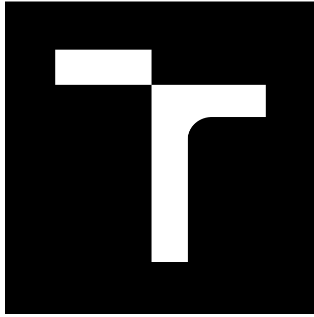
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2020 Bc. Lucie Chovančíková



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV MIKROELEKTRONIKY

DEPARTMENT OF MICROELECTRONICS

IMPLEMENTACE MIKROPROCESORU RISC-V S ROZŠÍŘENÍM PRO BITOVÉ MANIPULACE

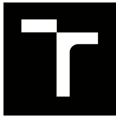
RISC-V MICROPROCESSOR IMPLEMENTATION WITH BIT MANIPULATIONS INSTRUCTION SET EXTENSION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Lucie Chovančíková**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Marián Pristach, Ph.D.**
SUPERVISOR

BRNO 2020



Diplomová práce

magisterský navazující studijní obor **Mikroelektronika**

Ústav mikroelektroniky

Studentka: Bc. Lucie Chovančíková ***ID:*** 173661
Ročník: 2 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Implementace mikroprocesoru RISC-V s rozšířením pro bitové manipulace

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s architekturou procesoru RISC-V. Navrhněte vhodnou mikroarchitekturu procesoru s instrukční sadou RISC-V RV32I a s instrukčním rozšířením pro bitové manipulace. Navržený procesor implementujte na úrovni RTL s využitím nástrojů firmy Codasip. Na vhodné sadě testů ověřte správnou funkci procesoru. Vytvořte jednoduchou aplikaci demonstrující funkci procesoru a zhodnoťte dosažené výsledky.

DOPORUČENÁ LITERATURA:

Podle pokynů vedoucího práce

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 1.6.2020

Vedoucí práce: Ing. Marián Pristach, Ph.D.
Konzultant: Bc. Marek Masařík

doc. Ing. Lukáš Fujcik, Ph.D.
předseda oborové rady

UPOZORNĚNÍ:

Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Diplomová práce se zabývá návrhem procesoru RISC-V rozšířeného o instrukce pro bitové manipulace. V této práci se věnuje pozornost popisu instrukční sady RISC-V a jazyka CodAL, který slouží k popisu instrukčních sad a procesorových architektur. Hlavním cílem práce je implementace modelu s 32-bitovým adresním prostorem, základní instrukční sadou RISC-V a rozšířením pro bitové manipulace na instrukční a RTL úrovni. Výsledné parametry navrženého procesoru jsou změřeny pomocí nástroje Genus Synthesis Solution. Do měření je také zahrnuta využitelnost bitových manipulací na základě pokrytí dekodéru.

KLÍČOVÁ SLOVA

Procesor, CodAL, RISC-V, instrukční rozšíření pro bitové manipulace, AHB sběrnice

ABSTRACT

This master thesis deals with the design of a RISC-V processor with bit manipulations instruction set extension. In this work, attention is paid to the description of the RISC-V instruction set and the CodAL language, which is used to describe the instruction sets and the processor architectures. The main goal of this work is to implement a model with a 32-bit address space, RISC-V basic instruction set and bit manipulations instruction set. The processor's design have two models, which one is instruction model and second is RTL model. The resulting parameters of the designed processor are measured using a Genus Synthesis Solution tool. The usability of bit manipulations based on decoder coverage is also included in the measurement.

KEYWORDS

Processor, CodAL, RISC-V, bit manipulation instruction set, AHB bus

CHOVANČÍKOVÁ, Lucie. *Implementace mikroprocesoru RISC-V s rozšířením pro bitové manipulace*. Brno, 2020, 86 s. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav mikroelektroniky. Vedoucí práce: Ing. Marián Pristach, Ph.D.

Vysázeno pomocí balíčku thesis verze 4.00; http://latex.feec.vutbr.cz

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Implementace mikroprocesoru RISC-V s rozšířením pro bitové manipulace“ jsem vypracovala samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autorka uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušila autorská práva třetích osob, zejména jsem nezasáhla nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědoma následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autorky

PODĚKOVÁNÍ

Ráda bych poděkovala vedoucímu diplomové práce panu Ing. Mariánu Pristachovi, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci. Dále bych ráda poděkovala firmě Codasip za umožnění zpracování diplomové práce a poskytnutí prostředků pro její zrealizování.

Obsah

Úvod **12**

1 Procesor **13**
- 1.1 CISC . . . 15
- 1.2 RISC . . . 16
- 1.3 ARM . . . 18

2 AHB sběrnice **19**
- 2.1 Průběh přenosu na AHB sběrnici . . . 20

3 RISC-V **22**
- 3.1 Instrukční sada RISC-V procesorů . . . 23
  - 3.1.1 Obecné standardní rozšíření . . . 23
  - 3.1.2 Specifická standardní rozšíření . . . 24
  - 3.1.3 Nestandardní rozšíření . . . 24
  - 3.1.4 Přehled rozšíření . . . 25
- 3.2 Formát instrukcí . . . 26
  - 3.2.1 Formáty instrukcí pro B rozšíření . . . 26

4 Jazyk CodAL **28**
- 4.1 IA a CA model . . . 28
- 4.2 Signály a registry . . . 29
- 4.3 Základní blok událostí *event* . . . 30
- 4.4 Adresní prostor a rozhraní . . . 32

5 Model na instrukční úrovni IA **34**
- 5.1 Popis instrukční sady ISA . . . 34
- 5.2 Implementace RV32I instrukcí . . . 38
  - 5.2.1 Výpočetní instrukce . . . 38
  - 5.2.2 Instrukce pro práci s pamětí . . . 41
  - 5.2.3 Skokové instrukce . . . 43
  - 5.2.4 Systémové instrukce . . . 45
- 5.3 Implementace instrukcí pro bitové manipulace . . . 47
  - 5.3.1 Základní bitové instrukce . . . 48
  - 5.3.2 Permutační instrukce . . . 50

6 Model na RTL úrovni 54
6.1 FE stupeň načtení 55
6.2 EX stupeň vykonání 55
6.3 WB zápis výsledku 56
6.4 Implementace B rozšíření 56
6.4.1 Implementace základních bitových instrukcí 56
6.4.2 Generické instrukce 58

7 Verifikace a testování 62
7.1 Testovaní pomocí sady testů 62
7.1.1 Architekturální testy pro bitové manipulace 63
7.2 Funkční verifikace 64

8 Dosažené výsledky 65
8.1 Parametry navrženého procesoru 65
8.2 Četnost instrukcí 67

9 Závěr 68

Literatura 70

Seznam symbolů, veličin a zkratek 72

A Schéma implemntováného procesoru RV32IB 74

B Implementace instrukcí BEXT A BDEP 75

C Funkce pro výpočet instrukcí SHFL 84

D Obsah přiloženého zip souboru 86

Seznam obrázků

1.1 Struktrura CPU [6] 13
2.1 Schéma zapojení sběrnice [11] 19
6.1 Zjednodušené schéma implementovaného procesoru 54
6.2 Blokové zapojení pro instrukce GREV, BEXT a BDEP[3] 59
6.3 Motýlí síť pro 8 prvků 60
6.4 Síť pro instrukci SHFL [2] 61
8.1 Procentuální zastoupení plochy největších komponet v rámci návrhu 66
8.2 Procentuální zastoupení plochy v rámci ALU 66
8.3 Graf využití instrukcí 67

Seznam tabulek

1.1 Proudové sekvenční instrukcí I v čase T procesorem CISC 15
1.2 Zřetězené provádění instrukcí I v čase T procesorem RISC 16
2.1 Odpovědi AHB sběrnice v rámci signálu HRESP [11] 21
2.2 Klasifikace přenosu AHB sběrnice [11] 21
3.1 RISC-V privilegované módy 22
3.2 Podporované kombinace privilegováných módů[10] 22
3.3 Přehled současných RISC-V rozšíření 25
3.4 Formát instrukcí 26
3.5 Instrukce s typem formátu I 26
3.6 Instrukce s typem formátu S 27
3.7 Formát instrukce s konstantou 27
5.1 Princip instrukcí BEXT a BDEP 53
7.1 Výsledky testování 63
7.2 Výsledky funkční verifikace (pokrytí kódu) 64
8.1 Parametry navrženého procesoru 65

Seznam algoritmů

4.1 Deklarace signálu . . . 29
4.2 Architekturální registr pro programový čítač . . . 29
4.3 Architekturální registr . . . 29
4.4 Alias registru . . . 29
4.5 Deklarace hlavního registrového pole . . . 30
4.6 Konstrukce události (*event*) . . . 31
4.7 Deklarace a použtí stupně (*pipeline*) . . . 31
4.8 Definice adresního prostoru . . . 32
4.9 Definice rozhraní pro program . . . 33
4.10 Definice rozhraní pro data . . . 33
5.1 Obecná definice elementu instrukce . . . 34
5.2 Funkce pro čtení z hlavního registrového pole . . . 35
5.3 Funkce pro zápis do hlavního registrového pole . . . 35
5.4 Funkce pro kontrolu přístupu do paměti . . . 36
5.5 Zkrácená funkce čtení z paměti . . . 37
5.6 Zkrácená funkce pro zápis do paměti . . . 37
5.7 Funkce halt . . . 38
5.8 Příklad popisu elementu pro výpočetní instrukci . . . 39
5.9 Popis instrukce ADDI . . . 40
5.10 Příklad instrukce pro odčítání . . . 40
5.11 Příklad instrukcí menší než . . . 40
5.12 Příklad instrukcí pro logické operace . . . 40
5.13 Příklad instrukcí pro posuvy vpravo . . . 41
5.14 Příklad instrukce AUIPC . . . 41
5.15 Příklad instrukcí LOAD a STORE . . . 42
5.16 Element instrukce LOAD . . . 42
5.17 Ukázka instrukcí skoků v testovacím makru . . . 43
5.18 Element pro podmíněný skok . . . 44
5.19 Elemet pro operaci s kontrolními stavovými registry . . . 46
5.20 Zkrácená funkce csr_read_write . . . 47
5.21 Implementace a příklad instrukcí CLZ a CTZ . . . 48
5.22 Implementace a příklad instrukce PCNT . . . 48
5.23 Implementace a příklad instrukcí posuvu s jedničkou SLO a SRO . . . 49
5.24 Implementace a příklad instrukcí pro rotace . . . 49
5.25 Příklad použítí instrukcí ANDC, ANDN, ORN a ORN . . . 50
5.26 Instrukce SHFLI a UNSHFLI . . . 50
5.27 Implementace funkce *shfl_stage* [3] . . . 51

5.28 Implementace funkce *shfl_comp* [3] 51
5.29 Příklad instrukce GREVI 52
5.30 Příklad instrukce GREVI [3] 52
5.31 Příklad instrukce GREVI [3] 53
6.1 Ukázka implementace základních bitových instrukcí CLZ a CTZ . . . 57
6.2 Ukázka implementace bitových rotací ROR a ROL 57
6.3 Ukázka výpočtu PCNT pro pěti bitový operand 57
6.4 Ukázka inline funkce pro logické posuvy s jedničkou 58
6.5 Inline funkce pro barel rotaci 8 bitové nuly 59

Úvod

Od doby prvních počítačů, které zabíraly obrovské prostory a uměly jen velmi jednoduché operace, již uplynulo mnoho let. V dnešní době si už ani neumíme představit, že před pár desítkami let byl počítač v domácnosti spíše raritou než samozřejmostí, jak je tomu dnes. Doba dospěla do bodu, kdy skoro všude využíváme tzv. chytré zařízení. Technologii můžeme najít v mobilu, hodinkách, lednici, hračkách, vysavači a dokonce i v oblečení. Bohužel pro všechny aplikace není možné použít stejný procesor jako do osobního počítače. Důvodů je hned několik, příliš velká enegetická náročnost, velké rozměry, příliš vysoká výkonnost a hlavně ekonomicky nepřijatelná cena. U procesorů pro specifické aplikace je kladen důraz na velikost, spotřebu a samozřejmě na požadovaný výkon vzhledem k aplikaci.

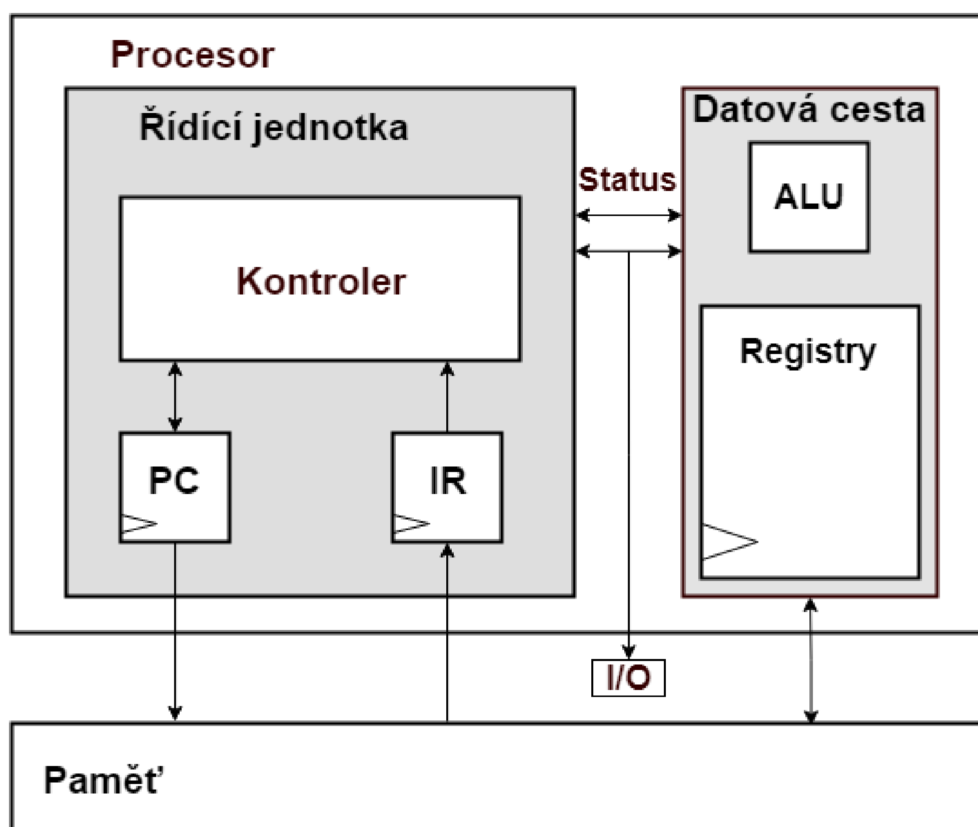
Pro potřeby těchto zařízení přišly na trh tzv. aplikačně specifické procesory (ASIP – Application-Specific Instruction Set Processor), které mohou dosáhnout vysokých výpočetních výkonů a nízké enegetické náročnosti. Instrukční sada ASIP je speciálně navržena pro urychlení složitých a nejpoužívanějších funkcí.

Cílem této diplomové práce je seznámit se s architekturou RISC-V, její základní instrukční sadou a instrukční sadou pro bitové manipulace. Poté za pomocí nástrojů od firmy Codasip navrhnout a otestovat mikroarchitekturu procesoru s instrukční sadou RISC-V RV32IB. Návrh procesoru je rozdělen do dvou částí. Nejprve bude navržen model na instrukční úrovni, který popisuje architekturu instrukční sady a je časově nezávislý. Poté bude navržen model na RTL (register-transfer level) úrovni, který je časově závislý a popisuje, jak bude mikroarchitektura ve skutečnosti vypadat.

1 Procesor

Procesor neboli CPU (centrální procesorová jednotka) je hlavní součástka, která je schopna vykonávat strojový kód (instrukce). Jednoduchou strukturu procesoru můžeme vidět na obr.1.1 Za elementární části CPU můžeme považovat:

- **ALU** (aritmeticko-logická jednotka), která je schopna vykonávat matematické a logické operace nad zpracovávanými daty,
- **řídící jednotka** řídí tok dat a to načtení instrukce a operandů z operační paměti, dekódování příslušné instrukce a uložení výsledku dané operace do paměti,
- **registry** slouží k uchování dat během vykonávání instrukce,
- **I/O** (vstupno-výstupní) jednotka.



Obr. 1.1: Struktrura CPU [6]

Zpracování každé instrukce probíhá v několika krocích. Nejprve se načte příslušná instrukce z paměti do instrukčního registru (IR) a přidělí se jí adresa z programového čítače. Obsah instrukčního registru je přesunut do dekodéru, kde je instrukce dekódována. Řídícím signálem pro ALU a jiné interní obvody jsou přiděleny příslušné

hodnoty dle dekódované instrukce. Jestliže instrukce disponuje operandy, jsou načteny z paměti do pracovních registrů. Jakmile dá řadič pokyn, instrukce je vykonána v ALU. Výstup dané instrukce je uložen do pracovních registrů a následně je uložen zpět do paměti.[6][7]

Procesory můžeme dělit podle:

- šířky slova (16-bit, 32-bit, 64-bit),
- architektury instrukční sady (RISC, CISC),
- architektury (harvardská, von Neumannova),
- podpory operačního systému,
- stupně integrace,
- počtu jader,
- specializace.

Von Neumannova architektura, která používá jednu operační paměť a jeden adresní prostor, je flexibilnější pro aplikace.[6]

Harvardská architektura má zvlášť paměť pro data a zvlášť pro instrukce. Umožňuje použít fyzicky rozdílné typy pamětí. Dále umožňuje použít rozdílnou velikost buňky paměti pro data a pro instrukce. V jednom taktu umožňuje získat instrukci a zároveň její operandy. [6]

U univerzálních počítačů vyhrává von Neumannova architektura. Pro návrh mikrokontrolérů se používají oba typy architektur.

1.1 CISC

Počítač se složitým souborem instrukcí (Complex Instruction Set Computer)[8]. Jak již z názvu vyplývá, instrukční sada této architektury je velmi rozsáhlá. Procesor je schopen vykonávat velmi specifické a komplexní instrukce, které jsou využity jen velmi zřídka. Instrukce mohou mít různé délky s mnoha parametry a modifikačními bity, což má za důsledek složité dekodéry instrukcí. Instrukce jsou realizovány pomocí mikroprogramů sestavených z mikro-instrukcí, přičemž každé instrukci odpovídá posloupnost mikro-instrukcí. Velmi vysoká variabilita má však za důsledek také velkou plochu.

Jestliže si představíme CISC architekturu jako zjednodušený sekvenční obvod, jsou instrukce zpracovány sekvenčně viz tab. 1.1. To znamená, že procesor zpracovává vždy jen jednu instrukci. Proto se snažíme jednotlivé instrukce rozdělit tak, aby jejich čas byl přibližně stejně dlouhý. Jestliže by byly některé instrukce zpracovávány příliš dlouho, zneprůchodnili bychom tím procesor a výrazně bychom tím zpomalili vykonání instrukcí.

Tab. 1.1: Sekvenční provádění instrukcí I v čase T procesorem CISC [8]

	T0	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**	**T11**
Výběr instrukce	I1						I2					
Dekódování		I1						I2				
Výpočet adresy			I1						I2			
Výběr operandu				I1						I2		
Provedení instrukce					I1						I2	
Uložení výsledku						I1						I2

1.2 RISC

Procesor s redukovaným souborem instrukcí (Reduced Instruction Set Computer). RISC procesory vznikly v 70. letech na základě výzkumu četnosti výskytu instrukcí [8]. Z tohoto výzkumu vyplynulo, že programátoři a kompilátory v 75 % používají pouze 8 instrukcí. Četnost ostatních instrukcí se pohybovala v řádu promile. Třemi nejpoužívanějšími instrukcemi byly načtení z paměti, zápis do paměti a podmíněný skok. Proto se rozhodli na konci 70. let vytvořit optimální redukovaný instrukční soubor. Touto problematikou se zabývaly především Univerzita Berkeley v Kalifornii, Stanfordova Univerzita a firma IBM. V roce 1982 na univerzitě v Berkeley vznikl procesor RISCI, který dal název celé této odnoži.

Typickým znakem těchto procesorů je malý instrukční soubor. Mezi jejich další charakteristické znaky patří dokončení instrukce v každém strojovém cyklu, povolena záměna mikroprogramovatelného řadiče za rychlejší obvodový řadič, zřetězené zpracování instrukcí, malý adresní prostor, ukládání a vybírání dat z hlavní paměti výhradně pomocí instrukcí LOAD a STORE, jednotný formát, délka instrukce a využití většího počtu registrů [8]. Jelikož tato architektura pracuje pouze se dvěma instrukcemi pro práci s hlavní pamětí, je nutno navýšit počet interních registrů, aby bylo možno poskytnout dostatek místa pro uložení pracovních dat.

Na rozdíl od CISC procesoru jsou instrukce zpravovány zřetězeně, což znamená, že procesor zpracovává několik různých instrukcí najednou viz tab.1.2. V každém kroku je vykonávána právě jedna instrukce. Na první pohled je viditelné, že zřetězením instrukcí zvýšíme výkon. Ovšem toto zřetězení není dokonalé, jak se může na první pohled zdát. Najdeme zde i jistá úskalí v podobě plnění fronty instrukcí, datových a strukturálních hazardů.

