Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra číslicového návrhu

Diplomová práce

Model VLIW procesoru

Bc. Hynek Blaha

Vedoucí práce: doc. Ing. Hana Kubátová, CSc.

23. června 2015

Poděkování

Děkuji doc. Ing. Haně Kubátové, CSc. za rady a vedení.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. Dále prohlašuji, že jsem s Českým vysokým učením technickým v Praze uzavřel dohodu, na základě níž se ČVUT vzdalo práva na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona. Tato skutečnost nemá vliv na ust. § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisů.

V Praze dne 23. června 2015 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Blaha, Hynek. *Model VLIW procesoru.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Tato diplomová práce obsahuje teoretický základ typologie procesorů, jejich vývoje a vlastostí s důrazem na procesory typu VLIW. Čtenář je seznámen s vývojovým prostředím Codasip, ve kterém je následně implementována mnou navržená mikroarchitektura procesoru Codix VLIW. Správnost implementace je ověřena funknční verifikací a výkon je porovnán s konkurencí na trhu. Výsledkem této práce je procesor Codix VLIW na platformě Xilinx Zynq.

Klíčová slova Procesor, Codix, VLIW, Velmi dlouhé instrukční slovo, IA, CA, Codal, Zynq, Implementace, FPGA, Codasip, Benchmark, Verifikace, Rozšiřitelné jádro

Abstract

This master thesis contains theoretical basics of processor typology with aim on processor VLIW family. The reader will get familiar with Codasip IDE, in which a design of Codix VLIW processor is implemented. The correct implementation of processor core is validated using functional verification. The result of this master thesis is processor Codix VLIW on Xilinx Zynq platform.

Keywords Processor, Codix, VLIW, Very long instruction word, IA, CA, Codal, Zynq, Implementation, FPGA, Codasip, Benchmark, Verification, Extensible core

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V roce 1965, v době kdy se pro sálové počítače teprve začínala používat technologie integrovaných obvodů, Dr. Gordon E. Moore publikoval článek v časopisu Electronics, ve kterém předpovídal počítačovému oboru zářnou budoucnost. Na základě sledování evoluce integrovaných obvodů predikoval dlouhodobý exponenciální růst hustoty tranzistorů.[1] Tuto předpověď dnes známe jako Moorův zákon.[2] O šest let později byl světu představen první mikroprocesor. Byl jím Intel 4004, čtyřbitový procesor tvořený přibližně dvěma tisíci tranzistory s taktovací frekvencí rovnající se pouhé desetině MHz.

Během následujících dekád prošel vývoj procesorů dechberoucími změnami. Výkon se v souladu s Moorovým zákonem, který letos oslaví padesátileté výročí, pravidelně zdvojnásoboval každých osmnáct až dvacet čtyři měsíců, zatímco cena průběžně klesala, a tak se zařízení stávala čím dál tím více dostupnější.

V dnešní vyspělé době jsme obklopeni elektronikou doslova na každém kroku. Každoročně je vyrobeno přes jednu miliardu procesorů. Z toho více než devadesát procent je využito ve vestavěných systémech. Vestavěným zařízením rozumíme jednoúčelový systém, jehož řídící počítač je zabudovaný do zařízení, které je jím ovládáno. Na rozdíl od univerzálních počítačů, které oplývají širokou funkčností, jsou vestavěná zařízení jednoúčelová s předem definovanou činností, kterou vykonávají po celou dobu své životnosti. Typickým představitelem takových zařízení je mobilní telefon, bezpečnostní alarm či router připojující domácnost k internetu.

Na vestavěná zařízení jsou kladeny často protichůdné požadavky. Vyžadujeme nízké náklady na výrobu (jsou vyráběny sériově ve velkých objemech), malou spotřebu (měřič rychlosti u silnice napájený solárními články se nesmí vybít, pokud je dlouhou dobu zataženo) a malé rozměry (vyšší výtěžnost z křemíkových wafferů, více plochy na čipu, která může být využita jinými prostředky). Na druhou stranu požadujeme vysoký výkon (HD kamera musí každou vteřinu zpracovat velký objem dat) či můžeme klást požadavky na fungování v reálném čase (detektory kouře chrání před požárem a urgují hasičskou

centrálu), na spolehlivost (roboti vyslaní do vesmíru se nesmí pokazit) či na bezpečnost (jaderný reaktor se nesmí přehřát přes kritickou mez).

Zdaleka nejzajímavějším se pro trh mikroprocesorů jeví raketový nástup internetu věcí.[3] Internet věcí je síť složená ze zařízení různých účelů, která v sobě obsahují elektroniku, software, senzory a která mezi sebou komunikují za účelem vyššího užitku. Jedná se o přirozený vývoj aktuální situace, kdy je již přes patnáct miliard zařízení napojených na internet. Jen během posledních pěti let vzrostl počet těchto zařízení tři sta krát. Například běžný moderní automobil využívá až sto senzorů různého typu.

Města v příštích dvaceti letech investují do vylepšení infrastruktury slučitelné s internetem věcí 41 biliónů dolarů. Přelomovou možností internetu věcí je mimo jiné projekt Smart City, jehož jednotlivé části se již testují například v Londýně či San José.

Dopravní tepny, křižovatky, mosty i parky, tato všechna místa vybavená detektory kvality ovzduší dovolí městům regulovat poplatek za vjezd do města v závažnosti na aktuálních rozptylových podmínkách. Osazení čidly s napojením na internet se už dočkala i známá dominanta Londýna Tower Bridge. Čidla na semaforech lze využít pro sledování dopravního toku, během špiček tak efektivně měnit propustnost v kritických lokalitách, a tím šetřit prostředky. Město s takovou inteligentní infrastrukturou bude jistě atraktivnější pro místní obyvatele i turisty.

Informace ze všudypřítomných čidel bude možné využívat i pro strategická plánování. Zjistíme, jak účinné je vysazování nových stromů v ulicích či která část města je nejlepším kandidátem pro vznik rekreační zóny.

S takovými jednoduše dostupnými zařízeními se nám otvírá netušené množství nových možností jak vylepšit kvalitu života, záleží opravdu jen na nápadu.

Jednou ze společností, které mají dispozice uspět na celosvětovém trhu internetu věcí, je původem brněnská společnost Codasip vyvíjející návrhové studio pro rychlé prototypování systémů na čipu včetně všech potřebných nástrojů pro komplexní verifikaci, profilaci, simulátorů i překladače.

Zákazníci se mohou rozhodnout pro návrh a realizaci specializovaného procesoru dle jejich požadavků nebo si mohou vybrat z portfolia rodiny již hotových procesorů Codix, které jsou jednoduše upravitelné. Díky efektivnímu popisu instrukční sady a mikroarchitektury v jazyce Codal je odebrání nepotřebné či přidání žádané funkcionality otázkou několika dní. Tím je výrazně zkrácena finančně i časově náročná doba vývoje a zařízení může být rychle přivedeno na trh.

Tématem mé diplomové práce je návrh mikroarchitektury procesoru Codix VLIW spojené s vytvořením jeho modelu v jazyce Codal a syntéza na platformu Xilinx ZYNQ. Výsledný model se zařadí do rodiny procesorů Codix a bude nabízen pro komerční účely.

Kapitola 1

Klasifikace procesorů

1.1 Podle instrukční sady

V dnešní době dělíme počítače do dvou základních kategorií podle toho, jaký typ sady strojových instrukcí používá jejich procesor. Počítače s komplexní instrukční sadou označujeme CISC (Complex Instruction Set Computer) a počítače s redukovaným souborem instrukcí nazýváme RISC (Reduced Instruction Set Computer). Dnes již prakticky neexistují čistě CISC nebo RISC procesory, téměř vždy jde o kompromis.[4][5][6]

1.1.1 CISC - Complex Instruction Set Computer

První počítače měly jednoduché a malé instrukční sady. Honba za zvýšením jejich výkonnosti vedla k vytváření složitějších a rozsáhlejších instrukčních sad. Navzdory delší době vykonávání jednotlivých komplexních instrukcí bylo pozorováním velice brzy zjištěno, že celý program se často provede rychleji. Příkladem komplexní instrukce je operace sčítání nad dvěma čísly v plovoucí řádové čárce nebo použití složitějšího adresného módu pro přímý přístup k prvkům pole. V případě, že se tyto dvě instrukce vyskytovaly v programu po sobě, bylo místo nich možné nahradit jinou, která sečetla všechna čísla v poli, a tím bylo dosaženo znatelného urychlení výpočtu. Snahou při tvorbě komplexních instrukcí byla lepší podpora konstrukcí vysokoúrovňových jazyků. Díky instrukcím ušitým na míru, se redukoval počet přístupů do pamětí, což vedlo k dalšímu navýšení výkonu.

