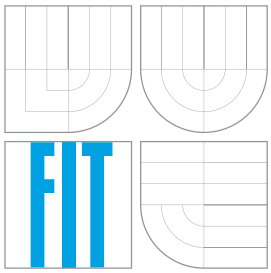
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

MODEL PROCESORU RISC-V

RISC-V PROCESSOR MODEL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ BARTÁK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MARCELA ZACHARIÁŠOVÁ, Ph.D.

BRNO 2016

Zadání diplomové práce/18674/2015/xbarta33

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Barták Jiří, Bc.**
Obor: Počítačové a vestavěné systémy
Téma: **Model procesoru RISC-V**
RISC-V Processor Model
Kategorie: Počítačová architektura

Pokyny:

1. Seznamte se s instrukční sadou procesoru RISC-V a s dostupnými dokumenty popisujícími jeho architekturu.
2. Seznamte se s existujícím modelem procesoru RISC-V popsaném v jazyku Chisel/Scala a proveďte analýzu generovaného kódu v jazyku Verilog.
3. Seznamte se s jazykem CodAL pro modelování procesorových architektur a systémů na čipu.
4. Vyberte podmnožinu vhodných instrukcí a podporovaných funkcí a vytvořte instrukční model procesoru RISC-V.
5. Dle dostupné dokumentace vytvořte obvodový model procesoru RISC-V pro zvolenou podmnožinu instrukční sady a podporovaných vlastností.
6. Na vhodné sadě testovacích aplikací ověřte funkčnost vytvořeného modelu.
7. Porovnejte výkonnost vytvořeného modelu RISC-V s jeho otevřenou podobou v jazyku Chisel/Scala na sadě výkonnostních testů a porovnejte výsledky syntézy obou modelů na zvoleném FPGA.
8. Zhodnoťte celkově vytvořené řešení a dosažené výsledky.

Literatura:

- WATERMAN, Andrew, Yunsup LEE, David A. PATTERSON a Krste ASANOVIĆ. *The RISC-V Instruction Set Manual: Volume I: User-Level ISA* [online]. 2014, (Version 2.0) [cit. 2015-08-15]. Dostupné z: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf
- UC Berkeley Architecture Research. UNIVERSITY OF CALIFORNIA BERKELEY. *RISC-V GitHub of University of California Berkeley*[online]. [cit. 2015-08-15]. Dostupné z: https://github.com/ucb-bar
- Codasip. *Codasip Studio User Guide.* Codasip s.r.o., 2015

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 4 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Šimková Marcela, Ing.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Abstrakt

V rámci snahy o minimalizaci spotřeby a plochy na čipu dochází k vývoji procesorů s aplikačně specifickou instrukční sadou. Dochází tak k vytváření nových instrukčních sad, které však často bývají tajné. Proti tomuto trendu stojí instrukční sada RISC-V, vyvinutá Kalifornskou univerzitou v Berkeley, která je plně otevřena. V této diplomové práci se pozornost věnuje analýze instrukční sady RISC-V a jazyků Chisel a CodAL, které slouží k popisu instrukčních sad a počítačových architektur. Jádrem práce je implementace modelu základní instrukční sady RISC-V a rozšíření pro dělení, násobení a 64-bitový adresový prostor a dále implementace modelu časování založeného na mikroarchitektuře Rocket Core. To vše v jazyce CodAL. Modely jsou dále využity ke generování překladače jazyka C a RTL reprezentace procesoru ve vývojovém prostředí Codasip Studio. Získaný překladač je porovnán s překladačem dostupným od tvůrců instrukční sady a výsledky použity k optimalizaci instrukční sady. RTL je syntetyzováno na FPGA Artix 7 a srovnáno s výsledky syntézy Rocket Core.

Abstract

The number of application specific instruction set processors is rapidly increasing, because of increased demand for low power and small area designs. A lot of new instruction sets are born, but they are usually confidential. University of California in Berkeley took an opposite approach. The RISC-V instruction set is completely free. This master's thesis focuses on analysis of RISC-V instruction set and two programming languages used to model instruction sets and microarchitectures, CodAL and Chisel. Implementation of RISC-V base instruction set along with multiplication, division and 64-bit address space extensions and implementation of cycle accurate model of Rocket Core-like microarchitecture in CodAL are main goals of this master's thesis. The instruction set model is used to generate the C compiler and the cycle accurate model is used to generate RTL representation, all thanks to Codasip Studio. Generated compiler is compared against the one implemented manually and results are used for instruction set optimizations. RTL is synthesized to Artix 7 FPGA and compared to the Rocket Core synthesis.

Klíčová slova

RISC-V, ASIP, modelování souboru instrukcí, instrukční model, CodAL, Codasip Studio, Chisel, Rocket Core, modelování mikroarchitektury, model časování

Keywords

RISC-V, ASIP, instruction set architecture modeling, instruction accurate model, CodAL, Codasip Studio, Chisel, Rocket Core, microarchitecture modelling, cycle accurate model

Citace

BARTÁK, Jiří. *Model procesoru RISC-V*. Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Zachariášová Marcela.

Model procesoru RISC-V

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Marcely Zachariášové, Ph.D.. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jiří Barták
23. května 2016

Poděkování

Děkuji Ing. Marcele Zachariášové, Ph.D., za vedení této diplomové práce. Dále děkuji Ing. Liboru Vašíčkovi, Ing. Michalu Kajanovi a Ing. Adamu Husárovi, Ph.D., za odborné konzultace.

© Jiří Barták, 2016.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Kapitola 1

Úvod

Roku 1946 byl dokončen první počítač `ENIAC`. Tento stroj zabíral plochu 167 m^2 a operoval na frekvenci 5 kHz. Bylo nutné pozměnit jednotlivá propojení v počítači, aby mohl provádět různé programy. Bylo tedy nutné vždy vytvořit nový počítač pro konkrétní úlohu. V padesátých a šedesátých letech došlo díky vynalezení tranzistoru k významnému omezení rozměrů počítačů a dosažení vyšších frekvencí. Roku 1965 vyslovil spoluzakladatel firmy Intel, Gordon Moore, myšlenku, kterou nyní známe jako Moorův zákon a to že počet tranzistorů, které mohou být umístěny na integrovaný obvod se při zachování stejné ceny přibližně každých 18 měsíců zdvojnásobí [13]. První komerčně úspěšný mikroprocesor `Intel 4004` skládající se z 2300 tranzistorů a dosahující frekvence 740 kHz byl dokončen o šest let později. V následujících desetiletích se prokázala platnost Moorova zákona. Výkon procesorů se zvyšoval, zatímco cena se spíše snižovala. Výpočetní výkon se tak stával dostupnějším.

Dříve byl tedy vývoj nových procesorů zaměřen na miniaturizaci technologie a s tím souvisejícím zvyšování hodinového kmitočtu. Tento přístup vede na zvýšení spotřeby a nárůst hustoty tepla na čipu do úrovní, které prakticky nelze uchladit a dosahujeme tak určitých fyzikálních limitů. V době mobilních zařízení je to velmi nepraktické. Je potřeba naopak snižovat spotřebu elektrické energie, což umožní delší výdrž baterií. Mimo to je také vhodné minimalizovat plochu, kterou čip zabírá. Tyto požadavky vedou na potřebu vytvoření nových počítačových architektur a to zejména procesorů s aplikačně specifickou instrukční sadou (ASIP). Takovéto specializované procesory už nemusí být použitelné pro obecné výpočty, ale díky většinou redukované instrukční sadě přináší zjednodušení dekodérů a zmenšení celkové plochy čipu. Instrukční sadu lze také rozšířit o specializované instrukce, které mohou zjednodušit výpočet ve smyslu redukce počtu instrukcí, respektive hodinových cyklů potřebných k provedení operace. Tímto navíc dochází k redukci velikosti kódu, což je také často používaná metrika. Příkladem takové specializované instrukce může být získání počtu bitů ve slově nastavených na 1[1], kterou lze využít třeba ke zrychlení genetických algoritmů [11].

Procesor se obecně skládá ze dvou hlavních částí. Z funkční specifikace v podobě instrukční sady (ISA) a rozhraní. Příkladem ISA jsou `x86` a `IA-32` společnosti `Intel`. Druhou složkou je konkrétní implementace specifikované architektury, jako jsou mikroarchitektury `Intel Sandy Bridge`, či `Haswell`. Z jedné ISA tedy může vzniknout více mikroarchitektur. Například skalární, superskalární, zpracovávající instrukce v programovém pořadí (in-order) nebo mimo něj (out-of-order).

Vývoj specializovaných procesorů v dnešní době úzce souvisí s pojmem Internet věcí.

[1] Population count.

Jedná se vzájemnou komunikaci různých zařízení a přínos nových možností jejich ovládání. Již dnes je například možné ovládat jas chytrých žárovek pomocí mobilního telefonu[2] nebo získat podrobné informace o zápasu pomocí chytré tenisové rakety analyzující hru[3]. Toto je pouze začátek integrace moderních technologií do každodenního života. Spotřeba a velikost je u takových technologií hlavním kritériem. Vytvoření nového procesoru však neznamená pouze definici ISA a její implementaci. Je potřeba také vytvořit program ovládající jeho činnost. Pro procesor využívající specializovanou instrukční sadu není možné využít běžné překladače vyšších programovacích jazyků jako je například `GCC` pro jazyk `C`. Programátor je tedy omezen na jazyk symbolických instrukcí. Vývoj aplikací v takovém nízkoúrovňovém jazyce je zdlouhavý a díky tomu drahý. Jinou možností je neméně nákladné vytvoření nového překladače nějakého vyššího jazyka, usnadňujícího vývoj aplikací. Jakákoliv následná změna ISA či mikroarchitektury potom však znamená potřebu provést změnu překladače, dále prodražující celý projekt. Řešení tohoto problému přináší brněnská společnost Codasip vyvíjející prostředí pro návrh ASIPů. Codasip Studio umožní efektivní zápis ISA, ze které je možné automaticky vygenerovat překladač jazyka `C` pro danou instrukční sadu. Tímto redukuje potřebný čas návrhu z mnoha týdnů na dny až hodiny. Studio dále umožňuje popis mikroarchitektury v intuitivním jazyce `CodAL` podobnému `C`. Z tohoto kódu je možné vygenerovat RTL reprezentaci a verifikační prostředí k ověření správnosti návrhu. Časová úspora získaná automatizací návrhu a zvýšením úrovně abstrakce znamená úsporu financí a také dřívější uvedení produktu na trh, což je základem úspěchu.

Cílem této diplomové práce je využití Codasip Studia pro vytvoření modelu instrukční sady ISA RISC-V zveřejněné University of California Berkeley (UCB) a porovnání dosažených výsledků s dostupnými nástroji v podobě překladače založeného na `GCC`. Nejprve je v kapitole 2 popsána ISA `RISC-V` a její různé implementace. Kapitola 3 se věnuje jazyku `Chisel` a analýze z něj generovaného kódu procesoru `Rocket Core`. V kapitolách 4 a 5 se věnuji popisu jazyka `CodAL` a vývojovému prostředí, které jej využívá. Následuje jádro mé samostatné práce a to implementace RISC-V ISA a v podobě instrukčního modelu (IA) v kapitole 6 a modelu časování (CA) v kapitole 7. Z IA modelu je poté generována sada nástrojů, jejichž výkon je porovnán s nástroji UCB. U CA modelu byla ověřena jeho ekvivalence s IA modelem. Dále byla z CA modelu vygenerována RTL reprezentace, po jejíž syntéze do FPGA bylo provedeno porovnání s mikroarchitekturou `Rocket Core`.

[2] `http://www.easybulb.com`
[3] `www.babolatplay.com`

Kapitola 2

RISC-V

RISC-V je soubor instrukcí původně vytvořený za účelem výuky a výzkumu počítačových architektur v roce 2010 na UCB ve Spojených Státech [17]. Nyní se však stal průmyslovým standardem pod záštitou RISC-V Foundation. Cílem RISC-V je kompletně otevřená ISA, která se nespecializuje na konkrétní cílovou technologii (ASIC, FPGA) a je rozdělena na moduly, které jsou vzájemně nezávislé a nesdílí operační kódy. Jsou podporovány čtyři privilegované módy: strojový (M-machine), virtualizační (H-hypervisor), režim jádra (S-supervisor) a uživatelský (U-user) [16]. Privilegované módy lze kombinovat následovně:

pouze M Vestavěné systémy.

M + U Vestavěné systémy s privilegovaným módem.

M + S + U Systémy umožňující provoz operačních systémů.

M + H + S + U Systémy pro virtualizaci operačních systémů.

Jednotlivé varianty RISC-V ISA využívají 32, 64 a 128-bitové režimy adresování. Volně k dispozici jsou: dokumenty popisující uživatelské a privilegované instrukční sady, sada nástrojů založená na `GCC`, překladač založený na `LLVM`, simulátor ISA, verifikační platforma a parametrizovatelný generátor mikroarchitektury `Rocket Core` generující RTL kód v jazyce `Verilog`. Název RISC-V byl zvolen z důvodu, že se jedná o pátou generaci instrukční sady typu RISC vyvinutou v Berkeley. Jejími předchůdci jsou `RISC-I`, `RISC-II`, `SOAR` a `SPUR` [17]. Římská číslice pět navíc symbolizuje vektor (vektorová rozšíření instrukční sady) a variabilitu.

2.1 ISA

Základem `RISC-V` je ISA pro celočíselné operace, kterou je nutné vždy implementovat a je možné ji doplnit velkým množstvím rozšíření. Jde o obdobu raných instrukčních sad RISC, ovšem s podporou různé délky instrukcí a bez opožděných skoků[1]. Základní ISA je omezena na minimální sadu instrukcí umožňující provoz překladačů a operačních systémů. Existují dvě stabilní varianty celočíselné ISA: RV32I[2] a RV64I, podporující 32, respektive 64-bitový adresový prostor. Obě varianty je možné použít pospolu. Je možnost využít i experimentální

[1] Branch delay slot.

[2] RV znamená RISC-V, I Integer a číslo velikost adresovatelného prostoru.

rozšíření RV128I poskytující 128-bitový adresový prostor k jehož potřebě může v budoucnosti dojít [17]. Celočíselná ISA obsahuje instrukce pro základní počítání s celými čísly (sčítání, odčítání, logické operace a porovnávání), instrukce pro načítání a ukládání dat a také pro změnu toku řízení (skoky). Rozšíření základní ISA se dělí na 2 skupiny:

Standardní Obecně použitelná rozšíření, jejichž použití není v konfliktu se základní ISA ani s ostatními základními rozšířeními.

Nestandardní Vysoce specializovaná rozšíření, mohou být v konfliktu s jinými rozšířeními.

2.1.1 Standardní rozšíření

Rozšíření pro násobení a dělení nese označení „M“ (Multiplication) a obsahuje instrukce pro dělení, násobení a zbytek po celočíselném dělení. Všechny pouze s registrovými operandy. „A“ (Atomic) značí skupinu instrukcí pro atomické čtení, zápis a instrukce typu načti-modifikuj-zapiš[3] umožňující meziprocesorovou synchronizaci [17]. Dvojice rozšíření „F“ a „D“ přináší registry pro operace s plovoucí řádovou čárkou s jednoduchou, respektive dvojitou přesností a dále výpočetní, načítací a ukládací instrukce, které s nimi operují. Pokud jsou použity všechny skupiny instrukcí „IMAFD“, užívá se zkrácené označení „G“ (General-purpose), tedy „RV32G“ nebo „RV64G“ podle základní sady.

2.1.2 Nestadardní rozšíření

Pro výpočty vysoce náročné na přesnost je možné použít rozšíření pro čtyřnásobnou přesnost plovoucí řádové čárky „Q“ (Quad). Jeho použití vyžaduje rozšíření registrů s plovoucí řádovou čárkou na 128 bitů a implementaci „RV64IFD“. Jedná se o přidání načítacích, ukládacích a výpočetních instrukcí pro nově rozšířené registry a také instrukcí konvertující hodnoty mezi různými přesnostmi. „C“ značí komprimovanou instrukční sadu redukující velikost instrukcí na 16 bitů. Všechny tyto instrukce jsou variantami skupiny „G“. Redukce délky instrukce je provedena zmenšením šířky přímého kódování konstant v instrukci, omezením množiny registrů, které mohou být použity a pevným stanovením registrových operandů. Tato sada obsahuje instrukce pro nejčastěji užívané operace, které mohou v běžném kódu nahradit více jak polovinu instrukcí a redukovat tím velikost kódu až o 30 % [18]. Instrukce redukované délky je možné kombinovat s klasickými, vyžaduje to však přizpůsobení mikroarchitektury, jelikož ta se nyní musí vyrovnat s 32-bitovými slovy zarovnanými na 16 bitů.

Zbytek rozšíření není plně definován a obsahuje pouze nástin možné činnosti. „L“ je plánováno pro podporu dekadické desetinné aritmetiky dle standardu IEEE 754-2008. „B“ je rezervováno pro bitovou manipulaci. Bude extrahovat, vkládat a rotovat bity. „T“ pro paměťové operace transakčního chování. RISC-V má místo rezervováno i na rozšíření pro vektorové operace „P“ (Packed SIMD), které by mělo operovat nad registry určenými pro výpočty s plovoucí řádovou čárkou. Kromě zde nastíněných rozšíření je v operačních kódech RISC-V ponechán prostor pro další.

2.2 Mikroarchitektury

RISC-V přináší pouze popis instrukční sady, nikoliv popis mikroarchitektury. Detaily mikroarchitektur vyvinutých na základě RISC-V pro komerční účely pochopitelně nejsou veřejně

[3] AMO - Atomic Modify.

dostupné. UCB vyvinula k výzkumným účelům 3 rozdílné architektury: Z-Scale, `Rocket Core` a BOOM. Ačkoliv je každá specificky zaměřená díky popisu v jazyce `Chisel` je podstatná část kódu sdílená mezi jednotlivými jádry [12][10]. Za zmínku také stojí projekt `lowRISC`, který se zaměřil na úpravy `Rocket Core`. Zejména jde o zjednodušení vstupních a výstupních operací. Původní procesor spoléhá na asistenci koprocesoru, zatímco upravená verze je schopna tyto operace provádět napřímo. Zdrojové kódy `lowRISC` jsou volně k dispozici[4].

2.2.1 Z-Scale

Z-Scale je rodina jader zaměřená podobně jako `ARM M0` na co nejmenší velikost a spotřebu [12]. Jádra implementují „RV32IM“ ISA, podporují strojové a uživatelské privilegované módy. Jedná se o skalární procesor s čtyřstupňovou zřetězenou linkou. Stupně linky jsou:

PC Generation Nastavení hodnoty programového čítače a zaslání požadavku na instrukci do paměti.

IF Dokončení načítání instrukce.

ID + EX Dekódování instrukce, provedení její operace a případné zaslání požadavku na data do paměti.

WB + MEM + MUL Paměťové operace a případné dokončení násobení.

Instrukční a datová sběrnice jsou standardu `AHB-Lite` [12]. Konkrétní mikroarchitektura nebyla prozatím zveřejněna, k dispozici je ale porovnání s procesorem ARM M0 uvedené v tabulce 2.1.

	RISC-V Z-Scale	**ARM Cortex-M0**
ISA	RV32IM	32-b ARM v6
Výkonnost	1.35 DMIPS/MHz	0.87 DMIPS/MHz
Výrobní proces	TSMC 40GPLUS	TSMC 40LP
Plocha bez pamětí cache	0.0098 mm^2	0.0070 mm^2
Efektivita na plochu	138 DMIPS/MHz/mm^2	124 DMIPS/MHz/mm^2
Frekvence	cca 500 MHz	50 MHz
Napětí	0.99 V	1.1 V
Dynamický příkon	1.8 μW/MHz	5.1 μW/MHz

Tabulka 2.1: Porovnání RISC-V Z-Scale s ARM Cortex-M0 [12].