Tab. 1.2: Zřetězené provádění instrukcí I v čase T procesorem RISC [8]

	T0	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**
Výběr instrukce	I1	I2	I3	I4	I5	I6					
Dekódování		I1	I2	I3	I4	I5	I6				
Výpočet adresy			I1	I2	I3	I4	I5	I6			
Výběr operandu				I1	I2	I3	I4	I5	I6		
Provedení instrukce					I1	I2	I3	I4	I5	I6	
Uložení výsledku						I1	I2	I3	I4	I5	I6

Strukturální hazardy se mohou například projevit, když bude potřeba poskytnout sběrnici více jednotkám. Jelikož sběrnice může být využívána pouze jednou

jednotkou, musí být proto provoz na sběrnici řízený, což má za následek zpomalení práce.

Jako příklad datového hazardu při zřetězeném zpracování může být neúplnost dat v daném čase. V praxi to znamená, že například instrukce pro výpočet své adresy potřebuje výsledek z předešlé instrukce, ale výsledek předešlé instrukce ještě není validní [8]. Proto při návrhu zřetězené jednotky je nutno počítat s tímto problémem a uzpůsobit tak příslušný návrh.

Při plnění fronty může nastat problém u instrukcích vykonávající skoky. U nepodmíněného skoku můžeme tento problém vyřešit vyvoláním podprogramů s pevnou adresou. Větší problém nastane, jedná-li se o nepodmíněný skok. Důvodem je výpočet adresy, protože příslušná adresa nemusí být v daném okamžiku validní. Tento problém lze řešit několika způsoby. První z nich je nezastavovat celou frontu a nečekat na validní výsledek, ale počkat, zda se skok provede či ne. Jestliže skok nebude proveden, fronta běží dále bez něho. Pokud se skok provede, fronta se smaže a začne se znovu plnit. Dalšími možnými řešeními jsou dvě paralelní fronty nebo predikce skoku. V obou těchto případech je nutno zajistit další jednotky prediktor nebo řídící jednotku pro paralelní fronty.

1.3 ARM

ARM procesory spadají pod architekturu RISC. Jedná se o nejvíce rozšířené typy RISC procesorů v současnosti. Jejich historie sahá do poloviny 80 let. První čip ARM Holdings byl ARM 1, jenž byl inspirován procesorem RISC I, ale s originálním řešením problematiky podmíněných skoků [9]. Vykonávání instrukce u ARM 1 bylo rozděleno do tří fází: načtení operačního kódu, dekódování a příprava operandů a vykonání instrukce a zpětný zápis. Instrukční sada ARM obsahovala instrukce o 32 bitové šířce, což přesně odpovídalo šířce datové sběrnice. To umožnilo načíst celou instrukci, což byla velká výhoda oproti tehdejším CISC procesorům. Tato instrukční sada je použitelná dodnes a její největší výhodou je možnost přidat ke každé instrukci podmínku pro splnění dané instrukce. Díky těmto podmínkám lze redukovat skoky, které jsou pro implementaci velice náročné u obou architektur.

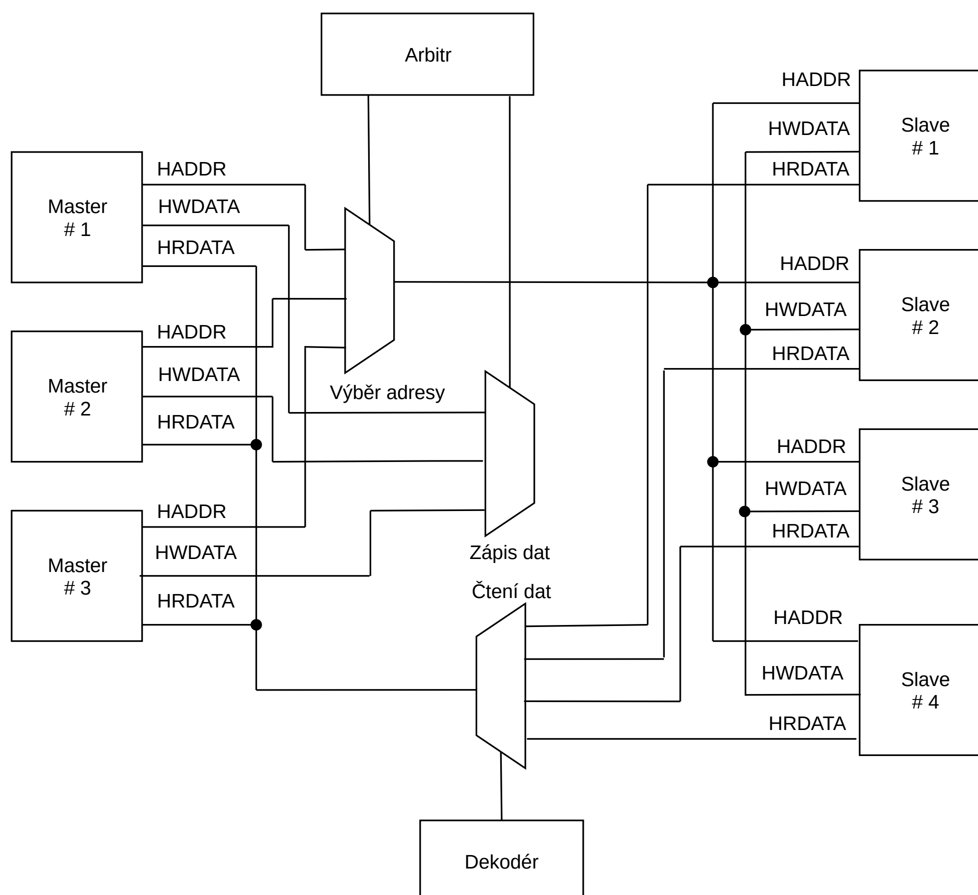
ARM instrukční sada má i své nevýhody a proto vznikla u novějších rodin nová instrukční sada Thumb. Oproti ARM zkrátila délku instrukce z 32 na 16 bitů. Na úkor tohoto zmenšení přišla istrukční sada Thumb o již zmiňované podmínky a musela se vrátit k tradičním řešením skoku. Ale díky těmto krokům dosáhla velice jednoduchého dekodéru, který může být nahrán do interní ROM paměti nebo může být přímo realizován na čipu. Tato sada byla vybrána na základě analýzy programů pro jazyky C a C++. Vzhledem k tomu, že v ARM nechtěli přijít o svou výhodu s podmínkami v původní instrukční sadě ARM, rozhodli se zrealizovat přepínání mezi ARM a Thumb sadou. Přepínání je realizované pomocí speciální instrukce skoku[9]. Přepnutí je umožněno i v rámci samotných funkcí.

2 AHB sběrnice

V této práci je použita pro komunikaci s pamětí AHB sběrnice. Jedná se o pokročilou vysoce výkonnou sběrnici. Uplatnění pro tuto sběrnici najdeme například při připojení komponentů, které potřebují větší šířku pásma na sdílené sběrnici, jako například interní paměť nebo externí paměťové rozhraní DMA, DSO atd. Tato sběrnice implementuje vlastnosti vyžadující vysokou výkonnost, vysokou frekvenci hodin zahrnující[11]:

- dávkový přenos dat,
- rozdělení transakcí,
- neimplementuje stav vysoké impedance,
- širší konfigurace datové sběrnice (64/128 bitů).

Propojení sběrnice je navrženo tak, aby zvládlo vícero master zařízení na sběrnici bez trojstavové logiky. Pro tento účel je nutná přítomnost centrálního multiplexoru a arbitru viz obr.2.1



Obr. 2.1: Schéma zapojení sběrnice [11]

Z obrázku 2.1 můžeme vyčíst hlavní prvky sběrnice. Patří zde master zařízení, slave zařízení, arbitr a dekodér. Popis jednotlivých prvků:

Master zařízení

Povoluje operaci čtení nebo zápisu, poskytuje adresu a řídící signály přenosu slave zařízení. AHB podporuje v systému více master zařízení, avšak v jednu chvíli může komunikovat pouze jeden master. Master může být například mikroprocesor nebo DMA (přímý přístup do paměti neboli direct memory Access).

Slave zařízení

Jeho povinností je odpovědět na požadavek čtení nebo zápisu přicházejícího z master zařízení. Od master zařízení dostane adresu. Slave zařízení monitoruje stav přenosu mezi master zařízením pomocí signálů. Za slave zařízení můžeme považovat například interní paměť nebo rozhraní externí paměti.

Arbitr

Arbitr určuje, který z připojených master zařízení může v daném okamžiku využívat sběrnici. Každý systém může mít pouze jeden arbitr. Algoritmus pro arbitr je specifikovaný v závislosti na využití dané aplikace.

Dekodér

Jeho funkcí je dekódovat adresu a aktivovat správné slave zařízení. V systému je pouze jeden dekodér, protože může být aktivní jen jedno master zařízení.

2.1 Průběh přenosu na AHB sběrnici

Před zahájením přenosu mezi master zařízením a slave zařízením master musí požádat arbitr o přístup na sběrnici pomocí signálu HBUSRQx. Po udělení přístupu master zařízení vystaví na sběrnici signál HREADY a přenos může začít. AHB přenos se skládá z adresy, kontrolního cyklu a jednoho nebo více cyklů pro data. Master začíná přenos nastavením adresy a kontrolních signálů. Kontrolní signály obsahují informace o směru komunikace (čtení nebo zápis), datové šířce a o typu přenosu (možnost dávkového přenosu). Tyto informace jsou vystaveny pouze jeden hodinový cyklus. Během jednoho hodinového cyklu musí slave zařízení tyto informace stihnout přečíst.

Ze sběrnice můžeme očekávat následující odpovědi viz tab. 2.1.

Tab. 2.1: Odpovědi AHB sběrnice v rámci signálu HRESP [11]

HRESP [1:0]	**Popis**
OKAY	Označení správného průběhu přenosu. Jestliže je HREDY jedna, přenos je úspěšně dokončen.
ERROR	Došlo k chybě přenosu
RETRY, SPLIT	Přenos nelze dokončit ihned

Každý přenos může být klasifikován do jednoho ze čtyř různých typů pomocí signálu HTRANS[1:0] viz tab. 2.2.

Tab. 2.2: Klasifikace přenosu AHB sběrnice [11]

HTRANS [1:0]		**Popis**
00	IDLE	Master zařízení nevyžaduje data.
01	BUSY	V případě dávkového přenosu značí, že probíhá komunikace.
10	NONSEQ	Začátek komunikace
11	SEQ	Na sběrnici je vystavena adresa z předchozího přenosu (dávkový přenos)

3 RISC-V

RISC-V je otevřená instrukční sada, která vznikla na akademické půdě Kalifornské univerzity Berkeley. V současné době je RISC-V spravován stejnojmennou neziskovou společností RISC-V, založenou v roce 2005, jejíž zakladatelé jsou Bluespec, Inc; Google; Microsemi; NVIDIA; NXP; Kalifornská univerzita v Berkeley a Western Digital. Cíl RISC-V je umožnit vývoj nových procesorů v otevřeném standardizovaném ekosystému a zajistit novou rozšiřitelnou volnou instrukční sadu.

Každé hardwarové vlákno může být spuštěno v jiném režimu oprávnění, tzv. privilegované módy. RISC-V umožňuje čtyři privilegované módy: strojový (M-machine), režim jádra (S-supervisor), uživatelský (U-user) a ladící (D-debug), viz tab.3.1. Úrovně oprávnění slouží k zajištění ochrany softwarového zásobníku před neoprávněným spuštěním operace z jiného privilegovaného módu. Instrukce spuštěné v M módu mají neomezený přístup ke strojové implementaci. Využíváme ho k zabezpečenému spuštění a spravování RISCV prostředí. Kombinace U a S módu se využívá pro účely operačního systému. Privilegované módy je možno zkombinovat viz tab. 3.2. Zvláštním privilegovaným režimem je mód D, který podporuje ladění jádra.

Tab. 3.1: RISC-V privilegované módy[10]

Level	Kodóvání	Jméno	Zkratka
0	00	uživatelský/ aplikační	U
1	01	Supervizor	S
2	10	Rezervováno	
3	11	Strojový	M

Tab. 3.2: Podporované kombinace privilegováných módů[10]

Mód	Použití
M	Vestavěné systémy
M+U	Vestavěné systémy s privilegovaným módem
M+U+S	Systémy umožňující provoz operačního systému

3.1 Instrukční sada RISC-V procesorů

Jako základ v RISC-V je nutno implementovat instrukční sadu pro celočíselné operace, kterou si může uživatel doplnit o libovolné rozšíření. V podstatě se jedná o souhrn dřívějších sad RISC s podporou variabilní délky instrukce bez opožděných skoků. Základní ISA obsahuje nejnutnější instrukce.

Pro ISA můžeme zvolit tři varianty adresního prostoru 32-bitový (RV32I), 64-bitový (RV64I) a 128-bitový. RV32I a RV64 jsou stabilní používané varianty, které je možno používat současně. RV32I má taky podmnožinu RV32E, což je základní sada instrukcí, která je učena pro mikrokontrolery [1]. RV32E nemá konečnou formu specifikace. RV128 je určena pro budoucí vývoj procesorů.

Základem RISC-V je celočíselná ISA, která je označovaná „I“, obsahuje celočíselné aritmetické operace, instrukce pro řízení toku (řízení skoku), instrukce pro načtení a uložení celého čísla z paměti. Tato sada je povinná po všechny implementace RISC-V. Instrukční sadu lze s ohledem na rozšíření rozdělit do dvou skupin:

Standardní instrukční sada obsahuje široce použitelné instrukce. Tyto instrukce nekolidují s instrukcemi v základní ISA ani s instrukcemi z jiných standardních rozšíření. V současné době zde patří instrukční rozšíření "MAFDQLCBTPV".

Nestandardní rozšíření mohou být vysoce specializované. Tato rozšíření mohou kolidovat s jinými standardními nebo nestandardními rozšířeními.

3.1.1 Obecné standardní rozšíření

Kromě základní ISA, která je označovaná „I“, lze implementovat i další obecné rozšíření. Jedná se o rozšíření označené písmeny [1]:

- „M“ (Integer multiply/divide) je určena pro dělení a násobení celých čísel,
- „A“ (Atomic memory operations) atomické instrukce, které umí atomicky číst, zapisovat a modifikovat paměť mezi několika běžícími RISC-V jádry ve stejném adresním prostoru,
- „F“ (Single-precision floating-point) zahrnuje instrukce s plovoucí řádovou desetinnou čárkou se základní přesností dle standardu IEEE 754-2008,
- „D“ (Double-precision flouting-point) jedná se o rozšíření s plovoucí řádovou desetinnou čárkou.

Všechna tato rozšíření lze shrnout pod jedno písmeno „G“.

3.1.2 Specifická standardní rozšíření

Specifická standardní rozšíření jsou označena písmeny [1]:

- „Q“ (Quad-precision floating-point) instrukce s pohyblivou desetinnou s čtyřnásobnou přesností pro 128-bitové instrukce. Spadá pod standard IEEE 754-2008.
- „L“ (Decimal floatig-point) je určeno pro podporu dekadické desetinné aritmetiky dle standardu IEEE 754-2008.
- „C“ (Compressed instructions) zahrnuje komprimovanou instrukční sadu redukující velikost instrukcí na 16 bitů. Komprimovanými instrukcemi může být nahrazeno 50% - 60% instrukcí RISC-V, výsledkem je 25% - 30% snížení velikosti kódu,
- „B“ (Bit Manipulation) je určeno pro bitové operace.
- „T“ (Transactional Memory) je učeno pro operace, které manipulují s pamětí,
- „P“ (Packed SIMD), přidá 8-bitové a 16-bitové instrukce SIMD pro zvýšení propustnosti výpočtů nad 8-bitovými a 16-bitovými DSP. Pracuje s registry pro práci s plovoucí řadovou desetinnou čárkou.
- „V“ (Vector Operations) je učeno pro práci s vektory. Podporuje konfigurovatelnou vektorovou jednotku, aby bylo možno dosáhnout kompromisu v počtu vektorových registrů.

3.1.3 Nestandardní rozšíření

Doposud známá nestandardní rozšíření jsou označena písmeny:

- „N“ rozšíření poskytuje možnost přidání uživatelských přerušení a zachytávání výjimek.
- „J“ (Dynamically translated languages) je učeno pro dynamický překlad jazyka.

Do budoucna se předpokládá, že přijde spousta nestandardních rozšíření, která se postupně překlopí v standardní rozšíření.

3.1.4 Přehled rozšíření

V tabulce 3.3 naleznete stručný přehled o stavu a verzi jednotlivých rozšíření [1]. RISC-V definuje přesné pořadí, které musí být použito k definování ISA:
RV [32, 64, 128] I, M, A, F, D, G, Q, L, C, B, J, P, V, N

Tab. 3.3: Přehled současných RISC-V rozšíření

Označení	Název rozšíření	Verze	Stav
A	Atomické instrukce	2.0	Uzavřeno
B	Bitové operace	0.37	Otevřeno
C	Komprimované instrukce	2.0	Uzavřeno
D	S plovoucí desetinnou čárkou, dvojitá přesnost	2.0	Uzavřeno
F	S plovoucí desetinnou čárkou, základní přesnost	2.0	Uzavřeno
I	Základní celočíselné operace	2.0	Uzavřeno
J	Dynamický překlad jazyků	0.0	Otevřeno
L	Dekadická plovoucí čárka	0.0	Otevřeno
M	Celočíselné násobení a dělení	2.0	Uzavřeno
N	Uživatelské přerušení	1.1	Otevřeno
P	Balíček pro SIMD instrukce	0.1	Otevřeno
Q	S plovoucí desetinnou čárkou, čtyřnásobná přesnost	2.0	Uzavřeno
V	Vektorové operace	0.2	Otevřeno

3.2 Formát instrukcí

V základní ISA existují čtyři formáty instrukcí (R, I, U, S) viz tab.3.4, jejichž délka je vždy zarovnána na 32 bitů[1]. Instrukce si uchovává zdroje ve zdrojových registrech (rs1 a rs2) a ukládá je do cílového registru (ds). Pro snadnou implementaci a následné dekódování instrukce jsou tyto registry na stejné pozici v kódování, kromě 5-bit instrukcí využívající CSR (Control status register). Nalezneme zde operační kód instrukce, který slouží pro identifikaci instrukce. U nejvýznamnějšího bitu nalezneme přímé kódování konstant. Tohoto faktu je využito k urychlení znaménkového rozšíření konstant, jelikož znaménkový bit je na MSB pozici. Ke čtyřem základním formátům přibývají ještě 2 rozšířené a to B a J založené na manipulaci s konstantami.

Tab. 3.4: Formát instrukcí [1]

31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		zdrojový reg 2		zdrojový reg 1	funct3	cílový reg		opcode	R-Typ
konstanta [11:0]				zdrojový reg 1	funct3	cílový reg		opcode	I-Typ
konstanta [11:5]		zdrojový reg 2		zdrojový reg 1	funct3	konstanta [4:0]		opcode	S-Typ
konstanta [12]	konstanta [10:5]	zdrojový reg 2		zdrojový reg 1	funct3	konstanta [4:1]	konstanta [11]	opcode	B-Typ
konstanta [31:12]						cílový reg		opcode	U-Typ
konstanta [20]	konstanta [10:1]		konstanta [11]	konstanta [19:12]		cílový reg		opcode	J-Typ

3.2.1 Formáty instrukcí pro B rozšíření

Jak již bylo zmíněno v kapitole 3.2, RISC-V má svoje standardizované formáty instrukcí a ani B rozšíření není výjimkou. U již zmiňovaných elementárních instrukcí se můžeme potkat s formátem typu I (viz tab.3.5) a S (viz tab.3.6 3.7).

Tab. 3.5: Instrukce s typem formátu I[2]

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	func3	rd	opcode
????????????	zdrojový reg.	*CLZ*	koncový reg.	OP-IMM
????????????	zdrojový reg.	*CTZ*	koncový reg.	OP-IMM
????????????	zdrojový reg.	*PCNT*	koncový reg.	OP-IMM

Tab. 3.6: Instrukce s typem formátu S[2]

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	func3	rd	opcode
10?????	zdrojový reg. 2	zdrojový reg. 1	*SRO*	koncový reg.	OP
10?????	zdrojový reg. 2	zdrojový reg. 1	*SRO*	koncový reg.	OP
10?????	zdrojový reg. 2	zdrojový reg. 1	*ROR*	koncový reg.	OP
11?????	zdrojový reg. 2	zdrojový reg. 1	*ROL*	koncový reg.	OP
11?????	zdrojový reg. 2	zdrojový reg. 1	*ANDC*	koncový reg.	OP

Tab. 3.7: Formát instrukce s konstantou[2]

31 27	26 20	19 15	14 12	11 7	6 0
imm[11:7]	imm[6:0]	rs1	funct3	rd	opcode
10???	konstanta	zdrojový reg.	*SLOI*	koncový reg.	OP-IMM
10???	konstanta	zdrojový reg.	*SROI*	koncový reg.	OP-IMM
11???	konstanta	zdrojový reg.	*RORI*	koncový reg.	OP-IMM

Podíváme-li se například na tab. 3.5, vidíme, že každá instrukce je podle typu rozčleněna na několik sekcí. Nejdůležitější částí je operační kód, který je například u tab. 3.5 poskládaný z 12 bitové konstanty (imm[11:0]), položky *funct3* a 7 bitového operačního kódu (opcode). Bohužel u rozšíření B nejsou tyto položky pevně zadány, jelikož ještě nebyly uveřejněny jejich přesné pozice. Specifikace nám ale ke každé skupině instrukcí dává doporučení, kde by měla být příslušná instrukce přibližně umístěna. Je tedy nutné navrhnout všechny operační kódy tak, aby nekolidovaly s již pevně danými operačními kódy. Drobný problém nastává u instrukcí s konstantou, kde se délka konstanty mění s nastavením bitové šířky. Proto je nutné do návrhu zahrnout i proměnlivou délku operačního kódu, jestliže se bude jednat o návrh RV64 nebo RV128. V mém případě tento problém zcela odpadne i společně s implementací instrukcí s koncovkou W.