Používání složitých instrukcí bylo lepší i proto, že provedení jednotlivých operací mohlo být v hardware často prováděno paralelně. V případě drahých, a v té době vysoce výkonných sálových počítačů, bylo přidání dalších hardwarových jednotek pro zvýšení schopností paralelního vykonávání instrukcí lehce ospravedlnitelné. Přímým důsledkem bylo, že velké a drahé počítače měly košatější sady strojových instrukcí než ty malé a levné.

Během několika dalších let se začaly projevovat problémy s kompatibilitou

napříč různě velkými instrukčními sadami. Vzrůstající cena vývoje softwaru vedla k potřebě co nejvíce sjednotit instrukční sady. To přímo vedlo k implementaci komplexních instrukčních sad i do malých počítačů, ve kterých se nekladl hlavní důraz na vysoký výkon, ale na co nejnižší cenu.

V padesátých letech minulého století rozpoznala společnost IBM, která v té době byla dominantní hráč v oblasti počítačové technologie, že podpora stejné instrukční sady v rodině procesorů vede k výhodám pro ně i jejich zákazníky. Pro tento stupeň kompatibility se vžil název architektura.

Nové procesorové řady, lišící se v ceně a rychlosti výpočtu, implementovaly stejnou architekturu a měly zaručenu programovou kompatibilitu.[7]

V roce 1951 Maurice V. Wilkes[8] publikoval knihu o programování počítačů, ve které navrhl nový přístup, který na dlouhou dobu vyřešil problém, jak na malých a levných počítačích zajistit plnou instrukční kompatibilitu. Návrh popisoval implementaci sady strojových instrukcí formou mikroprogramů v PROM paměti.

V roce 1964 přišla na světlo světa architektura IBM System/360, rodina procesorů s instrukcemi implementovanými tímtéž způsobem, který před více než dekádou navrhl Maurice V. Wilkes. Tato rodina počítačů, která zaznamenala velký úspěch, byla výkonnostně, ale i cenově, téměř o dva řády výše než konkurence. Implementace instrukcí formou hardware byla použita jen v nejdražších modelech.

Jednoduché procesory s takto interpretovanými instrukcemi přinesly řadu dalších výhod. Mezi tři nejdůležitější z nich patří možnost opravit špatně implementované instrukce. Druhou z možností bylo bezbolestné přidání nových instrukcí a to i poté, co procesor byl již převzat zákazníkem. Poslední z výhod byla možnost strukturovaného návrhu a testování nových komplexnějších instrukcí.

Technologie tranzistorů z polovodičových materiálů byla rok od roku levnější. Trh s počítači i jejich výkon rostly dramatickou rychlostí a požadavky na malé levné počítače hrály do karet procesorům s instrukční sadou implementovanou formou mikroprogramů. Ty po dlouho dobu staly běžnou metodou pro návrh standardních počítačů.

V sedmdesátých letech minulého století se analýzou vysokoúrovňových jazyků zjistila potřeba implementace nových strojových instrukcí za účelem zvýšení výkonnosti a efektivnosti tehdejších překladačů. Velká většina z nově přidaných instrukcí nikdy nebyla míněna pro ruční programování, ale nové vylepšené překladače je běžně používaly.

Za tímto účelem byly implementovány skupiny instrukcí s variabilní délkou, pomocí kterých se velice komfortně daly provádět konstrukce vyšších programovacích jazyků jako volání funkcí, kontrola smyček či složité adresné módy pro indexování polí či struktur. Velikou výhodou bylo značné omezení zdlouhavých přístupů do pomalých pamětí a zredukovaní velikosti programů.

Tento trend dosáhl svého vrcholu u procesoru VAX od Digital Equipment Corporation´s. Jeho architektura podporovala stovky instrukcí a více než dvě

stě různých adresných módů pro výběr instrukčních operandů. VAX byl od úplného počátku navržen pro interpretované vykonávání instrukcí a nikdo nebral v potaz vysoce výkonný model, který byl běžně řešen celý v hardware.

Díky velké specifičnosti bylo časté, že se podstatná část instrukcí téměř nepoužívala, případně přinášela jen marginální výhody. Procesory pracovaly s instrukcemi různých délek s mnoha parametry, kvůli tomu musely být implementovány složité dekodéry. Doba vykonání instrukce se výrazně lišila napříč komplikovanou instrukční sadou a vliv mělo i umístění uložených operandů.

Extenzivní rozvoj instrukčních sad, při kterém nové instrukce neustále rozšiřovaly původní sadu, vedl ke stavu, kdy rozšiřování pamětí PROM, bylo dále neúnosné.

1.1.2 RISC - Reduced Instruction Set Computer

Analýzou četností vykonávaných instrukcí na CISC počítačích se v sedmdesátých letech minulého století ukázalo, že tehdejší překladače a programátoři používají strojové instrukce velice nerovnoměrně. Nejfrekventovaněji používané z nich patřily jen do velmi úzké instrukční podsady.

		%
LOAD	čtení z paměti	26.6
STORE	zápis do paměti	15.6
Jcond	podmíněný skok	10.0
LOADA	načtení adresy	7.0
SUB	odčítání	5.8
CALL	volání podprogramu	5.3
SLL	bitový posun vlevo	3.6
IC	vložení znaku	3.2

		%
LOAD	čtení z paměti	27.3
Jcond	podmíněný skok	13.7
STORE	zápis do paměti	9.8
CMP	porovnání	6.2
LOADA	načtení adresy	6.1
SUB	odčítání	4.5
IC	vložení znaku	4.1
ADD	sčítání	3.7

Obrázek 1.1: Průměrná statická a dynamická četnost instrukcí procesoru IBM System/360, převzato z [4]

Na obrázku je vidět, že v programech procesoru IBM System/360 se více jak v polovině všech případů se používají pouze tři instrukce, ve třech čtvrtinách všech případů pak pouze osm. Četnost výskytu ostatních instrukcí byla malá, někdy jen v řádu promile. To vedlo ke snaze o vytvoření optimální instrukční sady její redukcí. V té době byl již přes deset let používán model procesoru s košatými instrukcemi, které byly implementované jako mikroprogramy v řadiči počítače, a málo kdo si dokázal představit lepší řešení.

Přesto v roce 1980 na univerzitě Berkeley, začal tým vedený Davidem Pattersonem navrhovat počítačové čipy, které interpretaci instrukcí nepoužívaly. O dva roky později spatřil světlo světa první procesor RISC I, zanedlouho následovaný vylepšenou verzí RISC II. Prvkem, který přitáhl pozornost širokého

publika byl fakt, že první z těchto procesorů měly instrukční sadu přibližně pouze šestinovou oproti VAX architektuře.

Tyto nové typy procesorů byly zřetelně rozdílné od těch, které byly v té době běžné. Jako veliká výhoda se ukázala absence potřeby zpětné kompatibility, neboť jiné stroje podobného typu neexistovaly. Návrháři měli volné ruce a mohli se soustředit na vytvoření nové instrukční sady, která dokáže zvýšit celkový výpočetní výkon počítače. Nejdříve byl kladen důraz na jednoduché instrukce, které by bylo možné rychle vykonat. Později začalo být z měření zřejmé, že hlavní roli ve výkonu nemá délka vykonávání instrukce, ale spíše celkový počet instrukcí, jejichž výpočet bude započatý během určitého času.

Následovala téměř náboženská válka mezi zastánci RISC a CISC architektur. První z nich argumentovali tím, že zatímco načtení operandů z paměti, jejich sečtení a uložení výsledku zabere v RISC architektuře čtyři instrukce a v CISC lze tyto operace provést jednou, průměrná doba vykonání jedné operace je desetinná, neboť instrukce nejsou interpretovány.

Proč tedy RISC architektury plně nenahradily své CISC předchůdce? Důvodem byly miliardy dolarů, které společnosti již dříve investovaly do řady CISC procesorů Intel. Ten agilně převzal nový nápad a podařilo se mu některé principy implementovat i do svých CISC procesorů. Počínaje s procesorem 486, začaly řídicí jednotky vykonávat jednoduché instrukce v pro ně navržených datových cestách, zatímco složité byly nadále implementovány formou mikroprogramů v řadiči procesoru. Výsledkem je, že jsou běžné instrukce provedeny rychle a méně časté pomalu. Přestože tento kombinovaný přístup není tak rychlý jako čistě RISC, získaný výkon byl dostatečně konkurenceschopný a byla zachována úplná kompatibilita pro původní programy.