2.2.2 Rocket Core

Rocket je 64-bitový skalární procesor zpracovávající instrukce v programovém pořadí. Je obdobou procesoru `ARM A5` [5]. Jejich srovnání je v tabulce 2.2. Rocket implementuje sadu instrukcí „RV64G“, obsahuje plnou podporu spekulace skoků[5] a podporuje virtuální paměť. Počet stupňů zřetězené linky je 6:

[4] Repozitáře jsou dostupné na: `https://github.com/lowrisc`.

[5] Tabulka cílů skoků (Branch Target Buffer), tabulka historie skoků (Branch History Table) a zásobník návratových adres (Return Address Stack).

PC Generation Nastavení hodnoty programového čítače.

IF Přístup do ITLB a instrukční paměti cache.

ID Dekódování instrukce a přístup k registrovým operandům.

EX Provedení instrukce pokud je celočíselná. Instrukce pracující s plovoucí řádovou čárkou jsou zpracovány jednotkou FPU umístěnou za posledním stupněm.

MEM Přístup do DTLB a datové paměti cache.

WB Dokončení paměťové operace.

V současné době je komunikace mezi procesorem a pamětí zprostředkována akcelerátorem, kterým může být jádro Z-Scale. Do budoucna se plánuje integrace standardu `AXI4` a přidání druhé cesty funkčních jednotek do řetězené linky[6] [4]. Zdrojové kódy tohoto jádra jsou plně uvolněny a jsou rozebrány v sekci 3.2.

	RISC-V Rocket	**ARM Cortex-A5**
ISA	RV64G	32-b ARM v7
Výkonnost	1.72 DMIPS/MHz	1.57 DMIPS/MHz
Výrobní proces	TSMC 40GPLUS	TSMC 40GPLUS
Plocha bez pamětí cache	0.14 mm^2	0.27 mm^2
Efektivita na plochu	4.41 DMIPS/MHz/mm^2	2.96 DMIPS/MHz/mm^2
Frekvence	> 1 GHz	> 1 GHz
Dynamický příkon	0.034 mW/MHz	0.08 mW/MHz

Tabulka 2.2: Porovnání RISC-V Rocket s ARM Cortex-A5 [5].

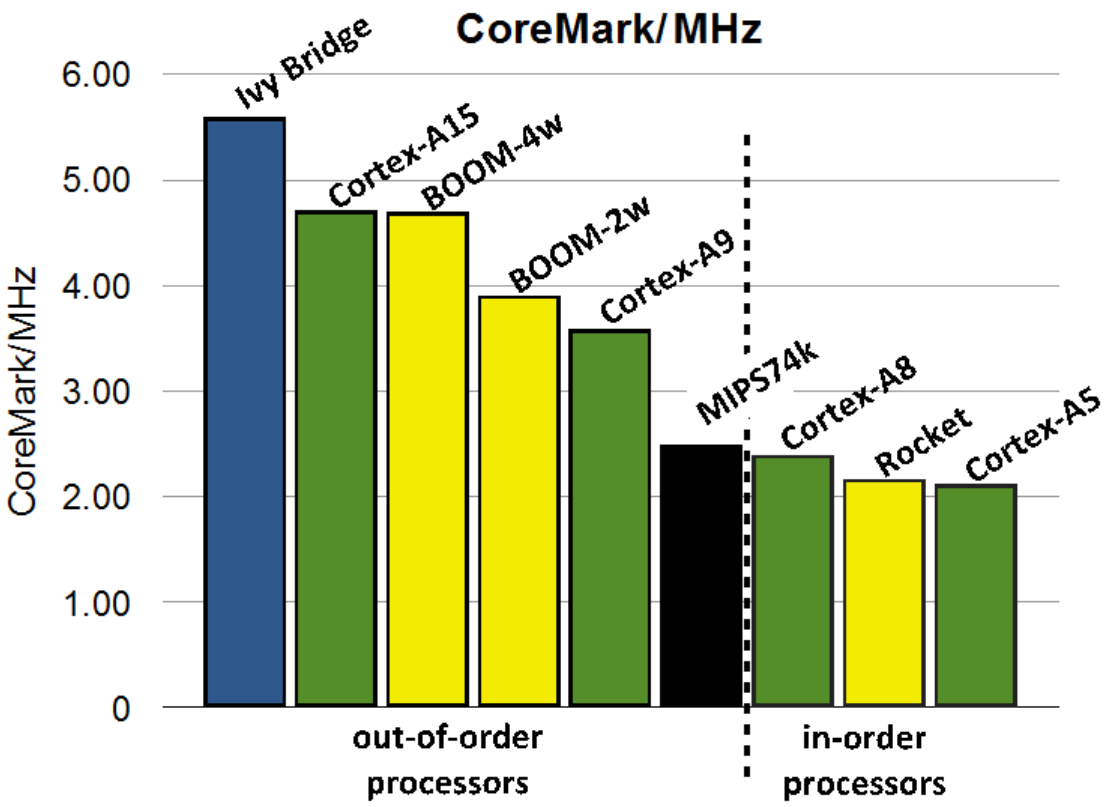
2.2.3 BOOM

Berkeley Out-of-Order Machine je 64-bitový superskalární procesor s možností vykonávání instrukcí mimo programové pořadí. Podporovaný počet cest je 1, 2 a 4. BOOM je obdobou procesoru `ARM A9` [10]. Výsledky jejich srovnání se nachází v tabulce 2.3. Stejně jako `Rocket Core` obsahuje plnou podporu spekulace skoků a sdílí s ním zdrojové kódy pro paměti cache a okolí jádra[7]. Zřetězená linka má 8 stupňů, ale konkrétní mikroarchitektura nebyla prozatím zveřejněna.

	RISC-V BOOM 2 linky	**ARM Cortex-A9**
ISA	RV64G	32-b ARM v7
Výkonnost	3.91 CoreMark/MHz	3.59 CoreMark/MHz
Výrobní proces	TSMC 40GPLUS	TSMC 40GPLUS
Plocha s 32 KB cache	1 mm^2	2.5 mm^2
Efektivita na plochu	3.9 CoreMark/MHz/mm^2	1.4 CoreMark/MHz/mm^2
Frekvence	1.5 GHz	1.4 GHz

Tabulka 2.3: Porovnání RISC-V BOOM s ARM Cortex-A9 [10].

[6] To by vedlo k vytvoření superskalárního procesoru.
[7] Takzvaný uncore. Jedná se o koherenční manažery a další podpůrné jednotky.



Obrázek 2.1: Porovnání procesorů na testu CoreMark [10].

Z grafu uvedeného na obrázku 2.1 vyplývá, že mikroarchitektury založené na relativně mladé ISA RISC-V jsou schopné konkurovat jádrům společnosti `ARM` a dokonce příliš nezaostávají ani za procesory `Intel` což je vzhledem k tomu, že RISC-V vyvíjí velmi malé týmy velký úspěch.

Kapitola 3

Jazyk Chisel

Donedávna byl běžný způsob návrhu hardwaru založen na grafických editorech schémat obvodu. Použití schémat je intuitivní, jde o modelování bloků pomocí základních primitiv, jako jsou logické členy. Výsledná schémata však mohou být pro osobu, která je nevytvořila matoucí a to zejména pokud není využita rozumná hierachie komponent. HDL jazyky umožnily návrh obvodů na vyšší úrovni abstrakce, což přináší vyšší produktivitu. Ovšem stále neumožňují vytváření hardwaru ve stejném duchu jako programovaní běžné aplikace. HDL jazyky umožňují popis na úrovni struktury a chování. Strukturální popis umožňuje propojování jednotlivých komponent podobně jako u schémat obvodů. Behaviorální popis zachycuje chování obvodu s využitím konstrukcí vyššího programovacího jazyka, jako jsou podmínky a cykly. Nejznámější HDL jazyky jsou VHDL a `Verilog`.

Počátek `Verilogu` se datuje na přelom let 1983-1984, kdy byl vytvořen jako soukromý projekt společnosti Gateway Design Automation za účelem simulování a verifikace. Později byla tato společnost odkoupena Cadence Design Systems. Roku 1995 byl `Verilog` poprvé standardizován jako `IEEE 1364-1995` [15].

Vývoj VHDL začal roku 1981 jako projekt ministerstva obrany Spojených Států kvůli řešení krize vývojového cyklu hardware. Tato krize spočívala v tom, že nahrazování zastaralé technologie bylo příliš nákladné. Jednak z důvodu chabé dokumentace a také díky tomu, že jednotlivé komponenty systémů byly navrhovány v různých jazycích a k jejich simulacím se používalo jiných technik. Bylo tedy třeba zavést standardizaci. VHDL bylo na rozdíl od Verilogu poskytnuto průmyslovému sektoru zadarmo již od svého počátku. Poprvé bylo standardizováno roku 1987 jako `IEEE 1076-1987` [9].

Většina programovacích struktur má svoje ekvivalenty v obou jazycích. Vyjímkou je datový typ záznamu[1], který ve Verilogu nenajdeme. V dnešní době je VHDL populární hlavně v Evropě, zatímco `Verilog` vede v Americe. Problémem těchto jazyků je, že mnoho programovacích struktur nelze syntetizovat. Běžní uživatelé ignorují většinu možností jazyka a drží se pouze jeho nejzákladnější podmnožiny. Dalším problémem je slabá podpora metaprogramování ztěžující vytváření dobře parametrizovatelného kódu. Tyto problémy se snaží řešit `Chisel`.

`Chisel` je nadstavba programovacího jazyka `Scala` vyvinutá pod vedením Jonathana Bachrache v UCB. Základem jazyka je velká podpora metaprogramování. Důvodem vytvoření pouhé nadstavby namísto nového jazyka byla snaha vyhnout se programování celého jazyka, tedy i částí, které s vývojem hardware vůbec nesouvisí, když je možné nějaký stávající jazyk pouze doplnit o potřebné části. `Scala` byla vybrána jako základ, protože spojuje

[1] Struktura v jazyce `C`.

dohromady objektově orientované programování, funkcionální programování, silnou typovou kontrolu, odvozování datových typu[2] a virtuální stroj jazyka `Java` [6]. `Chisel` neslouží přímo pro syntézu hardwaru, generuje se z něj kód v jazyce `Verilog`. Spuštěním programu v `Chiselu` dojde provedení následujících akcí: [8]

1. Generování interní datové struktury (graf buněk).
2. Výpočet šířky vodičů.
3. Kontrola propojení komponent.
4. Generování kódu v jazyce `Verilog`, případně C++.

3.1 Možnosti jazyka

3.1.1 Datové typy

Základní datové typy jazyka `Chisel` jsou:

Bits Vektor jednotlivých bitů.

Fix Číslo v pevné řádové čárce se znaménkem.

UFix Číslo v pevné řádové čárce bez znaménka.

Bool

Celá čísla jsou speciálním případem `Fix` a `UFix` bez desetinné složky. Čísla se znaménkem jsou reprezentována ve dvojkovém doplňku. Jednotlivé vodiče lze sdružovat do svazků (Bundles)[3]. `Chisel` dále nabízí vektorový datový typ `Vec` vytvářející indexovatelné pole objektů stejného typu. `Vec` i `Bundle` jsou potomci třídy Data. Jakýkoliv objekt, který je potomkem třídy `Data` lze v hardwaru reprezentovat jako vektor bitů. Jednotlivým objektům lze nastavit jejich směr a tím z nich vytvořit porty[4]. Nastavení směru lze provést během definice i instancování. `Chisel` automaticky nastavuje šířku dat výpočetních operací podle operandů a to způsobem uvedeným v tabulce 3.1.

Operace	**Bitová šířka výsledku**
+,–	Šířka delšího operandu + 1.
*, konkatenace	Součet šířek operandů.
NOT, AND, OR, XOR	Šířka delšího operandu.
konkatenace	Šířka delšího operandu.
posun doleva	Šířka prvního operandu + největší číslo reprezentovatelné na šířce druhého operandu.
posun doprava	Šířka prvního operandu + nejmenší číslo reprezentovatelné na šířce druhého operandu.

Tabulka 3.1: Bitová šířka výsledků operací v jazyce `Chisel` [7].

[2]Type inferrence.
[3]Obdobou struktur jazyka `C`.
[4]Sdružením portů do svazku získáme obdobu entity jazyka VHDL.

3.1.2 Kombinační logika

Kombinační logika se zapisuje následovně: **val** out = (a & ∼b) | (∼a & b). Tento výraz se při každé změně „a“ nebo „b“ aktualizuje. „a“ a „b“ zde figurují jako pojmenované vodiče. Kompilací tohoto výrazu vznikne jednoduchý obvod se dvěma logickými členy `AND` a `NOT` a jedním `OR` poskytující logickou funkci `XOR`. Větvení kódu se provádí pomocí bloků `When` a `Switch`. Chování `Switch` je stejné jako v jazyce `C`. `When` je ekvivalentní k podmínce `If`, přináší však kromě větve `elsewhen` také větev `unless`, která je provedena v případě, že její podmínka splněna není. Tohoto chování však lze jednoduše dosáhnout negací podmínky, proto se tato možnost zdá nadbytečná.

3.1.3 Funkce

Funkce umožňují znovupoužití definovaných obvodů. Následující výraz definuje funkci XOR: **def** XOR (a: **bits**, b: **bits**) = (a & ∼b) | (∼a & b). Lze ji použít v předchozím příkladu: **val** out = XOR(a , b). Jednoduché funkce jako je tato umožní vnést hierarchii do zdrojového kódu, nikoliv však do kódu, který se z něj generuje [7]. K tomuto účelu slouží komponenty.

3.1.4 Komponenty

Komponenty umožňují vnést hierarchii do generovaných obvodů [8]. Jde o obdobu modulů jazyka `Verilog`. Komponenty jsou třídy rozšiřující třídu Component. Definují rozhraní a spojují dohromady menší obvody. Ukázka kódu 3.1 prezentuje definici jedné ze základních hardwarových komponent, multiplexoru.

```
class Mux extends Component
{
  val io = new Bundle
  {
    val sel = Bits(width = 1, dir = input);
    val in0 = Bits(width = 1, dir = input);
    val in2 = Bits(width = 1, dir = input);
    val out = Bits(width = 1, dir = output);
  }
  io.out := (io.sel & io.in0) | (~io.sel & io.in1);
}
```

Ukázka kódu 3.1: Definice dvouvstupového multiplexoru.

Komponenty se instancují stejně jako v jazyce C++ klíčovým slovem new: **val** multiplexor = **new** Mux(). Základní komponenty jsou součástí standardní knihovny: dekodér, kodér, multiplexor, registr, paměť, posuvný registr, LSFR[5], převrácení pořadí bitů a čítač bitů nastavených na 1[6]. Znovupoužitelnost v `Chiselu` vychází z možnosti rozšířit komponentou třídu parametrů, která díky tomu může ovlivňovat celý návrh.

[5] Linear Feedback Shifting Register. Komponenta generující pseudonáhodné posloupnosti.
[6] Population count.

3.2 Analýza a generovaní Rocket Core

V současné době není zveřejněn žádný dokument popisující mikroarchitekturu `Rocket Core` a proto jsou informace dostupné pouze v podobě prezentací, zveřejněných záznamů e-mailové korespondence, zdrojových kódů v jazyce `Chisel` a vygerovaných kódů v jazyce `Verilog`. Zdrojové kódy Rocket Core jsou volně dostupné v systému `Github`[7]. Jsou však rozprostřené přes několik repositářů z důvodu oddělení jádra, jeho okolí a generických komponent a to zejména pro potřeby znovupoužití. Tento přístup však komplikuje analýzu. Řešením je využití vývojového prostředí `Scala IDE`[8] založeného na vývojovém prostředí `Eclipse`, které umožní pohodlné vyhledávání napříč repositáři. Samotné zdrojové kódy nejsou vůbec komentované, popřípadě velmi málo a obecně. Jelikož se v případě `Chiselu` jedná o funkcionální programování, představuje Rocket Core značnou překážku pro programátory orientující se na procedurální jazyky. Učící křivka `Chiselu` je navíc velmi strmá ve smyslu obtížnosti prvotního pochopení kódu.

Pro generování HDL popisu jsou v repositáři přichystané skripty. Pro jejich správné fungování je třeba nainstalovat sadu nástrojů RISC-V. Instalace se skládá především ze stažení několika potřebných repositářů a nastavení systémových proměnných. `Rocket Core` poskytuje velkou flexibilitu generovaného systému prostřednictvím konfiguračního souboru `Config.scala` a to včetně počtu jader. Lze jednoduše ovlivnit následující nastavení:

- Bitové šířky adres (fyzických i virtuálních) a dat.
- Počet úrovní tabulky stránek a jejich velikost.
- Počet cest a velikost skupin TLB.
- Maximální hardwarově podporovaný počet běžících procesů.
- Počet cest, velikost skupin a šířku řádku pamětí cache.
- Rozdělení paměti L2 cache a hlavní paměti do bank.
- Koherenční protokoly pro jednotlivé úrovně pamětí cache.
- Počet jader.

Experimentování s konfiguracemi je tedy přímočaré, ale pouze pokud není třeba provádět zásahy do zdrojových kódů. Přidání dalšího jádra je primitivní záležitostí, naproti tomu rozšíření zřetězené linky o další stupeň je komplikované. Lze si zvolit mezi generováním pro FPGA a ASIC.

Výsledkem generování je testovací skript a zdrojový kód v jazyce Verilog. Tento jeden zdrojový kód obsahuje celý výsledný systém, interně rozdělený do modulů. Při použití originální konfigurace má generovaný kód 73355 řádků. Názvy jednotlivých modulů jsou stejné jako názvy tříd v `Chiselu`. To samé platí pro rozhranní tříd. Implementace samotného modulu však nese typický znak generovaného kódu. Signály jsou nazvány ve stylu "Tx", kde "x"je celé číslo. Do těchto signálů se přiřazují hodnoty na základě ternárního výrazu či porovnání hodnot jiných signálů. Sledovaná hodnota je propagována přes nepřeberné množství signálů a čitelnost kódu je tudíž minimální. Možnosti vizualizace generovaného kódu prostřednictvím běžných nástrojů jako je Vivado společnosti Xilinx jsou omezené,

[7] Samotné jádro dostupné na adrese: `https://github.com/ucb-bar/rocket`.

[8] Dostupné na adrese: `http://scala-ide.org/`.

jelikož i menší moduly obsahují příliš velké množství primitiv a při pokusu o jejich vykreslení dochází k pádu aplikace. Efektivní analýza je tedy omezena pouze na zdrojové kódy v `Chiselu`. Analýzou zdrojových kódů jsem zjistil následující podrobnosti o mikroarchitektuře `Rocket Core`:

- Násobení a dělení může probíhat mimo programové pořadí.
- Podporované privilegované módy jsou M, S a U.
- Prediktor skoků je založený na korelačním prediktoru `McFarling`. Používá se registr globální historie skoků a pole dvoubitových prediktorů. `XOR` globální historie skoků se spodními bity adresy dává index do tabulky prediktorů.
- Modul pro průchod tabulky stránek používá pamět cache pro uchování nejčastěji vyhledávaných záznamů.
- TLB jsou plně asociativní a využívají PseudoLRU politiku výběru obětí.
- Paměti L1 cache používají pro výběr obětí politiku náhodného výběru realizovanou pomocí Fibonacciho 16-bitového lineárního posuvného registru se zpětnou vazbou.
- Vyřízení požadavku na data z paměti cache trvá 1 takt.
- Datová cache je neblokující, dokáže tedy přijímat požadavky, zatímco načítá chybějící záznam.
- Vždy dochází ke znaménkovému rozšíření hodnot.
- Každé jádro má svou L1 datovou a instrukční cache.
- Paměť L2 cache je sdílená pro všechna jádra a je společná pro instrukce a data.
- Pro paměťové operace je nutné připojit koprocesor na rozhranní `TileLink`.
- Vstupně-výstupní operace jsou identifikovány až po přístupu do L1 cache. Cíl těchto operací bude ve speciálním regionu, který se neukládá do cache. Přístup tedy vždy vyvolá výpadek, po kterém se provede kontrola cíle a případné odeslání požadavku do vstupně-výstupní jednotky.

