost instrukční paměti (pro mikroarchitekturu v základu aplikace platí, že instrukcí musí být 64).

Níže uvedená tabulka (Tabulka 1: Tabulka operací pro zjednodušenou architekturu MIPS32) poskytuje přehledný seznam všech podporovaných instrukcí ve zjednodušené MIPS32 mikroarchitektuře.

Instrukce	Syntaxe	Operace	Kódování	Poznámka
add	add d, s, t	d = s + t;	`0000 00ss ssst tttt` `dddd d000 0010 0000`	
sub	sub d, s, t	d = s - t;	`0000 00ss ssst tttt` `dddd d000 0010 0010`	
and	and d, s, t	d = s & t;	`0000 00ss ssst tttt` `dddd d000 0010 0100`	
or	or d, s, t	d = s | t;	`0000 00ss ssst tttt` `dddd d000 0010 0101`	
slt	slt d, s, t	d = (s<t) ? 1 : 0;	`0000 00ss ssst tttt` `dddd d000 0010 1010`	
addi	addi d, s, imm	d = s + imm;	`0010 00ss ssst tttt` `iiii iiii iiii iiii`	
lw	lw t, offset(s)	t = MEM[s + offset];	`1000 11ss ssst tttt` `iiii iiii iiii iiii`	
sw	sw t, offset(s)	MEM[s + offset] = t;	`1010 11ss ssst tttt` `iiii iiii iiii iiii`	
beq	beq s, t, offset	if s==t then PC=PC+4+(offset « 2); else PC=PC+4;	`0001 00ss ssst tttt` `iiii iiii iiii iiii`	
jal	jal target	$31=PC+4; PC=(PC & 0xf0000000) | (target « 2);	`0000 11ii iiii iiii` `iiii iiii iiii iiii`	neodpovídá původnímu MIPS32
jr	jr s	PC = s;	`0001 11ss sss0 0000` `0000 0000 0000 1000`	neodpovídá původnímu MIPS32, opCode má jinou hodnotu
addu.qb	addu.qb d, s, t	$d_{31:24} = s_{31:24} + t_{31:24}$; etc.	`0111 11ss ssst tttt` `dddd d000 0001 0000`	sčítání dvou vektorů bytových hodnot bez znaménka
addu_s.qb	addu_s.qb d, s, t	$d_{31:24} = sat(s_{31:24} + t_{31:24})$; etc.	`0111 11ss ssst tttt` `dddd d001 0001 0000`	se saturací
sllv	sllv d, t, s	d = t « s;	`0000 00ss ssst tttt` `dddd dxxx xx00 0100`	
srlv	srlv d, t, s	d = (unsigned)t » s;	`0000 00ss ssst tttt` `dddd d000 0000 0110`	nula se vsune do výsledku
srav	srav d, t, s	d = (signed)t » s;	`0000 00ss ssst tttt` `dddd d000 0000 0111`	znaménkový bit se vsune do výsledku

Tabulka 1: Tabulka operací pro zjednodušenou architekturu MIPS32

Tip: K ukončení programu (vytvoření nekonečné smyčky) se hodí využít instrukci *beq $zero, $zero, -1* (v hexadecimální podobě 1000ffff).

Následuje ukázka demo programu, který je součástí základní nabídky při výběru obsahu instrukční paměti. Jak je vidět na ukázce (Tabulka 2: Program k vyhledání maxima v poli), je možné k jednotlivým instrukcím přidat i komentář. Skutečný soubor také obsahuje prázdné instrukce, zakódované 00000000, aby byl naplněn limit 64 instrukcí.

```
2004000c //addi $a0, $zero, 12            int * arr = 0x0000000C
8c050008 //lw $a1, 8($zero)               int n = 8
0c000005 //jal max                        go to max function
ac020004 //sw $v0, 4($zero)               store result into memory
1000ffff //beq $zero, $zero, end          end cycle, stop here
8c820000 //lw $v0, 0($a0)                 function max begins here, res = *arr;
20880004 //addi $t0, $a0, 4               int * ptr = arr + 1, address for second element
20090001 //addi $t1, $zero, 1             int i = 1
11250007 //beq $t1, $a1, end_cycle        begins for cycle, if i == n, then end cycle
8d0a0000 //lw $t2,0($t0)                  int number = *ptr
004a582a //slt $t2,0($t0)                 tell if actual number is smaller than result
11600001 //beq $t2, $zero, add_constants  if previous result false, then skip next step
01401020 //add $v0, $t2, $zero            result = number
21290001 //addi $t1, $t1, 1               i++
21080004 //addi $t0, $t0, 4               ptr++ (next address, it means +4 bytes)
1000fff8 //beq $zero, $zero, for_cycle    go to the beginning of the cycle
1fe00008 //jr $ra                         end cycle here, jump back to main
00000000
00000000
```

Tabulka 2: Program k vyhledání maxima v poli

Tip: Pro rychlejší a pohodlnější vývoj programů je možné použít externí editor assembleru pro mikroarchitekturu MIPS32 a nechat si zobrazit kódování instrukcí, které se přepíšou (vhodný je třeba program MARS). Jen je nutno dát pozor na instrukce **jal** a **jr**, jelikož jejich kódování se může lišit.

5 Vývoj vlastní mikroarchitektury

FitMips je možné doplnit o vlastní mikroarchitekturu. K tomu je potřeba vyvinout popis hardwaru v jazyce Verilog a přidat XML soubor, který aplikace využije k načtení.

5.1 Jak na popis hardwaru ve Verilogu

Cílem této kapitoly není naučit čtenáře, jak psát popis hardwaru v jazyce Verilog, ale jak poupravit existující popis tak, aby byl integrovatelný s FitMips simulátorem.