To, co začalo jako projekt na univerzitě, významně ovlivnilo budoucnost návrhu procesorů. Je to již více než třicet let od doby, kdy se na trhu objevil první RISC procesor. Ideologie redukované instrukční sady s sebou přinesla řadu dobrých principů, jenž se staly fundamentální pro návrh procesorů s dnešní technologií. V případě, že by příští rok došlo k průlomu na poli pamětí a byl by vyvinut nový prototyp, který by byl schopen fungovat s dvakrát vyšší taktovací frekvencí, než je tomu u procesoru, téměř jistě by vznikly nové lépe vyhovující řešení. Proto je důležité, aby návrháři udržovali krok s technologickým vývojem, který může způsobit výraznou změnu v rovnováze počítačových dílů.

Podobně jako programátoři objektově orientovaného software využívají při programování návrhové vzory, tak i architekti systémů na čipu dodržují osvědčené principy návrhu za účelem zvýšení kvality výsledného zařízení.

Dedikovaný obvod pro všechny instrukce

Všechny běžné instrukce by mělo být možné provést na hardware a ne je interpretovat mikroinstrukcemi. Odstraněním interpretace se u většiny instrukcí dosáhne výrazného zrychlení. V případě procesorů s CISC instrukční sadou se komplikované instrukce rozdělí na elementární bloky, které mohou být vyko-

nány v řadě za sebou. Tento krok může způsobit zpomalení procesoru, ale u méně častých instrukcí to může být schůdné.

Maximální počet dokončených instrukcí za jednotku času

Jedním z klíčových triků pro zvýšení výkonnosti procesorů je snaha zahájit během každého hodinového taktu co nejvyšší počet nových instrukcí. Pokud dokážeme syntetizovat procesor do programovatelného hradlového pole FPGA a podaří se nám navrhnout mikroarchitekturu tak kvalitně, že každý takt hodin dokážeme při frekvenci 100 MHz započít novou instrukci, vytvoříme 100 MIPS (Milion Instruction Per Second) procesor nezávisle na délce jejich vykonání. Tento příklad ukazuje, že paralelismus, neboli vykonávání více instrukcí najednou hraje při zvyšování výkonnosti podstatnou roli.

Jednoduché dekódování instrukcí

Jestliže chceme v každém taktu rozpracovat co největší počet nových instrukcí, musíme je být schopni rychle dekódovat a připravit jejich operandy. Nejlepší možností je používat instrukce jednotné délky s malým počtem formátů, které vymezují význam jednotlivých bitů, a operačním kódem na stejné pozici.

Přístup do paměti pouze skrze load a store instrukce

Nejjednodušší způsob, jak rozbít komplexní instrukce do série jednoduchých, je požadovat, aby jejich operandy mohly pocházet pouze z registrového pole procesoru. Pro přenosy dat mezi registrovým polem a pamětí jsou vyhrazeny jednoúčelové instrukce load (načtení dat z paměti) a store (uložení dat do paměti). Tyto paměťové operace mají variabilní délku vykonání. Závisí na tom, zda se jsou dostupné v rychlých cache pamětech. V případě, kdy se s daty předtím nepracovalo nebo když byla v cache paměti nahrazena kvůli nedostatku místa jinými, musí se přistoupit do pomalé hlavní paměti. Jelikož přístup do paměti trvá dlouho a zpoždění může nebo nemusí nastat, vykonávají se tyto instrukce ve speciálních LSU (Load/Store Unit) jednotkách. Díky tomu se může souběžně probíhat zpracovávání jiných, na nich nezávislých, instrukcí.

Velké množství registrů

Navýšení počtu registru přímo souvisí s omezením komunikace s pamětí na load a store instrukce. Všechny ostatní instrukce používají buď přímé (hodnota operandu je zakódována přímo v binárním kódu instrukce) nebo registrové operandy. Načtená data z paměti či mezivýsledky výpočetních operací se musí nacházet v registrovém poli po celou dobu jejich aktuálnosti. V případě, kdy nám registry během výpočtu dojdou, musíme je uložit do paměti, abychom je po čase mohli zase obnovit. Tato operace se anglicky nazývá spilling a je velice pomalá. Proto navyšujeme počet registrů, aby k ní v ideálním případě docházelo jen ve výjimečných příležitostech.

Optimalizovaný překladač

Čím jednodušší navrhneme hardware, tím více čipů dokážeme vyrobit z jednoho křemíkového wafferu. S menší plochou často souvisí i nižší spotřeba, menší zahřívání, poruchovost a snadnější testování. Všechny tyto aspekty se

promítají do výsledné ceny produktu. Dlouhotrvající snahou je přenést složitost z hardware do software. Optimalizovaný překladač dokáže vhodně využít všechny dostupné prostředky a zachovat přibližně stejnou výkonnost.

Tyto principy se staly tradiční kostrou při návrhu RISC procesorů. Dnešní procesory jsou výhodnou kombinací dobrých vlastností RISC a CISC přístupů, které se nejvíce hodí pro konkrétní řešení. V souvislosti s tím mohou být dále rozšiřovány a jejich instrukční sada obohacována o specializované instrukce.

1.1.3 DSP - Digital Signal Processor

Digitální signálový procesor je specializovaný procesor, který je navržený pro účely zpracovávání digitálních signálů. Jeho hlavní úlohou je měření, filtrování a komprese spojitých analogových signálů. Tyto operace dokáže provést téměř každý univerzální počítač, ale digitální signálové procesory je předčí v rychlosti, spotřebě a efektivitě. Mezi hlavní domény DSP patří zpracování a komprese zvuku, zpracování obrazu, komprese videa, rozpoznávání mluveného slova, analýza finančních dat či biomedicína. V dnešní době se s tímto typem procesorů setkáváme zejména v mobilních telefonech, diktafonech či kamerách a fotoaparátech.[9][10]

Vstupem DSP je spojitý analogový signál, který je za pomoci A/D převodníku převeden do proudu diskrétních digitálních hodnot. Přestože je digitální zpracování signálů složitější, přináší výhody ve formě vyšší výpočetní síly, komprese a možnosti detekce a opravy chyb.

DSP dělíme podle použité aritmetiky na procesory pracující v celočíselné aritmetice a na procesory pracující v aritmetice s pevnou a plovoucí řádovou čárkou. Podpora celočíselné aritmetiky je sice levná, protože původní vstupní signál byl spojitý, ale algoritmy neustále narážejí na potřebu převádět reálná čísla na celá a výsledky operací se musí normalizovat. Vývoj digitálního signálního procesoru s čistě celočíselnou aritmetikou je náročný a drahý, proto se k němu přistupuje pouze ve vysoko objemové výrobě, kde klademe důraz na koncovou cenu součástky a vyšší počáteční náklady jsou tolerované. Digitální signální procesory s aritmetikou v plovoucí řádové čárce jsou složitější, mají vyšší spotřebu a jsou kusově dražší, ale zato poskytují vyšší komfort pro vývoj software. Podpora aritmetiky v pevné řádové čárce jsou kompromis mezi předchozími případy. Prakticky je však nelze jasně odlišit od procesorů pracujících v celočíselné aritmetice.

Na rozdíl od univerzálních procesorů je instrukční sada digitálních signálních procesorů značně omezená. Jelikož jsou kladeny požadavky na vysokou výkonnost a zařízení s DSP procesorem jsou zejména jednoúčelová, nepoužívá se pro překlad aplikací překladač, ale programy jsou ručně psané v jazyce strojových instrukcí. Za účelem zvýšení paralelismu a s tím související výkonnosti jsou podporovány SIMD (Single Instruction Multiple Data) instrukce, které dovolují provést stejnou operaci nad více registry, či jsou použity techniky více pipeline, které jsou běžné u VLIW (Very Long Instruction Processor) proce-

sorů, se kterými bude čtenář seznámen v jedné z následujících podkapitol. Dalším typickým znakem instrukční sady digitálních signálních procesorů je podpora operace MAC (Multiply-Accumulate), která je pravidelně používána při maticových operacích jako je filtrování signálů, jejich konvoluce, rychlá Fourierova transformace nebo výpočet polynomů pomocí algoritmu Hornerova schémata.