4 Jazyk CodAL

Jazyk CodAL (Codasip Architecture Language) byl vyvinut pod záštitou výzkumného projektu Lissom (Language for Instruction Set Simulator-Oriented Model) na Fakultě informačních technologií Vysokého učení technického v Brně. V současné době je jazyk CodAL vyvíjí firma Codasip. Jazyk je vhodný k vývoji procesorů se specifickou instrukční sadou (ASIP, application-specific instruction set processor) a multiprocesorových platforem [5].

Spolu s jazykem CodAL je souběžně vyvíjeno také návrhové prostředí Codasip Studio. Návrhové prostředí umožnuje souběžný návrh hardwaru a softwaru, možnost rychlých změn v mikroarchitektuře procesoru a v instrukční sadě díky vysoké míře automatizace a rychlému testovaní s možností optimalizace. Z popsaného modelu v tomto jazyce je možnost vygenerovat nástroje pro překlad programu a simulace. Codasip Studio umožňuje taky generování RTL (Register Transfer Level) popisu procesoru a to v jazycích VHDL (Very High Speed Integrated Circuit Hardware Description Language), Verilog a SystemVerilog.

4.1 IA a CA model

Model procesoru může být popsán buď na instrukční úrovni tzv. IA model (Instruction Accurate model) nebo na RTL úrovni tzv. CA model (Cycle Accurate model).

Model na instrukční úrovni (IA) popisuje architekturu instrukční sady. Celý model je časově nezávislý, tudíž paměť s daty je dostupná okamžitě. Totéž platí o zápisu do registru atd. Z popsaného IA modelu je následně generován assembler a překladač jazyka *C/C++* a simulátor. Nad IA modelem je možno spustit testování a simulátor pro ověření funkčnosti a možnosti ladění kódu.

Model na RTL úrovni (CA) popisuje, jak bude mikroarchitektura ve skutečnosti vypadat. Tento model je na rozdíl od IA modelu časově závislý a proto se v CA modelu projeví veškeré aspekty časových zavislostí. Z CA modelu se generuje HDL kód. Pro CA model stejně jako u IA molodelu je možné provést simulaci a testování, které společně s nástrojem *debugger* pomáhají s vývojem. Můžeme tak kontrolovat správnost implementace během vývoje. Podrobnější popis CA a IA modelu nalezne v praktické části diplomové práce to v kapitolách 5 a 6.

4.2 Signály a registry

Signály se používají v CA modelu s definovanou zřetězenou linkou. Zapsaná hodnota do signálu je dostupná ke čtení ve stejném hodinovém cyklu (deklarace signálu viz algoritmus 4.1). Na rozdíl od signálu se registry chovají jako klopný obvod D [5]. To znamená, že zapsaná hodnota je dostupná ke čtení až v následujímcím hodinovém cyklu.V každém hodinovém cyklu je povolen pouze jeden zápis do jednotlivých signalů a registrů. Pro lepší čitelnost jsou signály pojmenovány vždy písmenem „s“ na začátku názvu a registry jsou pojmenovány vždy písmenem „r“ na začátku názvu.

```
// signál o velikosti 5 bitů
signal bit[5] s_example;
```

Algoritmus 4.1: Deklarace signálu

Registry jsou základním stavebním prvkem každého procesoru. Jazyk CodAL podporuje deklaraci běžných registrů i speciálních registrů. Speciální registry jsou označeny klíčovým slovem[5]:
pc – architekturální registr pro programový čítač (viz algoritmus 4.2), tento registr musí obsahovat každý model procesoru
arch – architekturální registr (viz algoritmus 4.3), který je přístupný z pohledu programátora
alias – alias registru (viz algoritmus 4.4), používáme, když potřebujeme přistupovat k registru nebo jeho částem pomocí různých názvů [5].

```
// adresa je defaultně nastavena na hodnotu zavaděče
pc register bit[ADDR_W] r_pc { default = BOOT_START; };
```

Algoritmus 4.2: Architekturální registr pro programový čítač

```
// Příklad architekturálního registru
arch register bit[XLEN] r_arch_new;
```

Algoritmus 4.3: Architekturální registr

```
// alias pro druhý bit registru r_psw
alias register bit[1] a_carry
{
  second bit overlap = r_psw[2..2];
};
```

Algoritmus 4.4: Alias registru

Nedílnou součastí každého procesoru je registrové pole. Stejně jako u samostatných registrů se požadavek na čtení v registrovém poli provádí v rámci aktuálního hodinového cyklu (asynchronní čtení, latence je nula), zatímco zapsaná hodnota je viditelná až v příštím hodinovém cyklu (synchronní zápis, latence je jedna)[5]. Toto chování se aplikuje pouze v rámci CA modelu. U modelu IA se je zapis do registrového pole okamžitě viditelný.

K registrovému poli se přistupuje podobným způsobem jako v jazyce (pomocí závorek). Číslo v závorce označuje index v registrovém poli. Styl přístupu je stejný jak u IA tak u CA modelů. Pro CA model musí mít registrové pole zadefinováno počet datových portů pro čtení a zápis.

V následujícím příkladu algoritmu 4.5 je definován architekturální registrové pole *rf_gpr*, které je použito v rámci implementace procesoru. Registrové pole má 32 prvků o velikosti 32 bitů, jeden zápisový a dva čtecí datové porty.

```
// hlavní registrové pole
arch register_file bit[32] rf_gpr
{
 dataport r1, r2 { flag = R; };
 dataport w { flag = W; };
 size = RF_GPR_SIZE; // 32
 reset = true;
};
```

Algoritmus 4.5: Deklarace hlavního registrového pole

4.3 Základní blok události *event*

Jelikož je jazyk CodAL sekvenční, je potřeba model procesoru poskládat z bloku vyvolávající události. Pro tento účel slouží základní funkční blok označený klíčovým slovem ***event***. Jeho funkce je popsána v rámci sémantické sekce 4.7. Každá událost může aktivovat jinou událost. Událost může být přiřazena jednotlivým fázím zpracování. Obvykle mají stupně přiřazeny několik událostí. Pro každý model procesoru musí být definovaná základní událost (*event*) ***main***. Můžeme také přidat událost ***reset***.

Event reset se aktivuje pouze jednou a to na začátku simulace. *Event reset* určuje, co se stane po resetování procesoru. Například se zde mohou inicializovat registry nebo paměť. *Event main* se spouští s každým novým taktem procesoru a aktivuje všechny funkční bloky.

U IA modelu se obvykle využívá pouze událostí *reset* a *main*, jelikož funkcionalita instrukcí je popsána v rámci sémantik instrukcí. V rámci IA modelu je událost *main* využívaná k aktualizaci programového čítače, načtení instrukcí z paměti, obsluhy přerušení a aktivaci dekodéru.

Jak již bylo zmíněno, každá úloha může být přiřazena k stupni označené klíčovým slovem ***pipeline*** 4.6. Stupně jsou struktury, které dělí úlohy procesoru na jednotlivé etapy. Tyto stupně jsou autonomní a umožňují určitý druh paralelismu. Stupni lze přiřadit v definicích k registru a událostí. Každý stupně má dvě vestavěné funkce:
stall() - když je voláno zastavení v sémantické sekci, všechny registry přiřazené k tomuto stupni si zachovají své hodnoty, i když dojde k zápisu v aktuálním hodinovém cyklu.
clear() - jesliže je v sémantické sekci voláno vyčištění, veškeré registry přiřazené k tomuto stupni nastaví své hodnoty na výchozí, i když k zápisu dojde v aktuálním hodinovém cyklu.

```
event ex_module : pipeline(pipe.EX)
{
  // interní registr pro event ex_module
    register bit[32] r_sub_res;

  semantics
  {
    // funkce eventu
  };
};
```

Algoritmus 4.6: Konstrukce události (*event*)

```
// deklarace stupňů FE, EX, WB
pipeline pipe { FE, EX, WB };

// registr s přiřazeným stupněm EX
register bit[DATA_W] r_ex_data { pipeline = pipe.EX; };

//zastavení stupně EX v rámci sémantické části
pipe.EX.stall();
```

Algoritmus 4.7: Deklarace a použtí stupně (*pipeline*)

4.4 Adresní prostor a rozhraní

Adresní prostor definuje pohled na pamět, kterou vidí procesor. Nástroj podle adresního prostoru zjistí, jaká pamať by měla býť použitá pro program a data. Každá implementace procesoru musí zahrnovat adresní prostory pro program a data. V rámci této práce byla pro implentaci použita von Neumannova architektura. U von Neumannovy architektury je definován adresní prostor, který je společný pro program i data. Typ adresního prostoru *ALL* označuje von Neumannovu architekturu a může být definován pouze jednou v rámci popisu procesoru.

Definice adresního prostoru musí obsahovat nastavení, která se používají pro přístup do paměti. V následujícím příkladu algoritmu 4.8 je ukázka z implementace procesoru. Je definován jeden adresní prostor pro program a data *as_all*. Používají se rozhraní *if_fe* a *if_ldst*. Rozhraní *if_fe* je určeno pro načtení programu. Rozhraní *if_ldst* je určeno pro data. S rozhraním *if_ldst* pracují instrukce pro práci s pamětí.

```
address_space as_all
{
  type = ALL;  // von Neumann
  bits = { VADDR_W, DATA_W, LAU_W };
  endianness = LITTLE;

  interfaces =
  {
    DATA: ldst_interface,
    PROGRAM: fetch_interface
  };
};
```

Algoritmus 4.8: Definice adresního prostoru

Rozhraní se používá hlavně pro připojení sběrnice a paměti. Umožňuje rychlé prototypování procesoru, protože to skrývá podrobnosti implementace [5].

Každé rozhraní má položku typ, která se skládá ze dvou částí. První označuje protokol a druhá roli (master nebo slave). Existuje již několik předdefinovaných typů. Pro tuto páci byl použit typ protokolu AHB v roli master.

V ukázce algoritmu 4.9 můžeme vidět rozhraní pro program *if_fetch*, který je použit v rámci implementace. Rozhraní pracuje s malou endianitou, můžeme přes něho pouze číst. Data jsou zarovnána po 32 bitech.

```
interface if_fetch
{
  type = AHB3_LITE:MASTER;
  bits = { VADDR_W, DATA_W, LAU_W };
  endianness = LITTLE;
  flag = R;
  alignment = {
    data = {INSTR_W};
  };
};
```

Algoritmus 4.9: Definice rozhraní pro program

V ukázce algoritmu 4.10 můžeme vidět rozhraní pro data *if_ldst*, které je použito v rámci implementace. Rozhraní pracuje s malou endianitou, můžeme přes něho jak číst tak i zapisovat. Data jsou zarovnána na 8, 16 nebo 32 bitů.

```
interface if_ldst
{
  type = AHB3_LITE:MASTER;
  bits = { VADDR_W, DATA_W, LAU_W };
  endianness = LITTLE;
  flag = RW;
  alignment = {
    data = {LAU_W, 2*LAU_W, DATA_W};
  };
};
```

Algoritmus 4.10: Definice rozhraní pro data

5 Model na instrukční úrovni IA

Prvním bodem k dosažení cíle této práce, je naimplementovat model na instrukční úrovni s pomocí jazyka CodAL. V tomto případě implementovaná instrukční sada obsahuje instrukce pro bitové manipulace, aritmeticko-logické operace, práci s pamětí a v neposlední řadě systémové instrukce. IA model obsahuje funkční popis všech instrukcí.

5.1 Popis instrukční sady ISA

ISA (Instruction Set Architecture) obsahuje funkční popis všech instrukcí. Jejím základním stavebním prvkem je ***element***, který se skládá ze tří částí viz algoritmus 5.1. První část ***assembly*** definuje, jak bude vypadat instrukce assembler zápisu. Druhá část ***binary*** identifikuje jednotlivé fragmenty binárního kódu, jako jsou například konstanty, operační kód, registry atd. Poslední a nejdůležitější částí je ***semantics***. Ta nám popisuje funkcionalitu daného elementu.

Každá instrukce má svůj jedinečný operační kód, který slouží jako identifikátor. Tyto operační kódy jsou v tomto případě pevně dané specifikací RISC-V. Mnohokrát nastáva situace, že instrukce se stejným formátem, mají i podobné chování. Proto je vhodné tyto instrukce sloučit do logických celků, což nám poskytuje možnost popsat více podobných instrukcí najednou.

```
element i_my_instr
{
  use opc_my_insr as opc;
  use reg as dst, src1, src2;
  assembly { opc dst "," src1 "," src2 };
  binary { opc[FRAG2] src2 src1 opc[FRAG1] dst
     opc[FRAG0] };
  semantics
  {
    // funkce elementu
  };
};
set isa_MY += i_my_instr;
```

Algoritmus 5.1: Obecná definice elementu instrukce

Element můžeme využít také pro definici registrů a konstant, přičemž je můžeme využít v rámci elementu instrukce pomocí klíčového slova ***use***. Všechny elementy

instrukcí jsou sloučeny pomocí klíčového slova ***set*** a vznikne tak funkční popis instrukční sady ISA.

Pro snazší popisování ISA je vhodné si vytvořit funkce pro opakujíci se rutiny. Téměr každá instrukce provádí zápis do hlavního registrového pole. Pro tyto účely byly vytvořeny funkce pro zápis do registrového pole *rf_gpr_read* a pro čtení z registrového pole *rf_gpr_write*.

Funkce pro čtení *rf_gpr_read* 5.2 načte 32 bitovou hodnotu do proměnné. Jejím vstupním parametrem je adresa registru, ze kterého chceme číst. Uvnitř funkce se přečte hodnota z příslušného registru. V případě, že se jedná o registr *X0*, funkce vráti nulu, jelikož tomuto registru je trvale přiřazena nula.

```
uint32 rf_gpr_read(const uint5 i)
{
  if(i != 0)
  {
    return (rf_gpr[i]);
  }
  else // rf_gpr[0] = 0
  {
    return 0;
  }
}
```

Algoritmus 5.2: Funkce pro čtení z hlavního registrového pole

Funkce pro zápis *rf_gpr_write* 5.3 zapíše 32 bitovou hodnotu příslušného registru. Jejími vstupními parametry jsou adresa registru, kde chceme zapsat, a hodnota, kterou chceme zapsat. Uvnitř funkce se hodnota zapíše do příslušného registru. V případě, že se jedná o registr*X0*, zápis se neprovede, jelikož do registru *X0* není povolen zápis. Funkce nemá žádnou návratovou hodnotu.

```
void rf_gpr_write(const uint5 i, const uXlen val)
{
  if (i != 0)
  {
    rf_gpr[i] = val;
  }
}
```

Algoritmus 5.3: Funkce pro zápis do hlavního registrového pole

Kromě práce s registrovým polem je zapotřebí pracovat i s pamětí. Pro tyto účely slouží funkce *store_val* pro zápis do paměti, funkce *load_val* pro čtení z paměti a funkce *check_mem_access* pro kontrolu přístupu do paměti.

Funkce *check_mem_access* 5.4 pracuje s adresou a velikosti slova. Pokud funkce vrátí pravdivostní hodnotu pravda, adresa je mimo rozsah. V opačném případě je adresa v pořádku. Můžeme si zde všimnout použití direktivy *#pragma simulator* a *#pragma compiler*. Direktiva *#pragma simulator* určuje, jak se funkce přeloží pro simulátor a *#pragma compiler* jak funkce bude vypadat pro C kompiler.

```
bool check_mem_access(const uint32 sbc, const uint32 address)
{
  #pragma simulator
  {
    return
      ((sbc == WORD_BYTES) && (address & WORD_MASK)) ||
      ((sbc == HWORD_BYTES) && (address & HWORD_MASK));
  }
  #pragma compiler
  {
    return false;
  }
}
```

Algoritmus 5.4: Funkce pro kontrolu přístupu do paměti

Funkce pro čtení z paměti *load_val* 5.5 má dva vstupní parametry operační kód a adresu. Ve funkci se nejprve ověří správnost adresy pomocí *check_mem_access*. Jestliže je adresa mimo rozsah, vyvolá se obsluha přerušení a funkce vrátí nulu. Jesliže je adresa v pořádku, přístoupí přes rozhraní *ldst_interface* do paměti. Funkce vrátí 32 bitovou hodnotu načtenou z paměti.

Funkce pro zápis do paměti *store_val* má tři vstupní parametry operační kód, adresu a data. Stejně jako u funkce pro čtení z paměti se provede kontrola adresy. V případě, že je adresa správná, přistoupí přes rozhraní do paměti a uloží se data. Pro simulátor je potřeba zadefinovat chování u přístupu k systemovému volání. V případě že chceme vyvolat zastavení běhu programu, zapíšeme na adresu systémového volání jedničku a vyvoláme tím funkci *halt*.

Kvůli velkému rozsahu jsou ukázky algoritmů funkcí *load_val* 5.5 a *store_val* 5.6 zkráceny pouze na práci s velkostí slova 32 bitů. V rámci modelu byly naimplemetovány všechny varianty podle RISC-V specifikace.

```
uint32 load_val(const uint32 opc, const uint32 address)
{
  if (check_mem_access(WORD_BYTES, address)) &&
      (opc == OPC_I_LW)
  {
    take_trap(MISALIGNED_LOAD, address, DEC_PC);
    r_mem_error = 1;
    return 0;
  }
  return ((uint32)((int32)ldst_interface.read(address,
      WORD_BYTES)));
```

Algoritmus 5.5: Zkrácená funkce čtení z paměti

```
void store_val(const uint32 opc, const uint32 address, const
    uint32 data)
{
  // ověření adresy je stejné jako funkce load_val
  if (opc == OPC_S_SW)
  {
    ldst_interface.write(
      val & DATA_WORD_MASK, address, WORD_BYTES);
  }
  #pragma simulator
  {
    if (address == SYSCALL_ADDR)
      {
        if (val == 1){
          halt(); }
    }
    else if (val > 1){
      codasip_syscall(val); }
  }
}
```

Algoritmus 5.6: Zkrácená funkce pro zápis do paměti

Funkce *halt* 5.7 volá integrovanou funkci *codasip _halt*, která slouží k ukončení simulace. Funkce může být volána v jakékoliv události či elementu. Při volání funkce *halt* se přečte návratová hodnota z registru *X10*. Volání *codasip_store_exit_code* uvnitř funkce *halt* slouží pro automatické testování pomocí simulátoru. Funkce *codasip_store_exit_code* vytvoří soubor *sim_exit_code*, který obsahuje hodnotu si-

mulovaného programu.

```
void halt()
{
  uint32 val;
  codasip_halt();
  val = rf_gpr_read(RETURN_REG);
  iprintf("HALT: return value 0x%X\n", val)
  iprintf("r_csr_mcause is %d\n", r_csr_mcause);
  codasip_store_exit_code(val);
}
```

Algoritmus 5.7: Funkce halt

Po připravených obecných funkcích se můžeme přesunout k samotné implementaci jednotlivých instrukcí z I a B instrukčních rozšíření.

5.2 Implementace RV32I instrukcí

Implementované instrukce z RV32I můžeme rozdělit do čtyř skupin: výpočetní, skokové, pro práci s pamětí a systémové.

5.2.1 Výpočetní instrukce

Tyto instrukce vykonávají především aritmeticko-logické operace. Instrukce využívají dvou formátů instrukcí a to „R“ a „I“. To znamená, že instrukce pracují pouze s registry v případě „R“ formátu, anebo s registry a přímo kódovanou konstantou v případě „I“ formátu. Žádná s těchto instrukcí nemůže vyvolat výjimku.