Na následujícím obrázku (Ilustrace 14: Blok „initial“) je vidět hlavní modul našeho popisu. Důležité jsou tyto deklarace proměnných (ostatní nejsou pro integraci důležité):

- integer i
- reg [100:0] dmemFile
- reg [100:0] imemFile

Na obrázku je pro inspiraci vidět i část kódu ke generování hodin.

```
module testbench();
    integer i;
    reg         clk;
    reg         reset;
    wire [31:0] data_to_mem, address_to_mem;
    wire        memwrite;

    reg [100:0] dmemFile;
    reg [100:0] imemFile;

    top simulated_system (clk, reset, data_to_mem, address_to_mem, write_enable);

    initial    begin

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

Ilustrace 13: Přehled hlavního modulu popisu v jazyce Verilog

Nejdůležitější částí kódu je blok „initial“, který je na předchozím obrázku schován, na následujícím obrázku (Ilustrace 14: Blok „initial“) je vidět jeho obsah. Zde je seznam jednotlivých příkazů:

- **dumpfile** – specifikuje soubor s výstupem
- **dumpvars** – příkaz k uložení všech proměnných do výstupu
- **podmíněné bloky –** slouží načtení pamětí, které chceme nastavit zvenčí
    - testují přítomnost argumentu k načtení obsahu paměti

- ○ načtou argumenty do vlastní proměnné
- ○ vypíší na standardní výstup, že načetly proměnnou
- ○ načtou soubor typu HEX specifikovaný v argumentu jako obsah paměti

- **for cykly** – Icarus Verilog sám od sebe neukládá do výstupu paměťová pole, musí se to udělat pomocí těchto cyklů
- **reset** – tyto příkazy jsou implementačně závislé, není nutné je definovat ve vaší mikroarchitektuře
- **doba trvání skriptu** – sekvence příkazů „#<číslo>; $finish;“ znamená čekej <číslo> kroků, než skončíš. Určuje dobu trvání simulace. Číslo by nemělo být příliš malé (simulovaný program brzy skončí) ani příliš velké (simulace zabere větší množství času a paměti).

```
initial   begin

    $dumpfile("test.vcd");
    $dumpvars;

    if ($test$plusargs("dmem")) begin
        i = $value$plusargs("dmem=%s", dmemFile);
        $display("dmem=%s", dmemFile);
        $readmemh (dmemFile, simulated_system.dmem.RAM, 0, 63);
    end

    if ($test$plusargs("imem")) begin
        i = $value$plusargs("imem=%s", imemFile);
        $display("imem=%s", imemFile);
        $readmemh (imemFile, simulated_system.imem.RAM, 0, 63);
    end

    for (i = 0; i < 64; i = i+1) $dumpvars(0, simulated_system.imem.RAM[i]);
    for (i = 0; i < 64; i = i+1) $dumpvars(0, simulated_system.dmem.RAM[i]);
    for (i = 0; i < 32; i = i+1) $dumpvars(0, simulated_system.CPU.RF.i[i]);
    reset<=1; # 2; reset<=0;
    #10000; $finish;
end
```

Ilustrace 14: Blok „initial“

5.1.1 Vytvoření spustitelného skriptu

Pokud jsou vaše zdrojové kódy kompletní, je třeba vytvořit spustitelný skript. Skript se vytvoří pomocí následujícího příkazu, kde „my_architecture.vvp“ znamená jméno skriptu a „source_1.v“ až „source_n.v“ jsou vaše zdrojové soubory:

```
iverilog -o my_architecture.vvp source_1.v source_2.v source_3.v
```

5.2 Konfigurace architektury v XML

Druhým krokem je nakonfigurování XML souboru, který reprezentuje model konfigurace ve FitMipsu. Konfigurace je poměrně obsáhlá, proto jsou její jednotlivé části popsány v samostatných kapitolách. Vzorová konfigurace se nachází ve složce **FitMips/demo** pod názvem **architecture_configuration.xml** .

Důležitá informace: pokud není řečeno jinak, jsou všechny uvedené konfigurační elementy v této a následujících kapitolách povinné!

Na následujícím obrázku (Ilustrace 15: Celkový přehled XML konfigurace mikroarchitektury) je vidět struktura souboru. Nejvyšší element je má název „ArchitectureConfiguration“ a má následující elementy:

- **name** – název mikroarchitektury, jak se bude zobrazovat v seznamu
- **description –** popis architektury, který se bude zobrazovat v seznamu
- **engineConfiguration** – popisuje integraci s VVP
- **guiConfiguration** – nastavuje komponenty k vykreslení

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<ArchitectureConfiguration>
    <name>Mips32 simplified</name>
    <description>Simplified design of Mips 32-bit architecture.

These instructions are supported:
Arithmetic &amp; logical:
add, sub, and, or, slt, addi, addu.qb, addu_s.qb

Memory:
lw, sw

Branching &amp; jumping:
beq, jal, jr

Important note:
Instruction jr does not match exactly the original Mips specification. The opcode differs.
Encoding: 0001 11ss sss0 0000 0000 0000 0000 1000</description>
    <engineConfiguration>
    <guiConfiguration>
</ArchitectureConfiguration>
```

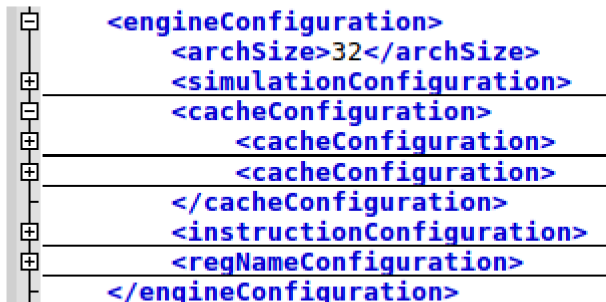
Ilustrace 15: Celkový přehled XML konfigurace mikroarchitektury