Kapitola 4

Jazyk CodAL

`CodAL` je jazyk pro popis počítačových architektur vyvinutý v rámci výzkumného projektu `Lissom` na Fakultě informačních technologií Vysokého Učení Technického v Brně. Účelem jazyka je velmi rychlé prototypování ASIPů a multiprocesorových platforem [2]. Předností jazyka je možnost souběžného vývoje software a hardware za pomoci vysokého stupně automatizace generování podpůrných nástrojů pro programování a simulování. Mezi vygenerované nástroje patří překladače jazyka `C` a jazyka symbolických instrukcí, profilovací nástroje, debugger a linker. `CodAL` poskytuje velkou míru abstrakce, která umožňuje provádět rychlé změny mikroarchitektury i instrukční sady. Díky tomu je možné efektivně zkoušet velké množství různých variant pro dosažení maximální spokojenosti. Syntaxe `CodALu` je velmi podobná jazyku `C`. Jeho předností je také možnost využití zdrojů před jejich deklarací v kódu a to bez potřeby dopředných deklarací. Model procesoru je rozdělen na popis platformy a samotného procesoru, což umožňuje znovupoužití procesoru na různých platformách. Nově lze kód organizovat do modulů sdílejících lokální zdroje. Díky modulům lze efektivněji využít úprav časování[1] syntetizačních nástrojů.

4.1 Popis platformy

Platforma definuje okolí procesoru, ve kterém je zpravidla paměť pro kód a data. Do okolí je možné přidat paměť cache, časovače, řadiče přerušení a libovolná další uživatelská rozšíření. Paměti a cache jsou součástí jazyka a umožňují parametrizaci. Je možné nastavit například jejich zpoždění, šířku datových bloků[2] a politiku výběru obětí. Jednotlivé komponenty jsou na platformě propojeny pomocí jednosměrných portů libovolné šířky, případně pomocí sběrnice Codasip Local Bus (CLB). CLB umožňuje vykonávat operace čtení, zápisu, zneplatnění a vypláchnutí[3]. Uživatelské komponenty poskytují velkou volnost při implementaci. Je třeba dodat pouze definici rozhranní komponenty v `CodALu`, její simulační model v jazyce C/C++[4] a hardwarovou reprezentaci ve VHDL či `Verilogu`. Ukázka kódu 4.1 demonstruje definici komponenty na platformě.

[1] Retiming.
[2] Cacheline.
[3] Flush.
[4] Musí implementovat virtuální metody dané definovaným rozhranním.

```
component pic
{
  // typ komponenty vyuzity pri propojeni komponenty
  // a jejiho modelu
  type = "pic_t";
  // vystupni port, znaci cekajici preruseni
  port bit[1] p_irq_out;
  // vstupni port pro zadosti o preruseni
  port bit[3] p_irq_in;
  // rozhranni pro pristup ke kontrolnim registrum komponenty
  interface if_clb
  {
    // bitova sirka adresy, dat a nejmensi adresovatelne jednotky
    bits = { 32, 32, 8 };
    // endianita rozhranni
    endianness = LITTLE;
    // typ rozhranni
    type = CLB:SLAVE;
    // povolene typy pristupu (cteni a zapis, pouze pro cteni)
    flag = RW;
  }
}
```

Ukázka kódu 4.1: Popis komponenty řadiče přerušení (Programmable Interrupt Controler) v jazyce `CodAL`.

4.2 Popis procesoru

Popis procesoru se skládá ze čtyř částí: [2]

Architekturální zdroje Programový čítač a registry.

Popis ISA Názvy instrukcí, jejich operandy a binární kódování.

Sémantický popis Chování všech instrukcí a výjímek. Popis jak ovlivňují viditelné (architekturální) registry.

Implementační část Chování (časování) zdrojů, které nejsou viditelné z úrovně instrukčního popisu, ale definují samotnou mikroarchitekturu[5].

Z těchto částí se vytváří dvě úrovně modelu procesoru:

Instrukční model Instruction Accurate Model (IA). Obsahuje sémantiku, ISA a architekturální zdroje. Slouží pro generování překladačů a reprezentuje referenční model verifikace.

Model časování Cycle Accurate Model (CA). Obsahuje ISA, architekturální zdroje a implementační část. Slouží pro generování HDL kódu a verifikačního prostředí.

[5]Například nastavování řídících signálů multiplexorů.

Velká část popisu obou modelů je tedy sdílena. IA a CA modely musí být ekvivalentní, pro ověření se využívá funkční verifikace. Každý procesor má zpravidla jeden IA model (jedna instrukční sada) a libovolný počet CA modelů (alternativní mikroarchitektury). Kód 4.2 ukazuje část popisu dekódovacího stupně zřetězené linky procesoru.

```
event id : pipeline(pipe.ID)
{
  use dec;
  use id_output;
  semantics
  {
    uint32 temp;
    // dokonceni nacitani instrukce
    if_fetch.ifinish(CP_FI_COMPLETE, temp);
    // pokud je stupen ID volny pouzit nactenou instrukci,
    // jinak vlozit NOP
    id_instruction = (id_clear) ? NOP_INSTRUCTION : temp;
    // dekompozice instrukce-uvedene signaly jsou globalni zdroje
    // operacni kod
    id_opcode = (uint6)(id_instruction >> 26);
    .
    .
    .
  };
  decoders
  {
    // zaslani operacniho kodu do dekoderu
    // pro nastaveni dalsich ridicich signalu
    { dec(id_opcode); }
  };
  .
  .
  .
};
```

Ukázka kódu 4.2: Část popisu ID stupně řetězené linky CA modelu procesoru Codasip uRISC.

4.3 Popis instrukcí

Objekt popisující instrukci, takzvaný `element`[6] musí obsahovat informace o své binární a textové reprezentaci pro potřeby překladače jazyka symbolických instrukcí, popis chování instrukce během simulace a také její popis pro potřeby generování překladače (může se lišit). Dále je třeba definovat hardware, který je s danou instrukcí asociovaný[7]. V ukázce kódu 4.3 je uvedena definice instrukce pro operaci bez efektu na stav procesoru. Od definovaných

[6]Element neslouží pouze k popisu instrukce jako celku, ale také k popisu jejích komponent jako jsou registry, přímé kódování konstant a operační kód.

[7]Registry, signály a jejich časování.

objektů lze odvozovat aliasy pro překladač jazyka symbolických instrukcí, které umožní různý textový popis téhož binárního kódování[8]. Je možné také vytvořit alias pro potřeby překladače. Takový element potom slouží pouze k extrakci sémantiky[9], pro potřeby simulace je použit původní objekt. Oba aliasy je možné kombinovat.

```
element i_nop
{
  assembler { "NOP" };
  // binarni kodovani je zvolene jako same nuly.
  binary { 0:bit[32] };
  semantics
  {
    // semanticka sekce pouze zavola vestavenou funkci, v pripade
    // instrukce pro secteni by zde byl proveden soucet.
    codasip_nop();
  }
}
```

Ukázka kódu 4.3: Popis instrukce NOP.

[8]Například od instrukce pro podmíněný skok s testem na rovnost (BEQ registr1, registr2, adresa) může být vytvořen assembler alias pro porovnání na nulu (BEQZ registr, adresa), za předpokladu, že některý registr je zadrátovaný na 0). Potom by existovaly dvě možné textové reprezentace stejného binárního kódu.

[9]Lze tak například donutit překladač, aby instrukci pro relativní skok chápal jako instrukci pro skok absolutní bez ovlivnění správného chování překladače i architektury.

Kapitola 5

Codasip Studio

Codasip Studio poskytuje vývojové prostředí pro jazyk `CodAL`. Využívá předností jazyka pro automatizaci úkonů, které by jinak bylo potřeba provádět manuálně. Tabulka 5.1 ilustruje typické trvání etap vývoje procesorů RISC a VLIW.

Etapa vývoje	**Codasip Studio**		**Tradiční postupy**	
	RISC	**VLIW**	**RISC**	**VLIW**
Výzkum architektury a modelování	40-60	60-80	40-60	60-80
Programování nástrojů	<1		80	480
Simulace			80	120
Hardwarová reprezentace			100	150
Vytvoření verifikačního prostředí			100-150	
Verifikace	40-60		100-150	
$\sum$	**80-120**	**100-140**	**420-500**	**930-1010**

Tabulka 5.1: Typické etapy vývoje procesorů a jejich trvání ve dnech [1].

Z tabulky je patrné, že Codasip Studio umožní zkrátit některé etapy vývoje z týdnů na hodiny. Celý vývoj potom z řádu měsíců na několik týdnů.

Studio je klient-server aplikací. Klient poskytuje vývojové prostředí založené na platformě `Eclipse` a rozhranní pro komunikaci se serverem v podobě Codasip Commandline, která je sama o sobě plně dostačující pro jakékoliv úkony a to prostřednictvím příkazové řádky. Není tedy nutné používat grafické prostředí. Server[1] obsahuje generátory jednotlivých nástrojů a po vygenerování umožňuje jejich export. Exportované nástroje je možné na kompatibilních platformách využít bez asistence Studia.

Pomocí Studia lze generovat následující nástroje:

Assembler Překladač jazyka symbolických instrukcí slouží k transformaci lidsky čitelného kódu v jazyce symbolických instrukcí do binárního objektového souboru. Objektové soubory jsou následně linkovány do binárního kódu čitelného cílovým procesorem. Formáty direktiv a symbolů jsou shodné s `GCC`, avšak je snadné je změnit. Výstupem assembleru je objektový soubor ve formátu `ELF`.

Disassembler Slouží k transformaci spustitelného kódu[2] do kódu v jazyce symbolických instrukcí. Jeho výstup lze bez úprav opětovně přeložit assemblerem. Datové sekce jsou

[1] Nazývaný Middleware.

[2] Nejčastěji se jedná o výstup assembleru nebo linkeru.

automaticky převedeny na definice konstant. U sekcí obsahujících kód se dissambler pokouší přiřadit binárním datům instrukci. Pokud uspěje, vyprodukuje její textovou reprezentaci. V opačném případě data reprezentuje jako konstanty.

Linker Slouží ke spojování objektových souborů dohromady. Dále řeší různé relokace adres, které nebyly známé během překladu. Linker nezávisí na cílové architektuře, tudíž jeho generování není nutné [3]. Codasip linker je založený na GNU. Jeho výstupem je spustitelný soubor[3] s příponou `.xexe`.

Překladač jazyků C a C++ Překladač je založený na populární volně dostupné platformě LLVM[4]. Ta poskytuje kód v produkční kvalitě s velkým stupněm optimalizace [3]. Při generování překladače dochází k doplnění LLVM o optimalizace závisející na cílové architektuře. Je poskytována i podpora pro překladač pro procesory VLIW. Unikátní vlastností překladače generovaného Codasip Studiem je možnost nastavení různých podmínek [3]. Například lze nastavit překladač tak, aby sám řešil datové závislosti mezi instrukcemi. Díky tomu je možné odstranit hardware, který má tuto funkcionalitu na starosti a tím dosáhnout menší spotřeby a plochy čipu. K dispozici je také standardní knihovna jazyka `C Newlib`. Její nízkoúrovňové části je však třeba upravit na základě cílové architektury. Překlad probíhá z jazyka `C/C++` do jazyka symbolických instrukcí ve formátu `GNU`.

Simulátory Simulátory jsou popsány samostatně v podkapitole 5.1.

Profilovací nástroje Zaznamenávají důležité informace[5] během simulace. Tyto informace mohou vést k dalšímu zlepšení výsledného procesoru. Je možné například odstranit nepoužité instrukce a pokusit se optimalizovat nejčastěji používané. Nasbírané informace jsou poskytovány v několika formátech včetně velmi přehledného `HTML` souboru.

Syntetizovatelné RTL Ekvivalence mezi simulační a hardwarovou reprezentací je zaručena použitím stejných algoritmů pro jejich generování. Tyto algoritmy jsou navíc založeny na formálních modelech.

Verifikační prostředí Slouží k ověření toho, že generované RTL skutečně implementuje instrukční model. RTL je generováno z modelu časování, díky tomu je nepřímo prověřen i tento model. Prováděna je funkční verifikace založená na UVM. Pro její účely je přítomen balík testů v jazyce `C` a také generátor náhodných aplikací v jazyce symbolických instrukcí. Programy jsou provedeny HDL simulátorem a referenčním simulátorem v podobě instrukčního modelu. Jsou porovnávány výsledky těchto simulací.

[3] Spustitelný simulátorem cílové architektury.

[4] Hojně využívaná i giganty jako jsou Apple a Google.

[5] Například kolik hodinových taktů trvá provedení funkce, použití instrukcí a nejčastěji se objevující sekvence instrukcí.

5.1 Simulátory

Simulátory umožňují testovat vyvíjený hardware i software. Jako ostatní nástroje jsou založeny na modelech v `CodALu`. Simulátory umožňují multiprocesorovou simulaci a průběžný přístup k vnitřním zdrojům procesoru. Jsou podporovány 3 typy simulátorů:

- IA
- CA
- QEMU

IA simulátor slouží k testování implementované ISA a je vytvořen z IA modelu. Veškeré načítání, dekódování a provádění instrukcí je okamžité. CA simulátor slouží k testování mikroarchitektury a vytváří se z CA modelu. Jeho běh se odehrává na úrovni hodinových cyklů. Oba tyto simulátory je možné exportovat ve formě dynamických knihoven a opatřit obálkou pro použití na simulačních platformách jazyka `SystemC`. Pro účely QEMU simulátoru generuje Studio pouze Tiny Code Engine z IA modelu [3]. Jednotlivé instrukce jsou překládány do nativních, což umožňuje velký počet vykonaných instrukcí za sekundu[6]. QEMU simulátor je však omezen na jednoprocesorovou simulaci.

Součástí simulátorů je také debugger. Ladění je možné na úrovni:

Zdrojového kódu Každý krok znamená posun na další řádek zdrojového kódu. Podpora pro `C` i jazyk symbolických instrukcí.

Instrukcí Každý krok znamená posun na další instrukci (v IA módu) nebo na další hodinový takt (CA mód).

Debugger umožňuje výpis posloupnosti volání funkcí i vkládání zarážek[7], u kterých je možné nastavení podmínek aktivace. Je založen na volně dostupné platformě `GDB` a jeho rozhranní je s ní kompatibilní.

[6] V řádech milionů.
[7] Breakpoint.

Kapitola 6

Instrukční model procesoru RISC-V

Prvním z cílů práce je implementace instrukčního modelu procesoru v jazyce `CodAL`. Ta se skládá z popisu jednotlivých instrukcí, základního popisu chování a platformy. Instrukční sada je plně definovaná. Kromě základní ISA v podobě „RV32I“ jsem se rozhodl implementovat rozšíření pro násobení a rozšíření pro 64-bitová data. Kompletní implementovaná ISA je tedy „RV64IM“.

Platforma nutná pro základní instrukční sadu je jednoduchá. Postačí na ni vložit procesor a paměť. Na velikosti paměti a endianitě nezáleží. Šířka adresy je 32 bitů, stejně tak šířka dat a nejmenší adresovatelná jednotka je bajt, tedy 8 bitů. Paměť má dvě rozhranní typu `CLB`. První je pouze pro čtení a slouží pro načítání instrukcí. Druhé je pro čtení a zápis. Slouží pro přístup k datům.

Instrukční model procesoru je založen na dvou blocích typu `event`. `Event reset` slouží k nastavení počátečních hodnot registrů. `Event main` je potom obdobou tradiční funkce `main`. Jeho úlohou je načtení instrukce z paměti, posun programového čítače a zaslání načtené instrukce do dekodéru, kterým je samotný popis ISA[1]. Jelikož tento model nezohledňuje časování, není třeba modelovat čekání na odpověď paměti. Dále je potřeba definovat dostupné zdroje. Základní sada architekturálních registrů se skládá z 31 registrů pro čtení a zápis číslovaných od 1 a speciálního registru `R0` mapovaného na konstantní nulu, do kterého není možný zápis. Šířka registrů je 32 bitů. Kromě architekturálních registrů jsou zapotřebí ještě registry pro uchování hodnoty programového čítače a načtené instrukce. Oba 32-bitové.

Všechny instrukce RISC-V musí být v paměti zarovnány na 32 bitů. Jsou děleny do čtyř hlavních a dvou odvozených formátů uvedených v tabulce 6.1. Je patrné, že u všech typů kódování jsou zdrojové a cílové registry umístěny shodně což usnadňuje implementaci dekodérů[2]. Operační kód má proměnnou délku a jeho jednotlivé části se opět nachází na stejných pozicích. Přímé kódování konstant je v instrukcích mapované směrem k nejvýznamnějšímu bitu. Toto je výhodné zejména u znaménkového bitu, který se vždy nachází na nejvýznamnějším bitu instrukce. Toto umožní významné urychlení znaménkového rozšíření konstant, což je jeden z hlavních přínosů RISC-V. Formát SB se od S liší pouze v automatickém doplnění nuly na nejnižší bitovou pozici. Tento formát je využíván pro podmíněné skoky. Umožní tak doskočit dále a zjednodušit kontrolu zarovnání adresy. Dále je zajímavé, že v kó-

[1] ISA taktéž provede požadované chování instrukce.

[2] Dekódování registrových operandů se obvykle nachází na kritické cestě [17].

31 25	24 20	19 15	14 12	11 7	6 0	formát
operační kód	zdroj 2	zdroj 1	operační kód	cíl	operační kód	R
konstanta[11:5]	[4:0]	zdroj 1	operační kód	cíl	operační kód	I
konstanta[11:5]	zdroj 2	zdroj 1	operační kód	[4:0]	operační kód	S
konstanta[12,10:5]	zdroj 2	zdroj 1	operační kód	[4:1,11]	operační kód	SB
konstanta[31:25]	[24:20]	[19:15]	[14:12]	cíl	operační kód	U
konstanta[20,10:5]	[4:1,11]	[19:15]	[14:12]	cíl	operační kód	UJ

Tabulka 6.1: Formáty instrukcí RISC-V [17].

dování SB nejsou bity přímého kódování pouze posunuty doleva. Namísto toho zůstávají všechny bity kromě druhého nejvýznamnějšího na svých místech, který je posunutý na uvolněnou pozici nultého bitu. Takto zůstává maximální počet bitů na fixních pozicích, což opět zjednodušuje dekódování. Podobně je tomu i s formáty U a UJ, kde U kóduje vrchní bity konstanty a UJ spodní. Následující podkapitoly se věnují rozboru a implementaci jednotlivých skupin instrukcí základní ISA a jejích rozšíření.