Samotná implementace těchto instrukcí je vcelku přímočará. Výpočetní instrukce jsou implementovány ve čtyřech elementech. První element slučuje instrukce využívající jako zdroj dat dva zdrojové registy (algoritmus viz 5.8). Nejprve je potřeba přečíst data z registru pomoci funkce *rf_gpr_read*. Na základě operačního kódu se provede příslušná operace a výsledek provedené operace se zapíše do registrového pole pomocí funkce *rf_gpr_write*. Druhý element je pro instrukce, které využívají jako zdroj dat konstantu a zdrojový registr. Element vypadá téměř stejně jako elemet v ukázce algoritmu 5.8. Rozdíl je v získáni dat. Využívá se pouze jedná funkce *rf_gpr_read* pro *data1*, do *data2* je přiřazen elemet konstanty. Ve třetím a čtvrtém případě jsou data poskytnuta buď jedním registrem nebo jednou konstantou.

```
element i_compute
{
  use opc_comp as opc;
  use reg_any as dst, src1, src2;
  assembly { opc dst "," src1 "," src2 };
  binary { opc[FRAG2] src2 src1 opc[FRAG1] dst opc[FRAG0]
      };
  semantics
  {
    uint32 data1, data2, result;
    // čtení dat ze zdrojových registrů
    data1 = rf_gpr_read(src1);
    data2 = rf_gpr_read(src2);
    //výběr operace podle operačního kódu
    // vykonání jenotlivých operací
    switch (opc)
    {
      case OPC_R_ADD:
        result = data1 + data2;
        break;
      case OPC_R_XOR:
        result = data1 ^ data2;
        break;
      case OPC_R_SRA:
        result = SEXT_XLEN(op1) >> (shift_t)op2;
        break;
      // další instrukce
    }
    rf_gpr_write(dst, result);
  };
};
```

Algoritmus 5.8: Příklad popisu elementu pro výpočetní instrukci

Mezi implementované výpočetní instrukce patří:

ADD(I) 5.9 Instrukce pro aritmetický součet. Instrukce ADD pracuje pouze s registry a instrukce ADDI pracuje s registry a znaménkovou konstantou. Dojde-li k přetečení výsledku, do paměti se uloží spodních 32 bitů.

```
addi dst, src, imm
// addi x5 , x6 , 1  -> x5 = x6 + 1;
```

Algoritmus 5.9: Popis instrukce ADDI

SUB 5.10 Instrukce, která odčítá dva registry. Tato instrukce nemá variantu s konstantou.

```
sub dst, src1 , rsc2
// sub x5 , x6 , x7  -> x5 = x6 - x7;
```

Algoritmus 5.10: Příklad instrukce pro odčítání

SLT(I)(U) 5.11 Tato instrukce zastává funkci operátoru menší než (<). Ve výsledku se objeví buď nula nebo jedna v závislosti na pravdivosti tvrzení. Varianta s „I" nám značí použití znaménkové konstanty. Varianta s „U" nám značí porovnání v absolutní hodnotě, tudíž bezznaménkově. Použitím instrukce SLTIU s konstantou jedna, ověříme, zda je hodnota registru nenulová.

```
li x7 -6
li x6  9
slti  x5 , x7 , -10  // x5 =  -7  < -10 = 0
sltu  x5 , x7 , x6   // x5 = |-7| < |9|  = 1
slt   x5 , x7 , x6    // x5 = -7 < 9 = 1
sltiu x5 , x7 , 1    // x5 = |-7| < |1| = 0
```

Algoritmus 5.11: Příklad instrukcí menší než

AND(I), OR(I), XOR(I) 5.12 Skupina logických operací jako jsou logický součet, součin a exkluzivní disjunkce. Pro logickou negaci je možno díky možnosti znaménkového rozšíření použít instrukci XORI s konstantou mínus jedna.

```
li x7  0b10110
li x6  0b1011
and  x5 , x7 , x6   // x5 = 0b10110 & 0b1011 = 0b10
ori  x5 , x7 , 9    // x5 = 0b10110 | 0b1001 = 0b11111
xori x5 , x7 , -1   // x5 = 0b10110 ^ -1 = 0b01001
```

Algoritmus 5.12: Příklad instrukcí pro logické operace

SLL(I), SRL(I), SRA(I) 5.13 Instrukce, které umožňují logické posuvy doprava (nuly jsou posunuty do nižších bitů), doleva (nuly jsou posunuty do horních bitů) a aritmetický posuv doprava, který si zachovává znaménkový bit. Velikost posuvu je kódována v dolních 5 bitech.

```
li x7  0b10110
srli x5 , x7 , 1  // x5 = 0b10110 >> 1 = 0b1011
srai x5 , x7 , 1  // x5 = 0b10110 >> 1 = 0b11011
```

Algoritmus 5.13: Příklad instrukcí pro posuvy vpravo

LUI používá se k vytvoření 32 bitových konstant. Instrukce konstantu uloží do horních 20 bitů a na spodních 12 bitů uloží samé nuly. Používá se předeším pro uložení konstanty do registru.

AUIPC 5.14 instrukci lze využít k relativnímu adresování. Instrukce vezme horních 20 bitů, přičte hodnotu programového čítače a uloží ho do registru.

```
  // hodnota PC = 0x103c
   auipc x5 , 2   // x5 = 0x103c + 0x2 = 0x103e
```

Algoritmus 5.14: Příklad instrukce AUIPC

5.2.2 Instrukce pro práci s pamětí

Pro RISC-V architekturu je typické, že s pamětí mohou operovat pouze instrukce pro načtení (**LOAD** 5.15) a uložení dat (**STORE**5.15). RV32I poskytuje 32 bitový adresní prostor, přičemž je adresován po bajtech. Využívá uspořádání "little endian", který ukládá na který ukládá na spodní bajt s nějmeně významným bitem a pokračuje po bajtech až k nejvýznamnějšímu bitu.

Instrukce pro ukládání a načítání přenášejí hodnoty mezi registry a pamětí. Instrukce pro načítání jsou kódovány ve formátu „I" a instrukce pro uložení jsou kódovány ve formátu „S". Adresy pro operace jsou získávány ze zdrojového registru, k němuž je přičten 12 bitový offset. Instrukce load načte hodnotu o velikosti danou typem instrukce z paměti do cílového registru. Instrukce pro ukládání provádí to samé, akorát v opačném pořadí. Načte si hodnotu z registru a uloží ji do paměti. Z důvodu výkonnosti musí být adresa zarovnaná na velikost slova.

Typy načítacích instrukcí LW (32bit), LH(16 bit), LHU(16 bit bezznaménkově), LB (8bit) a LBU (8bit bezznaménkově). Typy ukládacích instrukcí SW (32bit), SH(16 bit) SB (8 bit).

```
li x6 _data    // adresa začátku datové oblasti
li x7 0xbeef   // data
lw x14, 8 (x6) // data o velikosti slova uložené na
   adrese _data + 8 se načtou do registů x14.
sb x7, 0 (x6)  // na začátek datové oblati se uloží 8
   bitů registru x7 (0xef)
```

Algoritmus 5.15: Příklad instrukcí LOAD a STORE

Implentace instrukcí je rozdělena do dvou elementů. Jeden pro instrukci LOAD viz 5.16 a jeden pro instrukci STORE. V sémantické části elemetu se zavolá funkce *load_val* pro LOAD nebo *store_val* pro STORE. V případě čtení z paměti se zapíše načtená hodnota do registrového pole pomocí funkce *rf_gpr_write*.

```
element i_load
{
  use opc_loads as opc;
  use reg_any as dst, src;
  use simm12;
  assembly { opc dst "," simm12 "(" src ")" };
  binary { simm12 src opc[FRAG1] dst opc[FRAG0] };
  semantics
  {
    uint32 val;
    val = load_val(opc, rf_gpr_read(src) + simm12);
    #pragma simulator // pro simulátor
    {
      if (!r_pf_mem_error){ // nenastala chyba paměti
        rf_gpr_write(dst, val);
      }
    }
    #pragma compiler // pro kompiler
    {
      rf_gpr_write(dst, val);
    }
  };
};
```

Algoritmus 5.16: Element instrukce LOAD

5.2.3 Skokové instrukce

V RISC-V procesorech nalezneme dva typy skokových instrukcí a to podmíněné a nepodmíněné skoky.

První nepodmíněnou skokovou instrukcí je JAL. Tato instrukce využívá formátu typu J. U instrukce JAR se jedná o relativní skok. To v praxi znamená, že k aktuální hodnotě programového čítače se přičte konstanta. Konstanta u této instrukce má velikost 20 bitů. Posouvá se a 1 bit doleva. Díky této konstantě můžeme skočit v rozsahu o ±1 MiB. Druhou nepodmíněnou instrukcí je JALR. Oproti instrukci JAR se jedná o absolutní skok, což v praxi znamená, že programový čítač je bez ohledu na jeho hodnotu přepsán konstantou instrukce JALR. Obě tyto instrukce si ukládají adresu následující instrukce do svého cílového registru.

Druhou skupinou skokových instrukcí jsou podmíněné skoky. Pracují na principu porovnání hodnot dvou registrů a na základě typu skoku a obsahu registru provedou skok. RV32I obsahuje následující typy skoku respektující znaménko: BEQ (skočí na základě rovnosti), BNE (skočí na základě nerovnosti), BLT (menší než) a BGE (větší nebo rovno). Instrukce BLT a BGE máme v bezznaménkové variantě (BLTU a BGEU). Tyto varianty společně s možností prohazovat operandy nám pokryjí všechny možnosti podmínek. Všechny tyto skoky jsou relativní. Jejich rozsah je ±4 Kib, z čehož vyplývá, že disponují 12 bitovou konstantou posunutou o 1 bit vlevo. V ukázce algoritmu 5.17 vidíme využítí skoku při testovaní správnosti výpočtu.

```
TEST_INST_32(100, ADD, 0x15, 0x17, 0x0000002C)
// definice makra
#define TEST_INST_32(number, inst, op1, op2, expected) \
li x5, op1;\          // x5 = 0x15
li x6, op2;\          // x6 = 0x17
inst x6, x5, x6;\     // add x6, x5, x6 -> x6 = x5 + x6
li x5, expected;\     // x5 = 0x2c
// jesliže x5 = x6 provede se skok na návěští
   test_##number##
// jesliže x5 != x6 pokračuje se dál v programu
beq t0, t1, test_##number##;\
li x10, number;\     // x10 = number (číslo testu)
lui x5, %hi(_exit);\
jalr x0, x5, %lo(_exit);\ // skočí se na návěští _exit
_test_##number##:
```

Algoritmus 5.17: Ukázka instrukcí skoků v testovacím makru

Implementaci podmíněného skoku můžeme vidět na elemetu 5.18. V sémantické části elementu *i_control_conditional* se pomocí funkce *base_operation* rozhodne na základě logických operací, zda je podmínka skoku platná. V případě, že podmínka vyjde pravda, vypočte se adresa skoku. Před samotným skokem je potřeba zkontrolovat, zda vypočtená adresa je zarovnaná. V případě nezarovnané adresy se vyvolá výjimka. V opačném případě je nová adresa přepsána do registru programového čítače. Jesliže podmínka skoku vyšla nepravda, program pokračuje na následující instrukci.

```
element i_control_conditional
{
  use opc_control_conditional as opc;
  use reg_any as src1, src2;
  use rel_addr12 as addr;
  assembly { opc src1 "," src2 "," addr };
  binary { addr[11..11] addr[9..4] src2 src1 opc[FRAG1]
     addr[3..0] addr[10..10] opc[FRAG0]};
  semantics
  {
    if(base_operatin(opc, rf_gpr_read(src1), rf_gpr_read(
       src2))){
      uint32 jmp_addr;
      jmp_addr = (r_PC - INSTR_LAU_SIZE) + addr;
      #pragma simulator{
        if (jmp_addr[1..1]){ // ověření zarovnání adresy
          take_trap(MISALIGNED_FETCH, jmp_addr, DEC_PC);
          r_fetch_fault = 1;
        }
        else // přepis adresy do programového čítače
          r_PC = jmp_addr;
       }
       #pragma compiler {
          r_PC = jmp_addr;
       }
    }
  };
};
```

Algoritmus 5.18: Element pro podmíněný skok

Implementace u nepodmíněných skoků je téměř stejná jako u podminěných skoků 5.18. Jediný rozdíl v implemetaci u nepodmíněných skoků je, že se nevykoná podmínka, zda skok provést či nikoliv. U nepodmíněných skoků se pouze vypočte nová adresa. Ověří se, zda je adresa zarovnáná a přepíše se hodnota v programovém čítači.

5.2.4 Systémové instrukce

Systémové instrukce se používají k přístupu do systému. Přístup může vyžadovat privilegovaný přístup. Tyto instrukce využívají instrukční formát I. První skupina těchto systémových instrukcí jsou instrukce, které mohou atomicky číst a zapisovat do CSR (kontrolní a stavové registry). Pro atomické čtení a zápis do CSR registrů se používá instrukce CSRRW. Instrukce CSRRW načte starou hodnotu z CSR, rozšíří ji nulou na 32 bitů a zapíše do cílového registru. Jestliže je cílový registr X0, tak instrukce nesmí načítat z CSR. Instrukce CSRRS (atomické čtení a nastavení CSR bitů) načte hodnotu CSR, rozšíří hodnotu nulou na 32 bitů a zapíše ji do cílového registru. Hodnota zdrojového registru slouží jako bitová maska pro bitové pozice, které mají být nastaveny v CSR. Změna hodnoty může nastat v případě, že hodnota bitů v masce je rovna jedné a zároveň musí daný bit v CSR umožňovat zápis. Ostatní bity v CSR nejsou ovlivněny. Instrukce CSRRC (atomické čtení a mazaní bitů v CSR) čtení proběhne, jako u předchozích CSR instrukcí. Hodnota ve zdrojovém registru slouží jako bitová maska pro specifikaci bitů, které mají být vymazány v CSR. Smazány jsou pouze ty bity, pro které je v masce nastavena logická jedna. Ostatní bity v CSR nejsou ovlivněny. Pro instrukce CSRRS a CSRRC, pokud je zdrojový registr x0, pak instrukce nebude zapisovat do CSR vůbec. CSRRWI, CSRRSI a CSRRCI jsou varianty s konstantou.

Instrukce ECALL vytváří požadavek na systémové volání, které využívá operační systém. Instrukci EBREAK používají ladící nástroje pro přesměrování do ladícího prostředí. Codasip Studio umožňuje emulaci systémových volání pomocí funkce *codasip_syscall()*. Jako parametr přijímá adresu struktury, která obsahuje informace o požadované akci a její parametry.

Imlementace instrukcí pro práci s kontrolními stavovými registry obsahuje dva elementy. Jeden element pracuje s kostrantou jako zdroj dat a druhý s registrem jako zdrojem dat 5.19 . V případě zápisu do CSR registrů, přečteme pomocí funkce *rf_gpr_read* data nebo použijeme kostantu (záleží na zdroji dat). Po přečtení dat voláme funkci *csr_read_write* 5.20, která provede požadovaný zápis a čtení nad vybraným CSR registem. Jestliže se četlo z CSR registrů, je potřeba přečtenou hodnotu zapsat do registrového pole pomocí funkce *rf_gpr_write*.

```
element i_control_registers_reg
{
  use opc_control_registers_reg as opc;
  use reg_any as dst, src;
  use regs_csr as csr;
  assembly { opc dst "," csr "," src };
  binary { csr src opc[FRAG1] dst opc[FRAG0] };
  semantics
  {
    uint32 write_val;
    write_val = 0;
    #pragma simulator
    {
      if (opc ==  OPC_I_CSRRW ||  opc ==OPC_I_CSRRS ||
         opc == OPC_I_CSRRC){
        write_val = rf_gpr_read(src);
      }
      write_val = csr_read_write(write_val, csr, opc);
      if (!r_illegal_inst2){
        rf_gpr_write(dst, write_val);
      }
    }
  };
};
```

Algoritmus 5.19: Elemet pro operaci s kontrolními stavovými registry

Funkce *csr_read_write* 5.20 je využívána výhradně pro práci s kontrolními stavovými registry. Jejími vstupními parametry jsou data určena pro zápis (*write_val*), jméno kontrolního stavového registru (*csr*) a typ operace (*opc*). Ve funkci se nejprve zkotroluje, zda nenastala ilegální situace pro práci s CSR registrem. Po kontrole se může přistoupit k samotnému zápisu a čtení. Podle jména se vybere příslušný registr, přečte se stávající hodnota a až poté se může zapsat nová hodnota pomocí funkce *csr_write_val*. Pořadí: čtení pak zápis, musí být dodrženo pro případ atomického čtení (CSRRS). Funkce *csr_write_val* v případě atomického čtení se logicky sečtou stávající data a nová data. V připadě zápisu, funkce *csr_write_val* přepíše stávající data novými. V případě mazání bitů, funkce *csr_write_val* provede logický součin stávajících dat a negovaných dat určených pro zápis. Návratová hodnota funkce *csr_write_val* je *read_val* a obsahuje přečtená data z konkrétního CSR registru.

```
uint32 csr_read_write(uint32 write_val, const uint12 csr,
    const uint10 opc)
{
  uint32 read_val;
  bool write;
  write = write_val;
  if ((csr[11..10] == 3) && (write || opc == OPC_I_CSRRW
     || opc == OPC_I_CSRRWI))
  {
    r_illegal_inst2 = 1;
    return 0;
  }
  switch (csr)
  {
    case CSR_MTVAL:
      read_val = r_csr_mtval;
      r_csr_mtval = csr_write_val(opc, read_val,
         write_val);
      break;
    case CSR_MIP:
      read_Val = READ_MIP;
      break;
    // další csr registry
  }
return read_val;
}
```

Algoritmus 5.20: Zkrácená funkce csr_read_write

5.3 Implementace instrukcí pro bitové manipulace

Rozšíření s označením „B“ zahrnuje instrukce pro logické operace s bity, extrakci bitů, vkládání bitů a záměnu bitů [2]. Specifikace podporuje RV32, RV64 a RV128 architekturu. V této práci je implentována architektura RV32. V současné době je vývoj této specifikace stále otevřený. Specifikace obsahuje několik verzí bitové instrukční sady. V této práci jsou implementovány instrukce, které používá firma Codasip. Žádná z těchto instrukcí nemůže vyvolat výjimku.

5.3.1 Základní bitové instrukce

Tyto instrukce je velice snadné implementovat, jelikož zahrnují použití běžných operátorů nebo pro jejich implementaci existuje funkce v jazyce CodAL. Stejně jako u RV32I sady, instrukce označené písmenem „I“ na konci pracují s konstantou.

Mezi základní bitové instrukce patří:

CLZ, CTZ 5.21 slouží pro výpočet pozice nejvýznamnější jedničky nebo nejméně významné jedničky. Operace *CLZ* spočítá počet nulových bitů před jednotkovým bitem od nejvýznamnějšího bitu. Operace *CTZ* spočítá počet nulových bitů na konci LSB argumentu. Pokud jsou vstupy rovné -1, pak výstup je roven 0. Pokud je vstup roven 0, pak výstup je roven velikosti bitové šířky zvoleného adresního prostoru (v tomto případě 32 bitů). Tyto instrukce pracují pouze z registry. Implementace byla provedena pomocí integrovaných funkci *codasip_ctlz_uint32* a *codasip_cttz_uint32* jazyka CodAL.

```
li x6  0b1010
clz x7, x6      // x7 = 28
ctz x7, x6      // x7 = 1

//implementace v jazyce CodAL:
// instrukce CLZ
cnt_zero = codasip_ctlz_uint32(op1);
// instrukce CTZ
cnt_zero = codasip_cttz_uint32(op1);
```

Algoritmus 5.21: Implementace a příklad instrukcí CLZ a CTZ

PCNT 5.22 spočítá počet bitů nastavených na jedna. Tato instrukce pracuje s jedním zdrojovým a s jedním cílovým registrem. Implementace byla provedena pomocí integrované fukce *codasip_ctpop_uint32* jazyka CodAL.

```
li x6  0b1010
pcnt x7, x6      // x7 = 2

//implementace PCTN jazyce CodAL :
result = codasip_ctpop_uint32(op1);
```

Algoritmus 5.22: Implementace a příklad instrukce PCNT

SLO(I), SRO(I) 5.23 tyto instrukce jsou podobné logickým posuvům ze základní istrukční sady I. V tomto případě se však jedná o posuv s jedničkou. Tyto instrukce používají jako zdroj registr nebo jeden ze zdrojů může být konstanta. Implementační výpočet byl proveden následujícím způsobem. Nejprve se zneguje první operand, který je posunut o hodnotu druhého operandu. Výsledek posuvu je znegován.

```
li x6  0xa
li x5  7
slo x4, x6, x5  // x4 = 0xa << 7 = 0x57f
sro x4, x6, x5  // x4 = 0xa >> 7 = 0xfe000000
//varianty s~konstantou
sloi x4, x6, 3  // x4 = 0xa << 3 = 0x57
sroi x4, x6, 3  // x4 = 0xa >> 3 = 0xe0000001

// implementace v jazyce CodAL:
// instrukce SRO(I)
result = ~(~op1 >> op2);
// instrukce SLO(I)
result = ~(~op1 << op2);
```

Algoritmus 5.23: Implementace a příklad instrukcí posuvu s jedničkou SLO a SRO

ROL, ROR(I) 5.24 tyto instrukce slouží k rotaci bitů, kde *ROR* je rotace vpravo a *ROL* je rotace vlevo. Pouze rotace vpravo má variantu s konstantou. Jinak instrukce používají registry. Implementace byla provedena pomocí operátoru rotace.

```
li x6  0xa
li x5  7
rol x4, x6, x5 // x4 = 0xa <<< 7 = 0x500
//varianta s~konstantou
rori x4, x6, 3 // x4 = 0xa <<< 3 = 0x40000001

// implementace v jazyce CodAL:
// instrukce ROL
result = op1 <<< (uint32)op2;
// instrukce ROR(I):
result = op1 >>> (uint32)op2;
```

Algoritmus 5.24: Implementace a příklad instrukcí pro rotace

ANDC, ANDN, ORN, XORN 5.25 Tyto instrukce doplňují logické instrukce z ISA I o jejich negovanou verzi. ANDC instrukce je pro logický operátor AND s komplementem. Tyto instrukce májí dva zdrojové registry a jeden cílový. Implementace probíhala pomocí logických operátorů.

```
li x6  0x12b8ac
li x5  0x4532
ANDC x7, x6, x5 //x7 = 0x12B8ac & (~ 0x4532) =0x12b8c
ANDN x7, x6, x5 //x7 = ~(0x12b8ac & 0x4532) = 0xffffffdf
ORN  x7, x6, x5 //x7 = ~(0x12b8ac | 0x4532) = 0xffed0241
XORN x7, x6, x5 //x7 = ~(0x12b8ac ^ 0x4532) = 0x12fd9e
```