5.2.1 Konfigurace enginu

Engine má tyto položky:

- **archSize** – definuje kolikabitový je procesor (očekávaná číselná hodnota z těchto hodnot: 8, 16, 32, 64)
- **simulationConfiguration** – zanořený element, definuje hodnoty potřebné ke spuštění VVP a parsování

- **cacheConfiguration** – pole dalších elementů s názvem „cacheConfiguration“, které definují hodnoty důležité k výpočtu různých cache. Pole může být prázdné.
- **instructionConfiguration** – element, který definuje instrukce na dané platformě
- **regNameConfiguration** – zde se specifikují názvy pro jednotlivé registry



Ilustrace 16: Hierarchie konfigurace enginu

Element simulationConfiguration

Tento element má tyto nastavitelné položky:

- **tickLength** – souvisí s tím, jak funguje váš procesor ve Verilogu. Pokud se stav procesoru mění jenom při náběžné hraně hodin, nastavte dvojku, pokud se mění při každé změně hodin, nastavte jedničku
- **vvpFile** – odkaz na spustitelný skript pro VVP, relativní cesta k tomuto XML souboru
- **vcdOutputFile** – stejný text, jaký je použit v příkazu $dumpfile v kapitole Jak na popis hardwaru ve Verilogu
- **memoryConfigs** – pole dalších elementů, pojmenované opět „memoryConfigs“. Každý element v tomto poli nastavuje jednu paměť, která se nastavuje v „if blocích“ ve popisu procesoru ve Verilogu. Každá položka má tyto parametry:
  - **plusArg –** název argumentu pro tuto paměť použitý ve Verilogu
  - **name** – jméno paměti
  - **description** – popisek paměti
  - **size** – velikost paměti ve slovech

```xml
<simulationConfiguration>
    <tickLength>2</tickLength>
    <vvpFile>resource:defaults/architecture.vvp</vvpFile>
    <vcdOutputFile>test.vcd</vcdOutputFile>
    <memoryConfigs>
        <memoryConfigs>
            <plusArg>dmem</plusArg>
            <name>Data memory</name>
            <description>Data memory with size of 64 * 32 bits. Selected file must be large enough to fill the memory!</description>
            <size>64</size>
        </memoryConfigs>
        <memoryConfigs>
            <plusArg>imem</plusArg>
            <name>Instruction memory</name>
            <description>Instruction memory with size of 64 * 32 bits. Selected file must be large enough to fill the memory!</description>
            <size>64</size>
        </memoryConfigs>
    </memoryConfigs>
</simulationConfiguration>
```

Ilustrace 17: Element simulationConfiguration

Element cacheConfiguration

Každý prvek cacheConfiguration konfiguruje jednu cache paměť. Má tyto položky:

- **id** – jakýkoli text, musí být jedinečný v rámci konfigurací cache
- **name** – název cache
- **writeToMem** a **readFromMem** jsou proměnné s délkou 1 bitu. Mohou být buď typu „static“ nebo „path“. Proměnná „writeToMem“ označuje, zda je právě zapisováno do paměti cache, proměnná „readFromMem“ označuje, zda je právě čteno z paměti cache.
- **adressIn** (adresa přístupu), **writeIn** (zapisovaná hodnota) a **valueOut** (čtená hodnota) jsou proměnné se stejnou délkou jako má architektura procesoru. Opět mohou být typu „static“ nebo „path“
- **memoryPath** – cesta k celé paměti, jehož cache se snažíme simulovat

Pro proměnné s konstantní hodnotou platí, že jejich obalující element musí mít atribut s názvem „_type“ a hodnotou „static“. Vnitřní elementy jsou:

  - **value** – hodnota
  - **bitWidth** – počet bitů na proměnnou

Proměnné s variabilní hodnotou jsou typu „path“. Mají pouze jeden vnitřní element a ten se jmenuje „path“, uvnitř kterého se nachází elementy, jejichž hodnoty jednoznačně určují cestu k proměnné uvnitř Verilogu. Tyto cesty jsou popsány v následující kapitole.

Na obrázku Ilustrace 18: Element cacheConfiguration pro instrukční cache je zobrazena konfigurace pro instrukční cache. Pro instrukční cache chceme, aby proměnná „writeToMem“ byla konstantně rovna nule (nikdy nezapisujeme) a proměnná „readFromMem“ konstantně rovna jedné (vždy čteme). Pro datovou cache tato omezení neplatí, proto všechny její proměnné definujeme jako typ „path“.

```xml
<cacheConfiguration>
    <id>i_cache</id>
    <name>Instruction cache</name>
    <writeToMem type="static">
        <value>0</value>
        <bitWidth>1</bitWidth>
    </writeToMem>
    <readFromMem  type="static">
    <addressIn  type="path">
    <writeIn  type="static">
    <valueOut  type="path">
    <memoryPath>
        <wireType>REG</wireType>
        <modules>
            <modules>testbench</modules>
            <modules>simulated_system</modules>
            <modules>imem</modules>
        </modules>
        <name>\RAM</name>
        <position/>
    </memoryPath>
</cacheConfiguration>
```

Ilustrace 18: Element cacheConfiguration pro instrukční cache

Cesta proměnné

V některých situacích se hodí se odkázat na proměnnou, která se nachází v popisu procesoru ve Verilogu a musíme nastavit její cestu. Tato cesta obsahuje následující elementy:

- **wireType** – jedna z těchto hodnot: WIRE, REG, INTEGER (záleží na definici proměnné ve Verilogu, typ je stejný)
- **modules** – pole hodnot, každá hodnota je opět element s názvem „modules“. Tyto hodnoty určují cestu v hierarchii modulů
- **name** – jméno proměnné. Pozor, pro pole platí, že jejich jméno je prefixováno zpětným lomítkem
- **position** – pozice v poli, nepovinný parametr. Interně funguje tak, že změní jméno tak, že přilepí pozici ohraničenou v hranatých závorkách. Například při jméně „\RAM“ a pozici „10“ se změní jméno na „\RAM[10]“

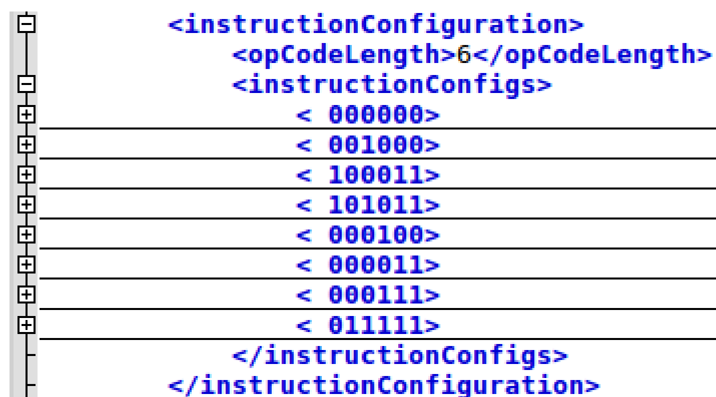
Příklad se nachází i třeba na obrázku Ilustrace 18: Element cacheConfiguration pro instrukční cache.

Element instructionConfiguration

Tento element má tyto položky:

- **opCodeLength** – počet bitů od počátku adresy, které jsou vyhrazeny pro operační kód
- **instructionConfig** – mapa, kde klíčem jsou operační kódy jednotlivých instrukcí ve tvaru „_<opCode>“, kde <opCode> je hodnota operačního kódu ve dvojkové soustavě s přesně

tolika číslicemi, kolik jich je uvedeno v hodnotě „opCodeLength“. Hodnotou je seznam všech instrukcí pro tento operační kód.



Ilustrace 19: Element instructionConfiguration

Seznam instrukcí pro daný operační kód je uveden v elementu s názvem „list“. I každá položka má element s názvem „list“. Každá položka má tyto další elementy:

- **name** – jméno instrukce
- **mask –** délka masky je daná vzorcem n = počet bitů mikroarchitektury – délka operačního kódu. Maska určuje kódování instrukcí, začíná za operačním kódem. Vyplňte 0 nebo 1 na místa, kde se v dané instrukci očekávají, znak ‚x‘ pro don‘t care hodnoty a libovolné znaky abecedy pro ucelenou hodnotu. Do masky nepatří mezery. Například maska „ssssstttttdddddxxxx1010101“ znamená, že prvních pět bitů je rezervováno pro proměnnou „s“, dalších pět bitů pro proměnnou „t“, dalších pět bitů pro proměnnou „d“, další 4 bity, kde je hodnota nezajímavá a instrukce vždy končí hodnotou 1010101.
- **printExpression** – vytiskne hodnotu podle speciálního jazyka výrazu.

Syntaxe jazyka k vytištění výrazu je jednoduchá – jednotlivé lexikální elementy jsou od sebe odděleny plusem. Lexikální elementy jsou tyto:

- **jméno** – vytiskne jméno instrukce, syntaxe: *name*
- **název registru** – vytiskne název registru, syntaxe: *reg(<X>)* , kde <X> značí nějakou proměnnou uvedenou v masce
- **číslo** – vytiskne číslo v daném základu, syntaxe: *<X>_<base>* , kde <X> opět značí proměnnou uvedenou v masce instrukce a <base> je jedna z těchto hodnot:
  - **B2** – dvojková soustava
  - **B10** – desítková soustava bez znaménka
  - **BS10 –** desítková soustava se znaménkem
  - **B16** – šestnáctková soustava

    - **BA** – ASCII text

- **text** – pevně daný text , syntaxe: ‘<*text*>‘

Ukázka definicí instrukcí se naléžá na obrázku Ilustrace 20: Seznam operačních instrukcí. Jsou zobrazeny jenom dvě instrukce, další jsou skryty. Jedna z těchto instrukcí je „NOOP“, což znamená žádná operace.

```
<_000000>
    <list>
        <list>
            <name>NOOP</name>
            <mask>00000000000000000000000000</mask>
            <printExpression>name</printExpression>
        </list>
        <list>
        <list>
        <list>
        <list>
        <list>
        <list>
        <list>
        <list>
            <name>srav</name>
            <mask>ssssstttttddddd00000000111</mask>
            <printExpression>name + ' ' + reg(d) + ', ' + reg(t) + ', ' + reg(s)</printExpression>
        </list>
    </list>
</_000000>
```

Ilustrace 20: Seznam operačních instrukcí

Element regNameConfiguration

Tento element s názvem „regNameConfiguration“ má tyto konfigurovatelné položky:

- **registers –** počet registrů celkově
- **nameMap** – mapa jmen (může být prázdná a nemusí pro každý registr obsahovat jménoú
    - **klíč** – podtržítko + číslo registru
    - **hodnota –** jméno registru