6.1 RV32I

Pro každou instrukci je třeba definovat `element`, který ji popisuje. Skupiny instrukcí, které sdílí jeden formát, mají často podobné chování. Proto je výhodné popis jejich chování sloučit do jednoho `elementu`. Aby to bylo možné, je potřeba definovat `elementy` popisující jednotlivé operační kódy a ty potom sloučit do skupiny označené klíčovým slovem `set`. Ten je potom použit v popisu samotné instrukce a zastupuje všechny své členy. Kód 6.1 zachycuje vytvoření operačního kódu pro součet a jeho přidání do `setu` sdružujícího operace používající registrový a konstantní operandy a ukládající výsledek do registru. Mimo operační kódy je potřeba definovat `elementy` reprezentující jednotlivá přímá kódování konstant (ukázka 6.2) a také elementy popisující registry (ukázka 6.3).

```
element opc_addi
{
  assembler { "ADDI" };
  binary { 0x013 };
  return { 0x013 };
};
set opc_comp_2reg_imm += opc_addi;
```

Ukázka kódu 6.1: Vytvoření operačního kódu a přiřazení do setu.

```
element simm12
{
  assembler { val:signed };
  binary { val:bit[12] };
  return { val };
};
```

Ukázka kódu 6.2: Příklad elementu přímého kódování konstanty se znaménkem.

```
element reg1 {
  assembler { "R1" };
  binary { 1:bit[5] };
  return { 1 };
};
```

Ukázka kódu 6.3: Příklad elementu registru.

Složením výše uvedeného dohromady lze vytvořit popis instrukce. V ukázce kódu 6.4 pro jednoduchost uvažujme, že popisuje chování instrukce součtu obsahu registru a konstanty.

```
element i_comp_2reg_imm
{
  use opc_comp_2reg_imm as opc;
  use reg_any as dst, src;
  use simm12;
  assembler { opc dst "," src "," simm12 };
  binary { simm12 src dst opc };
  semantics
  {
      // uvedene funkce zapisuji a ctou z registru
      rf_gpr_write(dst, rf_gpr_read(src) + int32(simm12));
  };
};
```

Ukázka kódu 6.4: Příklad instrukce součtu.

6.1.1 Výpočetní instrukce

Tyto instrukce pracují buď pouze s registry (instrukční formát „R“) nebo s registry a přímo kódovanou konstantou (instrukční formát „I“). Žádná z těchto instrukcí negeneruje výjimky. Přímo kódované konstanty jsou vždy znaménkově rozšířeny. Mezi výpočetní instrukce patří

ADD(I) Instrukce pro součet registrů a registru se znaménkovou konstantou. Přetečení jsou ignorována. Přičtením nuly se emuluje instrukce pro přesun mezi registry.

SUB Instrukce pro odečtení registrů. Není zde varianta s konstantou. Výpůjčka v nejvyšším bitu je ignorována.

SLT(I)(U) Nastavení výsledku porovnání zda je první operand menší než druhý. „I“ znamená porovnání s konstantou, „U“ porovnání bez znaménka. Výsledkem je 1 nebo 0. Použití SLTIU s operandem 1 zajišťuje porovnání na nulu.

AND(I),OR(I),XOR(I) Logický součin, součet a exkluzivní disjunkce. Díky znaménkovému rozšíření lze XOR s -1 využít k negaci bitů.

SLL(I),SRL(I),SRA(I) Instrukce pro logické posuny doleva a doprava a pro aritmetický posun doprava[3]. Posouvají první operand. Velikost posunu je uložena ve spodních pěti bitech druhého operandu.

LUI Načtení konstanty do vrchní části registru. Konstanta je posunuta o 12 b doleva a uložena do registru. Spodní bity jsou tedy nastaveny na 0.

AUIPC Instrukce obdobná LUI. Posune konstantu doleva o 12 b, přičte k ní obsah programového čítače a výsledek uloží do registru. Lze ji použít k relativnímu adresování.

NOP Operace, která nijak nemění obsah registrů ani paměti. Efektivně tedy nedělá nic, pouze posouvá programový čítač. V RISC-V je modelována jako ADDI R0, R0, 0.

Implementace těchto instrukcí je velmi přímočará. Spočívá pouze v popisu jejich kódování a chování v jazyce `C`. Je třeba si pouze dát pozor na používání správných datových typů.

6.1.2 Skokové instrukce

RISC-V poskytuje podmíněné a nepodmíněné skoky. Využívají formátů „U“, „S“ a „SB“. Nepodmíněné skoky ukládají adresu následující instrukce do cílového registru. Jsou to instrukce JAL a JALR. JALR je absolutní[4] skok na obsah registru, ke kterému je přičtena 12-bitová znaménková konstanta. JAL je relativní skok[5]. 20-bitová konstanta v této instrukci je posunuta o bit doleva, poskytuje tedy dosah skoku ± 1 MiB. Oba nepodmíněné skoky podporují pozičně nezávislý kód, jelikož instrukci JALR lze použít v páru s AUIPC.

Podmíněné skoky jsou provedeny na základě porovnání obsahu dvou registrů. Skok je relativní na adresu danou 12-bitovou znaménkovou konstantou posunutou o 1 bit doleva, dosah skoku je tedy ± 4 KiB. Jsou k dispozici porovnání na rovnost (BEQ), nerovnost(BNE), menší (BLT) a větší nebo rovno (BGE). Poslední dva skoky nabízí neznaménkové porovnání (BLTU a BGEU). Chybějící kombinace porovnání lze emulovat prohozením operandů.

Přímé kódování konstanty použité pro relativní adresování podmíněných skoků a JAL vyžaduje speciální syntaxi. Je nutné dát překladači jazyka symbolických instrukcí informaci o tom, že adresu cíle skoku je potřeba přepočítat na adresu relativní k pozici instrukce. Ukázka kódu 6.5 přináší definici `elementu` relativní adresy pro instrukci JAL.

Sekce `assembler` určuje transformaci při převodu do binárního kódu. Od adresy cíle je tedy nutné odečíst adresu instrukce a navíc ji posunout o bit doprava, protože při jejím dekódování bude přidána nula na pozici nejméně významného bitu. Sekce `binary` naopak slouží disassembleru pro převod binárního kódu do jazyka symbolických instrukcí. Provedené operace je tudíž potřeba obrátit. Tento přístup vede k tomu, že ve zdrojovém kódu se objeví skutečný cíl skoku, zatímco binární kódování bude takové, jako vyžaduje ISA.

[3] Zachovává znaménkový bit.
[4] Obsah programového čítače je modifikován nezávisle na jeho hodnotě.
[5] K programovému čítači je přičtena konstanta.

```
/// 20-bitova relativni adresa z formatu UJ
element rel_addr20
{
  assembler
  {
    val:label,signed { val = val - current_address >> 1; }
  };
  binary { val:bit[20] { val = (val << 1) + current_address; } };
  return { val };
};
```

Ukázka kódu 6.5: Přímé kódování konstanty pro relativní adresování.

6.1.3 Načítací a ukládací instrukce

Pouze tato skupina instrukcí přímo pracuje s pamětí. „RV32I“ poskytuje 32-bitový adresový prostor adresovatelný po bajtech. Data jsou ukládána dle principu `little-endian`[6]. Adresa pro načítání respektive ukládání je získána z hodnoty uložené v registru. K ní je přičtena 12-bitová znaménková konstanta. Ukládací instrukce jsou ve variantě pro uložení: bajtu (SB), poloviního slova (SH) a celého slova (SW). Varianty, které nepracují s celými slovy, ukládají spodní části hodnot v registrech. Podobně jsou rozděleny i načítací instrukce. Načtené hodnoty mohou a nemusí být znaménkově rozšířené. Jsou to instrukce: LB, LBU[7], LH, LHU a LW. Pro dosažení dobré výkonnosti je potřeba zarovnat paměťové přístupy na velikost slova, respektive půlslova [17].

Přístup do paměti, která je na platformě je proveden skrze volání funkcí `read` a `write` protokolu `CLB`. Tato paměť vyžaduje přístup zarovnaný na velikost slova. Umožňuje však přístup k jednotlivým po sobě jdoucím bajtům slova díky parametrům `SBI`[8] a `SBC`[9]. Pro implementaci přístupů na celá slova postačí nastavit `SBI` na 0 a `SBC` na počet bajtů ve slově, tedy 4. Pro přístupy na části slov jsou spodní dva bity transformovány na `SBI` a nahrazeny nulami. Před samotným přístupem do paměti simulační model kontroluje, zda je kombinace `SBI` a `SBC` validní[10].

6.1.4 Synchronizační instrukce

Základní ISA RISC-V podporuje běh více vláken v rámci jednoho uživatelského adresového prostoru. Každé vlákno má své hodnoty registrů a provádí nezávislý proud instrukcí. Vytváření vláken a jejich správu má na starosti operační systém. Jednotlivá vlákna spolu komunikují skrze jeho volání. Každé vlákno vidí následky jím provedených instrukcí, jako by byly provedeny v programovém pořadí. Aby bylo možné stejným způsobem pozorovat následky instrukcí jiných RISC-V vláken je nutné použít instrukci FENCE. Ta garantuje stejnou viditelnost vstupně-výstupních a paměťových operací pro všechna vlákna. Tuto garanci je možné omezit na jakoukoliv kombinaci: vstup, výstup, čtení z paměti a zápis do ní.

[6] Na nejnižší adresu se ukládá nejméně významný bajt a za něj se ukládají další bajty až po ten nejvýznamnější.

[7] Varianta bez znaménkového rozšíření.

[8] Sub-block Index. Index prvního požadovaného bajtu.

[9] Sub-block Index. Počet požadovaných bajtů.

[10] Validní kombinace SBI:SBC jsou: 0:4, 0:2, 0:1, 1:1, 2:2, 2:1 a 3:1.

Druhou synchronizační instrukcí je FENCE.I, která slouží k synchronizaci proudu instrukcí a dat. Obecně není garantováno, že uložení dat do instrukční paměti bude viditelné pro jednotku načítající instrukce. Důvodem je možná přítomnost dat v instrukční paměti cache během zápisu[11]. FENCE.I garantuje, že načtení jakékoliv následující instrukce bude brát v úvahu všechny doposud provedené zápisy.

Prostředí Codasip Studia prozatím neumožňuje běh více vláken na jednom procesoru, proto je popis chování instrukce FENCE prázdný. Samotná instrukce je definována pouze kvůli binární kompatibilitě s RISC-V ISA. Požadovaného chování FENCE.I lze dosáhnout zneplatněním obsahu instrukční paměti cache. Přestože platforma prozatím neobsahuje paměti cache, příkaz pro zneplatnění je sběrnicí `CLB` podporován a proto je možné instrukci implementovat. Paměť na platformě na daný příkaz nereaguje.

6.1.5 Systémové instrukce

Systémové instrukce umožňují volání privilegovaných systémových funkcí a přístup k hardwarovým čítačům. RISC-V definuje 64-bitové čítače hodinových cyklů a instrukcí provedených daným vláknem. Dále definuje čítač reálného času[12]. K jejich hodnotám je nutné přistupovat nadvakrát, jelikož šířka dat je pouze 32 b. K přístupu k těmto čítačům slouží instrukce RDCYCLE(H)(čítač cyklů), RDTIME(H)(čítač reálného času) a RDINSTRET(H) (čítač instrukcí). Pro získání jejich celé hodnoty je nutné nejprve načíst vrchní polovinu, poté spodní, následně opět vrchní a obě vrchní poloviny porovnat. Pokud se rovnají, byla načtena správná hodnota, jinak je nutné celý postup opakovat. Tyto čítače nemají pro instrukční model význam a proto implementace instukcí pro jejich čtení pouze zapisuje do cílového registru nulu.

Pro volání systémových funkcí slouží SBREAK a SCALL. SBREAK je instrukce pro debuggery, která má vrátit řízení debuggovacímu prostředí. Toto chování také prozatím není Studiem podporováno a proto je sémantická sekce instrukce prázdná. SCALL je voláním funkce operačního systému. ABI operačního systému určuje jakým způsobem jsou mu předány parametry volání. Codasip Studio umožňuje emulaci systémových volání pomocí funkce `codasip_syscall()`. Jako parametr přijímá adresu struktury, která obsahuje informace o požadované akci a její parametry[13].

6.2 RV32M

Rozšíření pro dělení a násobení nevyžaduje modifikaci zdrojů procesoru, pouze doplňuje další instrukce. Všechny pracují pouze s obsahy architekturálních registrů.

6.2.1 Instrukce pro násobení

Výsledek násobení dvou 32-bitových hodnot vyžaduje až 64 bitů. Pro získání spodní poloviny výsledku slouží instrukce MUL. U vrchní poloviny záleží na znaménkách operandů. Proto jsou k dispozici varianty instrukcí, které chápou své operandy jako dvojice: znaménkový a znaménkový (MULH), neznaménkový a neznaménkový (MULHU) a znaménkový a neznaménkový (MULHSU). Pokud je vyžadován celý 64bitový výsledek je doporučená

[11] Koherence mezi instrukční a datovou cache není zaručena.

[12] Perioda hodin tohoto čítače by neměla být delší než 100 ns [17].

[13] Například pro volání zápisu je to deskriptor souboru, ukazatel na řetězec a jeho délka.

sekvence instrukcí: MULH((S)U), MUL [17]. Tento přístup umožní upravit mikroarchitekturu pro provedení obou instrukcí současně.

Implementace těchto instrukcí je podobná výpočetním. Jelikož výsledek násobení je možné uložit do proměnné datového typu `long long`, který lze následně posunout, či vymaskovat pro získání jednotlivých polovin, není potřeba využívat žádných speciálních algoritmů.

6.2.2 Instrukce pro dělení

RISC-V poskytuje instrukce pro získání podílu a zbytku a to ve znaménkových (DIV a REM) a neznaménkových variantách (DIVU a REMU). Podobně jako u násobení je doporučena sekvence instrukcí pro získání obou hodnot. Je to: DIV(U) a REM(U) [17]. Dělení nulou ani přetečení[14] nevyvolávají výjimky. Sémantika těchto událostí je definována tabulkou 6.2.

	Dělenec	**Dělitel**	**DIVU**	**REMU**	**DIV**	**REM**
Dělení nulou	x	0	$2^{32}-1$	x	-1	x
Přetečení	-2^{31}	-1	Nemůže nastat		-2^{31}	0

Tabulka 6.2: Sémantika dělení nulou a přetečení [17].

Implementace v jazyce `CodAL` je opět přímočará.

6.3 RV64IM

Rozšíření „RV64IM“ mění šířku architekturálních registrů, programového čítače a adresovatelného prostoru na 64 bitů. Dochází ke změně chování doposud definovaných instrukcí, které nyní pracují se 64-bitovými hodnotami a produkují stejně široké výsledky. Dále jsou přidány nové instrukce především pro přístup ke spodní polovině 64-bitových slov. Aby bylo možné vytvářet instrukce pro dvě různé datové šířky, je nutné definovat další architekturální zdroj. Tím jsou 32 -bitové registry, které překrývají architekturální 64-bitové. Definice takových zdrojů se nachází v ukázce 6.6.

```
register_file bit[64] rf_gpr {
  size = 32;
};
register_file bit[32] rf_gpr32 {
  size = 32;
  overlap = rf_gpr[0][31..0];
};
```

Ukázka kódu 6.6: Definice architekturálních registrů.

Jelikož se zvětšila šířka dat, je nutné modifikovat paměť na platformě. Šířka obou rozhranní paměti musí být stejná. Instrukce si však zachovávají šířku 32 b. Je tedy třeba přidat komponentu provádějící přemostění mezi různými šířkami dat. Pro řešení tohoto problému lze s výhodou použít instrukční paměť cache.

[14] Dělení nejnižší možné hodnoty hodnotou -1.

6.3.1 Výpočetní instrukce

Pro přístup ke spodním 32 bitům jsou přidány instrukce s příponou „W“. Ty ignorují vrchních 32 b a produkují 32-bitové výsledky znaménkově rozšířené na 64 b. Nové a upravené instrukce jsou:

ADDIW, ADDW, SUBW Součet či odečtení spodních 32 b s 12bitovou znaménkovou konstantou. Výsledek je znaménkově rozšířen na 64 b. Přetečení jednatřicátého bitu je ignorováno. Použitím ADDIW s konstantou 0 se získá znaménkové rozšíření hodnoty.

SLL, SRL, SRA, SLLI, SRLI, SRAI Namísto spodních 5-ti bitů se pro hodnotu posunutí použije spodních 6.

SLLW, SRLW, SRAW, SLLIW, SRLIW, SRAIW Spodních 32 b prvního operandu se posouvá o hodnotu určenou spodními 5-ti bity druhého operandu. Spodních 32 b výsledku je znaménkově rozšířeno na 64 b.

LUI, AUIPC Stále nahrávají konstantu do bitů 12 až 31. Opět znaménkově rozšiřují.

Implementace těchto změn a dodatků spočívá pouze ve změně datových typů, důsledném přetypovávání a maskování.

6.3.2 Načítací a ukládací instrukce

Tyto instrukce byly doplněny o možnost načítat (LD) a ukládat (SD) 64-bitové hodnoty. Byla přidána také neznaménková varianta načtení 32-bitového slova (LWU). Implementace si vyžádala modifikaci paměťových přístupů. Ty je potřeba zarovnávat na 8 B. Dále je třeba přidat nové povolené kombinace `SBI` a `SBC`[15].

6.3.3 Systémové instrukce

Šířka hardwarových čítačů zůstává stejná. Přístup k celkové hodnotě však nyní lze provést pomocí jediné instrukce. Instrukce pro přístup k vrchní polovině čítačů jsou tedy v „RV64“ zbytečné.

6.3.4 Instrukce pro dělení a násobení

Opět jsou doplněny instrukce pro práci se spodní půlkou registrů: DIV, DIVUW, REMW, REMUW a MULW. MULW poskytuje znaménkově rozšířenou spodní polovinu výsledku násobení spodních polovin operandů. Rozšíření dat na 64 bitů má za následek že výsledek je třeba vyjádřit 128 bity. K tomu už běžné datové typy nestačí a proto je potřeba zvolit jiný způsob implementace instrukcí pro získání vrchních 64 bitů výsledku. Lze využít vzorce: $a \cdot b = (a_1 + a_2) \cdot (b_1 + b_2)$. Kde a_1, b_1 a a_2, b_2 jsou čísla reprezentovaná vrchní, respektive spodní polovinou bitů. Kód 6.7 zobrazuje funkci pro získání hodnot MULH((S)U).

[15] Tyto kombinace jsou: 0:8, 4:1, 4:2, 4:4, 5:1, 6:1, 6:2 a 7:1.

```
uint64 multiply(const uint64 op1, const uint64 op2)
{
  uint64 tmp64, op1_l, op1_h, op2_l, op2_h;
  uint32 tmp32_1, tmp32_2;
  op1_l = (uint32)op1;
  op1_h = op1 >> 32;
  op2_l = (uint32)op2;
  op2_h = op2 >> 32;
  // a1 * b2 tvori cast bitu 95-32 vysledku
  tmp64 = op1_h * op2_l;
  // soucin spodnich polovin ovlivnuje bity 63-0,
  // z nich je dulezita vrchni polovina
  tmp64 += ((op1_l * op2_l) >> 32);
  // uchovani mezivysledku bitu 63-32 a 95-64
  tmp32_1 = tmp64;
  tmp32_2 = tmp64 >> 32;
  // a2 * b1 tvori dalsi cast bitu 95-32 vysledku
  tmp64 = op1_l * op2_h;
  // pricteni mezivysledku 63-32
  tmp64 += tmp32_1;
  // v tuto chvili je znam vliv spodnich polovin operandu
  // a1 * a2 tvori cast bitu 128-65 vysledku
  // k teto hodnote se prictou mezivysledky bitu 95-64
  tmp64 = op1_h * op2_h + tmp32_2 + (tmp64 >> 32);
  return tmp64;
}
```

Ukázka kódu 6.7: Získání vrchní poloviny výsledku 64-bitového násobení.