Algoritmus 5.25: Příklad použítí instrukcí ANDC, ANDN, ORN a ORN

5.3.2 Permutační instrukce

První v zastoupení těchto instrukcí je **SHFL(I)** 5.26, která realizuje zobecněné operace, známé jako shuflle (promíchání bitů) a jeho inverzi známou jako unshufle. Všeobecně mají tyto operace $\log_2(32) - 1$ řídících bitů, jeden pro každý pár sousedních bitů. Když je řídící bit nastaven na jedna, zamění shuflle dva indexované bity. Záměny jsou provedeny v pořadí od nevýznamnějšího bitu (MSB) po nejméně významný bit (LSB). Nejpoužívanější variantou je varianta s konstantou, která vytyčuje jednu souvislou oblast nastavených bitů tzv. zip. Pro tyto varianty jsou implementovány pseudo-instrukce.

```
li x6  0xA5A56969
li x5  0x99996969
shfli x4, x6, 4   // x4 = 0xAA556699)
unshfli x4, x5, 2 // x4 = 0xA5A56969)
```

Algoritmus 5.26: Instrukce SHFLI a UNSHFLI

Implementaci pro IA model můžeme vidět na algoritmech 5.27 a 5.28. Výpočet instrukce SHLF zahrnuje dvě funkce *shfl_comp* a *shfl_stage*. Funkce *shfl_comp* má dva vstupní parametry data a mód. Ve funci *shfl_comp* se na základě módu vybírá, kterými stupni sítě (viz obr. 6.4.2) projdou data a jestli bude síť inverzní či nikoliv. V každém stupni je volána funkce *shfl_stage*, která na základě masky a módu provede požadované promíchání bitů.

```
uint32 shfl_stage(const uint32 src, const uint32 maskL,
   const uint32 maskR, const uint8 mode)
{
  uint32 x;
  x = src & ~(maskL | maskR);
  x |= ((src << mode) & maskL) | ((src >> mode) & maskR);
  return x;
}
```

Algoritmus 5.27: Implementace funkce *shfl_stage* [3]

```
uin32 shfl_comp(const uint32 op1, const shift_t mode)
{
  uint32 shfl_mode, x;
  x = op1;
  if (zip_mode & 1) {
    if (mode & 2)
      x = shfl_stage(x, 0x44444444u, 0x22222222u, 1);
    if (mode & 4)
      x = shfl_stage(x, 0x30303030u, 0x0c0c0c0cu, 2);
    if (mode & 8)
      x = shfl_stage(x, 0x0f000f00u, 0x00f000f0u, 4);
    if (mode & 16)
      x = shfl_stage(x, 0x00ff0000u, 0x0000ff00u, 8);
  }
  else {
    if (mode & 16)
      x = shfl_stage(x, 0x00ff0000u, 0x0000ff00u, 8);
    if (mode & 8)
      x = shfl_stage(x, 0x0f000f00u, 0x00f000f0u, 4);
    if (mode & 4)
      x = shfl_stage(x, 0x30303030u, 0x0c0c0c0cu, 2);
    if (mode & 2)
      x = shfl_stage(x, 0x44444444u, 0x22222222u, 1);
  }
  return x;
}
```

Algoritmus 5.28: Implementace funkce *shfl_comp* [3]

Další ze skupiny těchto instrukcí je **GREV(I)** 5.29, která poskytuje všechny možné záměny bitů. Této funkcionality dosáhneme pomocí kontroly jednotlivých úrovní rekurzivního stromu, díky němuž můžeme obracet bity. Operace iterativně kontroluje každý bit zdrojového registru od nultého bitu až po 31. bit. Pokud je odpovídající bit nastaven na jedna, provede se obrat *2i* sousedních bitů.

```
li x6  0x87654321
grevi x4, x6, 31   // x4 = 0x84C2A6E1
```

Algoritmus 5.29: Příklad instrukce GREVI

Implementaci pro IA model můžeme vidět na algoritmu 5.30. Vstupními parametry funkce *g_reverse* jsou data a výber módů, ketrými projdou data. Data se promíchávají na základě masky a zvoleného módu.

```
uint32 g_reverse(const uint32 data, const shift_t mode)
{
  uint32 x;
  x = op1;
  if (mode & 1)
    x = ((x & 0x55555555u) << 1) | ((x & 0xaaaaaaaau)
    >> 1);
  if (mode & 2)
    x = ((x & 0x33333333u) << 2) | ((x & 0xccccccccu)
    >> 2);
  if (mode & 4)
    x = ((x & 0x0f0f0f0fu) << 4) | ((x & 0xf0f0f0f0u)
    >> 4);
  if (mode & 8)
    x = ((x & 0x00ff00ffu) << 8) | ((x & 0xff00ff00u)
    >> 8);
  if (mode & 16)
    x = ((x & 0x0000ffffu) << 16) | ((x & 0xffff0000u)
    >> 16);
  return x;
}
```

Algoritmus 5.30: Příklad instrukce GREVI [3]

Instrukce **BEXT** a **BDEP** implementují extrakci a vložení bitů viz příklad tab.5.1 BEXT shromáždí bity na nejméně významnou pozici do cílového registru.

Bity jsou vybrány pomocí extrakční masky ze zdrojového registru. BDEP zapisuje od nejméně významného bitu na pozice pomocí depozitové masky ze zdrojového registru. Implementaci pro IA model můžeme vidět na algoritmu 5.31. Funkce *bit_ext_dep* na základě operačního kódu provede extrakci nebo vložení bitů.

Tab. 5.1: Princip instrukcí BEXT a BDEP

Vstup	**Výběr bitů**	**Výsledek BEXT**	**Výsledek BDEP**
aaaattttthhhh	111100000000	00000000aaaa	hhhh00000000
aaaattttthhhh	000111100000	000att000000	000hhhh000000
aaaattttthhhh	000000001111	00000000hhhh	00000000hhhh

```
uint32 bit_ext_dep(const uint32 opc, const uint32 data,
    const uint32 mask)
{
 uint8 i, j;
 uint32 result = 0;
   if (opc == OPC_B_BEXT){
   for (i = 0, j = 0; i < 32; i++){
     if ((mask >> i) & 1){
       if ((data >> i) & 1)
         result |= (uint32)1 << j;
       j++;
     }
   }
 }
 if (opc == OPC_B_BDEP){
   for (i = 0, j = 0; i < 32; i++)
     if ((mask >> i) & 1){
       if ((data >> j) & 1)
         result |= (uint32)1 << i;
       j++;
     }
   }
 }
 return result;
}
```

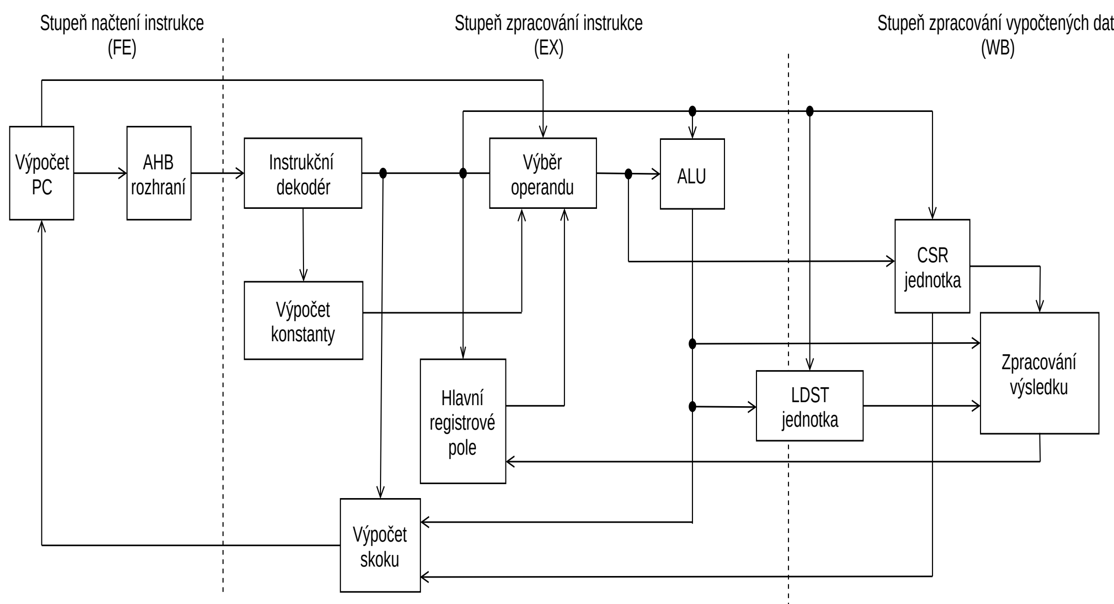
Algoritmus 5.31: Příklad instrukce GREVI [3]

6 Model na RTL úrovni

Druhým bodem k dosažení cíle diplomové práce je návrh modelu na RTL úrovni neboli CA model. CA model je stejně jako IA model popsán v jazyce CodAL. CA model byl vyvíjen pomocí nástrojů simulátoru a debugeru, které poskytuje pro vývoj Codasip studio. Na závěr je pomocí nástrojů Codasip studia vygenerován RTL popis v jazyce Verilog. Mikroarchitektura je navržena pro sadu instrukcí, které byly popsány ve výše uvedeném IA modelu.

Model je rozdělen do tří stupňů zpracování instrukce. První stupěň FE (fetch) načte jednu instrukci z paměti za takt a výpočet hodnoty programového čítače. Druhý stupěň EX (execute) dekóduje a zpracovává instrukce. Stupěň WB (write-back), která provádí zápis výsledku a zpracovává požadavky na kontrolní/stavové registry (CSR). Jednotlivé stupně se vykonávají v pořadí WB, EX, FE. Tím zajistíme předání potřebných informací od instrukce na konci fronty.

Model využívá dva hlavní eventy: reset a main. Reset inicializuje používané registry na počáteční hodnotu. Main slouží k řízení jednotlivých stupňů zpracování instrukce. Zjednodušené schéma implementovaného procesoru viz obr. 6.1. Podrobnější schéma implementovaného procesoru nalezneme v příloze A



Obr. 6.1: Zjednodušené schéma implementovaného procesoru

6.1 FE stupeň načtení

Stupeň FE slouží k vypočtení nové hodnoty programového čítače a k načtení nové instrukce z paměti programu. Počáteční hodnota programového čítače je učena atributem *BOOT START*. Atribut *BOOT START* je nastaven na hodnotu 0x1000.Ve standardním případě (není vykonávána skoková instrukce) je k aktuální hodnotě programového čítače navýšena o 4 bajty. V případě vykonávání skokové instrukce je hodnota programového čítače nastavena na hodnotu, která je vrácena z EX stupně. Hodnotu programového čítače mohou ovlivnit výjimky vyvolané v rámci zpracování kontrolních/stavových registrů. V tomto případě je hodnota programového čítače určena adresou z WB stupně. RISC-V nepodporuje nezarovnanou adresu.

Nově výpočtená hodnota programového čítače je adresou nové instrukce. Přes rozhraní se do paměti odešle adresa a požadavek na novou instrukci. Po ukončení FE stupně se vrátí z paměti nová instrukce.

6.2 EX stupeň vykonání

Stupeň EX slouží k dekódování instrukce, přípravě operandů a výpočtu pomocí ALU (arimeticko-logická jednotka). Z předešlého stupně se předá do EX stupně hodnota programového čítače a načtená instrukce z paměti. Instrukce je přivedena do dekodéru, kde je následně dekódována. Dekodér uloží do signálů informace potřebné k vykonání instrukce. Pro ALU nastaví funkci, která se bude počítat (defaultně je nastavena na funkci sčítání). Vybere se zdroj dat (registrové pole, konstanta, hodnota PC) pro operandy. Dále pak dostaneme adresu cílového registru. V případě paměťových a CSR instrukcí, dekódujeme, jestli budeme provádět zápis nebo čtení. U skokových instrukcí dekodér uloží do signálu typ skoku. V případě neznámé instrukce dekodér nastaví hodnotu signálu ilegální instrukce na logickou 1.

Před dalším zpracováním instrukce je potřeba načíst do operandů data na základě signálu z dekodéru. V případě konstanty se vybere příslušný segment z instrukce ve kterém je uložena konstanta. V případě čtení z registrového pole je potřeba ověřit, zda nedojde k RAW (read after write) hazardu (na adresu, ze které se mají číst data, se budou ukládat data z rozpracované instrukce). V případě, že hazard neexistuje, můžeme načíst hodnotu z registrového pole. Ovšem jestliže hazard existuje, je potřeba zastavit vykonávání FE a EX stupňů a počkat až se instrukce v rámci WB dokončí. Pokud je možné použít neuložený výsledek z WB stupně, není potřeba stupně FE a EX zastavovat. V případě nepodmíněného skoku se jako zdroj dat bude používat hodnota programového čítače. Poté jsou operandy předány do ALU. Pomocí signálu z dekodéru ALU vybere příslušnou funkci. V rámci aritmeticko-logické

jednotky jsou zpracovány i instrukce z rozšíření pro bitové manipulace (viz kapitola 6.4).

V rámci EX stupně je v případě paměťových operací provedena komunikace s pamětí pomocí AHB sběrnice. Do paměti se odešle adresa, ze které bude v následujícím taktu číst data nebo zapisovat data. Na konci EX stupně se uloží hodnoty do registrů pro následovné zpracování v dalších stupních.

6.3 WB zápis výsledku

V konečném stupni WB se provádí zápis dat do registrového pole, zpracovávájí se požadavky na CSR registry nebo se dokončují paměťové operace. Jestliže v předchozím cyklu v EX stupni byla odeslána přes rozhraní do paměti adresa, můžeme ve WB stupni očekávat data na výstupu rozhraní. V případě, že se jedná o výpočetní operaci, je příjmut výsledek z EX stupně a uložen na příslušnou cílovou adresu do registrového pole.

Poslední možnost, která se výkoná v rámci WB stupně je zpracování požadavku na CSR registry. V případě čtení, je z příšlušného registru přečtena jeho aktualní hodnota a následně zapsána na příslušnou cílovou adresu do registrového pole. V případě zápisu do CSR registru je hodnota zapsána do konkrétního CSR registru. Můžeme se setkat s požadavkem atomický zápis, což znamená čtení aktuální hodnoty z konkrétního CSR registru a následný zápis do stejného CSR registru.

6.4 Implementace B rozšíření

Výpočet instrukcí pro bitové manipulace se vykonává ve stupni EX v rámci aritmeticko-logické jednotky. Implementace tohoto rozšíření rozdělíme na dvě části a to implementaci elementarních instrukcí viz kapitola 6.4.1 a implementaci generických instrukcí viz kapitola 6.4.2. V IA modelu jsou generické instrukce popsány pomocí cyklů, které při implementaci v CA modelu nelze využít. Proto tyto instrukce musí být popsány pomocí sítí. Bohužel tyto sítě jsou velmi náročné na počty použitých zdrojů (větší náročnost na plochu). Proto bylo nutné pouvavažovat, jak vhodně sloučit generické instrukce tak, aby při výpočtu sdílely co nejvíce zdrojů.

6.4.1 Implementace základních bitových instrukcí

Jak již z názvu vyplýva, elementární bitové instrukce nejsou nijak složité na implementaci. Většina z nich jde popsat použitím jednoho operátoru, stejně jako v jazyce C. Nalezneme zde instrukce, které počítají pozici nul (CLZ a CTZ). Instrukce CLZ

a CTZ byly naiplementovány pomocí integrovaných a optimalizovaných funkcí jazyka CodAL (viz ukázka algoritmu 6.1).

```
case ALU_CTZ:
 s_ex_alu_res = codasip_cttz_uint32(ex_alu_op1);
 break;
case ALU_CLZ:
 s_ex_alu_res = codasip_ctlz_uint32(ex_alu_op1);
 break;
```

Algoritmus 6.1: Ukázka implementace základních bitových instrukcí CLZ a CTZ

Podobně snadno lze naimplementovat i instrukce pro rotaci bitů (ROR a ROL). U instrukcí pro rotaci bitů bylo využito operátoru pro rotaci (>>> a <<<), kterým taktéž disponuje jazyk CodAL (viz ukázka algoritmu 6.2).

```
  case ALU_ROR:
   s_ex_alu_res = ex_alu_op1 >>> (uint5)ex_alu_op2;
   break;
  case ALU_ROL:
   s_ex_alu_res = ex_alu_op1 <<< (uint5)ex_alu_op2;
   break;
```

Algoritmus 6.2: Ukázka implementace bitových rotací ROR a ROL

Další instrukcí této skupiny je PCNT, která spočitá počet jedniček operandu. Tato instrukce je zrealizovaná pomocí sčítání tak, že je postupně sčítáno všech 32 bitů. Výsledkem je pak počet logických jedniček. Ukázka principu výpočtu je provedena na pěti bitovém operandu viz alegoritmus 6.3

```
result =
  ((uint3)((uint2)src[0..0] + src[1..1] + src[2..2]))+
  ((uint2)src[3..3] + src[4..4] + src[5..5]));
```

Algoritmus 6.3: Ukázka výpočtu PCNT pro pěti bitový operand

Pro instrukce posuvu s jedničkou (SLO a SRO) byla vytvořena inline funkce *ones_shifter_32*, která na základě funkce pro ALU jednotku vybere směr posuvu. Operand, který budeme posouvat, je nutné před posunem znegovat. Znegovaný posunutý operand je potřeba znegovat ješte jednou. Tímto způsobem zachováme původní hodnoty bitů, které po posuvu zůstanou ve výsledku a nasuneme příslušný počet jedniček viz ukázka algoritmu 6.4

```
inline uint32 ones_shifter_32(uint5 opr, uint32 op1,
   uint5 op2)
{
  uint32 shamt;

  if (opr == ALU_SLO) {
    shamt = op2 & (uint32)31;
    return (~(~op1 << shamt));
  }
  else if (opr == ALU_SRO){
    shamt = op2 & (uint32)31;
    return (~(~op1 >> shamt));
  }
}
```

Algoritmus 6.4: Ukázka inline funkce pro logické posuvy s jedničkou

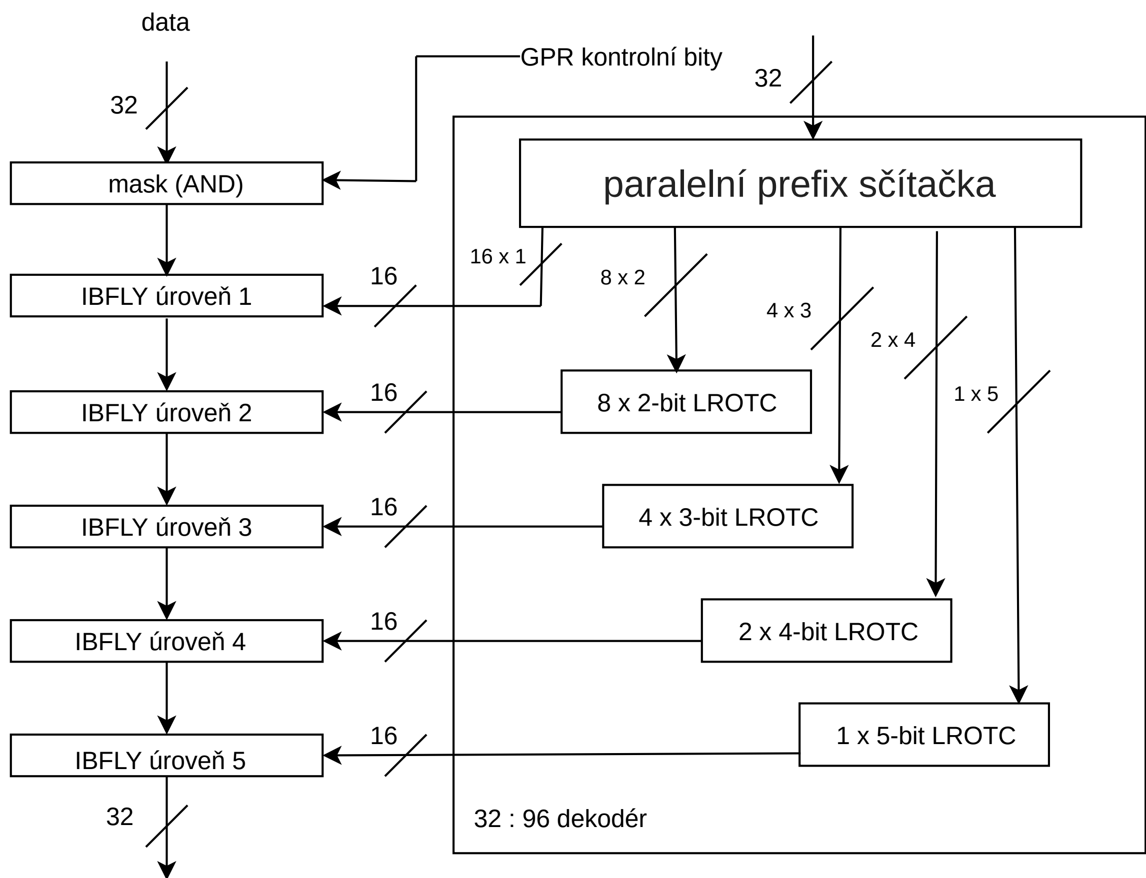
6.4.2 Generické instrukce

Druhou polovinu bitových instrukcí reprezentují generické bitové instrukce. Tyto instrukce jsou především náročné na plochu, jelikož využívají různé vícestupňové sítě. Patří mezi ně instrukce GREV, BEXT a BDEP, které pro svoje zpracování využívají tzv. motýlí síť (butterfly network). Instrukce SHFL má svou samostatnou síť, jelikož by sloučení s ostatními generickými instrukcemi nemělo výrazný efekt vzhledem prostředkům a výrazně by přidalo kódu na jeho složitosti.