Ukázka tohoto elementu je na obrázku Ilustrace 21: Konfigurace jmen registrů.

```xml
<regNameConfiguration>
    <registers>32</registers>
    <nameMap>
        <_0>zero</_0>
        <_1>at</_1>
        <_2>v0</_2>
        <_3>v1</_3>
        <_4>a0</_4>
        <_5>a1</_5>
        <_6>a2</_6>
        <_7>a3</_7>
        <_8>t0</_8>
        <_9>t1</_9>
        <_10>t2</_10>
        <_11>t3</_11>
        <_12>t4</_12>
        <_13>t5</_13>
        <_14>t6</_14>
        <_15>t7</_15>
        <_16>s0</_16>
        <_17>s1</_17>
        <_18>s2</_18>
        <_19>s3</_19>
        <_20>s4</_20>
        <_21>s5</_21>
        <_22>s6</_22>
        <_23>s7</_23>
        <_24>t8</_24>
        <_25>t9</_25>
        <_26>k0</_26>
        <_27>k1</_27>
        <_28>gp</_28>
        <_29>sp</_29>
        <_30>fp</_30>
        <_31>ra</_31>
    </nameMap>
</regNameConfiguration>
```

Ilustrace 21: Konfigurace jmen registrů

5.2.2 Konfigurace GUI

Nastavení GUI části má tyto elementy, které budou popsány v dalších kapitolách:

- **modelItems** – stará se o vykreslení schématu mikroarchitektury.
- **pcCounter –** cesta k PC (program counter)
- **registerConfiguration** – cesta k registrům
- **dataMemory** – cesta k datové paměti
- **instructionMemory** – cesta k instrukční paměti
- **otherValues** – proměnné, které se zobrazí v tabulce s názvem „Interesting values“ (viz. Tabulka zajímavých hodnot)

```
<guiConfiguration>
    <modelItems>
    <pcCounter>
    <registerConfiguration>
    <dataMemory>
    <instructionMemory>
    <otherValues>
</guiConfiguration>
```

Ilustrace 22: GUI konfigurace

Element modelItems

Tento element je pole, které může být prázdné. Pokud by ale bude prázdné, bude též schéma mikroarchitektury prázdné. Je možné vykreslit tyto objekty: drát, linku, obdélník, textový popisek

Vykreslení drátu je konfiguračně nejnáročnější (viz. náhled Ilustrace 23: Vykreslení drátu). Obalující element se napíše ve tvaru „<modelItems _type="wire">". Vnitřek vypadá takto:

- **valueLabel** – hodnota drátu
  - **x** – x-ová souřadnice
  - **y –** y-ová souřadnice
  - **base –** základ, příklad viz. kapitola Element instructionConfiguration
- **points** – pole linek vyznačující drát. Obsahuje elementy „points" (linky), který obsahuje další elementy s názvem „points" (body linky). Každá linka musí mít aspoň dva body. Tyto body mají hodnoty s těmito názvy:
  - **x** – x-ová souřadnice
  - **y** – y-ová souřadnice
- **wirePath** – cesta k verilogové proměnné – viz. kapitola Cesta proměnné

Dalším typem objektu je linka, jejíž atribut „_type" má hodnotou „line". Vnitřek má tyto hodnoty:

- **x1** – x-ová souřadnice počátečního bodu
- **y1** – y-ová souřadnice počátečního bodu
- **x2** – x-ová souřadnice koncového bodu
- **y2 –** y-ová souřadnice koncového bodu

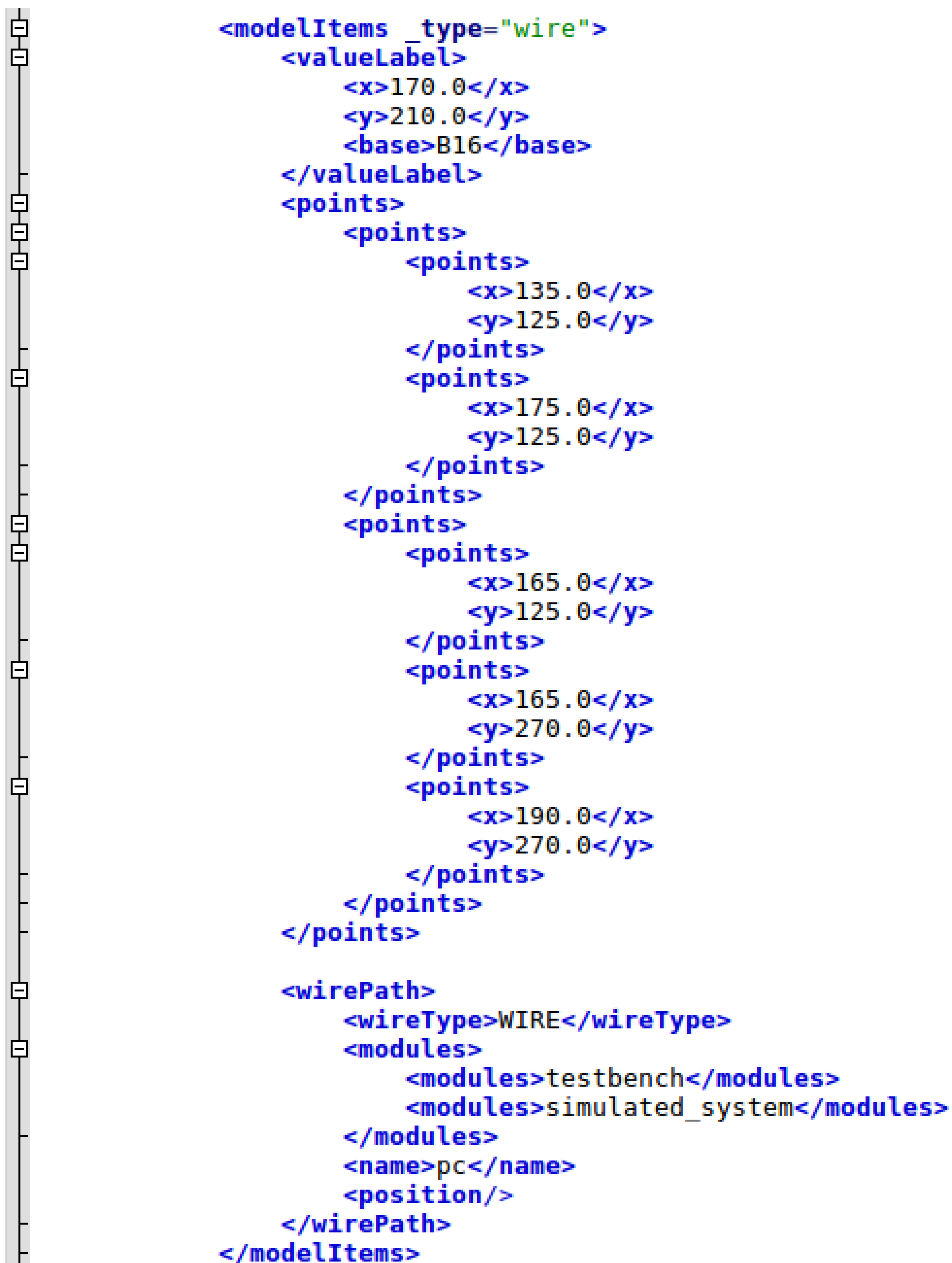
Popisek se rozliší pomocí hodnoty „label" v atributu „_type". Dá se nastavit pomocí těchto hodnot:

- **x** – x-ová souřadnice popisku
- **y** – y-ová souřadnice popisku

- **label** – text popisku

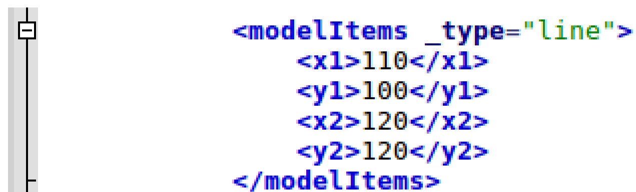
Obdélník se nastaví díky hodnotě „box“ v atributu „_type“ a má tyto elementy:

- **x –** x-ová souřadnice horního levého bodu
- **y –** y-ová souřadnice horního levého bodu
- **width** – šířka obdélníku
- **height** – výška obdéníku



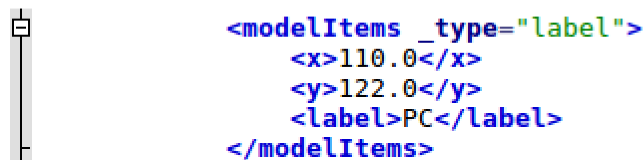
```
<modelItems _type="wire">
    <valueLabel>
        <x>170.0</x>
        <y>210.0</y>
        <base>B16</base>
    </valueLabel>
    <points>
        <points>
            <points>
                <x>135.0</x>
                <y>125.0</y>
            </points>
            <points>
                <x>175.0</x>
                <y>125.0</y>
            </points>
        </points>
        <points>
            <points>
                <x>165.0</x>
                <y>125.0</y>
            </points>
            <points>
                <x>165.0</x>
                <y>270.0</y>
            </points>
            <points>
                <x>190.0</x>
                <y>270.0</y>
            </points>
        </points>
    </points>

    <wirePath>
        <wireType>WIRE</wireType>
        <modules>
            <modules>testbench</modules>
            <modules>simulated_system</modules>
        </modules>
        <name>pc</name>
        <position/>
    </wirePath>
</modelItems>
```

Ilustrace 23: Vykreslení drátu



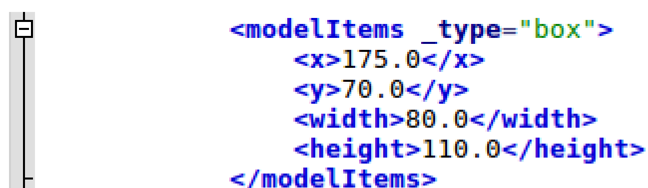
```
<modelItems _type="line">
    <x1>110</x1>
    <y1>100</y1>
    <x2>120</x2>
    <y2>120</y2>
</modelItems>
```

Ilustrace 24: Linka



```
<modelItems _type="label">
    <x>110.0</x>
    <y>122.0</y>
    <label>PC</label>
</modelItems>
```

Ilustrace 25: Popisek

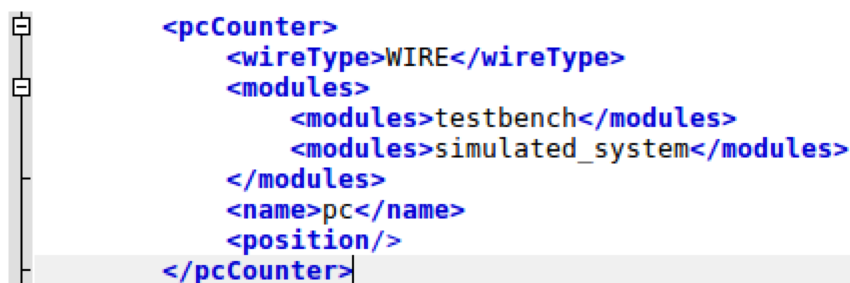


```
<modelItems _type="box">
    <x>175.0</x>
    <y>70.0</y>
    <width>80.0</width>
    <height>110.0</height>
</modelItems>
```

Ilustrace 26: Obdélník

Element pcCounter

Element „pcCounter“ je ukázkový příklad specifikace cesty, popsané v kapitole Cesta proměnné.