Algoritmy používané pro zpracování signálů jsou tvořeny velkým množstvím matematických operací, které se v rychlém sledu opakují nad proudem dat. Velmi často jsou kladeny požadavky na malou latenci, neboť zdrojová data jsou neustále nahrazována novými vzorky. Proto pokud jsou prováděné operace nad diskretizovanými digitálními daty tak výpočetně náročné, že je ani vysoko výkonný DSP procesor nestihne vykonat v reálném čase, je třeba data nejdříve uložit do paměti. Tento typ počítačů se vyznačuje Harvardskou architekturou. Díky odděleným pamětem je možné během jednoho hodinového taktu načíst větší množství operandů i instrukcí. Běžná je také podpora indexování kruhových čítačů, které jsou používány pro ukládání diskrétních hodnot signálů, pomocí operace modulo (zbytek po dělení).

Digitální signální procesory implementují mnoho zajímavých vlastností, které zlepšují zaplnění pipeline a zvyšují výkonnost:

Streamovaná data jsou ukládány do FIFO ve formě kruhového bufferu. Kruhový buffer je v hardware adresován za použití modulární aritmetiky, tím je obejita nutnost neustálé kontroly přetečení. Struktura pamětí je navržena pro lepší podporu streamovaných dat. Je podporován přímý přístup do paměti a programátoři využívají toho, že znají zapojení hierarchie cache pamětí a zpoždění v jejich částech. Digitální signální procesory nepoužívají virtuální paměť ani její ochranu. Zvýšený paralelismus díky vícenásobné přítomnosti aritmeticko-logických jednotek. Oddělená paměť pro program a data a vícenásobný přístup ke sběrnicím během jednoho taktu. Speciální bit-reversed adresný mód užitečný zejména při výpočtu rychlé Fourierovy transformace. Hardware podpora speciálních smyček bez penalizace při skocích. Dalším typickým rysem této architektury je použití saturované aritmetiky. Kde by aritmetická operace způsobila přetečení a z maximální hodnoty by se stala minimální, tam při použití saturované aritmetiky zůstane maximální možná hodnota.

1.1.4 Out of Order

Ve staticky řízené pipeline jsou instrukce vykonávány v řadě za sebou. Pokud mezi dvěma rozpracovanými instrukcemi existuje datová závislost, nezbývá než pipeline pozastavit, dokud nebude výsledek první operace k dispozici. Pozastavení pipeline blokuje rozpracování dalších instrukcí, které by mohly být zpracovány ve volných jednotkách, a dochází ke znatelným ztrátám výkonnosti.

Dynamické plánování je schopnost hardware měnit inteligentně pořadí instrukcí tak, aby byl redukován počet zastavení pipeline a byla zachována vysoká výkonnost procesoru, díky tomu je překladač velice zjednodušen. Nejvíce se výhody dynamického plánování projeví v případě proměnlivých zpoždění u datových cache pamětí, kdy prostoj v řádu desítek taktů může být využitý pro vykonávání užitečné práce. Další z výhod hodných zmínky je schopnost bezchybně vykonat program přeložený pro stejnou architekturu s jiným typem řízení pipeline.

Dynamické řízení pipeline přináší nové hazardy. Problém vzniká, pokud se za účelem vyšší výkonnosti vykoná pozdější instrukce, která změní hodnotu registru, jenž je zdrojem pro některou z předchozích instrukcí. Tyto Write after Read (WAR) a Write after Write (WAW) hazardy uměle vyžadují serializaci instrukcí a drasticky snižují užitek dynamického plánování instrukcí.

Tento problém však lze elegantně řešit technikou, která se nazývá přejmenování registrů. Na registr se obvykle díváme jako na pevně dané místo paměti. Netradičně na něj však můžeme pohlížet jako na identifikátor s pevným jménem, který může mít více míst v paměti. Základním požadavkem techniky přejmenování registrů je vyšší počet fyzických registrů než architekturálních (těch, které lze indexovat v instrukčních formátech). Zdrojové registry předcházejících instrukcí jsou namapovány na staré hodnoty, ty následující již bezproblémově používají nové hodnoty. Všechny informace potřebné pro přejmenování registrů (adresa instrukce, závislosti) jsou uloženy v překladové tabulce. Fyzické registry, které nemají žádné nevyřízené závislosti v překladové tabulce jsou následně uvolněny.[11]

1.1.5 VLIW - Very Long Instruction Word

S přibývajícími funkčními jednotkami pro vykonání jednotlivých instrukcí současně rostly nároky na propojení a řízení. To způsobovalo prodloužení kritických cest a kladlo vyšší omezení na pracovní frekvenci procesoru. Čím více bylo v jeden moment rozpracováno instrukcí, tím větší byly ztráty výkonnosti při skocích. Dynamické plánování instrukcí za běhu vyžadovalo speciální hardware bloky a velký počet registrů. Neseřazené výsledky ze všech funkčních jednotek musely být během jednoho hodinového taktu uloženy do rezervační stanice (synchronizační prvek Out of Order architektur). Aby bylo možné výsledky rychle uložit, rezervační stanice obsahovaly asociativní paměti, která byla složitá. Zvyšování paralelismu na úrovni instrukcí se v jedno jádrových superskalárních procesorech stávalo čím dál tím více komplikovanější.

Podobně jako instrukce procesorů CISC zaštiťují větší počet elemenetárních operací, může být paralelismus na úrovni instrukcí skryt očím programátora. Hlavní myšlenka, na které stojí procesory s velmi dlouhým instrukčním slovem, je explicitní specifikace instrukčního paralelismu v jazyce strojových instrukcí. Procesory typu VLIW spoléhají na překladač, který plánuje instrukce staticky, a tím přesouvají složitost z hardware do software. Inteli-

gentní překladač zná v době překladu strukturu mikroarchitektury, všechny instrukce a doby jejich vykonávání. Díky tomu má přehled o tom, které operace se mohou vykonávat souběžně, doby jejich vykonávání a tuto informaci explicitně zahrne do velmi dlouhé strojové instrukce.

Původně se procesory s velmi dlouhým instrukčním slovem vyskytovaly pouze ve vysoce výkonných univerzálních počítačích, neboť jejich nároky na tranzistory a spoje byly příliš vysoké pro běžné používání v přenosných vestavěných zařízeních. Do dnešní doby však technologie výroby hardware dostatečně postoupila a díky jejich velké atraktivitě se postupně začínáme setkávat s implementací procesorů VLIW i do přenosných vestavěných zařízení. Návrhář hardware může využít paralelismu instrukcí i tak, že zvýší výkonnost systému nad mez, která je nezbytná pro splnění nároků na zařízení, a následně sníží pracovní frekvenci. Zpomalení hodinového taktu je jeden z nejefektivnějších způsobů snižování spotřeby energie. Protože doba výdrže baterie na jedno nabití je u přenosných vestavěných zařízení důležitý aspekt, hraje tu paralelismus větší roli, než je tomu v případě serverů, u kterých se na celkovou spotřebu takový důraz neklade.

Každý typ architektury má své výhody a nevýhody. Výhodou staticky plánovaných instrukcí je komplexní kontrola instrukčních závislostí. Hardware může být následně bezstarostně zjednodušen o bloky, které se starají o detekci hazardů. V každém stupni pipeline se řeší pouze nejnutnější výpočty elementárních částí elementárních instrukcí. Aby bylo řízení procesoru co nejjednodušší, všechny instrukce mají konstantní šířku. V případě, kdy dojde k zastavení byť jen jediné funkční jednotky, musí být pozastaveno rozpracování celé paralelní série instrukcí ve všech pipeline, neboť procesory s velmi dlouhým instrukčním slovem zpracovávají instrukce ve všech pipeline synchronně.

Přestože překladače mohou instrukce se statickou prodlevou naplánovat pro zpracování v funkčních jednotkách tak, aby k pozastavení pipeline docházelo minimálně předpovídání, které přístupy k datům způsobí cache miss a s ním spojené dynamické zpoždění je velmi složité. Cache paměti mohou způsobit zastavení celé paralelně zpracovávané skupiny instrukci a při častých operacích s tímto typem paměti tak může docházet k neustálým synchronizacím, které radikální snižují celkovou výkonnost procesoru.

Souběžně vykonávané instrukce naráz načítají operandy z registrového pole a současně do něj zapisují při dokončení. Tím jsou kladeny požadavky na vysoký počet čtecích a zápisových portů, které se v hardware implementují duplikací paměťových buněk. Výsledné registrové pole je násobně veliké, spotřebovává více energie a je pomalejší. V případě kvalitního překladače s upraveným alokátorem registrů je možné tento problém částečně řešit rozdělením registrového pole na menší, která budou přidělena jednotlivým pipeline, a přidáním instrukcí pro přenos mezi registrovými poli.