6.4 Generování překladače

Aby bylo možné Codasip Studiem vygenerovat překladač jazyka `C` je potřeba doplnit model o několik dalších konstrukcí.

6.4.1 Compiler aliasy

LLVM nepodporuje výpočet cílové adresy skoku z více jak jednoho operandu. Díky tomu je instrukce JALR použita pouze s nulovou konstantou, nebo se zdrojovým registrem R0. Je tedy vhodné pro tyto dva případy vytvořit `assembler` a `compiler` aliasy. RISC-V neobsahuje žádné přímé[16] absolutní skoky. Respektive takový skok je možný pouze prostřednictvím JALR, kde zdrojový registr je R0. Potom lze využít 12 bitů přímo kódované konstanty pro přímý absolutní skok na nejnižších a nejvyšších 512 B paměti. Pro pokrytí větší části paměti je možné využít přímého relativního skoku JAL. Vytvořením `compiler` aliasu se dá popsat chování, které sémantický extraktor rozpozná jako požadovaný skok, zatímco `element` pro relativní adresování se postará o správné přepočtení konstanty pro skok na požadované místo. Skok z ukázky kódu 6.8 je již dostatečný pro běžné použití překladače.

[16] Skok na konstantu.

```
element i_jal_abs : compiler_alias(i_jal)
{
  use opc_jal as opc;
  use reg_any as dst;
  use rel_addr20 as addr;
  assembler { opc dst "," addr };
  binary { addr dst opc };
  semantics
  {
      rf_gpr_write(dst, r_PC);
      r_PC = (int64)((int21)addr << 1);
  };
};
```

Ukázka kódu 6.8: Absolutní skok pomocí relativního.

Přestože několik základních instrukcí pro práci s 32-bitovými hodnotami je v ISA definováno, není jich dostatek pro uznání `i32` jako legálního datového typu. Řešením je vytvoření skupiny `compiler` aliasů definujících požadované chování. Je nutné je vytvořit pro načítací a ukládací instrukce. Pro načtení slova je využita varianta se znaménkovým rozšířením a to z důvodu, že přestože překladač uvidí registr jako 32-bitový, ve skutečnosti bude hodnota zapsána do 64-bitového architekturálního registru, proto je nutné zachovat znaménko. Dále je třeba doplnit aliasy pro podmíněný skok na základě porovnání 32-bitových registrů[17], nastavení příznaku na základě jejich porovnání[18] a logické operace na nich.

6.4.2 Pseudoinstrukce

Pseudoinstrukce poskytují možnost poradit překladači užití instrukce, či sekvence instrukcí v případě, že se nedaří extrakce požadovaného chování z běžného popisu. Jedná se o zápis ve formě, v jaké se instrukce nacházejí po extrakci. Tento popis se skládá z:

- názvu instrukce a seznamu operandů (hodnoty v registrech, konstanty),
- zápisu sémantiky ve formátu vyžadovaném LLVM,
- řetězce jazyka symbolických instrukcí, který bude použití této intrukce generovat.

V RISC-V je nutné použít pseudoinstrukce pro modelování načítání 32 a 64-bitových konstant do registrů. Generátor překladače je obvykle schopen potřebnou posloupnost instrukcí nalézt sám, avšak všudypřítomné znaménkové rozšiřování tento proces komplikuje. Na běžných architekturách se načtení 32-bitové konstanty provádí pomocí kombinace LUI a ORI. Díky znaménkovému rozšíření konstanty by však ORI při nastaveném znaménkovém bitu způsobilo modifikaci hodnoty nahrané pomocí LUI. Kvůli tomuto problému je nutná delší sekvence operací:

1. Nahrání vrchních 20 bitů konstanty do cílového registru pomocí LUI.

[17] Díky znaménkovým rozšířením bude znaménkové i neznaménkové porovnání vždy fungovat korektně.
[18] Zde je důležité, aby výsledek porovnání byl zapsán do 64-bitového registru.

2. Nahrání spodních 12 bitů konstanty do pomocného registru pomocí ORI. Zrušení nastavených znaménkových bitů posunutím doleva o 52 b a následně doprava o stejnou hodnotu.

3. Spojení obou hodnot pomocí OR.

Tento postup je možné použít pro načtení 32-bitové hodnoty do 32 i 64-bitového registru. Postup nahrávání libovolné konstanty je ještě složitější. Vychází až na 14 instrukcí a je popsán v ukázce kódu 6.9.

```
instr load_imm64, ok,
{ regs_0 = regop(regs), imm_1 = immop() },
// semanticka sekce
regs_0 = i64 imm_1;,
// cilovy assembler
"lui" regs_0 "," imm_1 ">> 44\n\t"
"or r28, r0," imm_1 ">>32 & 0xfff\n\t"
"sll r28, r28, 52\n\t"
"srl r28, r28, 52\n\t"
"or" regs_0 "," regs_0 ", r28\n\t"
"sll" regs_0 "," regs_0 ", 32 \n\t"
"lui r28," imm_1 ">> 12 &0xfffff\n\t"
"or r29, r0," imm_1 "& 0xfff\n\t"
"sll r29, r29, 52\n\t"
"srl r29, r29, 52\n\t"
"or r28, r28, r29\n\t"
"sll r28, r28, 32 \n\t"
"srl r28, r28, 32 \n\t"
"or" regs_0 ","regs_0 ", r28 \n"
```

Ukázka kódu 6.9: Pseudoinstrukce pro nahrání 64-bitové konstanty.

Kromě nahrávání konstant zbývá už jen poskytnutí instrukce pro konverzi 64-bitové hodnoty na 32-bitovou. K tomu lze použít stejně jako ke znaménkovému rozšíření ADDIW s nulovou konstantou. Po doplnění těchto aliasů a pseudoinstrukcí je možné vygenerovat překladač jazyka `C`.

6.4.3 Optimalizační průchody překladače

Další možností jak přímo ovlivnit kvalitu překladačem generovaného kódu jsou uživatelsky definované optimalizační průchody. Jde o kód v jazyce `C++` operující nad takzvanými základními bloky programu. Tyto bloky je možné libovolně modifikovat, takové jednání však znamená převzetí zodpovědnosti za korektní chování výsledného programu. Každý základní blok obsahuje informace dostupné během kompilace jako jsou například hodnoty konstant. Toho lze využít k optimalizaci způsobu načítání konstant pomocí pseudoinstrukcí popsaném v předchozí kapitole. Metodu však lze použít pouze pro konstanty, nikoliv pro hodnoty závisející na návěštích, jelikož pozice kódu není v takto rané fázi překladu známa. Pro svou implementaci jsem se zaměřil na úpravu bloků pro načítání konstant, jako je `load_imm64`. Díky znalosti konkrétní načítané konstanty je možné obejít bitové posuny tím, že místo instrukce ORI využijeme ADDI. V případě, že je 12-tý LSB konstanty u instrukce ADDI

nastaven na 1, provede se odečtení této hodnoty. S tímto faktem se dá počítat a namísto původní hodnoty načíst pomocí předchozí instrukce LUI hodnotu o 0x1000 větší. Odečtením pomocí ADDI je získána požadovaná hodnota pomocí dvou instrukcí namísto pěti. Je třeba zvážit všechny kombinace, které mohou nastat při načítaní 64-bitové konstanty. Znaménkové bity mohou být nastaveny u dvou instrukcí ADDI a u jedné LUI. Zvýšení hodnoty konstanty u dílčí instrukce může navíc vést k přetečení její hodnoty, nebo nastavení znaménkového bitu. Průchodem celého stavového prostoru, při kterém jsem analyzoval nejkratší možné posloupnosti instrukcí pro načtení hodnoty jsem došel k šedesáti variantám. Nejdelší se skládají z šesti instrukcí a tedy jsou ekvivalentní ke konzervativní posloupnosti: LUI, ORI, SLLI, LUI, ORI, OR. Ukázka kódu 6.10 zobrazuje interní popis instrukce zahrnující největší počet transformací a jeho použití v optimalizačním průchodu. Většina transformací je použitelná pro interval hodnot, nikoliv jen pro jednu hodnotu. Daná posloupnost slouží k načtení konstanty `0xXXXXX7ff7ffffXXX`, kde X značí libovolnou hodnotu.

```
def load_imm64_lui_inc_add_inc_sll_lui_inc_add_add:
    CodasipMicroClass_<(outs regs:$op0), (ins i64imm:$op1)>
{
    let AsmString = "lui␣$op0,␣((($op1␣>>␣44)␣+␣1)␣&␣0xfffff\n"
        "add␣$op0,␣$op0,␣((($op1␣>>␣32)␣+␣1)␣&␣0xfff\n"
        "sll␣$op0,␣$op0,␣32\n"
        "lui␣r28,␣((($op1␣>>␣12)␣+␣1)␣&␣0xfffff\n"
        "add␣r28,␣r28,␣$op1␣&␣0xfff\n"
        "add␣$op0,␣$op0,␣r28\n";
    let Size = 24;
    let isReMaterializable = 1;
    let mayLoad = 0;
    let mayStore = 0;
    let isMoveImm = 1;
}
const MCInstrDesc &ImmLoadLuiIncAddIncSllLuiIncAddAdd =
    get(Codasip::load_imm64_lui_inc_add_inc_sll_lui_inc_add_add);
for(MachineBasicBlock::iterator MII = MBB->begin(),
    MIE = MBB->end(); MII != MIE; ++MII)
{
    if ((MII->getOpcode() == Codasip::load_imm64) &&
        ((MII->getOperand(1).getImm() & 0xfff80000000) ==
        0xfff80000000))
    {
        MII->setDesc(ImmLoadLuiIncAddIncSllLuiIncAddAdd);
    }
}
```

Ukázka kódu 6.10: Úprava základního bloku.

6.5 Testování

Pro základní otestování instrukčního modelu jsem nejprve vytvořil krátký program v jazyce symbolických instrukcí, který využívá všechny implementované instrukce. Pomocí krokování

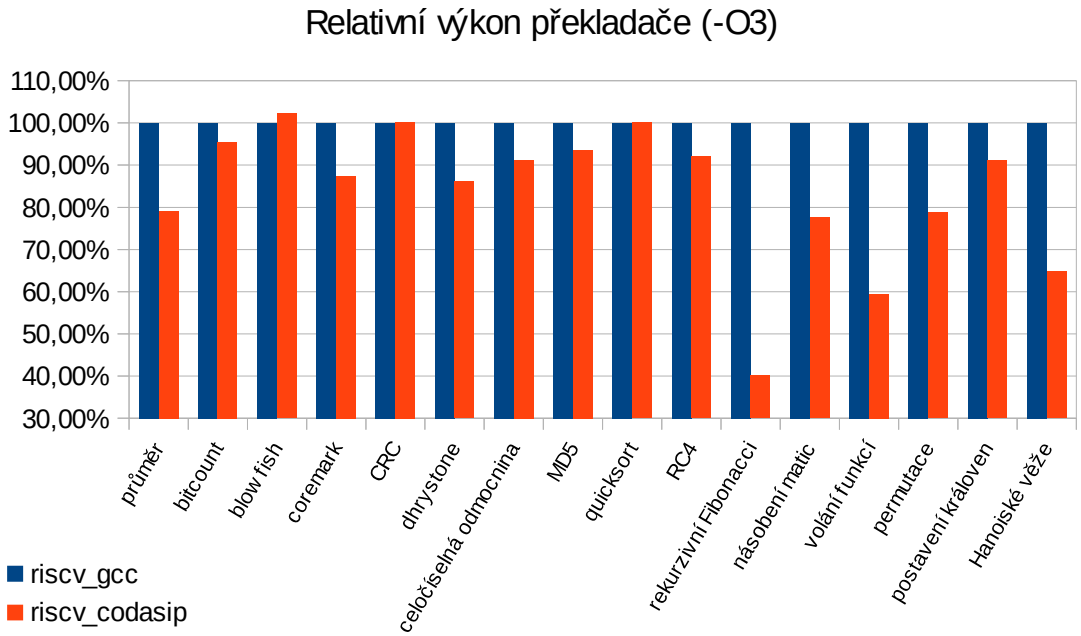
v ladícím prostředí Codasip Studia jsem se ujistil, že chování instrukcí souhlasí s jejich specifikací. Stavový prostor kombinací instrukčních operandů je však poměrně rozsáhlý pro ruční testování a některé nesrovnalosti mohou uniknout. Pro podrobnější testování jsem využil sadu 704 aplikací v jazyce `C` dodaných společností Codasip, určených pro všeobecné otestování překladače. Návratový kód programu je nasteven na 0 pouze v případě úspěšného provedení, což umožňuje pohodlné automatické testování. Celou sadu jsem otestoval se všemi stupni optimalizací od „-O0“ do „-O3“. Všechny testy kromě jediného byly provedeny úspěšně. Neúspěšný test se nepodařilo přeložit a to z důvodu pokusu o vytvoření příliš velkého statického pole[19].

Po ověření funkčnosti překladače jsem přešel k jeho porovnání s `GCC` dodávaným tvůrci RISC-V. Aby bylo možné je věrohodně porovnat je potřeba zajistit pro oba stejné podmínky, tedy testování na stejném simulátoru. Kód produkovaný RISC-V `GCC` je však spustitelný pouze na simulátoru `SPIKE`. Přímé spuštění na simulátoru Codasip Studia tedy není možné, protože spustitelný soubor není v kompatibilním formátu. Samotná datová část souboru obsahující posloupnost instrukcí však kompatibilní být musí, jelikož binární kódování je dáno ISA. Možným řešením je použít `GCC` pouze pro překlad do jazyka symbolických instrukcí a výsledné soubory přeložit nástroji Codasip Studia. Zjistil jsem však, že zdrojové kódy v jazyce symbolických instrukcí generovaném `GCC` prochází během práce linkeru masivní úpravou a samy o sobě nejsou použitelné. Navíc by tento přístup nezohlednil rozdíly v linkerech a tím by porovnání celých překladačů nebylo objektivní. Rozhodl jsem se tedy pro transformaci hotového spustitelného souboru pro `SPIKE` do podoby spustitelné simulátorem Studia. Nejprve jsem pomocí nástroje `objdump` zjistil názvy sekcí označených jako `ALLOC`[20]. Tyto sekce jsem extrahoval v binárním formátu pomocí nástroje `objcopy` a skrze direktivy „.incbin“ je vložil do nově vytvořeného zdrojového kódu v jazyce symbolických instrukcí. Tento kód tedy obsahuje pouze názvy sekcí a vkládání extrahovaných binárních souborů. Zdrojový kód v této podobě mohou nástroje Codasip Studia přeložit do souboru spustitelného na jeho simulátoru. Pro porovnání jsem vždy simuloval program přeložený oběma překladači a kontroloval počet provedených instrukcí a velikost generovaného kódu[21]. Testování probíhalo na sadě k tomuto účelu běžně používaných problémů, jako jsou: Coremark, MD5, CRC a Dhrystone. Programy byly překládány s optimalizací „-O3“ pro zjištění maximálního výkonu překladačů a poté s „-Os“ pro vypnutí optimalizací, které zvětšují kód. Získané hodnoty jsem zanesl do grafu ve formě relativního porovnání. RISC-V `GCC` je považováno za referenci, proto je nastaveno na 100 %. V grafu pro porovnání výkonu znamená větší hodnota lepší výkon. U porovnání velikosti kódu je naopak lepší nižší hodnota. Naměřené výsledky jsou zaneseny v grafech na obrázcích 6.1, 6.2, 6.3 a 6.4.

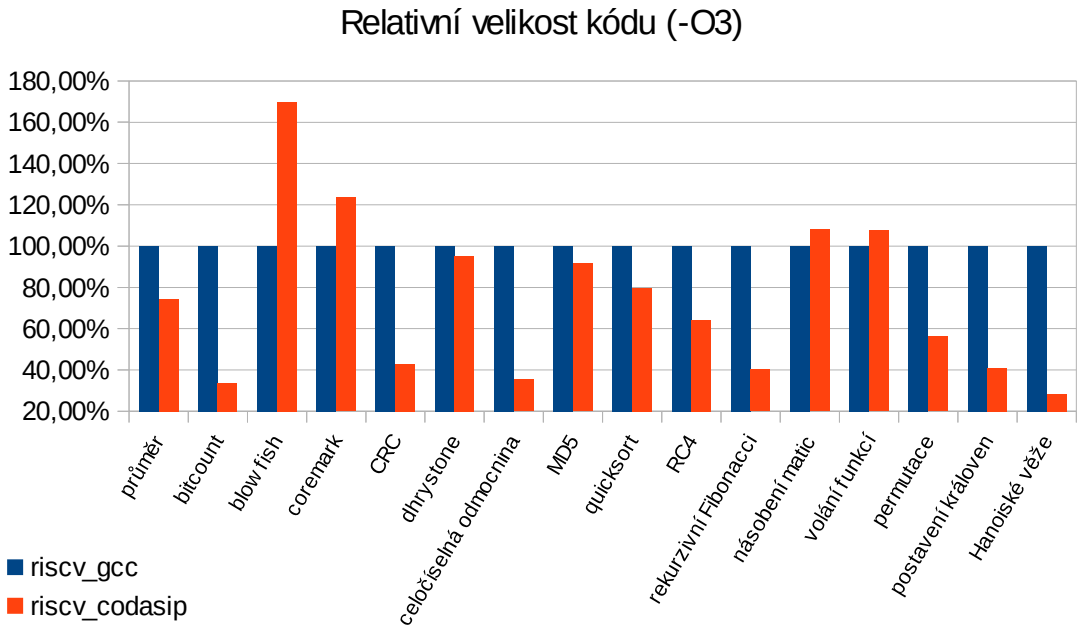
[19] Velikost pole byla odvozena od velikosti datového typu `size_t`. Požadovaný počet položek byl přibližně $7,5 \cdot 10^{15}$.

[20] Sekce, které jsou v paměti při běhu programu.

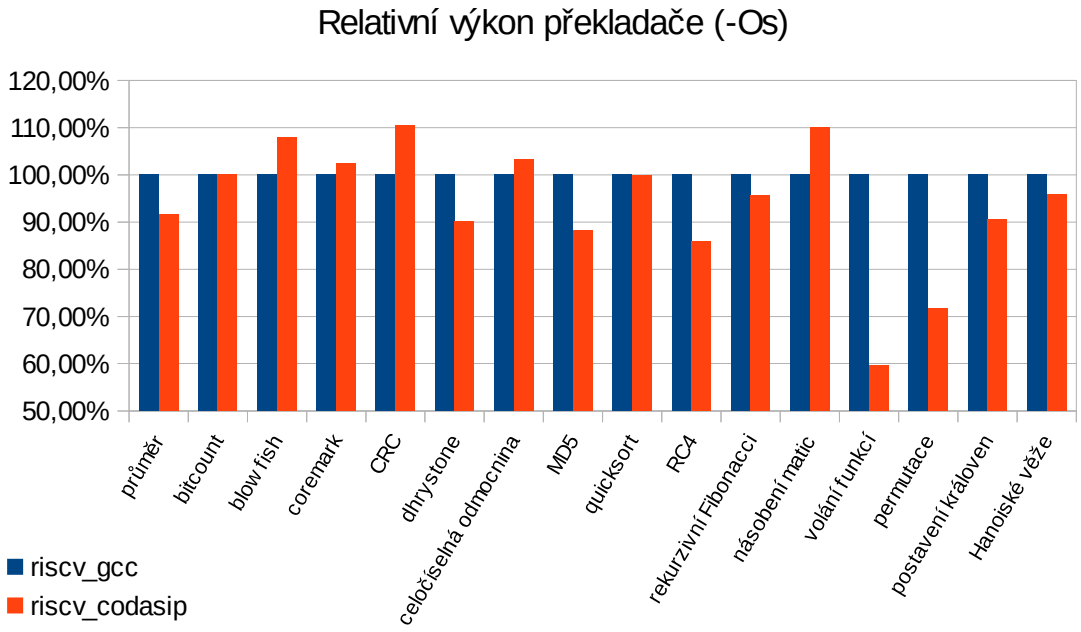
[21] Velikost sekce `.text`.