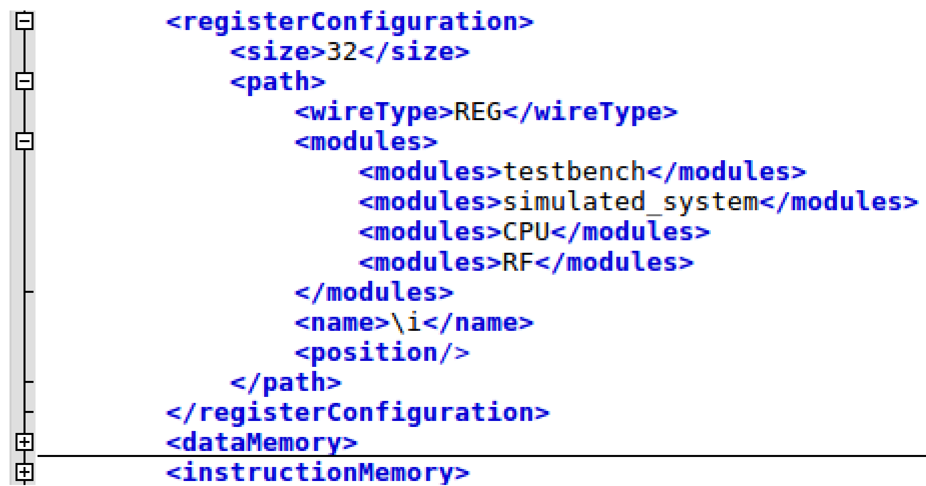
```
<pcCounter>
    <wireType>WIRE</wireType>
    <modules>
        <modules>testbench</modules>
        <modules>simulated_system</modules>
    </modules>
    <name>pc</name>
    <position/>
</pcCounter>
```

Ilustrace 27: Element pcCounter

Elementy registerConfiguration, dataMemory a instructionMemory

Tyto elementy mají stejnou konfiguraci a to:

- **size** – počet slov v těchto pamětech
- **path** – viz. kapitola Cesta proměnné



Ilustrace 28: Elementy registerConfiguration, dataMemory a instructionMemory

Element otherValues

Tento element je pole, které obsahuje další elementy s názvem „otherValues“. Toto pole může být prázdné. Každý prvek pole se dá nadefinovat pomocí těchto dalších elementů:

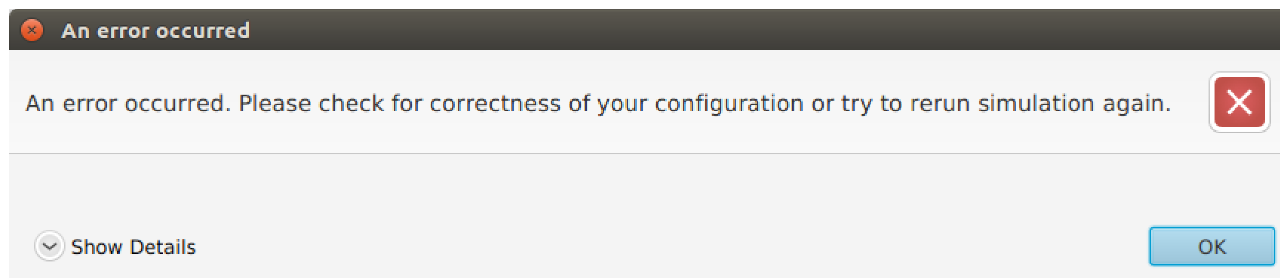
- **name** – jméno proměnné, jak se zobrazí v tabulce
- **base** – číselný základ hodnoty
    - **B2** – dvojková soustava
    - **B10** – desítková soustava bez znaménka
    - **BS10 –** desítková soustava se znaménkem
    - **B16** – šestnáctková soustava
    - **BA** – ASCII text
- **path –** cesta k proměnné (viz. kapitola Cesta proměnné)

```xml
<otherValues>
    <otherValues>
        <name>PC</name>
        <base>B16</base>
        <path>
            <wireType>WIRE</wireType>
            <modules>
                <modules>testbench</modules>
                <modules>simulated_system</modules>
            </modules>
            <name>pc</name>
            <position/>
        </path>
    </otherValues>
    <otherValues>
    <otherValues>
    <otherValues>
    <otherValues>
    <otherValues>
    <otherValues>
    <otherValues>
    <otherValues>
    <otherValues>
    <otherValues>
</otherValues>
```

Ilustrace 29: Náhled na element otherValues

6 Řešení problémů

V případě, že některá část aplikace selže, vyskočí následující dialogové okno (viz. Ilustrace 30: Dialogové okno s problémem). Obecně jsou tři známé problémy, které mohou nastat. Tyto problémy jsou popsané v následujících sekcích.



Ilustrace 30: Dialogové okno s problémem

6.1 Nejde spustit program na mikroarchitektuře

Tento problém může mít dvě různé známé příčiny.

6.1.1 Není známá cesta k VVP

Pokud se v logu vyskytuje hláška podobná této (Tabulka 3: Chybová hláška při neexistujícím programu), pak to znamená že FitMips nezná cestu k nainstalovanému Icarus Veriogu. Ujistěte se, že je Icarus Verilog nainstalován na vašem systému a že je známá cesta ke spustitelnému VVP programu. Cesta se dá specifikovat pomocí proměnné prostředí VVP_PATH. Více informací lze nalézt v sekci Spuštění přes příkazovou řádku.