Výkonnost procesorů s velmi dlouhým instrukčním slovem úzce závisí na schopnosti překladače naplánovat co nejvyšší využití funkčních jednotek v každé pipeline. Vysoké míry paralelismu je dosaženo zvětšením těl základních

bloků. Základní blok je část kódu, která má jeden vstup a jeden výstup. Je tedy zajištěno, že pokud se vykoná jeho první instrukce, vykonají se i všechny ostatní. Díky tomu jsou lehce analyzovatelné a jsou základním stavebním prvkem programů.

Čím více je instrukcí v základním bloku, tím efektivněji je může překladač naplánovat. Mezi techniky, které se používají k zvětšení těl základních bloků patří rozbalení smyček, při kterém je kód uvnitř smyčky násobně rozkopírován do sekvenční podoby a tím je odstraněna režie spojená s vyhodnocením podmínek a na nich závislých skocích. Existují i agresivní techniky globálního plánování instrukcí napříč více základních bloků.

Algoritmus globálního plánování je však značně složitější co se týká rozhodování, kdy je vhodné instrukce přesunout mezi základními bloky a kam je v souvislosti s přesunem nutné přidat opravný kód. V případě, že se velmi dlouhé instrukční slovo nepodaří naplno využít, jsou volné operace zakódovány jako instrukce NOP (No Operation). Rozbalováním smyček, ve kterých dochází k duplikaci často rozlehlých částí kódu, můžou při špatném plánování vzniknout velmi dlouhá instrukční slova, které obsahují množství zbytečných NOP instrukcí a násobně zvyšují paměťové nároky programů bez žádné přidané efektivity.

Komplikovaná je i binární kompatibilita mezi novými verzemi stejných procesorů s velmi dlouhým instrukčním slovem. Při striktním VLIW přístupu jsou ve strojovém kódu naplno využívány dostupné funkční jednotky a instrukce jsou plánovány se zřetelem na dobu jejich vykonávání. Proto přidání či ubrání funkčních jednotek či změna mikroarchitektury a s ní související změna dob vykonávání jednotlivých instrukcí vyžaduje nový binární program. Tento požadavek činí přenos programu napříč různými verzemi stejné VLIW architektury mnohem komplikovanější než tomu je u ostatních superskalárních architektur. Samozřejmě, pro využití výhod vyššího výkonu je nutné programy pro superskalární počítače opětovně přeložit, nicméně jsou až na výjimky zpravidla schopné vykonávat původní kód bez jakýchkoliv zásahů.[12]

1.2 Podle zaměření

1.2.1 CPU - Univerzální procesory

Univerzální procesory se téměř výhradně používají v osobních počítačích. Poskytují bohatý okruh softwarově implementovaných funkcí, neboť není dopředu zřejmé, které programy budou po dobu své aktivity vykonávat. Cenou za vysokou flexibilitu je vyšší cena a menší výkon.

1.2.2 ASIP - Aplikačně specifické procesory

Procesory s aplikačně specifickou instrukční sadou jsou silně optimalizované procesory vyráběné za účelem řešení jednoho specifického problému. Návrh

aplikačně specifických procesorů je dlouhodobá záležitost, která stojí nemalé peníze. Vysoké počáteční náklady se však záhy vrací ve formě levnějšího, rychlejšího a i menšího procesoru, který je dokonale splňuje počáteční požadavky.[13]

Tento typ kombinuje výhody flexibility univerzálních procesorů a výkonnosti aplikačně specifických obvodů ASIC. Jádro těchto procesorů je obvykle rozděleno na dvě části. V první z nich je obvodově realizována základní instrukční sada, ve druhé je ponechán prostor pro implementaci vlastních instrukcí, které bývají řešeny za použití syntézy do programovatelného hradlového pole.

Následující kapitola popisuje způsoby návrhu procesorů a jazykové prostředky, které jsou v průběhu vývoje používány za účelem automatizace.

1.3 Podle cílové platformy

Procesory se nejčastěji vyrábí jako aplikačně specifický integrovaný obvod (ASIC) nebo se syntetizují do programovatelného hradlového pole (FPGA). ASIC jsou zpravidla čistě digitální obvody ve standardní křemíkové CMOS technologii, které jsou navrženy a vyrobeny za účelem vykonávání určité aplikace. Oproti tomu FPGA poskytuje univerzální logické bloky, které lze opakovaně programově rekonfigurovat a spojit do obvodu, který bude vykonávat téměř libovolnou činnost.

Protože jsou ASIC zařízení ušitá na míru určité aplikaci, vyznačují se v porovnání s FPGA vyšší pracovní frekvencí. Rychlost provádění operací je však důležitá pouze v určitých zařízeních. Hlavním faktorem je hospodárnost výroby kusových zařízení v případě velkoobjemové sériové výroby. Aplikačně specifické integrované obvody jsou z principu vyrobeny pouze z nezbytně požadovaného množství tranzistorů, které dovolí vykonávání požadované práce za daných podmínek. V miliónových množstvích se ušetřená plocha na křemíkovém wafferu projeví až nečekaným způsobem. Jednorázové náklady na výrobu masek a licenční poplatky na návrhové a testovací prostředky stojí milióny dolarů. Navíc je třeba počítat i s platy vysoce kvalifikovaných návrhářů, které nejsou zanedbatelné. Někdy se vyplatí zaintegrovat několik poměrně běžných obvodů do jednoho, jenom aby se ušetřila plocha desky plošných spojů a počet osazovaných obvodů. Výrobní náklady na kus jsou ve výsledku velice malé. Aplikačně specifické integrované obvody se tedy vyrábí zejména v případě vysoko objemové produkce, kdy se vyplatí vynaložit vysoké počáteční fixní náklady na vývoj a výrobu, neboť kusová cena ASIC výrobku je velice příznivá.

Hlavní výhoda programovatelných hradlových polí spočívá v možnosti jejich rekonfigurace a absence potřeby drahých masek, díky tomu často používají k prototypování nových zařízení. Pokud je finální řešení syntetizováno do FPGA, je výrazně zkrácena jeho doba uvedení na trh, která je v silně konkurenčním prostředí důležitým aspektem. Oproti ASIC mají FPGA při stejné

funkci řádově vyšší spotřebu, což u větších obvodů způsobuje vyšší zahřívání a je třeba řešit problémy s chlazením. Ačkoliv jsou fixní náklady menší o výrobu masek, je cena finálního výrobku v technologii FPGA poměrně vysoká. FPGA má obrovskou výhodu v možnosti provádět vývoj a opravovat chyby pouhým přeprogramováním firmware, a to i v případě kdy je zařízení na trhu. V případě specifických integrovaných obvodů jsou náklady na opravu chyby astronomické. Zajímavou možností je i možnost konverze návrhu v FPGA do ASIC, která umožňuje spojit výhody FPGA v malosériové výrobě s výhodou ASIC pro velké série. Té se využívá zejména v případech, kdy se produkt na trhu uchytil a je po něm očekávaná dlouhodobá poptávka.

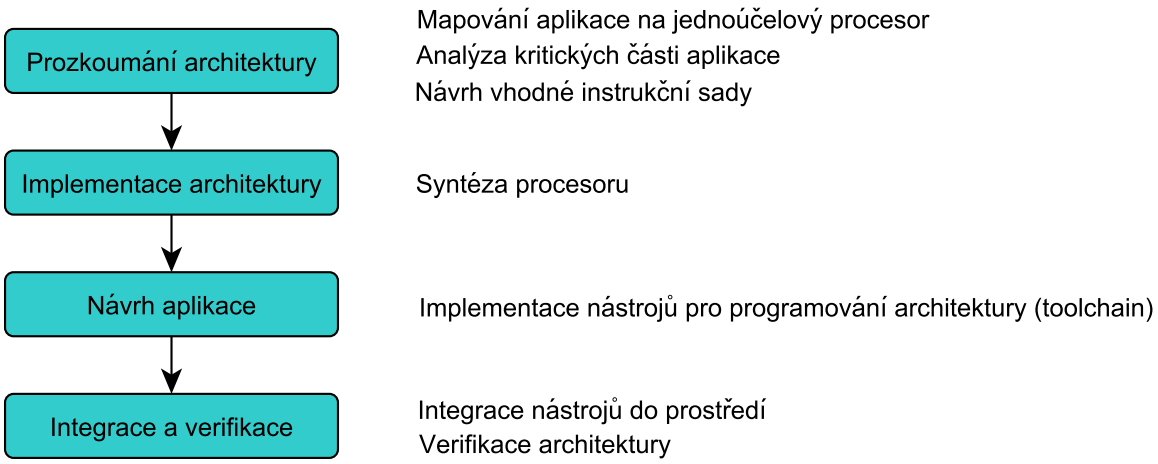
Je zřejmé, že obě technologie mají své výhody a je třeba pečlivě zhodnotit, který z nich více vyhovuje dané situaci.