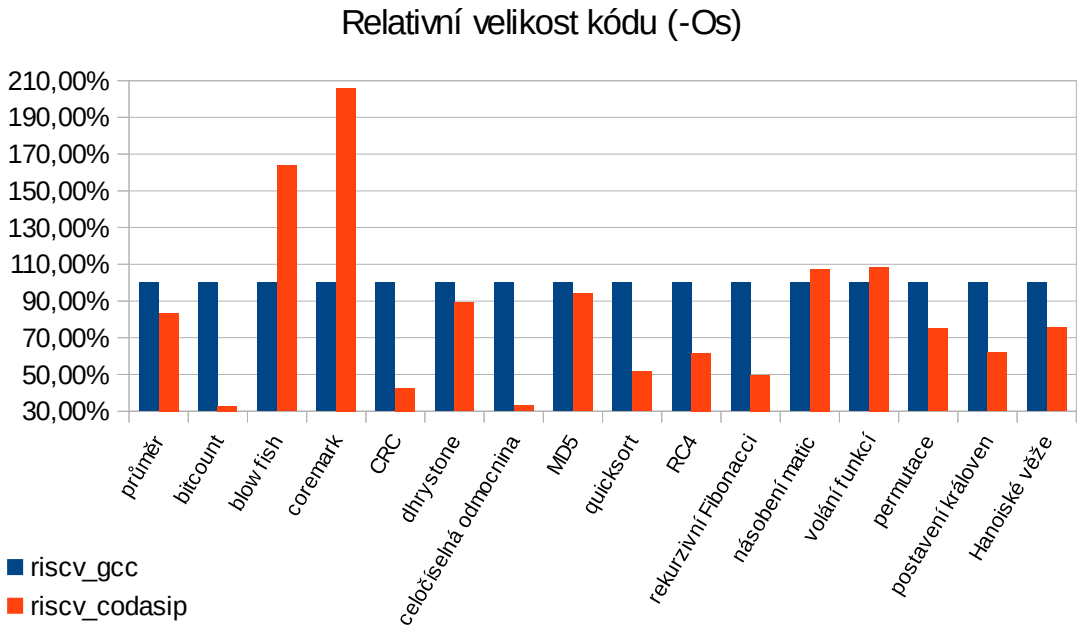
Obrázek 6.1: Relativní výkon překladačů při optimalizacích „-O3“.



Obrázek 6.2: Relativní velikosti kódu generovaného překladači při optimalizacích „-O3“.



Obrázek 6.3: Relativní výkon překladačů při optimalizacích „-Os“.



Obrázek 6.4: Relativní velikost kódu generovaného překladači při optimalizacích „-Os“.

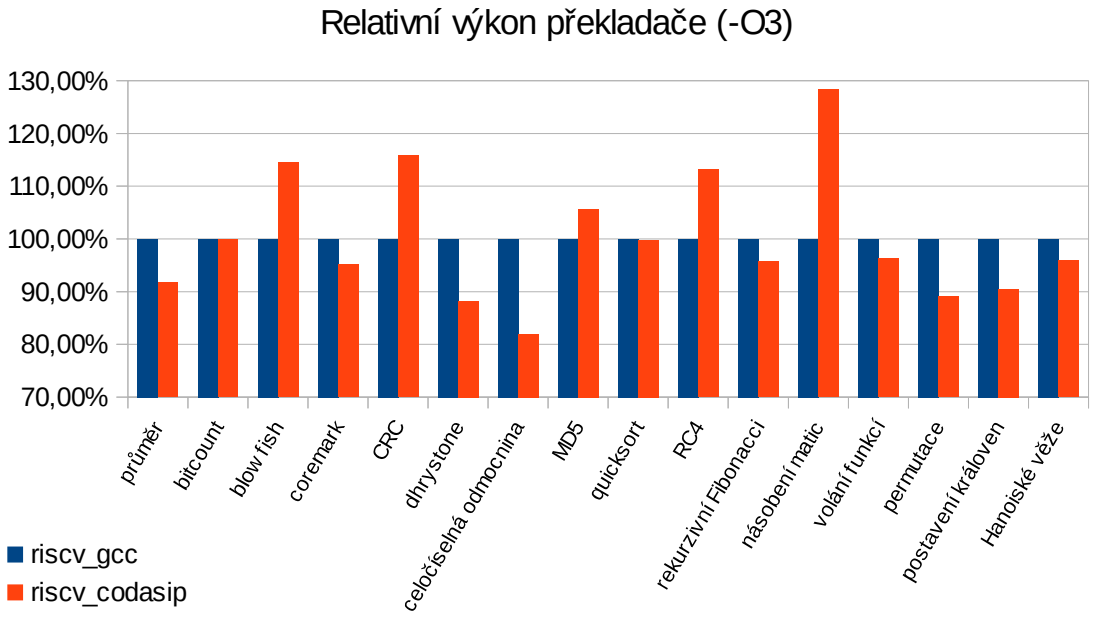
6.6 Optimalizace instrukční sady

Z naměřených hodnot vyplývá, že RISC-V `GCC` produkuje o přibližně 26 % výkonější kód, zatímco kód generovaný Codasip Studiem je o 23 % menší. Větší velikost kódu `GCC` je zřejmě způsobena rozsáhlejšími runtime knihovnami. U testů zaostávajích za výkonem `GCC` jsem analyzoval kódy v jazyce symbolických instrukcí. Přetrvává problém načítání ukazatelů na návěští, který `GCC` řeší skrze speciální relokace v linkeru. Toto v současné době není pomocí nástrojů Codasipu možné. `GCC` často volí pro načítání konstant přístup přes globální ukazatel. Načte tak konstantu z paměti jedinou instrukcí LOAD, která pracuje s 12-ti bitovým offsetem. Globální ukazatel tak umožní přístup k 256-ti 64-bitovým konstantám. Práce s globálním ukazatelem však také není Codasip Studiem umožněna. Ze srovnání testování s optimalizacemi „-O3“ a „-Os“ je patrné, že překladač `GCC` dokáže při některých testech najít v kódu vzorce, které za cenu nárůstu velikosti kódu razantně zvýší výkon.

Po konfrontaci s limity nástrojů jsem se rozhodl pro experimentování s doplněním instrukční sady o nové instrukce. Pro lokalizaci slabých míst instrukční sady jsem použil opětovných simulací se zapnutými profilovacími nástroji. Tyto nástroje umožňují sledovat četnosti posloupností instrukcí zadané délky. Volil jsem posloupnosti délky 2, 3 a 4. Delší posloupnosti by vedly na přílišnou specializaci s problematickou implementací v hardwaru.

Optimalizovat posloupnosti stejných instrukcí nepřináší užitek. Například spojení časté trojice instrukcí STORE pro zálohování stavu registrů do jediné instrukce je problematické. Bylo by třeba více zápisových portů paměti a čtecích portů registrového pole. Navíc není možné zakódovat nutné množství adres registrů do jediného instrukčního slova. Tento přístup je vhodný spíše pro architektury VLIW. Namísto toho jsem se zaměřil na posloupnosti instrukcí modifikujících pouze jeden registr, tedy pracujících nad stejnými daty. Nejčastější takovou posloupností, u které dává smysl spojení operací, je posun registru o jeden nebo dva bity doleva a jeho následné přičtení k dalšímu registru. Vytvořil jsem instrukce ADDSLL1 a ADDSLL2 s touto sémantikou. Mimo to jsem instrukční sadu doplnil ještě o variantu instrukce ORI bez znaménkového rozšíření (ORU), která pohodlně řeší problém načítání konstant. Modifikace mikroarchitektury pro potřeby těchto instrukcí nejsou zásadní a nevedou k vytvoření nové kritické cesty. Nové instrukce byly kódovány do prostoru vyhrazeného pro uživatelská rozšíření instrukční sady RISC-V, nemají tedy vliv na kompatibilitu se základní sadou ani jejími standardními rozšířeními. Výsledná instrukční sada tedy stále může být označena jako RISC-V.

Po dokončení optimalizací jsem opět provedl simulace na sadě programů popsané v sekci 6.5. Vliv nových instrukcí na výkon překladače je zobrazen v grafu na obrázku 6.5. Je patrné, že nové instrukce pomohli zmírnit ztrátu generovaného překladače oproti ručně psanému na 9 %.



Obrázek 6.5: Relativní výkon překladačů při optimalizacích „-O3“ po modifikaci instrukční sady.

Kapitola 7

Model časování procesoru RISC-V

2Druhým cílem mé práce je vytvoření obvodového modelu procesoru RISC-V na základě modelu časování popsaného v jazyce `CodAL`. Strukturu modelu jsem navrhl na základě mikroarchitektury `Rocket Core` popsané v sekci 2.2. Mikroarchitektura je implementována pro stejnou instrukční sadu a rozšíření jako v případě instrukčního modelu.

Platformu instrukčního modelu lze bez úprav použít i pro model časování. Byla však doplněna o datové a instrukční paměti cache dostupné v Codasip Studiu. Jejich konkrétní konfigurace byla zvolena tak, aby se co nejvíce blížila `Rocket Core` z důvodu férového porovnání plochy na čipu. Konfigurace obou použitých pamětí cache je stejná:

- Velikost 4 KiB.
- 4 cesty.
- Velikost cacheline 16 B

Šířka programového čítače byla po vzoru `Rocket Core` omezena na 39 bitů. Umožňuje tak adresování terabajtu paměti a zároveň úsporu zdrojů oproti práci s celou 64-bitovou hodnotou. Behaviorální modely instrukcí definované pro instrukční model je zapotřebí modifikovat. Elementy popisující instrukce pro model časování již neobsahují informace o sémantice a namísto toho definují hodnoty hodnoty řídících signálů, na které se instrukce dekóduje. Pro model jsem definoval následující řídící signály:

muldiv Instrukce pracuje s modulem pro dělení a násobení.

alu_function Signál vybírající operaci použitou v ALU jednotce.

aluop1 Kontrolní signál multiplexoru na vstupu prvního operandu ALU jednotky.

aluop2 Kontrolní signál multiplexoru na vstupu druhého operandu ALU jednotky.

sel_imm Výběr dekódování konstanty z instrukčního slova.

simm Signál ovládající znaménkové rozšíření konstant.

reg_write Instrukce zapisuje do registru.

memop Instrukce přistupuje do paměti.

mem_type Šířka přístupu do paměti (bajt, půlslěovo, slovo, dlouhé slovo).

mem_cmd Typ přístupu do paměti (čtení, zápis).

branch Instrukce podmíněného skoku.

jal Instrukce přímého skoku.

jalr Instrukce nepřímého skoku.

dw Instrukce pracuje s dlouhým slovem.

Ukázka kódu 7.1 prezentuje zápis instrukce LUI pro potřeby modelu časování.

```
element i_hw_lui
{
    use opc_lui as opc;
    assembler { opc };
    binary { bit[IMM20_W] bit[RF_GPR_ADDR] opc };
    semantics
    {
        s_id_muldiv = NO;
        // instrukce secte 0 a konstantu
        // v kodovani typu U
        s_id_alu_function = ALU_ADD;
        s_id_aluop1_mux = ALUOP1_ZERO;
        s_id_aluop2_mux = ALUOP2_IMM;
        s_id_sel_imm = IMM_U;
        s_id_simm = YES;
        s_id_reg_write = YES;
        s_id_memop = NO;
        s_id_mem_type = MEM_BYTE;
        s_id_branch = NO;
        s_id_jal = NO;
        s_id_jalr = NO;
        s_id_mem_cmd = MEM_RD;
        s_id_dw = YES;
    };
};
```

Ukázka kódu 7.1: Popis instrukce pro vytvoření dekodéru.

I v tomto modelu se využívají eventy reset a main. Reset opět nastavuje počáteční hodnoty registrového pole a pomocných registrů. Main slouží k řízení simulace řetězené linky procesoru sekvenční aktivací eventů reprezentujících její jednotlivé stupně. Nově je přidán také kontrolní event halt využívaný pro výpis návratové hodnoty (obsah daného registru) po skončení simulace. Architekturální zdroje je potřeba doplnit o mezistupňové registry. V následující sekcích jsou popsány jednotlivé stupně řetězené linky. Zvláštní pozornost je věnována modulu pro dělení a násobení, predikci skoků a vlivu paměťového subsystému na řízení řetězené linky.

7.1 Řetězená linka

Řetězená linka `Rocket Core` se stejně jako většina RISC-ových procesorů skládá z pěti stupňů: IF, ID, EX, MEM a WB. Vytvořil jsem tedy pětici eventů, které je reprezentují.

7.1.1 Načítání instrukce (IF)

Tento stupeň slouží k odeslání požadavku na instrukci do paměťového subsystému. Požadavek je odeslán na adresu danou aktuálním programovým čítačem pro tento stupeň. Dva nejnižší bity adresy jsou vynulovány pro zajištění zarovnaného přístupu. Požadovaná hodnota je také předána stupni ID. Požadavek na čtení je zasílán v každém taktu. Další zodpovědností tohoto stupně je výpočet nové hodnoty programového čítače. Tato hodnota je dána signálem ze stupně MEM v případě skoku, jinak je získána přičtením 4 (velikost instrukce v bajtech) k jeho aktuální hodnotě.

7.1.2 Dekódování instrukce (ID)

Stupeň dekódování instrukce čeká na odpověď paměťového subsystému a zasílá načtená data do dekodéru instrukcí. Instrukce je dekódována na dříve popsané řídící signály. Dále jsou dekódovány adresy zdrojových a cílových registrů. Data zdrojových registrů jsou přečtena z registrového pole, popřípadě ze stupně WB v případě že je v tomto stupni instrukce zapisující do registru s adresou shodnou s některým ze zdrojových registrů. Implementovaná architektura nevyžaduje řešení datových a strukturálních hazardů překladačem. O řešení hazardů se stará právě stupeň ID. Na základě cílových registrů instrukcí ve stupních EX a MEM je provedena kontrola, zda je nutné provést takzvaný bypass, tedy předání hodnoty před jejím fyzickým zápisem do registrového pole. Pokud ano, je tato informace předána stupni EX. Na spodních dvou bitech předaného registrového operandu je zakódován zdroj dat pro bypass. Strukturální hazard přináší modul pro dělení a násobení. Modul je popsán v sekci 7.3 společně s řešením hazardu. Další takový hazard přináší instrukce LOAD a JALR. Pokud instrukce v ID potřebuje data produkovaná JALR nebo LOAD je nutné zpracování v tomto stupni na jeden takt pozdržet, protože není možné tato data předat hned následující instrukci, jelikož:

LOAD Data jsou připravena až ve stupni WB. Je nutné aby se instrukce LOAD dostala do stupně MEM pro správnou detekci příležitosti pro bypass.

JALR JALR vypočítává ve stupni EX cíl skoku a návratovou adresu až v MEM.

Všechny informace dekódované v ID, programový čítač pro ID, hodnoty registrových operandů a samotné instrukční slovo jsou předány stupni EX.

7.1.3 Provedení instrukce (EX)

Stupeň EX extrahuje jednotlivé varianty přímého kódování z instrukčního slova předaného z ID a na základě řídícího signálu sel_imm vybere správnou variantu. U konstanty je provedeno znaménkové rozšíření. Dvojice nejnižších bitů registrových operandů předaných z ID určí zda a odkud bude proveden bypass. Řídící signály aluop1 a aluop2 nakonec vyberou vstupní data do ALU. Prvním operandem mohou být:

- Hodnota programového čítače stupně EX.

- Registrový operand, u kterého je v případě instrukce pracující pouze se spodní polovinou data nutné provést znaménkové rozšíření bitu 31 do vyšších pozic.
- Nula.

Druhým operandem mohou být:

- Registrový operand.
- Registrový operand posunutý o 1 bit doleva.
- Registrový operand posunutý o 2 bity doleva.
- Přímo kódovaná konstanta.
- Konstanta 4, využívaná instrukcí JAL k výpočtu návratové adresy.
- Nula.

Řídící signál alu_function vybírá operaci ALU:

- Sčítání, odečítání.
- Logický součet, součin, XOR.
- Porovnání na rovnost, nerovnost.
- Porovnání na větší nebo rovno, menší. Znaménkové a neznaménkové varianty.
- Logický posun doleva, doprava.
- Aritmetický posun doprava.

Stupeň EX je mimo to také vstupním bodem modulu pro dělení a násobení. Vypočtená data a původní vstupy EX jsou přeposlány stupni MEM. Instrukce podmíněného skoku využívají ALU k výpočtu podmínky, instrukce nepřímého skoku k výpočtu adresy cíle a instrukce přímého skoku k výpočtu návratové adresy.

7.1.4 Přístup do paměti (MEM)

MEM má zodpovědnost za řešení skoků a přístupy do paměti. Do paměťového subsystému je zaslán požadavek na adresu vypočtenou v EX. Typ požadavku je dán řídícím signálem mem_cmd (čtení, nebo zápis) a množství požadovaných dat řídícím signálem mem_type. Požadavky do paměti jsou zaslány pouze v případě, že je nastaven řídící signál mem_op.

Pro potřeby skoků jsou z instrukčního slova opět dekódovány konstanty ovšem pouze typu UJ a SB. Tato operace není na logiku náročná, proto jsem zvolil opakované dekódování namísto předávání těchto hodnot z EX přes registry. Splnění podmínky podmíněného skoku je dáno hodnotou nejnižšího bitu výsledku ALU předaného z EX. Nová hodnota programového čítače je pomocí řídících signálů jal, jalr a branch pro jednotlivé typy skoků dána následovně:

JAL Hodnota programového čítače MEM + konstanta typu UJ.

JALR Hodnota výsledku předaného z EX.

Podmíněné skoky Hodnota programového čítače MEM + konstanta typu SB.

Řídící signál pro provedení skoku je stupni IF zaslán pouze v případě, že cílová adresa je zarovnána na šířku instrukce. Sčítačka potřebná k výpočtu cílových adres pro podmíněné skoky a JAL je pro potřeby JALR použita k výpočtu návratové adresy. Vypočtená data a původní vstupy MEM jsou přeposlány stupni WB.

7.1.5 Dokončení zápisu (WB)

Stupeň WB čeká na dokončení paměťových operací při nastavení řídícího signálu mem_op a provádí zápis nové hodnoty do registrového pole na základě řídícího signálu reg_write. Zároveň je tento stupeň výstupním bodem modulu pro dělení a násobení.