2018.05.12 19:15:02.31 [ERROR] [GuiUtils] java.io.IOException: Cannot run program "/usr/lib/vvp" (in directory "/tmp/FitMips_1815019323346009861"): error=2, Adresář nebo soubor neexistuje

Tabulka 3: Chybová hláška při neexistujícím programu

6.1.2 Neodpovídající verze

Pokud spouštění spadne na následující hlášku „java.lang.IllegalStateException: Simulation process has ended with an invalid exit value: 1“, znamená to, že se nepovedlo spustit VVP správně. Nejpravděpodobnější vysvětlení je to, že nesedí verze programu Icarus Verilog. Script pro Icarus Verilog je generován ve verzi 0.9.7, není garantováno, že poběží na jiných verzích.

Zda je to tento problém zjistíme z logů. Výstup z externího procesu je vidět pouze v logu na standardním výstupu (náhled Tabulka 4: Špatná verze programu Icarus Verilog). Avšak je možné z logu zjistit pracovní složku (v řádku s popiskem „Created temporary directory XXX“), ve které se nachází soubor s názvem output.txt, kde najdeme výstup z běhu procesu.

```
2018.05.12 21:02:54.24 [INFO] [SimulationRunner] Executing system command: [vvp,
runtime.vvp, +dmem=mem_0.hex, +imem=mem_1.hex]
2018.05.12 21:02:54.72 [DEBUG] [SimulationRunner] ===Beginning of process output===
Error: VVP input file 9.7 can not be run with run time version 10.1 (stable)

===End of process output===
2018.05.12 21:02:54.73 [ERROR] [GuiUtils] An error occurred. Please check for correctness of
your configuration or try to rerun simulation again.
2018.05.12 21:02:54.73 [ERROR] [GuiUtils] java.lang.IllegalStateException: Simulation process
has ended with an invalid exit value: 1
```

Tabulka 4: Špatná verze programu Icarus Verilog

6.2 Program nenajde dodatečnou mikroarchitekturu

Při vývoji vlastní architektury je doporučeným postupem spouštění programu přes příkazovou řádku (viz. kapitola Spuštění přes příkazovou řádku). Jestli program nenajde dodatečnou architekturu, může to mít několik příčin.

6.2.1 Špatně nastavená složka

Tento problém se zjistí jednoduše. Při startu aplikace se v logu objeví informace, kterou složku skenuje k vyhledání konfigurace mokroarchitektury. Pokud je složka jiná, než očekáváme, je možné složku při příštím spuštění změnit pomocí proměnné prostředí s názvem „SCAN_PATH“. Ujistěte se také, že soubor má příponu „.xml“.

6.2.2 Nevalidní XML soubor

Pokud program narazí soubor typu XML, snaží se ho deserializovat do modelu v interní paměti. Jestli se to nepovede, je to buď z důvodu, že tento soubor vůbec nepopisuje tento model, nebo je v něm chyba. V každém případě lze nalézt v logu místo, kde poprvé nastala chyba při vytváření modelu. Z hlášky je možné vyčíst řádek a pořadí znaku, kde nastala chyba.

```
2018.05.12 23:04:39.43 [DEBUG] [ArchitectureLoader]
com.fasterxml.jackson.core.JsonParseException: Unexpected character 'i' (code 105) in prolog;
expected '<'
 at [row,col {unknown-source}]: [1,1]
```

Tabulka 5: Chyba parsování xml souboru

6.3 Vlastní nová mikroarchitektura nejde spustit

Nejprve se ujistěte, zda nedochází žádnému z problémů popsaných v kapitole Nejde spustit program na mikroarchitektuře. Pokud ne, pravděpodobně je v XML souboru sémantická chyba. Projděte si prosím znova návod v kapitole Vývoj vlastních programů ke spuštění nebo se podívejte na ukázkový soubor, který najdete podle následující cesty:
FitMips/demo/architecture_configuration.xml

Pokud se pouze spletete při párování hodnoty s hodnotou ve Verilogu a nastavíte špatnou cestu, program vás na to upozorní. Ukázka je na obrázku Ilustrace 31: Neexistující cesta ve Verilogu.

```
▼ Logs
2018.05.12 18:53:15.63 [ERROR] [GuiUtils] java.lang.NullPointerException
	at org.cvut.fit.simulator.gui.runtime.SimulatorGuiSessionFactory.lambda$createChangeableModelItemList$0(SimulatorGuiSessionFactory.java:170)
	at java.util.ArrayList.forEach(ArrayList.java:1249)
	at org.cvut.fit.simulator.gui.runtime.SimulatorGuiSessionFactory.createChangeableModelItemList(SimulatorGuiSessionFactory.java:164)
	at org.cvut.fit.simulator.gui.runtime.SimulatorGuiSessionFactory.createSession(SimulatorGuiSessionFactory.java:57)
	at org.cvut.fit.simulator.gui.controller.ArchitectureConfigurationController$1$1.call(ArchitectureConfigurationController.java:185)
	at org.cvut.fit.simulator.gui.controller.ArchitectureConfigurationController$1$1.call(ArchitectureConfigurationController.java:182)
	at javafx.concurrent.Task$TaskCallable.call(Task.java:1423)
	at java.util.concurrent.FutureTask.run(FutureTask.java:266)
	at javafx.concurrent.Service.lambda$null$492(Service.java:725)
	at java.security.AccessController.doPrivileged(Native Method)
	at javafx.concurrent.Service.lambda$executeTask$493(Service.java:724)
	at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
	at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
	at java.lang.Thread.run(Thread.java:745)
2018.05.12 18:53:15.63 [ERROR] [GuiUtils] An error occurred. Please check for correctness of your configuration or try to rerun simulation again.
2018.05.12 18:53:15.63 [WARN] [WireResolver] Attempting to resolve non-existing wire path. Modules: [testbench, simulated_system], Type: WIRE, Name: pcA
```

Ilustrace 31: Neexistující cesta ve Verilogu