Kapitola 2

Jazyky pro popis procesorů

Neustále se zdokonalující technologie výroby tranzistorů vede ke vzniku stále složitějších procesorů.

Tradiční metody návrhu, při kterých tým odborníků navrhuje zařízení s nízkou úrovní automatizace, je zachován dodnes. Jedná se o velice zdlouhavý proces, při kterém spolu často musejí spolupracovat vzdálené skupiny a výsledkem často není nejlepší možné řešení.[14]



Obrázek 2.1: Tradiční návrh architektury

2.1 HDL - Jazyky pro popis hardware

Lepší volbou je návrh za pomoci nástrojů, které poskytují vyšší úroveň automatizace. Jazyky pro popis hardware specifikují element času ve formě hran signálu hodinového taktu. Díky této vlastnosti bývají používány jak pro návrh, tak i pro simulaci.

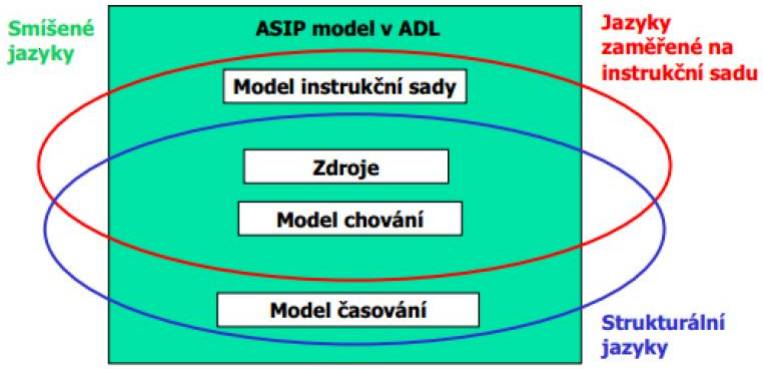
Stále komplexnější procesory či dokonce celé systémy na čipu se jimi popisují komplikovaně a pracně. Navíc jazyky pro popis hardware nepodporují automatické generování nástrojů pro programování (překladač, assembler, linker) a simulaci (debugger, profiler) architektury.

Nejznámějšími jazyky pro popis hardware je VHDL a Verilog.

2.2 ADL - Jazyky pro popis architektury

Předchozí problémy překonávají jazyky pro popis architektury.

Ty dovolují popisovat procesory s vyšší mírou abstrakce. Zároveň podporují konstrukce určené pro kontrolu výkonu, konzistence a jednoznačnosti formálního popisu. Z něj mohou některé jazyky pro popis architektur automatizovaně generovat toolchain.



Obrázek 2.2: Typy ADL jazyků. Převzato z [14]

Jazyky zaměřené na instrukční sadu jsou používány za účelem vývoje překladače z vyšších programovacích jazyků do jazyka strojových instrukcí dané architektury.

Strukturální jazyky popisují části procesoru a jejich propojení. Nástroje, které jsou pomocí těchto jazyků generované, pracují na nízké úrovni abstrakce a jsou řádově pomalejší.

Smíšené jazyky pro popis architektur kombinují oba výše zmíněné přístupy. Do této skupiny patří jazyk Codal, ve kterém jsem namodeloval procesor Codix VLIW na úrovni instrukcí a cyklů.

Kapitola 3

Jazyk Codal

Jazyk Codal, který spadá do kategorie smíšených jazyků pro popis architektur, byl vyvinut v rámci výzkumného projektu Lissom na Fakultě informačních technologií v Brně.

Je používán za účelem rychlého prototypování procesorů s aplikačně specifickou instrukční sadou. Kvůli vysoké automatizaci při generování toolchainu vyhovuje dlouhodobým požadavkům na souběžný návrh hardware a software.[15]

Popis procesoru v jazyce Codal je přeložen do interního formálního modelu, který je následně zpracováván generátory nástrojů (debugger, překladač, profiler, assembler, linker a další).

Vysoká míra abstrakce a podpora automatizace dovoluje rychlé změny v instrukční sadě i mikroarchitektuře. Typicky tak být vyzkoušeno více variant a pro výsledný procesor vybrána ta nejefektivnější kombinace.

Model procesoru je možné popsat na jedné ze čtyř úrovní složitosti:

- Instrukční model
- Model chování
- Časový model
- Strukturní model

Přestože se jednotlivé modely liší úrovní detailu, základní struktura je u všech stejná. Každý model musí obsahovat popis zdrojů, instrukční sady a událostí.

3.1 Popis zdrojů

Každý model povinně obsahuje všechny architekturální zdroje, mezi které patří registry, signály, porty, sběrnice, paměti a pipeline.

Zdroje mohou být propojeny s jinými, lze také specifikovat mapování paměti do určitého adresného prostoru.

3.2 Popis instrukcí a událostí

Povinné je i specifikování instrukční sady a popsání událostí Reset, Main a Halt.

Instrukce jsou popisovány za použití elementů, základních stavebních prvků jazyka Codal. Ty pak mohou být slučovány do skupin, ze kterých je vybudována celá instrukční sada procesoru.

Element obsahuje následující sekce:

- Assembler - popis textové reprezentace elementu, používána ke generování assembleru
- Binary - popis binární reprezentace instrukce či její části
- Semantics - popis akce v jazyce ANSI C, používána pro simulátor a překladač
- Return - popis návratové hodnoty elementu v jazyce ANSI C
- Timing - popis aktivace událostí provázaných se zpracováním elementu
- Start - obsahuje skupinu reprezentující instrukční sadu, používána pro assembler a disassembler
- Decoders - popisuje architekturu instrukčního dekodéru
- Reset - popis událostí, které se vykonají při resetu procesoru (nastavení programového čítače, nulování registrového pole ...)
- Main - popis událostí, které se vykonají s hodinovým taktem (načtení instrukce v IA modelu, pomocné výpisy ...)
- Halt - popis událostí, které se vykonají po skončení simulace (uložení výsledků do souboru, výpis registrového pole ...)

Kapitola 4

Codasip Studio

Codasip Studio je integrované vývojové prostřední, které pokrývá všechny aspekty návrhu procesorů s aplikačně specifickou instrukční sadou a víceprocesorových systémů na čipu popsaných v jazyce Codal.[16]

Codasip Studio poskytuje dva typy vysoce automatizovaných generátorů. Prvním z nich je generátor toolchainu, který by jinak musel být vytvářen programátory. Tím významně redukuje čas vývoje procesoru a s ním vynaložené prostředky. Pomocí druhého generátoru, který je hardwarový, se automatizovaně generuje plně syntetizovatelný systém na čipu v jazyce VHDL či Verilog.

Vysoká míra automatizace vede k jednoduchému a rychlému návrhu. Přirozený iterativní návrh poskytuje jednoduchý posun od algoritmu ke konečné implementaci procesoru s aplikačně specifickou instrukční sadou. Během každého kroku jsou automatizovaně generovány nástroje pro souběžný návrh hardware a software.

Vytvořit model na instrukční úrovni (IA) trvá pouhých několik dní. Z něj se generuje kompletní toolchain, se kterým okamžitě můžou začít pracovat návrhářské týmy.

Toolchain je sada nástrojů využívaná v procesu vývoje návrhu architektury:

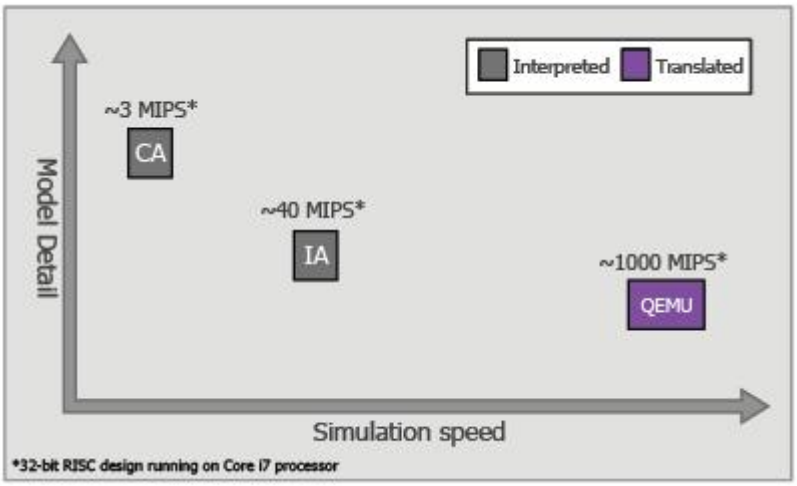
- Assembler
- Linker
- Překladač jazyka C/C++
- Simulátor
- Profiler
- Debugger
- Disassembler

Simulátor se používá pro rychlé odhalení chyb během počátečních fází návrhu. Codasip Studio vytváří několik různých typů simulátorů, které se liší v úrovni kontroly zdrojů.