7.2 Paměťový subsystém a řízení linky

Paměťový subsystém má zásadní dopad na řízení řetězené linky. Pokud paměť není připravena na požadavek zaslaný ve stupni MEM, nebo pokud vynutí čekání na data ve stupni WB, je třeba, aby žádný z předchozích stupňů neprovedl nevratné operace. Toto chování lze zajistit funkcemi stall a reset, které jsou k dispozici pro každý stupeň řetězené linky. Jednotlivé mezistupňové registry lze spojit se stupni. Jejich chování je potom spjato s těmito funkcemi. Funkce stall vynutí přechod signálu povolujícího zápis do registru do neaktivní hladiny. Tuto funkci je třeba volat na všechny stupně po ten, který požaduje zastavení linky, včetně jeho samého, aby se zabránilo jakékoliv ztrátě informace. Funkce reset ovládá resetovací vstup registru. Je nutné resetovat stupeň, který následuje po tom, který žádá zastavení linky. Uvažme situaci, kdy EX vyžaduje zastavení linky a obsahuje řídící signály pro provedení zápisu do paměti. Pokud se neprovede reset stupně MEM, v následujícím taktu dojde k zahájení zápisu do paměti, ovšem na základě signálů, které by nejsou kvůli požadavku na zastavení platné (například adresa může být neplatná). Resetování příštího stupně zajistí jeho nečinnost. Další použití této funkce je pro potřeby vyklizení stupňů IF až EX při provedení skoku. Priorita stupňů pro zastavení linky roste od IF k WB. Pokud tedy zároveň žádají stupně ID a MEM, bude zavolána funkce reset nad WB a stall nad IF, ID, EX a MEM.

Použití instrukčních pamětí cache přináší riziko vynechání, nebo naopak zdvojení instrukcí. Jelikož zpracování požadavků trvá různou dobu na základě přítomnosti dat v cache je zapotřebí pružně reagovat na aktuální situaci pro zajištění korektního chodu programu. Požadavek na instrukci je vydáván v každém taktu, bez ohledu na zastavování linky a to z důvodu zkrácení kritických cest. Zastavení linky pro stupeň ID prakticky znamená ztrátu instrukce, jelikož právě dekódované instrukci není povolen zápis do dalšího stupně. Úspěšná žádost o instrukci v IF při zastavení linky má za následek duplikaci instrukce jelikož programový čítač se nemůže během zastavení pohnout a tudíž bude v dalším taktu požádáno o stejnou instrukci. Tato fakta je možné kompenzovat. Způsob modifikace jednotlivých stupňů je popsán v následujících odstavcích.

7.2.1 IF

Stupeň IF zastavování linky nevynucuje, jelikož je prvním stupněm. Je třeba pouze zabránit posunu programového čítače v případě, že instrukční paměť není připravena přijmout požadavek na další instrukci. Jelikož je požadavek na instrukci odesílán v každém taktu včetně situace, kdy linka stojí, je třeba nastavit příznak, že příští instrukce musí být ignorována,

aby nedošlo k její duplikaci. Stejný problém nastává v případě skoku, kde by došlo k provedení instrukce následující za skokovou a tím k nevalidnímu chování programu. Ignorovat nastavenou instrukci je tedy třeba v při skoku pokud uspěl požadavek na instrukci nebo je třeba čekat na jeho dokončení. Ignorování je nutné také pokud požadavek na instrukci uspěl a linka stojí. Posun programového čítače je povolen pouze pokud není zastavena linka. Jeho modifikace je možná i skrze skoky a to i při zastavené lince.

7.2.2 ID

Ve stupni ID je pro kompenzaci zastavování linky implementován jednopoložkový instrukční buffer. Do toho se vždy ukládá načtená instrukce a to i v případě, že je linka zastavena. Díky tomu není třeba opakovat požadavek na instrukci, u níž před dokončením načítání došlo k zastavení linky. ID zastavuje linku v případě, že mezi operandy dekódované instrukce je výsledek instrukce JALR, nebo LOAD, která je ve stupni EX. Dále je linku potřeba zastavit pokud instrukční buffer neobsahuje požadovanou instrukci a paměťový subsystém na požadavek na dokončení načtení instrukce odpoví zprávou WAIT.

7.2.3 EX

Stupeň EX potřebuje zastavit linku pouze v případě, že obdržel instrukci násobení a modul pro dělení a násobení je již obsazen jinou instrukcí. Linka tedy bude stát dokud se modul neuvolní. Modul může být zastaven pouze zápisy do registrového pole ve WB. Zastavení EX způsobí resetování MEM, odkud se v dalším taktu efekt resetu propaguje do WB a tím umožní modulu pro dělení a násobení dokončit operaci zápisem do registrového pole. Nemůže tedy dojít k uváznutí.

7.2.4 MEM

Stupeň MEM žádá zastavení linky pokud paměť není připravena přijmout požadavek. V takovém případě je navíc nutné předat na vstup WB jeho výstupní data, protože v EX může být instrukce, která je skrze bypass potřebuje. Díky zastavení linky je však nemůže využít. V dalším taktu by tato data byla sice zapsána do registrového pole, ale instrukce v EX již počítá s jejich předáním z WB.

7.2.5 WB

Stupeň WB žádá zastavení linky v případě, že paměť na požadavek na dokončení operace odpoví zprávou WAIT.

7.3 Modul pro dělení a násobení

Implementovaný modul pro dělení a násobení vychází z modulu použitého v `Rocket Core`, popsaného v jazyce `Chisel`. Modul je umístěn mimo řetězenou linku. Jeho vstupní bod je ve stupni EX a výstupní ve stupni WB. Instrukce vstupující do modulu postupuje dále i řetězenou linkou a ve stupni WB způsobí zápis chybné hodnoty do registrového pole. Důvodem pro postup linkou je detekce přítomnosti instrukce využívající tento modul v řetězené lince ve stupni ID. Pokud instrukce v ID potřebuje výsledek dělení či násobení, musí být první dva stupně zastaveny. Zápis neplatné hodnoty do registrového pole je povolen kvůli snaze vyhnout se zbytečné logice. Neplatná hodnota nebude nikdy použita, protože příchod

násobící, či dělící instrukce do WB způsobí nastavení příznaku neplatnosti cílového registru. Tyto příznaky jsou také kontrolovány stupněm ID a potřeba využít daná data vede opět na zastavení linky. Modul po dokončení výpočtu čeká na možnost zápisu do registrového pole. Instrukce nacházející se ve WB má vždy přednost zápisu. Výsledek dělení a násobení je tedy zapsán až ve chvíli, kdy je WB okupován instrukcí, která nezapisuje do registru (STORE, NOP) nebo ve chvíli kdy datová závislost způsobí zastavení linky a tím uvolnění WB. Příznak neplatných dat je zrušen zároveň se zápisem platné hodnoty do registrového pole. Modul pro dělení a násobení tedy při vhodné skladbě programu může provádět instrukce mimo programové pořadí a tím překrýt latenci výpočtu. Vhodná skladba je taková, kde výsledek násobení či dělení není využíván hned následující instrukcí. Čím později je výsledek potřeba, tím lépe a to zejména u dělení. Modul je pro obě operace společný a dovoluje pouze jednu rozpracovanou operaci. Práce modulu je řízena konečným automatem s následujícími stavy:

IDLE Modul je připraven na nový požadavek.

IN_PROGRESS Probíhá operace.

DONE Čekání na zápis výsledku do registrového pole.

SHIFT Posun registru s výsledkem pro operace zbytku a vrchních polovin násobení.

NEG_IN Negování vstupních operandů.

NEG_OUT Negování výsledku operace.

Pro potřeby dělení i násobení je v modulu 65-ti bitový registr pro dělitele (činitele) a 130-ti bitový registr společný pro dělence a zbytek (druhého činitele a součin). Bity navíc slouží k uchování znamének. Dále je zde registr pro uchování čísla registru, do kterého má být uložen výsledek.

Dělení probíhá sekvenčně s maximální dobou výpočtu 67 taktů[1] a to na základě následujících kroků:

1. Vrchní polovina registru dělence slouží k uchování zbytku. Od ní je odečten dělitel.
2. Pokud je výsledek záporný, celý registr dělence je posunut doleva a zprava doplněn nulou, která reprezentuje výsledek dělení v daném kroku. Pokud je výsledek kladný, nahradí aktuální zbytek, celý registr se poté posune doleva a zprava doplní jedničkou.
3. Navýší se hodnota počítadla kroků dělení.
4. Pokud je počet kroků dělení roven 64, dělení je u konce, spodní polovina registru dělence obsahuje podíl a vrchní polovina zbytek. V opačném případě se pokračuje krokem 1 v dalším taktu.

Sekvenční povaha dělení na 64 bitech přináší velké množství zbytečných kroků pokud je hodnota dělence a dělitele příliš malá. Uvažme výpočet $\frac{255}{3}$. Prvních 57 taktů bude dělenec pouze posouván doleva rychlostí 1 bit za takt. Aby bylo možné tomuto předejít, obsahuje dělička speciální logiku, která v prvním kroku dělení na základě pozic nejvyšších nastavených bitů dělence a dělitele vypočítá o kolik bitů je potřeba dělence posunout pro začátek

[1] 64 taktů pro samotné dělení, 2 pro případné negování vstupu a výstupu a 1 pokud je požadována hodnota zbytku.

reálného výpočtu. Přidaná logika výrazně zrychluje výpočet, jelikož jen zřídka je využit celý 64-bitový rozsah dat a proto ji považuji za velice vhodnou přestože nemá správnost výsledku nemá vliv a zvětšuje plochu procesoru.

Plně paralelní násobení na 64 b vede na příliš rozměrnou logiku a velké zpoždění. To je problém, protože jíž 32-bitové paralelní násobičky se často nacházejí na kritické cestě. Násobení je častou operací, proto plně sekvenční násobička působí výkonnostní problémy příliš velkým zpožděním. V `Rocket Core` a tedy i v mém modulu je proto zvolen kompromis ve formě částečně rozbalené sekvenční násobičky. V každém taktu je provedeno násobení $64 \cdot 8$ bitů. Výpočet probíhá následovně:

1. Vrchní polovina registru pro dělence slouží k uložení střádače, který obsahuje část mezivýsledku, který je relevantní pro aktuální krok výsledku. Spodní polovina potom obsahuje část prvního činitele, kterým je ještě třeba násobit. Druhý činitel je umístěn v registru pro dělitele.

2. Druhý činitel je vynásoben spodními osmi bity prvního činitele.

3. Součin je sečten s aktuálním mezivýsledkem.

4. Součet má bitovou šířku 73 (násobení $64 \cdot 8$ a součet). Jeho spodních 8 bitů je zleva nasunuto do prvního činitele. Díky tomu tento činitel nebude obsahovat bity, kterými se již násobilo a v jeho vrchní části budou uloženy bity výsledku, které se již nemohou měnit.

5. Navýší se hodnota počítadla kroků násobení.

6. Pokud je počet kroků násobení roven 8, násobení je u konce a registr pro dělence obsahuje součin. V opačném případě se pokračuje krokem 1 v dalším taktu.

I pro potřeby násobení je implementována logika, která vyhodnocuje možnost předčasného ukončení operace. V každém kroku je ke spodní polovině registru pro dělence přikládána bitová maska se všemi bity nastavenými na 1. Tato maska je v každém kroku posunuta o 8 bitů doprava. Nastavené bity tedy vždy korespondují s částí činitele, kterou se teprve bude násobit. Pokud je výsledek logického součinu masky a činitele roven nule, jsou další kroky násobení zbytečné, jelikož se bude vždy jednat o násobení nulou. V takovém případě násobení končí a výsledek je dán hodnotou registru dělence posunutou doprava. Počet bitů o které je výsledek potřeba posunout je dán povahou algoritmu. Je nutné dokončit posunutí, které by proběhlo ve zbývajících krocích. Počet bitů tedy je $8 \cdot (8 - x)$, kde x je hodnota počítadla kroků násobení.

7.4 Predikce skoků

Implementovaná architektura trpí výraznou ztrátou výkonu při skocích. Při provedení skoku je nutné vyklidit stupně IF, ID a EX. Pokuta za skok je tedy 3 takty. Taková pokuta výrazně zvyšuje počet cyklů na instrukci a je tedy nutné ji kompenzovat. K tomuto účelu se běžně využívají prediktory skoků, které vyhodnotí skok již na základě jeho adresy, ještě před samotným dekódováním instrukce.

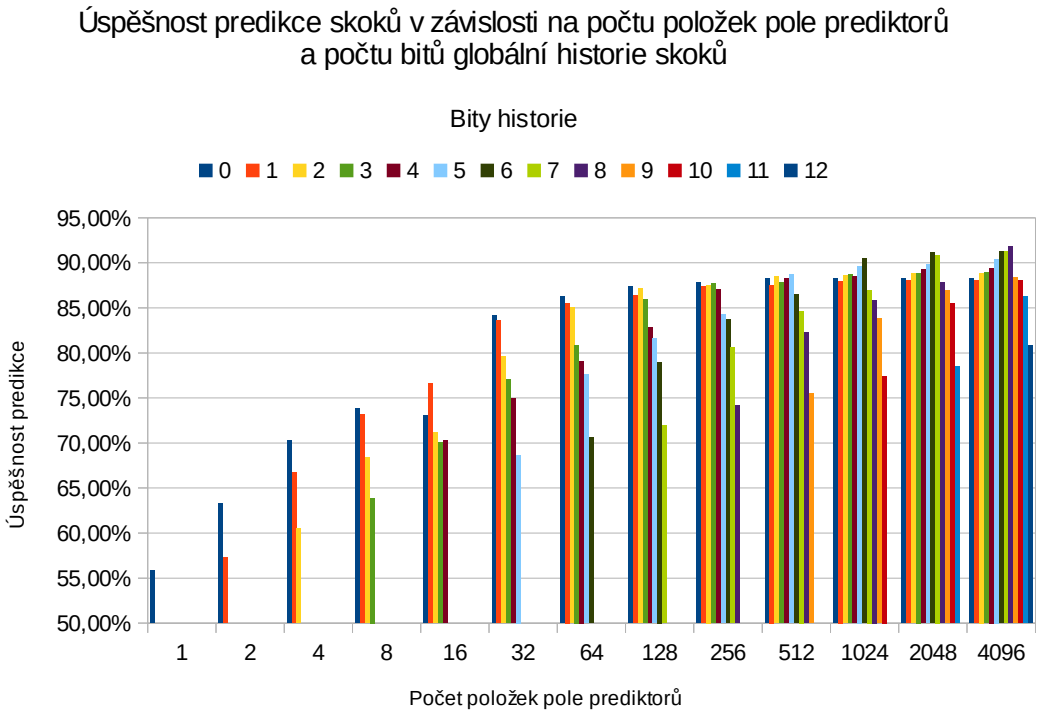
V první fázi jsem se snažil o implementaci prediktoru skoků založeného na modulu BHT popsanému v jazyce `Chisel`. Tento modul představuje velmi výkonný korelační prediktor skoků použitý v `Rocket Core`. Jeho tabulka historie zdrojů a cílů skoků je plně asociativní. V ní je uloženo pouze spodních 12 bitů adres, a to z důvodu zmenšení potřebných

komparátorů. Každá položka je doplněna o registr uchovávající index do menší tabulky s vrchními bity adres. Dochází tedy ke stránkování adresového prostoru. Prediktor může velice efektivně zpracovávat velké množství skoků, pokud se všechny nachází v prostoru, který dokáže stránkovat. Velikost jedné stránky je 4 KiB. V základním nastavení `Rocket Core` má tabulka těchto stránek 6 položek, tabulka spodních bitů 62 položek a pro adresování tabulky stránek je zapotřebí tří bitů. Pro predikci skoků je využito jednorozměrné pole 128 dvoubitových prediktorů, které je indexované hodnotou získanou logickou operací XOR nad posuvným registrem globální historie skoků a spodními sedmi bity adresy skoku. Registr globální historie skoků zaznamenává informaci o provedení posledních sedmi skoků.

Snaha o implementaci tohoto prediktoru v jazyce `CodAL` odhalila některá jeho slabá místa. Například nemožnost zapsat pole signálů a registrů. `Chisel` navíc obsahuje velké množství vestavěných funkcí, které je možné využít na různá mapování signálů. Funkcionalita je potom ukryta v samotném jazyce. V `CodALu` bylo nutné tato mapování napsat ručně pomocí podmínek a ternárních operátorů. Zápis prediktoru skoků ekvivalentního k tomu v `Rocket Core` v jazyce `CodAL` zabíral devětkrát více řádků, než zápis v jazyce `Chisel`. Ani výsledky syntézy nedopadly dobře. Procesor s prediktorem vygenerovaným z `CodALu` dosahoval o třetinu menší frekvence než při použití prediktoru vygenerovaného z popisu v `Chiselu`. Jazyk `CodAL` však cílí na velmi efektivní popis samotného jádra procesoru a jeho využití pro popis a generování vysoce specializovaných komponent jako je například prediktor skoků nebo FPU se nedoporučuje. Existuje však možnost integrovat je do řešení pomocí pluginů.

Namísto prediktoru `Rocket Core` jsem se rozhodl implementovat klasický dvouúrovňový prediktor skoků. První úroveň je opět posuvný registr globální historie skoků a druhou úrovní jednorozměrné pole dvoubitových prediktorů [14]. Index do této tabulky je získán konkatenací spodních bitů programového čítače a globální historie skoků. Samotná predikce nestačí, je nutné znát i adresu, na kterou se bude skákat. Každý dvoubitový prediktor je doplněn o vrchní bity adresy zdroje skoku, celou adresu cíle skoku a příznak platnosti záznamu. Příznak platnosti záznamu je třeba implementovat formou registru, kvůli možnosti okamžitého zneplatnění všech položek. Zbývající data je možné ukládat v paměti. Počet bitů historie a stejně tak počet bitů adresy, použitých k indexování přináší prostor k experimentování k nalezení optimální konfigurace. Více použitých bitů vede k rozšíření pole prediktorů, ale často také ke zlepšení predikce. Pro nalezení optimální konfigurace jsem experimentoval s IA simulací na stejné sadě testovacích programů jaká byla použita k měření výkonu překladače. Během IA simulace nedochází k žádné penalizaci za skoky, přináší však možnost rychle a jednoduše modelovat prediktor bez ohledu na časování při zachování informace o úspěšnosti predikce. V experimentech jsem zkoumal vliv počtu dvoubitových prediktorů na úspěšnost predikce. Pro každý zkoumaný počet jsem testoval i jednotlivé možnosti počtu bitů registru globální historie skoků. Zbývající bity nutné k adresaci jsou nejnižší bity programového čítače. Bity 0 a 1 jsou vynechány, jelikož by se jednalo o nezarovnané adresy. Dosažené úspěšnosti predikce jsou zaneseny do grafu na obrázku 7.1. Je nutné vzít v úvahu také cenu jednotlivých konfigurací z hlediska potřebné paměti. Požadavky na paměť jsou dány vzorcem: $x = (2 + 37 + 37 - (z - y)) \cdot 2^z$, kde „x“ je požadovaná velikost paměti, „y“ je počet bitů globální historie skoků, „z“ je celkový počet bitů použitý k indexování pole prediktorů, „2“ reprezentuje prediktor a „37“ počet bitů programového čítače. Více bitů globální historie skoků vyžaduje uložení většího počtu bitů zdrojové adresy skoku v paměti. Velikost registru globální historie skoků a registru s bity platnosti záznamů je zanedbatelná.