Implementace BGREV, BEXT a BDEP

Blok pro zpracování těchto instrukcí se skládá ze dvou hlavních komponent dekodéru masky a motýli sítě. Úkolem dekodéru masky je vypočíst masku pro všechny uzly v motýli síti. Jako vstup je v případě instrukcí BEXT a BDEP maska ze zdrojového registru. V případě GREV je maska hodnota mínus jedna, což značí zapnutí všech uzlů. Tato 32 bitová maska je vstupem do třiceti úplných sčítaček viz obr. 6.2. Ze sčítaček potřebujeme: 16x1, 8x2, 4x3, 2x4, 1x5 bité výsledky.

Výsledky z těchto sčítaček jsou vstupem do komponent barelový rotátor, které specifickým způsobem rotují 16, 8, 4 a 2 bitovou nulu. Počet rotací vpravo je dán výsledkem z příslušné sčítačky. Před každou rotací se nejvýznamnější bit zneguje. Příklad alogoritmu 6.5 pro rotaci 8 bitové nuly.



Obr. 6.2: Blokové zapojení pro instrukce GREV, BEXT a BDEP[3]

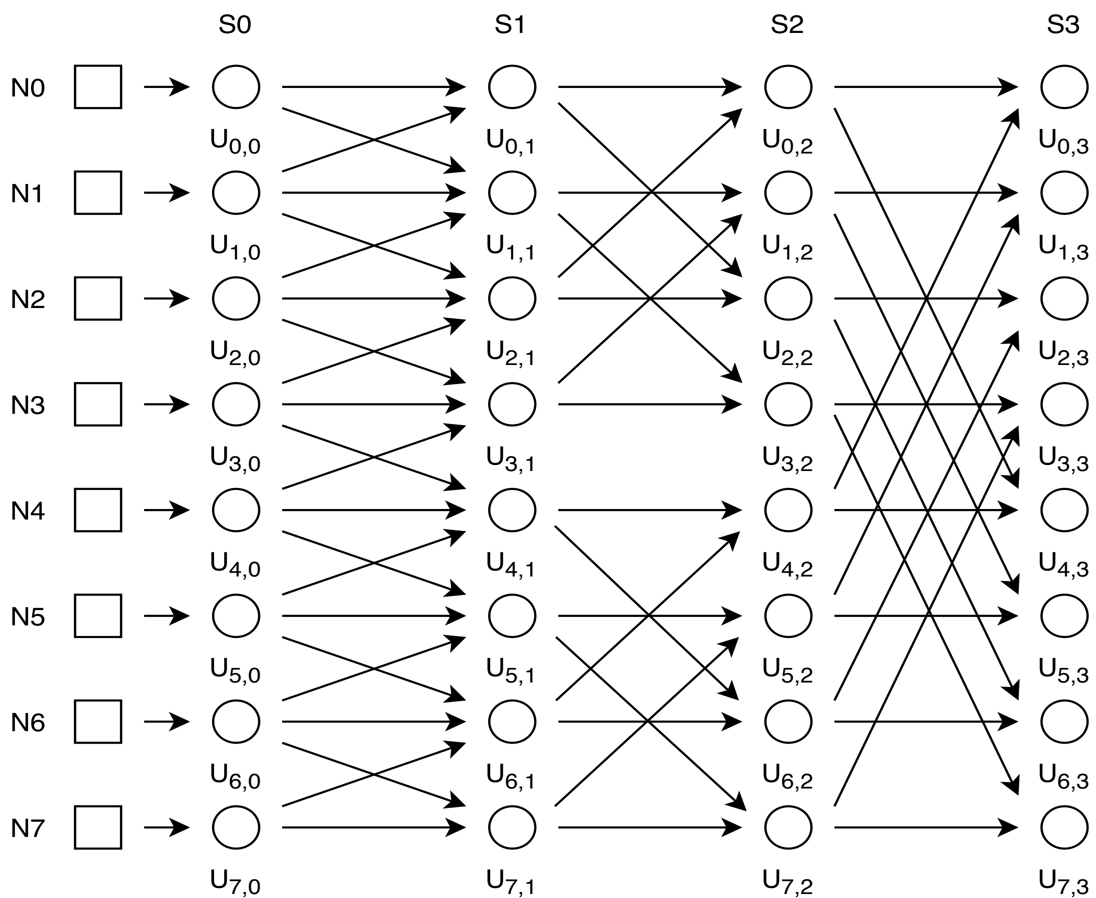
```
inline uint8 barrel_rotator_8(uint5 popcnt)
{
  uint1 stage1;
  uint3 stage2;
  uint8 stage3, stage4;
  stage1 = popcnt[0..0] ? 1 : 0;
  stage2 = popcnt[1..1] ? (stage1 :: (uint2)3) :
   ((uint2)0 :: stage1);
  stage3 = popcnt[2..2] ? ((uint1)0 :: stage2 :: (uint4)
     15) :
  ((uint5)0 :: stage2);
  stage4 = popcnt[3..3] ? ~stage3 : stage3;
  return ~(stage4);
}
```

Algoritmus 6.5: Inline funkce pro barel rotaci 8 bitové nuly

Po provedení těchto rotací nad všemi výsledky ze sčítaček dostaneme 80 bitovou masku. Tato maska společně s daty je vstupem do motýlí sítě 6.3.

Motýlí síť:
Označíme-li si prvky vstupující do sítě N $\in \mathbb{N}$, jednotlivé stupně sítě S $\in \mathbb{N}$ a jednotlivé uzly sítě $U_{Ni,Si}$. Můžeme určit počet uzlů a stupňů tak, že počet stupňů v sítí odpovídá $2^S = N$ a počet uzlů $U = N \cdot S$. Každý prvek N se musí promítnout na základě podmínky do možných dvou uzlů:

1. z uzlu $U_{Ni,Si}$ do uzlu $U_{Ni,Si+1}$,
2. z uzlu $U_{Ni,Si}$ v případě sudého prvku do uzlu $U_{Ni+i^2,Si+1}$ a případě lichého prvku do uzlu $U_{Ni-i^2,Si+1}$.



Obr. 6.3: Motýlí síť pro 8 prvků

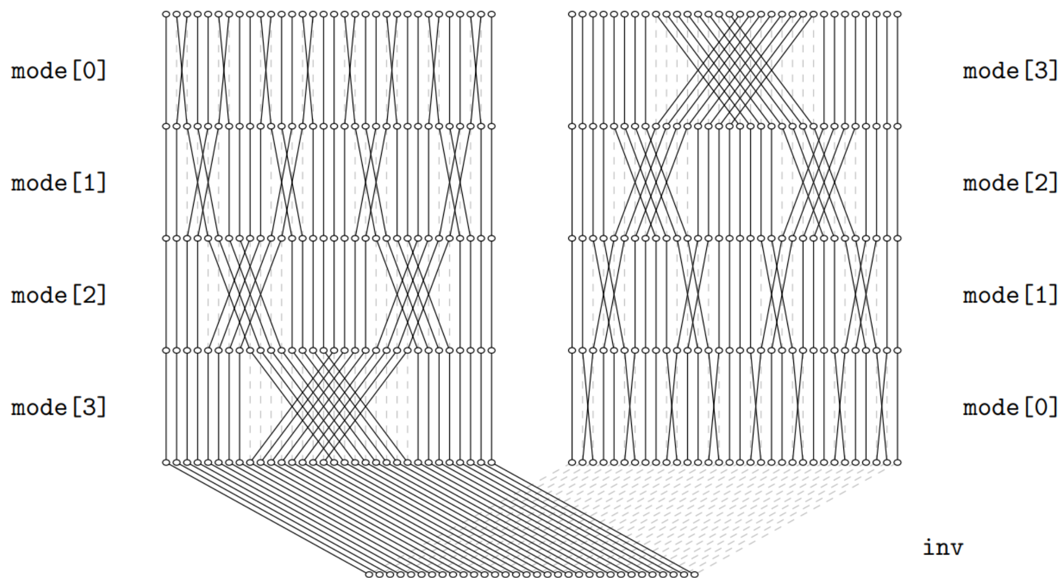
V mém případě motýli síť má 32 vstupů o 5 stupních. Instrukce BEXT využívá všechny stupně ve směru zleva doprava a ovládáme pouze směr přechodu do uzlů: v případě nuly rovný směr (bezezměny), v případě 1 směr zešikma (prohození). Instrukce GREV používá pouze vybrané stupně ve směru zleva doprava, přičemž všechny uzly ve vybraných stupních jsou nastaveny na jedna. Instrukce BDEP je

opakem instrukce BEXT. Motýli síť využívá v opačném směru než BEXT. Ovládání bitů zůstává stejné jako u BEXT instrukce.

Implementace SHFL(I)

Jak již bylo zmíněno, tato instrukce má svojí vlastní síť viz obr. 6.4.2. Na rozdíl od předchozích generických instrukcí nemusíme masku vypočítávat. Maska je pevně daná topologii sítě. U instrukce SHFL je realizovaná pomocí funkce *shfl*.

Kompletní implemetaci funkce *shfl* nalezneme v příloze C. V prvním operandu jsou data, která budou procházet sítí. Druhý operand nese informaci o módu sítě, krerým projdou data. Funkce *shfl* nejprve vybere na základě módu definované masky, které se použijí při výpočtu a v jakém pořadí se vybrané masky použijí. Poté se v každém módě zavolá funkce *hw_shfl_stage*, která umístí bity podle masky na příslušné pozice.



Obr. 6.4: Síť pro instrukci SHFL [2]

7 Verifikace a testování

Po dokončení implementace je nutné ověřit funkčnost celého návrhu. Verifikace a testováni jsou důležitou součástí při vývoji procesoru. Veškeré chyby se snažíme odhalit pravě ve verifikační fázi vývoje, jelikož výroba čipu je velmi finančně náročná a první prototypy by měly být funkční. V ideálním případě by měl verifikaci a testování provádět jiný člověk něz ten, kdo design implementoval. Jiná osoba může jinak pochopit specifikace, dojít k výsledkům odlišným způsobem a tím zvýšit pravděpodobnost odhalení chyb v návrhu.

Verifikaci můžeme rozdělit na funkční verifikaci a formální verifikaci. Formální verifikace je založena na formálních matematických metodách, které ověřují správnost systému podle formální specifikace [4]. V této práci však využijeme pouze funkční verifikaci, která je především založena na simulaci.

7.1 Testovaní pomocí sady testů

Integrované nástroje Codasip studia usňadňují otestování implementace pomocí integrované sady testů v rámci Codasip studia. Integrovaná sada testů obsahuje 703 programů a to ve 4 úrovních optimalizace překladače. Jedná se o testování na úrovni simulátorů. Po ukončení běhu každého programu se vyhodnocuje návratová hodnota programu uložená v registru. Program je ukončen a vyhodnocen pomocí funkce *halt*. Jestliže je návratová hodnota rovna 0, pak se test vyhodnotí jako úspěšný (PASS). Pokud návratová hodnota nabývá jiné hodnoty než 0, je test považován za neúspěšný (FAIL). V případě neúspěšného testu je ve výpisu viditelný chybový návratový kód. Některé testy mohou být označeny jako (SKIP). Tato varianta nastane v případě, že test nedokončí svůj běh ve stanoveném časovém limitu. Obvykle se s tou situací setkáme při testování modelu na RTL úrovni. Řešení v tomto případě je snadné. Spočívá ve zvýšení časového limitu pro běh testu.

Mimo integrované testy se k testování přidávají architekturální testy. Architekturální testy obsahují krátké sekvence assembler instrukcí, které jsou specifické pro dané instrukční rozšíření nebo popisují ojedinělou sekvenci instrukcí, u kterých je zvýšena pravděpodobnost chyby. K architekturálním testům se také přidávají testy, které jsou vytvořeny v rámci verifikace řízené pokrytím. V rámci implementace bitového instrukčního rozšíření bylo nutno dodělat architekturální test, který nově přidané instrukce otestuje. Test je postaven na principu, který můžeme vidět na algoritmu 5.17.

Testování probíhalo následujícím způsobem. Nejprve byl otestován instrukční model a případné chyby byly odstraněny. Po úspěšném otestování instrukčního modelu pokračoval vývoj a testovaní CA modelu.

Dalším nástrojem Codasip studia, který usnadňuje testování, je nástroj *consitency checker*. Tento nástroj pomocí porovnání IA a CA modelu ověří, zda je chování obou modelů shodné. Nejprve se přeloží testovací program, poté se provedou simulace obou modelů a porovnají se jednotlivé zápisy vybraných CSR registrů a registrového pole. Jestliže se hodnoty v kontrolovaných registech liší, běh nástroje Consitency Checker proběhne s chybou. Ve výpisu se dozvíme informace, ve kterém cyklu a registru byla nalezena neshoda. Tímto způsobem můžeme lépe lokalizovovat chyby nebo odhalit chyby, které se neprojeví při testování na sadě testů (např. nechtěný zdvojený zápis do registru).

Dále pak nástroj *consitency checker* využíváme při testování programů vygenerovaných pomocí nástroje *randomgen*. Nástroj *randomgen* generuje náhodně instrukce, které jsou implementovány v instrukčním modelu. Jelikož náhodně vygenerovaný program může mít libovolnou návratovou hodnotu, nelze ho testovat jako napříkald architekturální testy. K testovaní implementovaného modelu bylo vygenerováno tisíc náhodných programů o třista instrukcích. Výsledky testování viz tabulka 8.1.

Tab. 7.1: Výsledky testování

model	**typ testu**	**počet testovaných položek**	**výsledek simulace**	**výsledek verifikace**
IA	architekturální	18	OK	-
	testovací sada	703	OK	-
	random programy	1000	OK	-
CA	architekturální	18	OK	OK
	testovací sada	703	OK	OK
	random programy	1000	OK	OK

7.1.1 Architekturální testy pro bitové manipulace

Kompletní testy nalezneme v přiloženém souboru v adresáři *testing_B_instr* (adresářová struktura viz příloha D). Architekturálních testů pro bitové manipulace je celkem 5.
Test *tests_count_check.c* testuje instrukce CLZ CTZ a PCNT.
Test *tests_rot_check.c* testuje instrukce pro rotace a bitové posuvy.
Test *tests_shfl_grev_check.c.* testuje instrukce SHFL a GREV.
Test *tests_bext_bdep_check.c* testuje instrukce SHFL a GREV.
V rámci všech zmíněných testů je pro každou instrukci popsána funkce v jazyce C.

Výsledky jednotlivých funkcí v testech jsou srovnány s výsledkem testovaného modelu a následně vyhodnocena jejich správnost. Poslední test *B_extension_cover_32.S* obsahuje jednotlivé instrukce, které byly přidány na základě chybějícího pokrytí.

7.2 Funkční verifikace

Přestože je funkční verifikace založena na simulaci a není tak účinná v odhalení chyb jako formalní verifikace, je v praxi nejčastěji používaná. K verifikaci nejprve potřebujeme vytvořit verifikační prostředí, které nám bude sloužit k simulaci verifikovaného obvodu tzv. DUT (Design Under Test). DUT musí být popsán v HDL jazyku (Hardware Description Language).

Při verifikaci implementovaného procesoru byl použit simulátor HDL jazyků Questa od firmy Mentor. Verifikační prostředí bylo vygenerováno automaticky za pomocí Codasip studia podle metodologie UVM (Universal Verification Methodology). Během verifikace je kontrolován DUT vůči referenčnímu modelu (tzv. golden model). V našem případě jako referenční model slouží IA model. Na rozdíl od běžného testování se verifikace neřídí návratovým kódem programu, ale srovnává zapsané hodnoty registrů a paměti po dokončení simulace programu.

Jednou z technik, jak zvýšit účinnost verifikace, je použít metriky pokrytí simulovaného modelu tzv. verifikace řízená pokrytím (angl. coverage). Tato technika nám říká, které logické výrazy, stavy podmínek, stavy stavových automatů atd. byly v rámci simulace DUT vykonány. Pro zjištění konečného pokrytí jsou sloučena všechna měřená pokrytí simulovaných programů. V ideálním případě by mělo pokrytí dosahovat 100%. Ideální výsledek pokrytí nám ovšem nezaručuje 100% správnost návrhu.

Ke zjištění pokrytí u implementovaného návrhu byly použity stejné testovací programy jako u testování a tisíc programů vygenerovaných pomocí nástroje *randomgen*. Výsledky pokrytí viz tabulka 7.2. Jelikož bylo nedostačující pokrytí v rámci bitových instrukcí, bylo nutné doplnit architekrurální test pro bitové manipulace o chybějící kombinace a provést měření znova.

Tab. 7.2: Výsledky funkční verifikace (pokrytí kódu)

měření	pokrytí kódu
1.	82,12%
2.	89,48%

Výsledné změřené pokrytí nedosahovalo idealní hodnoty. Pro zvýšení pokrytí by muselo být zahrnuto testování s přerušením.

8 Dosažené výsledky

8.1 Parametry navrženého procesoru

Měření parametrů výsledného procesoru probíhalo pomocí softwaru Genus Synthesis Solution od firmy Cadence s knihovnou GLOBALFOUNDRIES 22 FDX® 22nm FD-SOI. Knihovna 22FDX je navržena pro 22 nm tranzistorovou technologii FD-SOI (Fully-Depleted Silicon-On Insulator) [12]. Knihovna je určena pro nízkopříkonové aplikace. V rámci knihovny lze zvolit několik technologických variant prahového napětí. První technologická varianta *lvt*, je zaměřena na vysoký výkon. Technologická varianta *rvt*, představuje kompromis mezi výkonem a spotřebou. Třetí meřená technologická varianta je kombinace *rvt* a *lvt*.

Při syntéze byly zvoleny následující parametry: teplota 125 °C a -40 °C, napětí VDD 0,72 V a proces SSG (nejhorší parametry tranzistorů při výrobě). Změřené parametry procesoru nalezneme v tabulce 8.1. Výsledná plocha po syntéze (100% využití plochy) dosahovala hodnot v rozmezí 3760 μm^2 až 4211 μm^2.

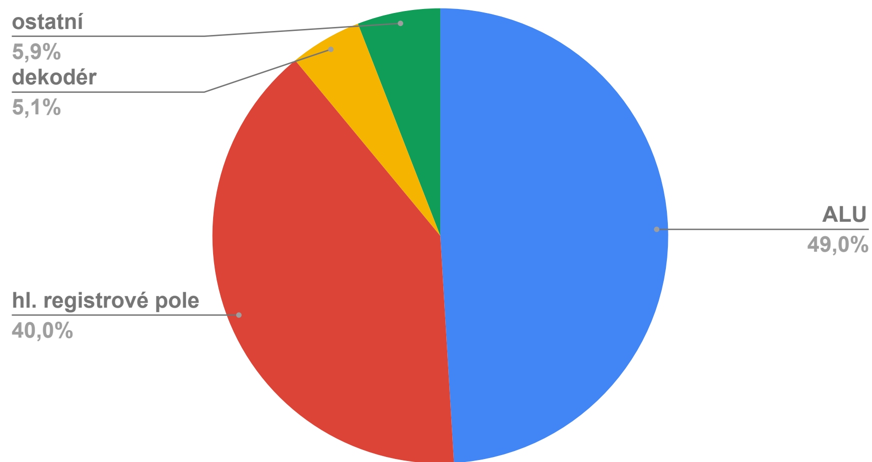
Tab. 8.1: Parametry navrženého procesoru

teplota $[°C]$	**technologie**	**frekvence** $[MHz]$	**celková plocha** $[\mu m^2]$
125	rvt a lvt	900	3760.041
	rvt	900	3914.061
	lvt	900	4021,107
-40	rvt a lvt	900	3828.598
	rvt	900	3815.552
	lvt	900	4211,548

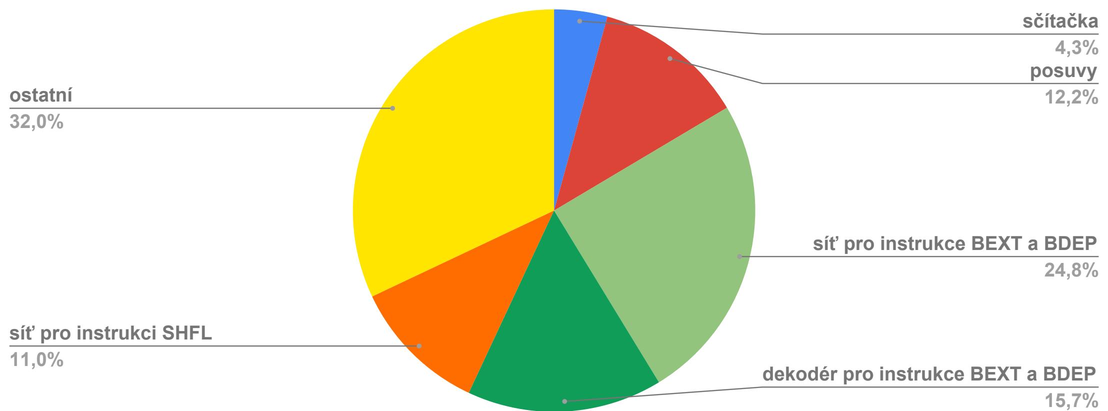
Jestliže se podíváme podrobněji na plochu navrženého procesoru, zjistíme, že největší komponenty jsou ALU, hlavní registrové pole a instrukční dekodér. Procentuální zastoupení plochy můžeme vidět na obr 8.1.