Interpretovaný simulátor je založen na opakovaném načtení, dekódování a provedení instrukcí z paměti.

V případě IA modelu, simulátor pracuje na úrovni instrukcí a zanedbává spojitost s mikroarchitekturou procesoru. CA simulátor operuje na nižší úrovni a věrohodně využívá mikroarchitekturální zdroje procesoru. Cenou za větší složitost je pomalost.

Nejrychlejší je překládaný QEMU simulátor, který pracuje na úrovni instrukcí. Jeho generování probíhá ve dvou následujících krocích. Během prvního se analyzuje simulovaná aplikace a je jí vytvořen odpovídající popis v jazyce ANSI C. Ten je následně se zdroji procesoru a adresami začátků a konců všech základních bloků přeložen do formy simulátoru. Vysoké rychlosti se dosahuje přímým mapováním instrukční sady modelovaného procesoru na procesor vykonávající simulaci.



Obrázek 4.1: Srovnání rychlostí generovaných simulátorů, převzato z [17]

Assembler překládá jazyk strojových instrukcí procesoru do binárního objektového souboru. Assembler generovaný pomocí Codasip Studia naráz překládá dva jazyky. První z nich je jazyk specifikovaný syntaxí instrukční sady, druhým jazykem jsou popisovány symboly a direktivy.

Linker propojuje binární objektové soubory a doplňuje adresy, které v době překladu do assembleru nebyly známé.

Překladač transformuje jazyk C/C++ do architektury procesoru s aplikačně specifickou instrukční sadou. V první fázi se z IA modelu procesoru extrahuje sémantika všech instrukcí do jazyka LLVM. Ten může být následně

upraven podle potřeb nebo z něj lze ihned generovat překladač. Optimalizovaný překladač pracuje se specifikovanými latencemi pro přístup k instrukcím i datům. Díky tomu vhodně za sebou plánuje instrukce tak, aby byl výkon co nejvyšší. Tato vlastnost je klíčová zejména u VLIW architektur.

Profiler je klíčový nástroj při návrhu aplikačně specifických procesorů, který během simulace sbírá statistiky o počtu vykonaných instrukcí, průchodu kódem či době výpočtu. Tyto informace využívají návrháři k optimalizaci výkonu pomocí změn v instrukční sadě či částí kódu, které se nejčastěji opakují.

Disassembler poskytuje inverzní funkcionalitu k assembleru. Zpracovává binární objektový souboru a generuje jazyk strojových instrukcí daného procesoru.

Pokročilá vysokoúrovňová syntéza Codasip Studia generuje procesory, které výkonem předčí ručně optimalizované. Přidané aplikačně specifické instrukce jsou plně integrované do mikroarchitektury procesoru. Díky tomu poskytují výkon než je běžné u tradičních rozšiřitelných procesorů.

Součástí Codasip Studia je i kompletní generované UVM prostředí pro automatizovanou verifikaci, validaci procesorů včetně periferií.

Deliverables	Codasip ASIP		Traditional ASIP	
	RISC	VLIW	RISC	VLIW
Architecture Exploration & Modeling	40-60	60-80	40-60	60-80
Programming tool-chain	<1	<1	80	480
Virtual platform			80	120
Hardware representation (RTL)			100	150
UVM Verification environment			20-30	20-30
Verification	40-60		100-150	
Total	**80-120**	**100-140**	**420-500**	**930-1010**
Design Derivative*	**days**		**weeks**	

Obrázek 4.2: Srovnání doby návrhu a vývoje, , převzato z [17]

Z obrázku 4.2 je patrné, že úkoly, které běžně trvají týdny jsou při práci v Codasip Studiu redukovány na jednotky dnů.

Kapitola 5

Návrh

Z důvodu ochrany duševního vlastnictví společnosti Codasip, již vlastní veškerá práva na mnou vytvořený procesor Codix VLIW, budou v této kapitole odhaleny pouze obecné vlastnosti procesoru.

Základní vlastnosti architektury:

- Von Neumannova architektura (společná paměť pro program i data)
- Little endian formát dat
- Paralelní zpracování 4 souběžných instrukcí
- Dobrá podpora přidávání nových instrukcí
- Datové hazardy řešeny v překladači
- Řídicí hazardy řešeny v hardware
- Strukturální hazardy řešeny v překladači i hardware

Registry:

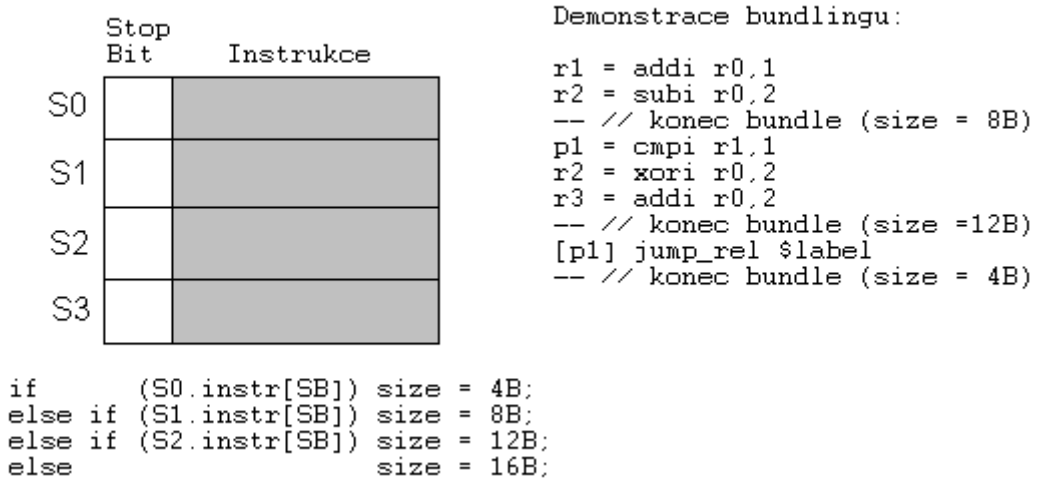
- 32x32b obecné registry (reg0 nese hodnotu 0)
- 8x1b predikátové registry (pred0 nese hodnotu false)
- 32x32b konfigurační registry

Paměti:

- instrukční cache (jeden port)
- datová cache (dva porty)

Procesor Codix VLIW je navržen s podporou predikce skoků. Ta je možná formou hardwarového dynamického prediktoru skoků či využitím statických profilovacích informací v těle skokových instrukcí za účelem zvolení pravděpodobnější cesty. Pokud je predikce správná, neztrácíme při skoku žádný výkon.

Druhou zajímavou vlastností je podpora komprimace programu v instrukční paměti. Během překladu z assembleru do binárního objektového souboru jsou z účinně nevyužité paralelní čtveřice instrukcí odstraněny NOP instrukce a tím dochází k významné redukci velikosti programu. Přímým následkem je zvýšení výkonu, neboť menší program se s vyšší pravděpodobností vměstná do programové cache paměti. Během načtení instrukce je pomocí prvního zarážecího bitu rozpoznána původní skupina instrukcí, která je následně doplněna instrukcemi NOP a rozeslána do příslušných linek.