Z naměřených dat vyplývá, že samotný počet položek bez uvážení historie skoků má



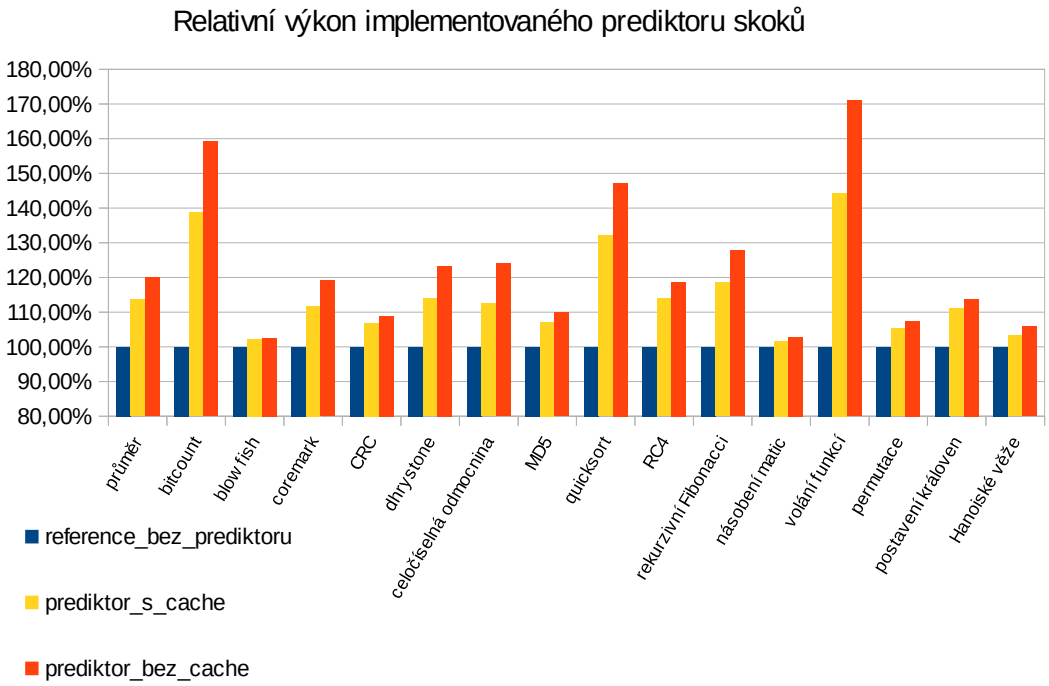
Obrázek 7.1: Úspěšnost predikce skoků v závisloti na počtu bitů globální historie skoků a dvoubitových prediktorů.

zásadní vliv pouze do 512. S dalším rozšiřováním se již výsledky nezlepšují. Nejlépe dopadly výsledky velkých prediktorů s vyrovnaným počtem bitů historie a programového čítače pro indexaci. Tuto skutečnost přisuzuji největší možnosti adaptace na vzorce skoků a jejich pozice. Pokud je použito příliš mnoho bitů historie, mnoho stejných vzorců skoků se navzájem ovlivňuje. Pokud je jich málo, globální chování není dostatečně bráno v úvahu. Pro menší prediktory je naopak využití historie skoků spíše na škodu, protože dochází k aliasu velkého množství adres na malé vzorce skoků a tím pádem k častému přepisu položek. Na základě dosažených výsledků jsem zvolil konfiguraci 256 položek a žádné bity globální historie skoků. Více položek přináší lepší predikci, avšak cena v podobě nároku na paměť je již pro 256 položek hraniční. Dosahuje poloviny velikosti využité paměti cache. Pro tento počet položek dává nejlepší výsledky predikce bez historie skoků. Tato skutečnost je výhodná z hlediska menších nároků na paměť a jednodušší logiky.

Ve stupni IF je prediktoru zaslána hodnota programového čítače. Jeho spodní bity tvoří adresu požadavku na čtení z paměti s prediktory. Vrchní část adresy je v příštím taktu v ID porovnána se záznamem. Pokud se adresy rovnají, příznak platnosti záznamu je nastaven a nejvyšší bit prediktoru také, je proveden skok na uloženou cílovou adresu. Cílová adresa je tedy použita v aktuálním požadavku na instrukci v IF. Jinak se neskáče. Informace o predikci je poslána řetězenou linkou. Ve stupni MEM je provedena kontrola správného vyhodnocení skoku a korekce stavu prediktoru zodpovědného za daný skok. Nová adresa cíle je do prediktoru nahrána pokud skok nebyl predikován, ale proběhl, nebo pokud proběhl a byl predikován, ale byl proveden na špatnou adresu. Při nahrání nové cílové adresy skoku je prediktor nastaven na nejvyšší hodnotu. Stav dvoubitového prediktoru je aktualizován

vždy. V případě, že byl skok predikován špatně, dojde k němu nyní. Stejná situace nastává pokud byl skok proveden na špatnou adresu.

Přínos prediktoru z hlediska snížení počtu taktů CA simulace nutných k běhu programu jsem měřil na stejné sadě programů použitých k jeho profilování. V rámci CA simulace je nutné vzít v úvahu, že model obsahuje paměti cache, které i přes správnou predikci mohou způsobit zastavení linky a to i na dobu srovnatelnou se špatnou predikcí skoku. Z tohoto důvodu jsem měření provedl na variantách modelu s a bez pamětí cache. Efekt prediktoru na chod programů shrnuje graf na obrázku 7.2. Jako reference je zvolena doba chodu programu bez prediktoru. Efekt prediktoru je zobrazen procentuálně vůči své referenční variantě. Latence hlavní paměti je nastavena na 1 takt, simuluje tedy ideální situaci, kdy nenastávají žádné výpadky a udává tak maximální zrychlení dosažitelné implementovaným prediktorem skoků.



Obrázek 7.2: Efekt implementovaného prediktoru skoků na výkon CA simulace.

7.5 Testování

Pro účely otestování správného fungování modelu časování jsem využil stejných testů jako v případě instrukčního modelu. Nejprve jsem provedl ladění modelu skrze kontrolu návratových kódů programů a krokování v debuggeru. Po odstranění nalezených chyb v podobě ztrácejících a zdvojujících se instrukcí jsem přešel k využití nástroje Equivalency Checker, který je dostupný v rámci Codasip Studia. Tento nástroj slouží k ověření shodného chování IA a CA modelu. Nejprve přeloží testovací program a poté ho simuluje nejprve na IA a CA simulátoru. Nástroj kontroluje zápisy do jednotlivých zdrojů (registry, paměti). Přístupy musí proběhnout ve stejném pořadí v obou simulátorech, jinak není chování jed-

noho z nich korektní (zpravidla CA). Pokud dojde k neshodě, je umožněn start vybrané simulace od místa, kde k ní došlo. To výrazně zrychluje ladění modelů oproti běžnému krokování a kontrolním výpisům. Drobné chyby odhalené tímto nástrojem byly opraveny. Korektní chování obvodového modelu RISC-V v podobě modelu časování bylo tedy ověřeno prostřednictvím kontroly ekvivalence s instrukčním modelem.

7.6 Syntéza

Pro syntézu jsem zvolil FPGA Artix 7 a provedl ji prostřednictvím nástroje Xilinx ISE. Z popisu `Rocket Core` v jazyce `Chisel` jsem vygeneroval jeho popis v jazyce `Verilog`. Použil jsem základní konfiguraci procesoru bez FPU jednotky. Mimo to jsem odstranil modul s kontrolními registry sloužícími k podpoře U a S módů procesoru a to z důvodu, že tyto módy nejsou v mém modelu podporovány. Provedl jsem dvě srovnání syntézy. Nejprve syntézu celého `Rocket Core` včetně pamětí cache vůči VHDL vygenerovaného z jádra a platformy mého modelu s pamětmi cache, ale bez hlavní paměti. Srovnání výsledků přináší tabulka 7.1. Syntéza celého modelu zahrnuje i cache, které nejsou popsány v jazyce `CodAL` a tedy ovlivňují výsledky získané efektivním popisem procesoru v tomto jazyce. Součástí `Rocket Core` jsou také moduly pro podporu virtuální paměti, průchod tabulky stránek a různé vstupní a výstupní fronty. Proto jsem se rozhodl srovnat také syntézy samotných jader. Zatímco v případě modelu v jazyce `CodAL` je velmi jednoduché oddělit jádro od platformy, v případě `Rocket Core` je to náročnější. První stupeň řetězené linky `Rocket Core` je pevně zabudován do modulu instrukční paměti cache a to včetně prediktoru skoků, který je velmi rozměrný. Rozhodl jsem se tedy pro samostatnou syntézu takzvaného backendu procesoru (ID, EX, MEM a WB). Srovnání opět není optimální, jelikož neobsahuje logiku pro výpočet nového programového čítače, komunikaci s pamětí cache ani instrukční buffer. Ztráta této logiky je však menší než přidání celé instrukční cache. Mé samostatné jádro neobsahuje paměti pro prediktor skoků, avšak obsahuje samotnou logiku prediktoru, která opět vyžaduje zdroje navíc a ovlivňuje porovnání. Výsledky druhé syntézy zobrazuje tabulka 7.2.

	CA model	**Rocket Core**
Počet registrů	2826	5089
Počet LUT	9265	14997
Počet RAM32M[2]	20	30
Počet RAM32X1D[3]	8	8
Počet RAM32X1S[4]	20	0
Počet RAM64M[5]	0	672
Počet RAM64X1D[6]	0	200
Počet RAM128X1D[7]	0	2
Počet RAMB36E1[8]	21	9
Počet DSP bloků	5	5
Frekvence	89.018 MHz	97.836 MHz

Tabulka 7.1: Porovnání syntézy procesoru `Rocket Core` a mého modelu.

	Jádro CA modelu	Jádro Rocket Core
Počet registrů	1551	957
Počet LUT	5974	3577
Počet RAM32M	20	20
Počet RAM32X1D	8	8
Počet DSP bloků	5	5
Frekvence	89.018 MHz	97.836 MHz

Tabulka 7.2: Porovnání syntézy samotných jader `Rocket Core` a mého modelu.

Kritická cesta mého modelu je v jádru procesoru a nachází se v modulu pro dělení a násobení. Vede od výstupu registru pro dělence po jeho vstup, skrze paralelní $8 \cdot 64$ násobičku a 72-bitovou sčítačku. Stejná kritická cesta se nachází i v `Rocket Core`, avšak zde nezahrnuje vrchních 8 bitů výsledku. Při generování kódu `Rocket Core` tak zřejmě dochází k určitým optimalizacím zajišťujícím efektivnější zápis násobičky.

[2]Distribuovaná RAM $32 \cdot 6$ bitů s jedním synchronním zápisovým portem a jedním asynchronním čtecím portem.

[3]Distribuovaná RAM $32 \cdot 1$ bit s jedním portem pro synchronní zápis a asynchronní čtení a jedním asynchronním čtecím portem.

[4]Distribuovaná RAM $32 \cdot 1$ bit s jedním portem pro synchronní zápis a asynchronní čtení.

[5]Distribuovaná RAM $64 \cdot 3$ bitů s jedním synchronním zápisovým portem a jedním asynchronním čtecím portem.

[6]Distribuovaná RAM $64 \cdot 1$ bit s jedním portem pro synchronní zápis a asynchronní čtení a jedním asynchronním čtecím portem.

[7]Distribuovaná RAM $128 \cdot 1$ bit s jedním portem pro synchronní zápis a asynchronní čtení a jedním asynchronním čtecím portem.

[8]Bloková RAM velikosti 36 Kb.

Kapitola 8

Závěr

V rámci této diplomové práce jsem provedl rozbor instrukční sady RISC-V, na ní založené mikroarchitektury `Rocket Core` a jazyků `CodAL` a `Chisel`, které slouží k popisu instrukčních sad a mikroarchitektur. Jazyk `Chisel` se ukázal jako nepřehledný, avšak podává kvalitní výsledky a značně zjednodušuje zápis mapování hodnot na jiné. Další jeho výhodou je automatické stanovení bitové šířky signálu. `Chisel` je však možné použít pouze pro popis mikroarchitektury a generování její RTL reprezentace. Jazyk `CodAL` je naproti tomu velmi uživatelsky přívětivý a umožňuje nejen popis mikroarchitektury, ale i instrukční sady. Z popisu instrukční sady je možné vygenerovat velké množství nástrojů, zejména překladač jazyka C. Díky generování vývojových nástrojů poskytuje `CodAL` možnost velmi rychlého prototypování instrukčních sad a mikroarchitektur.

Jazyk `CodAL` jsem využil k implementování základní instrukční sady RISC-V ve verzi s 64-bitovým adresovým prostorem a rozšíření pro dělení a násobení. Z modelu jsem vygeneroval překladač jazyka C a porovnal ho s volně dostupným ručně psaným překladačem RISC-V založeným na GCC na sadě výkonnostních testů. Výkon generovaného překladače zaostával o 21 %. Je však potřeba vzít v úvahu, že jde o překladač, který lze vytvořit pro libovolnou architekturu a to velmi rychle, na rozdíl od nákladného a dlouhého vývoje ručně psaných překladačů. Pro zlepšení výsledků jsem využil profilovacích nástrojů Codasip Studia, které mi umožnily nalézt slabá místa instrukční sady. Do prostoru instrukční sady, určeného pro uživatelská rozšíření, jsem doplnil instrukce, jejichž využití umožnilo snížit ztrátu generovaného překladače na 9 %.

Jazyk `CodAL` jsem dále využil pro implementaci modelu časování procesoru založeném na mikroarchitektuře `Rocket Core`. Mikroarchitektura má pět stupňů řetězené linky, modul pro dělení a násobní, který umožňuje zpracování mimo programové pořadí a také dvouúrovňový prediktor skoků. Z tohoto modelu jsem vygeneroval RTL reprezentaci v jazyce VHDL a využil ji k syntéze na FPGA Artix 7. Z volně dostupného popisu `Rocket Core` v jazyce `Chisel` jsem vygeneroval RTL reprezentaci v jazyce `Verilog` v konfiguraci co nejlépe odpovídající mému modelu a taktéž provedl syntézu. Výsledky syntézy ukázaly, že můj model dosahuje o 10 % nižší frekvence než `Rocket Core`. Plochu výsledného řešení není možné přímo porovnat, jelikož některé části `Rocket Core`, které nejsou v mém modelu implementovány, není možné z procesoru odstranit. Ačkoliv je frekvence mého řešení srovnatelná s referenční mikroarchitekturou, je třeba vzít v úvahu, že přestože lze `Rocket Core` syntetizovat na FPGA, je jeho popis optimalizován pro technologii ASIC.

Přes o něco horší výsledky je velkou výhodou modelu v jazyce `CodAL` možnost jednoduché modifikace, která díky generování nástrojů přináší okamžitý efekt. V rámci dalšího vývoje procesoru se hodlám zaměřit na rozšíření instrukčního modelu o instrukce pro práci s čísly v plovoucí řádové čárce a rozšíření modelu časování o FPU jednotku. Dále mám v plánu rozšíření mikroarchitektury o možnost využívat privilegované módy procesoru.

Literatura

[1] Codasip Studio Brochure. http://www.codasip.com, 2014 [cit. 2016-03-01].

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

[3] *CodAL Studio Technical Reference Manual*. Codasip, 2015.

[4] Asanovic, K. : RISC-V Update. ORCONF 2015, CERN, Geneva, 2015-10-10 [cit. 2016-02-01].

[5] Asanovic, K. : Workshop Introductions and RISC-V Update. http://riscv.org/workshop-jun2015/riscv-intro-workshop-june2015.pdf, 2015-30-06 [cit. 2016-02-01].

[6] Asanovic, K.; Bachrach, J.; Vo, H.; aj. : Chisel: Constructing Hardware in a Scala Embedded Language. https://chisel.eecs.berkeley.edu/chisel-dac2012.pdf, 2012-03-06 [cit. 2016-03-01].

[7] Asanovic, K.; Wawrzynek, J. : Chisel Introduction part1. http://www-inst.eecs.berkeley.edu/ cs250/fa11/lectures/lec03.pdf, 2011-07-07 [cit. 2016-02-01].

[8] Asanovic, K.; Wawrzynek, J. : Chisel Introduction part2. http://www-inst.eecs.berkeley.edu/ cs250/fa11/lectures/lec04.pdf, 2011-12-07 [cit. 2016-02-01].

[9] Ashenden, P. J. *The designer's guide to VHDL*. 3rd ed. Boston: Morgan Kaufmann Publishers, c2008. ISBN 0120887851.

[10] Celio, C.; Asanovic, K.; PaLerson, D. : The Berkeley Out-of-Order Machine (BOOM!). http://riscv.org/workshop-jun2015/riscv-boom-workshop-june2015.pdf, 2015-30-06 [cit. 2016-02-01].

[11] Dalalah, A.; Baba, S.; Tubaishat, A. New Hardware Architecture for Bit-counting. 2006.

[12] Lee, Y.; Ou, A.; Magyar, A. : Z-scale: Tiny 32-bit RISC-V Systems. http://riscv.org/workshop-jun2015/riscv-zscale-workshop-june2015.pdf, 2015-30-06 [cit. 2016-02-01].

[13] Moore, G. : Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, roč. 86, č. 1, Leden 1998: s 82–85, ISSN 0018-9219.

[14] Shen, J. P. *Modern processor design : fundamentals of superscalar processors*. Long Grove: Waveland Press, c[2013]. ISBN 9781478607830.

[15] Thomas, D. E.; Moorby, P. R. *The Verilog hardware description language*. 5. ed. New York: Springer, 2002. ISBN 1402070896.

[16] Waterman, A.; Lee, Y.; Avizienis, R.; aj. : The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.7. Technická Zpráva UCB/EECS-2015-49, EECS Department, University of California, Berkeley, Květen 2015. Dostupné z: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-49.html>

[17] Waterman, A.; Lee, Y.; Patterson, D. A.; aj. : The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0. Technická Zpráva UCB/EECS-2014-54, EECS Department, University of California, Berkeley, Květen 2014. Dostupné z: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html>

[18] Waterman, A.; Lee, Y.; Patterson, D. A.; aj. : The RISC-V Compressed Instruction Set Manual, Version 1.9. Technická Zpráva UCB/EECS-2015-209, EECS Department, University of California, Berkeley, Listopad 2015. Dostupné z: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-209.html>

Slovník

ABI Application Binary Interface. Soubor pravidel, která definují komunikaci mezi procesy a operačním systémem.

ASIC Application Specific Integrated Circuit. Integrovaný obvod navržený pro určitý specifický účel.

ASIP Application Specific Instruction set Procesor. Procesor s aplikačně specifickou instrukční sadou.

CA Cycle Accurate Model. Model popisující časování instrukcí a samotnou mikroarchitekturu.

DMIPS Dhrystone Million Instructions Per Second. Počet Dhrystone testů vykonaných za sekundu v miliónech.

DTLB Data Translation Lookaside Buffer. Paměť cache umístěná v procesoru za účelem zrychlení překladu virtuálních adres dat na fyzické.

FPGA Field Programmable Gate Array. Integrovaný obvod obsahující programovatelné bloky s konfigurovatelným propojením.

FPU Floating-Point Unit. Matematický koprocesor pro vykonávání operací nad čísly s plovoucí řádovou čárkou.

HDL Hardware Description Language. Programovací jazyk sloužící pro popis hardware.

IA Instruction Accurate Model. Model popisující instrukční sadu a chování jednotlivých instrukcí.

ISA Instruction Set Architecture. Soubor instrukcí. Zpravidla definuje binární kódování instrukcí a jejich požadované chování.

ITLB Instruction Translation Lookaside Buffer. Paměť cache umístěná v procesoru za účelem zrychlení překladu virtuálních adres instrukcí na fyzické.

RISC Reduced Instruction Set Computing. Procesorová architektura zaměřená na jednoduchou, vysoce optimalizovanou instukční sadu.

RTL Register Transfer Level. Úroveň abstrakce návrhu číslicových obvodů. Jde o popis logických operací nad signály a jejich tok mezi registry.

UVM Universal Verification Methodology. Standardizovaná metodologie pro verifikaci integrovaných obvodů.

VHDL Very High Speed Integrated Circuit Hardware Description Language. Konkrétní programovací jazyk sloužící pro popis hardware. Používá se pro návrh a simulaci číslicových obvodů obvodů.

VLIW Very Long Instruction Word. Procesorová architektura s dlouhým instrukčním slovem, která umožňuje instrukční paralelismus sloučením vzájemně nezávislých operací do instrukčních balíků.