Jak již vyplývá z obr. 8.1 ALU je největší komponentou v rámci návrhu. Zabírá 49% z celkové plochy navrženého procesoru. Důvodem je, že ALU zpracovává věskeré výpočetní operace včetně výpočtů bitových instrukcí. Generické bitové instrukce zabírají 51,5% celkové plochy ALU, což je jedna čtvrtina celkové plochy. Procentuální zastoupení plochy v rámci ALU můžeme vidět na obr 8.2.

Maximální frekvenci nám určuje kritická cesta. V tomto návrhu kritická cesta vede z dekodéru instrukcí přes funkci pro výpočet generických instrukcí BEXT



Obr. 8.1: Procentuální zastoupení plochy největších komponet v rámci návrhu

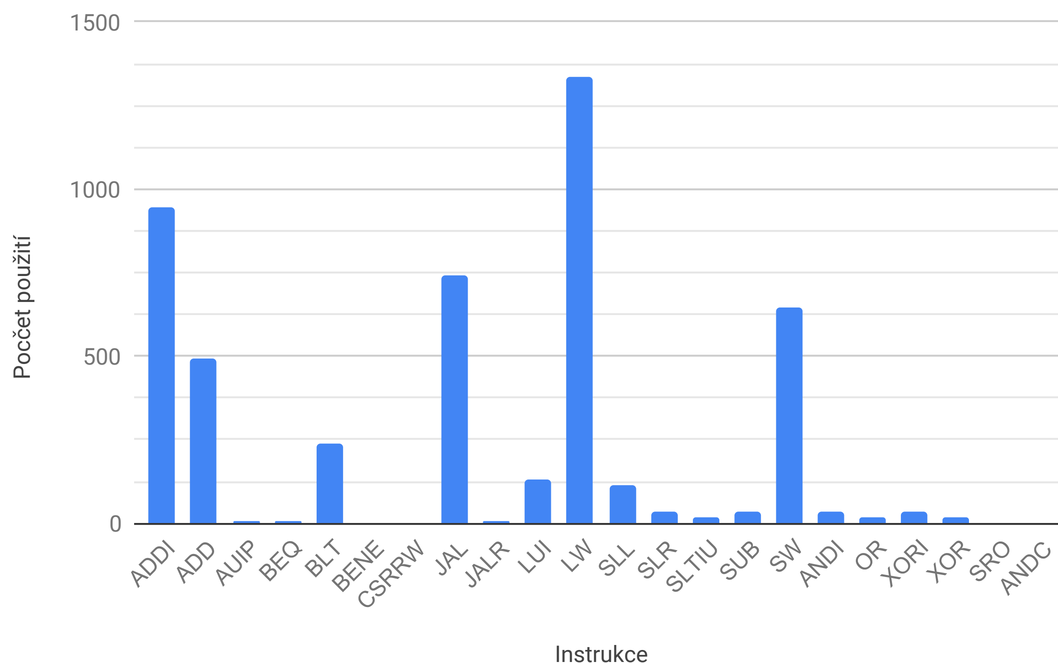


Obr. 8.2: Procentuální zastoupení plochy v rámci ALU

a BDEP až na výstup z ALU. Tato cesta by se dala zkrátit oddělením dekodéru od ALU. Tento krok by vedl k vytvoření nové fáze pro dekódování instrukce. U složitějších návhů by bylo vhodné rozdělit výpocet generických instrukcí do dvou cyklů.

8.2 Četnost instrukcí

Jelikož jsou instrukce pro bitové manipulace velmi specifické, je dobré se podívat na jejich využitelnost při překladu z jazyka C. K těmto účelům byl využit jeden z nástrojů Codasip studia *profiler*. *Profiler* mám umožňuje zjistit pokrytí instrukčního dekodéru. Což v praxi znamená, že profiler zaznamenává, jaké instrukce byly dekodovány v rámci běhu programu. Tímto způsobem můžeme zjistit, jak je optimalizován překladač pro instrukční sadou pro bitové manipulace.



Obr. 8.3: Graf využití instrukcí

K měření byly použity programy, u kterých bylo pravdepodobné, že překladač využije některou z bitových instrukcí. Jak můžeme vidět na grafu viz obr 8.3, nejpoužívanějšími instrukcemi jsou instrukce pro práci s pamětí, skoková instrukce JAL a instrukce pro součet. Instrukce pro bitové manipulace se projevily velmi zřídka. Podařilo se využít instrukce ANDC a SRO. Důvodem tak nízkého využití je neoptimalizovanost překladače v rámci Codasip studia, pomocí nehož byly programy kompilovány. Sémantika instrukcí je velmi složitá, díky čemuž extraktor sémantiky neumí rozpoznat sémantiku instrukce nebo jí vůbec nepoužije.

9 Závěr

Před začátkem implementace procesoru byly důkladně nastudované specifikace o instrukční sadě (Instruction Set Manual Volume I: Unprivileged ISA [1]), architektuře RISC-V (Instruction Set Manual Volume II: Privileged Architecture [10]) a návrhu specifikace pro instrukční rozšíření o bitové manipulace (RISC-V bitmanip extension [2]).

V praktické části diplomové práce byl za pomocí jazyka CodAL a nástrojů Codasip studia naimplementován procesor se základní instrukční sadou RISC-V ve verzi s 32-bitovým adresním prostorem a rozšíření o bitové manipulace.

Implementace začala návrhem instrukčníko modelu (kapitola 5) . Instrukční model popisuje architekturu instrukční sady a nezohledňuje časování instrukcí. Tento model slouží jako základ pro instrukční dekodér a definuje nám sémantiku instrukcí. V případě verifikace slouží jako referenční model.

Po dokončení a úspěšném otestování navrženého instrukčního modelu jsem mohla přejít k návrhu modelu na RTL úrovni. Mikroarchitektura má třístupňovou zřetězenou linku a v rámci aritmeticko-logické jenotky zpracovává bitové manipulace (blokové schéma viz příloha A). Pro vývoj modelu na RTL úrovni jsem zvolila metodiku vývoje TDD (Test Driven Development). Tato metodika spočívá v tom, že na začátku je vytvořen jednoduchý, a ne zcela funkční prototyp a sada testů. Tento prototyp se testuje a upravuje, dokud se testy úspěšně nedokončí. V mém případě se jednalo o 2 základní testy. Jeden pro základní sadu instrukcí I a druhý pro instrukční sadu s bitovými manipulacemi. Po ukončení a testování implementace modelu na RTL úrovni, jsem vygenerovala pomocí nástrojů Codasip studia RTL reprezentaci v jazyce Verilog a využila ji k verifikaci kapitola 7.2 a k syntéze pomocí nástroje Genus Synthesis Solution kapitola 8.1.

Měření parametrů výsledného procesoru probíhalo pomocí nástroje Genus Synthesis Solution od firmy Cadence s knihovnou GLOBALFOUNDRIES 22FDX® 22nm FD-SOI. V rámci knihovny lze zvolit několik technologických variant prahového napětí. První technologická varianta *lvt*, je zaměřena na vysoký výkon. Technologická varianta *rvt*, představuje kompromis mezi výkonem a spotřebou. Při syntéze byly zvoleny následující parametry: teplota 125 °C a -40 °C, napětí VDD 0,72 V a proces SSG (nejhorší parametry tranzistorů při výrobě). Změřené parametry procesoru nalezneme v tabulce 8.1. Výsledná plocha po syntéze (100% využití plochy) dosahovala hodnot v rozmezí 3760 μm^2 až 4211 μm^2. Změřená kritická cesta vede z dekodéru instrukcí přes funkci pro výpočet generických instrukcí BEXT a BDEP až na výstup z ALU.

U navrženého procesoru jsem provedla měření četností instrukcí pomocí nástroje profiler z Codasip studia. Z měření vyplynulo, že instrukce pro bitové manipulace

použije překladač Codasip studia jen velni zřítka. Důvodem je složitost instrukcí, se kterou se neumí vyspořádat extraktor sémantiky. Sémantiku nedokáže rozpoznat nebo jí vůbec nepoužije. Dalším důvodem je úzce specifický okruh použití instrukcí pro bitové manipulace. Instrukce by mohly být využity u algoritmu zameřených na komprimaci či šifrování dat.

Do budoucna bude po ustálení specifikace RISC-V Bitmanip Extension [2] realizována implementace instrukčního rozšíření pro bitové manipulace aktualizovaná.

Literatura

[1] ASANOVIČ, K., WATERMAN, A., ed.: *The RISC-V Instruction Set Manual: Volume I: User-Level ISA.* [online]. CS Division, EECS Department, University of California, Berkeley, 2017. [cit. 8. 12. 2018] Dostupné z URL: https://riscv.org/specifications

[2] WOLF, C., ed.: *RISC-V XBitmanip Extension: Document Version 0.37-draft.* [online]. Symbiotic GmbH. 2018. [cit. 8. 12. 2018] Dostupné z URL: https://github.com/riscv/riscv-bitmanip

[3] HILEWITZ, Y., ZHIJIE JERRY SHI a R.B. LEE. *Comparing fast implementations of bit permutation instructions.* Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers, 2004[online]. IEEE, 2004, , 1856-1863 [cit. 11. 12. 2018]. DOI: 10.1109/ACSSC.2004.1399486. ISBN 0-7803-8622-1. Dostupné z URL: http://ieeexplore.ieee.org/document/1399486/

[4] MEENAKSHI, B.: *Formal verification [online].* 2015, , 13 [cit. 2020-05-29]. Dostupné z: Dostupné z URL: https://link.springer.com/article/10.1007/BF02871329

[5] *CodAL Language Reference Manual. Codasip, 2015.*

[6] RŮŽIČKA, R.: *přednáška VUT FIT, Vestavěné systémy, Úvod*

[7] In: ManagementMania.com. Wilmington (DE) 2011-2018: *CPU (Central Processing Unit) - procesor.* [online]., poslední aktualizace 31. 12 2015 [cit. 10. 12. 2018]. Dostupné z: https://managementmania.com/cs/cpu-central-processing-unit-procesor

[8] OLIVKA, P.: *Procesory CISC a RISC: Studijní materiál pro předmět Architektury počítačů.* [online]. Katedra informatiky FEI VŠB-TU Ostrava, 2010 [cit. 10. 12. 2018]. Dostupné z URL: http://poli.cs.vsb.cz/edu/arp/down/procrisc.pdf

[9] TIŠNOVSKÝ, P.: *Mikroprocesory s architekturou ARM* [online]., poslední aktualizace 6. 3. 2012 [cit. 8. 12. 2018]. Dostupné z URL: https://www.root.cz/clanky/mikroprocesory-s-architekturou-arm

[10] ASANOVIČ, K., WATERMAN, A., ed.: *The RISC-V Instruction Set Manual: Volume II: Privileged Architecture, Privileged Architecture Version 1.10.* [online]. CS Division, EECS Department, University of California, Berkeley, 2017.

[cit. 8. 12. 2018] Dostupné z URL: https://riscv.org/specifications/privileged-isa

[11] ARM.: *AMBA Specification: Rev 2.0 [online]. , 230* [cit. 2020-05-29]. Dostupné z: https://developer.arm.com/docs/ihi0011/a/amba-specification-rev-20

[12] GLOBAL FOUNDRIES.: *22FDX® 22nm FD-SOI Technology [online].*[cit. 2020-05-29]. Dostupné z:https://www.globalfoundries.com/technology-solutions/cmos/fdx/22fdx

Seznam symbolů, veličin a zkratek

AHB ARM High Performance Bus, vysoce výkonna sběrnice

ALU Arithmetic Logic Unit, aritmeticko-logická jednotka

ARM Acorn RISC Machine, typ procesoru

ASIP Application-Specific Instruction Set Processor, aplikačně specifické procesory

CA Cycle Accurate, (model) na RTL úrovni

CISC Complete Instruction Set Computing, procesor s kompletní instrukční sadou

CodAL Codasip Architecture Language, programovací jazyk pro popis architektúry

CPU Central Processing Unit, centrální procesorová jednotka

CSR Control and Status Register, kontrolní a stavový registr

DMA Direct Memory Access, přímy přístup do paměti

DSO Digital Storage Scope, digitálni paměť

DUT Design Under Test, verifikovananý obvod

EX Execute, stupeň linky - vykonávání instrukce

FE Fetch, stupeň linky - načítání instrukce

HDL Hardware Description Language, jazyk pro popis hardware

IBM International Business Machines, americká mezinárodní technologická společnost

I/O Input/Output, vstupno-výstupní

IA Instruction Accurate, (model) na instrukční úrovni

IR Instruction Register, instrukční registr

ISA Instruction Set Architecture, instrukční sada architektury

MSB Most Significant Bit, nejvýznamnější bit

PC Program Counter, programový čítač

RAW Read After Write, hazard „čtení po zápise“

RISC Reduced Instruction Set Computing, procesor s redukovanou instrukční sadou

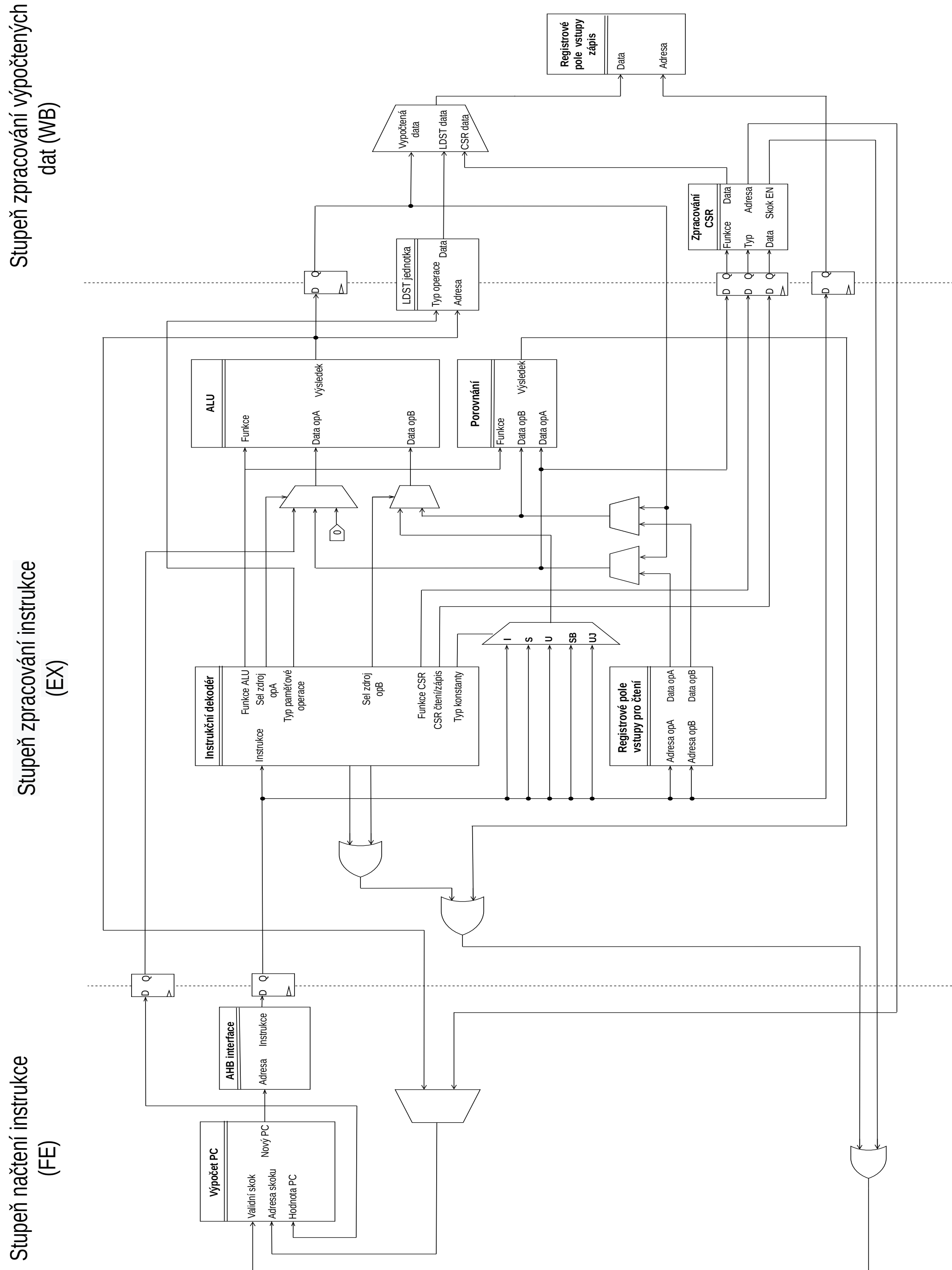
RTL Register Transfer Level, popis na úrovni přenosu signálu s využitím registrů

TDD Test Driven Development, programování řízené testy

UVM Universal Verification Methodology, univerzálni verifikační metodologie

WB Writeback, stupeň linky - zápis do registrů

A Schéma implemntováného procesoru RV32IB



B Implementace instrukcí BEXT A BDEP

```
inline void paraller_prefix_popcount (uXlen rsc)
{
  uint1 sum_bit_31, sum_bit_C_30_29, sum_bit_C_28_27,
     sum_bit_C_26_25, sum_bit_C_24_23;
  uint1 sum_bit_C_22_21, sum_bit_C_20_19, sum_bit_C_18_17,
     sum_bit_C_16_15;
  uint1 sum_bit_C_14_13, sum_bit_C_12_11, sum_bit_C_10_9,
     sum_bit_C_8_7;
  uint1 sum_bit_C_6_5 ,sum_bit_C_4_3, sum_bit_C_2_1,
     sum_bit_C_0;
  uint2 sum_0_1, sum_C_2_5, sum_C_6_9, sum_C_10_13, sum_C_14_17,
      sum_C_18_21, sum_C_22_25, sum_C_26_29;
  uint3 sum_0_3 ,sum_C_4_11, sum_C_12_19,sum_C_20_27;
  uint4 sum_C_8_23, sum_0_7;

  // PPC1 16x1bit
  // sčítaní dvojic, LSB je carry bit
  sum_bit_C_0     =  rsc[0 .. 0];
  sum_bit_C_2_1   =  (uint2)sum_bit_C_0 + rsc[2 .. 2] +
  rsc[1 .. 1];
  sum_bit_C_4_3   =  (uint2)sum_bit_C_2_1 + rsc[4 .. 4] +
  rsc[3 .. 3];
  sum_bit_C_6_5   =  (uint2)sum_bit_C_4_3 + rsc[6 .. 6] +
  rsc[5 .. 5];
  sum_bit_C_8_7   =  (uint2)sum_bit_C_6_5 + rsc[8 .. 8] +
  rsc[7 .. 7];
  sum_bit_C_10_9 =  (uint2)sum_bit_C_8_7 + rsc[10..10] +
  rsc[9 .. 9];
  sum_bit_C_12_11 = (uint2)sum_bit_C_10_9 + rsc[12..12] +
  rsc[11..11];
  sum_bit_C_14_13 = (uint2)sum_bit_C_12_11 + rsc[14..14] +
  rsc[13..13];
  sum_bit_C_16_15 = (uint2)sum_bit_C_14_13 + rsc[16..16] +
  rsc[15..15];
  sum_bit_C_18_17 = (uint2)sum_bit_C_16_15 + rsc[18..18] +
  rsc[17..17];
  sum_bit_C_20_19 = (uint2)sum_bit_C_18_17 + rsc[20..20] +
  rsc[19..19];
  sum_bit_C_22_21 = (uint2)sum_bit_C_20_19 + rsc[22..22] +
```

```
rsc[21..21];
sum_bit_C_24_23 = (uint2)sum_bit_C_22_21 + rsc[24..24] +
rsc[23..23];
sum_bit_C_26_25 = (uint2)sum_bit_C_24_23 + rsc[26..26] +
rsc[25..25];
sum_bit_C_28_27 = (uint2)sum_bit_C_26_25 + rsc[28..28] +
rsc[27..27];
sum_bit_C_30_29 = (uint2)sum_bit_C_28_27 + rsc[30..30] +
rsc[29..29];

s_ppc1 =  sum_bit_C_30_29 :: sum_bit_C_28_27 ::
    sum_bit_C_26_25 :: sum_bit_C_24_23 :: sum_bit_C_22_21 ::
    sum_bit_C_20_19 :: sum_bit_C_18_17 :: sum_bit_C_16_15 ::
    sum_bit_C_14_13 :: sum_bit_C_12_11 :: sum_bit_C_10_9 ::
    sum_bit_C_8_7 :: sum_bit_C_6_5 :: sum_bit_C_4_3  ::
    sum_bit_C_2_1 :: sum_bit_C_0;

// PPC2 8x2bit
sum_0_1 = (rsc[1 .. 1] & rsc[0 .. 0]) :: (uint2)rsc[0 .. 0] +
    rsc[1 .. 1];
sum_C_2_5 = (uint2)(sum_0_1 + rsc[2 .. 2]) +
   (uint2)rsc[3 .. 3] + (uint2)rsc[4 .. 4] +
   (uint2)rsc[5 .. 5];
sum_C_6_9 = (uint2)(sum_C_2_5 + rsc[6 .. 6]) +
   (uint2)rsc[7 .. 7] + (uint2)rsc[8 .. 8] +
   (uint2)rsc[9 .. 9];
sum_C_10_13 = (uint2)(sum_C_6_9  + rsc[10..10]) +
   (uint2)rsc[11..11] + (uint2)rsc[12..12] +
   (uint2)rsc[13..13];
sum_C_14_17 = (uint2)(sum_C_10_13 + rsc[14..14]) +
   (uint2)rsc[15..15] + (uint2)rsc[16..16] +
   (uint2)rsc[17..17];
sum_C_18_21 = (uint2)(sum_C_14_17 + rsc[18..18]) +
   (uint2)rsc[19..19] + (uint2)rsc[20..20] +
   (uint2)rsc[21..21];
sum_C_22_25 = (uint2)(sum_C_18_21 + rsc[22..22]) +
   (uint2)rsc[23..23] + (uint2)rsc[24..24] +
   (uint2)rsc[25..25];
sum_C_26_29 = (uint2)(sum_C_22_25 + rsc[26..26]) +
   (uint2)rsc[27..27] + (uint2)rsc[28..28] +
   (uint2)rsc[29..29];
```

```
s_ppc2 = sum_C_26_29 :: sum_C_22_25 :: sum_C_18_21 ::
    sum_C_14_17 :: sum_C_10_13 ::sum_C_6_9 :: sum_C_2_5 ::
    sum_0_1;

// PPC3 4X3bit
sum_0_3 = (rsc[0..0] & rsc[1..1] & rsc[2..2] & rsc[3..3]) ::
    ((uint2)rsc[0..0] + rsc[1..1] + rsc[2..2] + rsc[3..3]);

sum_C_4_11 = (uint3)(sum_0_3 + rsc[4..4]) +
   (uint3)((uint2)rsc[5..5] + (uint2)rsc[6..6]) +
   (uint3)((uint2)rsc[7..7] + (uint2)rsc[8..8]) +
   (uint3)((uint2)rsc[9..9] + (uint2)rsc[10..10]) +
   (uint3)rsc[11..11];

sum_C_12_19 = (uint3)(sum_C_4_11 +rsc[12..12]) +
   (uint3)((uint2)rsc[13..13] + (uint2)rsc[14..14]) +
   (uint3)((uint2)rsc[15..15] + (uint2)rsc[16..16]) +
   (uint3)((uint2)rsc[17..17] + (uint2)rsc[18..18]) +
   (uint3)rsc[19..19];

sum_C_20_27 = (uint3)(sum_C_12_19 +rsc[20..20]) +
   (uint3)((uint2)rsc[21..21] + (uint3)rsc[22..22]) +
   (uint3)((uint3)rsc[23..23] + (uint3)rsc[24..24]) +
   (uint3)((uint3)rsc[25..25] + (uint3)rsc[26..26]) +
   (uint3)rsc[27..27];

s_ppc3 = sum_C_20_27 :: sum_C_12_19 :: sum_C_4_11 :: sum_0_3;

// PPC4 2X4bit
sum_0_7 = (rsc[0..0]& rsc[1..1] & rsc[2..2]& rsc[3..3] & rsc
    [4..4] & rsc[5..5] & rsc[6..6] & rsc[7..7]) :: ((uint3)rsc
    [0..0]+ rsc[1..1] + rsc[2..2] + rsc[3..3]+ rsc[4..4]+ rsc
    [5..5] + rsc[6..6] + rsc[7..7]);

sum_C_8_23 = (uint4)(sum_0_7 + rsc[8 ..8]) +
   (uint4)rsc[9 .. 9] + rsc[10..10] +
   (uint4)rsc[11..11] + rsc[12..12] + rsc[13..13] +
   (uint4)rsc[14..14] + rsc[15..15] + rsc[16..16] +
   (uint4)rsc[17..17] + rsc[18..18] + rsc[19..19] +
   (uint4)rsc[20..20] + rsc[21..21] + rsc[22..22] +
```

```
    (uint4)rsc[23..23];

  s_ppc4 =  sum_C_8_23 :: sum_0_7;

  // PPC5 1x5bit
  s_ppc5 = (rsc[0 .. 0] & rsc[1 .. 1] & rsc[2 .. 2] &
    rsc[3 .. 3] & rsc[4 .. 4] & rsc[5 .. 5] & rsc[6 .. 6] &
    rsc[7 .. 7] & rsc[8 .. 8] & rsc[9 .. 9] & rsc[10..10] &
    rsc[11..11] & rsc[12..12] & rsc[13..13] & rsc[14..14] &
    rsc[15..15]) :: ((uint4)rsc[0 .. 0] + rsc[1 .. 1] +
    rsc[2 .. 2] + rsc[3 .. 3] + rsc[4 .. 4] + rsc[5 .. 5] +
    rsc[6 .. 6] + rsc[7 .. 7] + rsc[8 .. 8] + rsc[9 .. 9] +
    rsc[10..10] + rsc[11..11] + rsc[12..12] + rsc[13..13] +
    rsc[14..14] + rsc[15..15]);
}

// funkce pro rotace
inline uint16 barrel_rotator_16(uint5 popcnt)
{
  uint1 stage1;
  uint3 stage2;
  uint8 stage3;
  uint16 stage4, stage5;

  stage1 = popcnt[0..0] ? 1 : 0;
  stage2 = popcnt[1..1] ? (stage1 :: (uint2)3) :
    ((uint2)0 :: stage1);
  stage3 = popcnt[2..2] ? ((uint1)0 :: stage2 :: (uint4)15) :
    ((uint5)0 :: stage2);
  stage4 = popcnt[3..3] ? (stage3 :: (uint8)255) :
    ((uint8)0 :: stage3);
  stage5 = popcnt[4..4] ? ~stage4 : stage4;
  return ~stage5;
}
```

```
inline uint8 barrel_rotator_8(uint5 popcnt)
{
  uint1 stage1;
  uint3 stage2;
  uint8 stage3, stage4;

  stage1 = popcnt[0..0] ? 1 : 0;
  stage2 = popcnt[1..1] ? (stage1 :: (uint2)3) :
    ((uint2)0 :: stage1);
  stage3 = popcnt[2..2] ? ((uint1)0 :: stage2 :: (uint4)15) :
    ((uint5)0 :: stage2);
  stage4 = popcnt[3..3] ? ~stage3 : stage3;
  return ~(stage4);
}

inline uint4 barrel_rotator_4(uint4 popcnt)
{
  uint1 stage1;
  uint4 stage2, stage3;

  stage1 = popcnt[0..0] ? 1 : 0;
  stage2 = popcnt[1..1] ? ((uint1)0 :: stage1 :: (uint2)3) :
    ((uint3)0 :: stage1);
  stage3 = popcnt[2..2] ? ~stage2 : stage2;
  return ~(stage3);
}

inline uint2 barrel_rotator_2(uint2 popcnt)
{
  uint2 stage1,stage2;

  stage1 = popcnt[0..0] ? 0b01 : 0b00;
  stage2 = popcnt[1..1] ? ~stage1 : stage1;
  return ~stage2;
}
```

```
inline bfly_mask_t decoder(uXlen rsc2)
{
  bfly_mask_t mask;

  paraller_prefix_popcount (rsc2);
  //32: 1x16bit rotace pro ppc5 1x5bit
  s_barrel_rot16 =
    barrel_rotator_16(s_ppc5[4..0]);

  //32: 2x8bit rotace pro ppc4 2x4bit
  s_barrel_rot8 =
    barrel_rotator_8(s_ppc4[7 .. 4]) ::
    barrel_rotator_8(s_ppc4[3 .. 0]);

  //32: 4x4bit rotace pro ppc3 4x3bit
  s_barrel_rot4 =
    barrel_rotator_4(s_ppc3[11..9]) ::
    barrel_rotator_4(s_ppc3[8 ..6]) ::
    barrel_rotator_4(s_ppc3[5 ..3]) ::
    barrel_rotator_4(s_ppc3[2 ..0]);

  //32: 8x2bit rotace pro ppc2 8x2bit
  s_barrel_rot2=
    barrel_rotator_2(s_ppc2[15..14]) ::
    barrel_rotator_2(s_ppc2[13..12]) ::
    barrel_rotator_2(s_ppc2[11..10]) ::
    barrel_rotator_2(s_ppc2[9 .. 8]) ::
    barrel_rotator_2(s_ppc2[7 .. 6]) ::
    barrel_rotator_2(s_ppc2[5 .. 4]) ::
    barrel_rotator_2(s_ppc2[3 .. 2]) ::
    barrel_rotator_2(s_ppc2[1 .. 0]);

  mask = s_barrel_rot16 :: s_barrel_rot8 :: s_barrel_rot4 ::
     s_barrel_rot2 :: ~s_ppc1;
  return (mask);
}
```

```
inline uint32 butterfly_network(bool mode, uint32 rsc1, uint30
   shamt , bfly_mask_t mask)
{

uint16 mask_L0, mask_R0, mask_All0, mask_L1, mask_R1, mask_All1,
    mask_L2, mask_R2, mask_All2, mask_L3, mask_R3, mask_All3;
uint16 mask_L4, mask_R4, mask_All4, stage1, stage2, stage3,
   stage4, stage0;
uint5 N0, N1, N2, N3,N4;

//výběr kontrolních bitov pro stage0
mask_L0   = ((mode & 1) && (shamt & 1)) ?
  INTERLEAVE_2X16_32_1 (mask[15.. 0], (uint16)0):
  (!(mode & 1) && (shamt & 16)) ?
  INTERLEAVE_2X16_32_16(mask[79..64], (uint16)0):
  MASK_GREV_LR_1;
mask_R0   = ((mode & 1) && (shamt & 1)) ?
  INTERLEAVE_2X16_32_1 ((uint16)0, mask[15..0]):
  (!(mode & 1) && (shamt & 16)) ?
  INTERLEAVE_2X16_32_16((uint16)0, mask[79..64]):
  MASK_GREV_LR_1;
mask_All0 = ((mode & 1) && (shamt & 1)) ?
  INTERLEAVE_2X16_32_1 (mask[15..0], mask[15..0]) :
  (!(mode & 1) && (shamt & 16)) ?
  INTERLEAVE_2X16_32_16(mask[79..64], mask[79..64]) :
  MASK_GREV_LR_1;
//výběr kontrolních bitov pro stage1
mask_L1 =  ((mode & 1) && (shamt & 2)) ?
  INTERLEAVE_2X16_32_2(mask[31..16], (uint16)0):
  (!(mode & 1) && (shamt & 8)) ?
  INTERLEAVE_2X16_32_8(mask[63..48], (uint16)0):
  MASK_GREV_LR_1;
mask_R1 =  ((mode & 1) && (shamt & 2)) ?
  INTERLEAVE_2X16_32_2((uint16)0, mask[31..16]):
  (!(mode & 1) && (shamt & 8)) ?
  INTERLEAVE_2X16_32_8((uint16)0, mask[63..48]):
  MASK_GREV_LR_1;
mask_All1= ((mode & 1) && (shamt & 2)) ?
  INTERLEAVE_2X16_32_2(mask[31..16], mask[31..16]):
  (!(mode & 1) && (shamt & 8)) ?
  INTERLEAVE_2X16_32_8(mask[63..48], mask[63..48]):
```

```
  MASK_GREV_LR_1;
//výběr kontrolních bitov pro stage2
mask_L2 =  (shamt & 4) ? INTERLEAVE_2X16_32_4(mask[47..32],
  (uint16)0) : MASK_GREV_LR_1;
mask_R2 =  (shamt & 4) ? INTERLEAVE_2X16_32_4((uint16)0,
  mask[47..32]) : MASK_GREV_LR_1;
mask_All2= (shamt & 4) ? INTERLEAVE_2X16_32_4(mask[47..32],
  mask[47..32]) : MASK_GREV_LR_1;
//výběr kontrolních bitov pro stage3
mask_L3 =  ((mode & 1) && (shamt & 8)) ?
  INTERLEAVE_2X16_32_8(mask[63..48], (uint16)0):
  (!(mode & 1) && (shamt & 2)) ?
  INTERLEAVE_2X16_32_2(mask[31..16],
  (uint16)0): MASK_GREV_LR_1;
mask_R3 =  ((mode & 1) && (shamt & 8)) ?
  INTERLEAVE_2X16_32_8((uint16)0, mask[63..48]):
  (!(mode & 1) && (shamt & 2)) ?
  INTERLEAVE_2X16_32_2((uint16)0, mask[31..16]):
  MASK_GREV_LR_1;
mask_All3= ((mode & 1) && (shamt & 8)) ?
  INTERLEAVE_2X16_32_8(mask[63..48], mask[63..48]):
  (!(mode & 1) && (shamt & 2)) ?
  INTERLEAVE_2X16_32_2(mask[31..16],
  mask[31..16]): MASK_GREV_LR_1;
//výběr kontrolních bitov pro stage4
mask_L4 =  ((mode & 1) && (shamt & 16)) ?
  INTERLEAVE_2X16_32_16(mask[79..64], (uint16)0):
  (!(mode & 1) && (shamt & 1)) ?
   INTERLEAVE_2X16_32_1 (mask[15..0], (uint16)0):
   MASK_GREV_LR_1;
mask_R4 =  ((mode & 1) && (shamt & 16)) ?
   INTERLEAVE_2X16_32_16((uint16)0, mask[79..64]):
  (!(mode & 1) && (shamt & 1))  ?
  INTERLEAVE_2X16_32_1 ((uint16)0, mask[15..0])  :
  MASK_GREV_LR_1;
mask_All4= ((mode & 1) && (shamt & 16)) ?
  INTERLEAVE_2X16_32_16(mask[79..64], mask[79..64]):
  (!(mode & 1) && (shamt & 1))  ?
  INTERLEAVE_2X16_32_1 (mask[15..0],
  mask[15..0]): MASK_GREV_LR_1;
```

```
  // výběr hodnoty posunu pro stage 0
  N0 = ((mode & 1) && (shamt & 1))  ? (uint5)1 :
     (!(mode & 1) && (shamt & 16)) ? (uint5)16 : (uint5)0;
  // výběr hodnoty posunu pro stage 1
  N1 = ((mode & 1) && (shamt & 2))  ? (uint5)2 :
     (!(mode & 1) && (shamt & 8))  ? (uint5)8 : (uint5)0;
  // výběr hodnoty posunu pro stage 2
  N2 = ((shamt & 4))? (uint5)4 : (uint5)0;
  // výběr hodnoty posunu pro stage 3
  N3 = ((mode & 1) && (shamt & 8)) ? (uint5)8 :
     (!(mode & 1) && (shamt & 2)) ? (uint5)2 : (uint5)0;
  // výběr hodnoty posunu pro stage 4
  N4 = ((mode & 1) && (shamt & 16)) ? (uint5)16 :
     (!(mode & 1) && (shamt & 1))  ? (uint5)1 : (uint5)0;

  stage0 = ((rsc1   & mask_L0) >> N0) | ((rsc1   & mask_R0) <<
      N0) |
     (rsc1   & ~mask_All0);
  stage1 = ((stage0 & mask_L1) >> N1) | ((stage0 & mask_R1) <<
      N1) |
     (stage0 & ~mask_All1);
  stage2 = ((stage1 & mask_L2) >> N2) | ((stage1 & mask_R2) <<
      N2) |
     (stage1 & ~mask_All2);
  stage3 = ((stage2 & mask_L3) >> N3) | ((stage2 & mask_R3) <<
      N3) |
     (stage2 & ~mask_All3);
  stage4 = ((stage3 & mask_L3) >> N4) | ((stage3 & mask_R4) <<
      N4) |
     (stage3 & ~mask_All4);
  return stage4;
}
```

C Funkce pro výpočet instrukcí SHFL

```
inline uint32 shfl(uint32 rsc1, uint32 rsc2)
{
  uint32 maskL0, maskR0, maskL1, maskR1, maskL2, maskR2, maskL3,
         maskR3, stage1, stage2, stage3, stage4;
  uint4 N0, N1, N2, N3;
  uint5 mode;
  mode = rsc2 & 31;

  // výběr masky pro mód 0 shuffle/unshuffle
  maskL0 = (mode & (uint5)1  && mode & (uint5)2) ? MASK_ZIP2_L :
       (!(mode & (uint5)1) && mode & (uint5)16) ? MASK_ZIP16_L :
       (uint32)0;
  maskR0 = (mode & (uint5)1  && mode & (uint5)2) ? MASK_ZIP2_R :
       (!(mode & (uint5)1) && mode & (uint5)16) ? MASK_ZIP16_R :
       (uint32)0;
  // výběr masky pro mód 1 shuffle/unshuffle
  maskL1 = (mode & (uint5)1  && mode & (uint5)4) ? MASK_ZIP4_L :
       (!(mode & (uint5)1) && mode & (uint5)8) ? MASK_ZIP8_L   :
       (uint32)0;
  maskR1 = (mode & (uint5)1  && mode & (uint5)4) ? MASK_ZIP4_R :
       (!(mode & (uint5)1) && mode & (uint5)8) ? MASK_ZIP8_R   :
       (uint32)0;
  // výběr masky pro mód 2 shuffle/unshuffle
  maskL2 = (mode & (uint5)1  && mode & (uint5)8) ? MASK_ZIP8_L :
       (!(mode & (uint5)1) && mode & (uint5)4) ? MASK_ZIP4_L   :
       (uint32)0;
  maskR2 = (mode & (uint5)1  && mode & (uint5)8) ? MASK_ZIP8_R :
       (!(mode & (uint5)1) && mode & (uint5)4) ? MASK_ZIP4_R   :
       (uint32)0;
  // výběr masky pro mód 3 shuffle/unshuffle
  maskL3 = (mode & (uint5)1 && mode & (uint5)16) ? MASK_ZIP16_L
     : (!(mode & (uint5)1) && mode & (uint5)2) ? MASK_ZIP2_L   :
       (uint32)0;
  maskR3 = (mode & (uint5)1 && mode & (uint5)16) ? MASK_ZIP16_R
     : (!(mode & (uint5)1) && mode & (uint5)2) ? MASK_ZIP2_R   :
       (uint32)0;
```

```
  // výběr hodnoty pro posuv mód 0 shuffle/unshuffle
  N0 = (mode & (uint5)1  && mode & (uint5)2)  ? (uint5)1 :
  (!(mode & (uint5)1) && mode & (uint5)16) ? (uint5)8 : (uint5)
     0;
  // výběr hodnoty pro posuv mód 1 shuffle/unshuffle
    N1 = (mode & (uint5)1  && mode & (uint5)4)  ? (uint5)2 :
  (!(mode & (uint5)1) && mode & (uint5)8)  ? (uint5)4 : (uint5)
     0;
  // výběr hodnoty pro posuv mód 2 shuffle/unshuffle
  N2 = (mode & (uint5)1  && mode & (uint5)8)  ? (uint5)4 :
  (!(mode & (uint5)1) && mode & (uint5)4)  ? (uint5)2 : (uint5)
     0;
  // výběr hodnoty pro posuv mód 3 shuffle/unshuffle
  N3 = (mode & (uint5)1  && mode & (uint5)16) ? (uint5)8 :
  (!(mode & (uint5)1) && mode & (uint5)2)  ? (uint5)1 :
  (uint5)0;

  stage1 = hw_shfl_stage(rsc1   , maskL0, maskR0, N0);
  stage2 = hw_shfl_stage(stage1, maskL1, maskR1, N1);
  stage3 = hw_shfl_stage(stage2, maskL2, maskR2, N2);
  stage4 = hw_shfl_stage(stage3, maskL3, maskR3, N3);

  return stage4;
}
// funkce pro výpočet v jednom stupni
inline uint32 hw_shfl_stage(uint32 src, uint32 maskL, uint32
   maskR, uint5 N)
{
  uint32 unchanged;

  unchanged = src & ~(maskL | maskR);
  return ((src << N) & maskL) | ((src >> N) & maskR)
  | unchanged;
}
```

D Obsah přiloženého zip souboru

V přiloženém souboru najdeme vygenerované RTL schéma navrženého procesoru s rozšířením o bitové manipulace v jazyce Verilog. Architekturální testy pro bitové manipulace.

```
/ ...................................................... kořenový adresář
├── testing_B_instr ......................... architekturální testy pro B instrukce
│   ├── arch_covarage_B
│   │   ├── B_extension_cover_32.S
│   │   └── macro.h
│   └── arch_test_B
│       ├── includes
│       │   └── defines.h
│       ├── tests_bext_bdep_check.c
│       ├── tests_count_check.c
│       ├── tests_rot_check.c
│       └── tests_shfl_grev_check.c
└── gen_RTL ................................ generované RTL navrženého procesoru
```