Obrázek 5.1: Zjednodušená ukázka principu bundlingu

Kapitola 6

Implementace

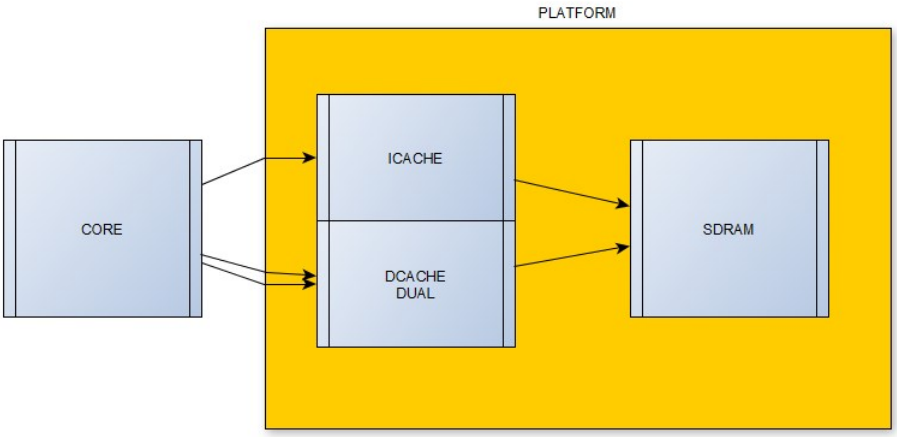
Procesor Codix VLIW jsem modeloval na úrovni instrukcí (instruction accurate model) i cyklů (cycle accurate model).

Obecně platí, že modelování na úrovni instrukcí je rychlejší a jednodušší. V případě prvního modelu je chování jednotlivých instrukcí popsáno v jazyce ANSI C a jsou opomenuty detailnější vlastnosti procesoru, jako je vnitřní struktura či časování. Z modelu popsaného na úrovni instrukcí jsou automatizovaně generované nástroje potřebné pro následné fáze vývoje jako je rychlý simulátor, překladač z jazyka C/C++ do jazyka strojových instrukcí procesoru Codix VLIW, assembler, disassembler či profiler.

K modelování na úrovni cyklů přistupujeme až v momentě, kdy za pomoci profileru obsaženého v IA toolchainu důkladně zkontrolujeme kvalitu instrukční sady. Při CA modelování popisujeme model na úrovni bližší reálnému hardware. Pracujeme s prvkem časování, chování jednotlivých instrukcí je v superskalárních procesorech rozděleno do různých stupňů pipeline. Model na úrovni instrukcí spolu s modelem na úrovni cyklů tvoří pár, ze kterého jsou vygenerovány veškeré nástroje nutné pro kvalitní vývoj a ve finále exportován popis v HDL jazyce, který ihned může být syntetizovatelný do hradlového pole FPGA či použit pro výrobu masek ASIC.

6.1 Implementace platformy

Kompletní systém na čipu je modelován ve dvou projektech. V prvním je popsáno jádro procesoru, v druhém pak zapojení procesoru a dalších komponent v rámci platformy. Toto rozdělení odpovídá metodologii firmy Codasip a je použito pro zvýšení podpory vícejádrových systému na čipu. Jednou namodelované jádro procesoru lze snadno vícenásobně referencovat.



Obrázek 6.1: Schéma platformy

6.1.1 Reference jádra

Na samotnou implementaci jádra se budu soustředit v následujících kapitolách. Při popisu platformy je nutné provést referenci procesorového jádra, které je součástí systému na čipu.

```
asip codix_vliw
{
    ia = "codix_vliw.ia";
    ca = "codix_vliw.ca";
};
```

6.1.2 Paměť

Pro implementaci paměti je použito klíčové slovo "memory". Každá paměť lze konfigurovat změnou jejích parametrů a ovlivňovat tak její výsledné vlastnosti.

Codix VLIW má Von Neumannovu architekturu. Paměť tedy slouží k uchování programu i dat. V popisu platformy jsou dobře viditelné dvě rozhraní, první slouží pro načítání instrukcí, druhé pro práci s daty. Paměť je adresována 32b adresou a dokáže během jednoho taktu vrátit celé velmi dlouhé instrukční slovo. Aby nebyla paměť moc složitá, podporuje pouze zarovnaný přístup.

```
memory mem {
    // 32b adresa, 128b data
    bits = { WORD32_W, QWORD128_W, BYTE8_W };
    size = MEMORY_SIZE;          // Velikost paměti
    endianness = "little";       // Little-Endian architektura
```

```
    unaligned = "no";           // Pouze zarovnaný přístup
    latency = { 1, 1 };         // Synchronní čtení i zápis

// Rozhranní pro program
    interface ibus {
        type = "clb:slave";     // Codasip Local Bus
        flag = "r";
    };

    // Rozhranní pro data
    interface dbus {
        type = "clb:slave";
        flag = "rw";
    };
};
```

Paměť pro program i data vede k velkému zpoždění. Platforma Codix VLIW proto obsahuje i malé oddělené cache paměti, které zkracují prodlevy při načítání a ukládání dat. V následujícím úseku kódu demonstruji implementaci dvouportové data cache paměti.

```
cache dcache {
// 32b adresa, 32b data
    bits = { WORD32_W, WORD32_W, BYTE8_W };
    size = DCACHE_SIZE;
    endianness = "little";
    unaligned = "no";
    latency = { 2, 2 };
    numways = DCACHE_NUMWAYS; // Míra asociativy
    linesize = DCACHE_LINESIZE; // Velikost cache line v bytech

    // Při plné cache paměti dojde k zápisu do hlavní paměti
    write_back = "yes";
    // Pokud při zápisu dat dojde ke cache miss, data jsou načtena z hlavní paměti
    write_allocate = "yes";
    rpl_policy = "round_robin"; // Politika nahrazování

    // Rozhraní do ASIP
    interface dbus_s0, dbus_s1 {
        type = "clb:slave";
        flag = "rw";
};

// Rozhranní do SDRAM
```

```
    interface sdram {
        bits = { WORD32_W, QWORD128_W, BYTE8_W };
        type = "clb:master";
        flag = "rw";
    };
};
```

6.1.3 Propojení komponent

Po instanciaci procesoru a nadefinování hlavní paměti a pomocných cache pamětí pro program a data nás čekají na úrovni platformy poslední dva úkoly. Těmi jsou propojení komponent a mapování jejich adres na sběrnici.

Pro propojení je použito klíčové slovo "connect". Na samotném pořadí rozhraní nezávisí.

```
connect dcache.dbus_s0 => codix_vliw.dbus_s0;
connect dcache.dbus_s1 => codix_vliw.dbus_s1;
connect dcache.sdram => mem.dbus;
```

V popisu se odkazuji na rozhraní patřící Codix VLIW procesoru. Ty jsou stejným způsobem nadefinována v ASIP projektu.

Na závěr je třeba specifikovat mapování adres pamětí do adresného prostoru.

```
address_space as_all
{
    bits = { WORD32_W, WORD32_W, BYTE8_W };
    interfaces = {
        ibus { type = "program"; };
        dbus_s0 { type = "data"; };
        dbus_s1 { type = "data"; };
};
endianness = "little";
};
```

Tímto je projekt platformy procesoru Codix VLIW hotov. V následující podkapitole budou popsány aspekty modelování procesorového jádra.

6.2 Implementace jádra

Procesor Codix VLIW je modelován ve dvou verzích modelu. Prvním z nich je model na úrovni instrukcí, druhým je model na úrovni cyklů. Přestože tyto modely popisují architekturu procesoru v různých úrovních abstrakce, využívají k výpočtu stejné základní zdroje.

6.2.1 Registry

Procesor Codix VLIW obsahuje pole obecných registrů, registrové pole predikátů a sadu speciálních registrů. Jelikož CA model pracuje s nízkou mírou abstrakce obsahuje navíc určité zdroje (např. interstage registry použité při pipeliningu), které se při modelování na úrovni instrukcí nepoužívají. Všechny architekturální registry, které může překladač použít pro uchovávání hodnot jsou označeny pomocí klíčového slova "arch".

Nearchitekturální registry slouží jako globální proměnné, kterých může být využito pro jednoduché manipulování s informacemi napříč modelem. Pro překladač jsou ale neviditelné.

```
// Programový čítač je speciální typ registru
program_counter bit[WORD32_W] pc;

// Registrové pole pro obecné operandy
arch register bit[WORD32_W] regs[RF_GPR_SIZE]
{ dataport = { RF_GPR_READ_PORT, RF_GPR_WRITE_PORT }; };

// Nearchitektuální registry pro uchování VLIW instrukce
register bit[WORD32_W] fetched_instr_slot0,
                       fetched_instr_slot1,
                       fetched_instr_slot2,
                       fetched_instr_slot3;
```

V případě CA modelu je nutné specifikovat vnitřní strukturu procesoru detailněji. Mezi jednotlivými stupni pipeline jsou umístěny tzv. interstage registry. Ty slouží jako předěl mezi paralelně vykonávanými instrukcemi. Procesory s velmi dlouhým instrukčním slovem jsou tvořeny clustery. Cluster je část hardware, která se vztahuje výhradně k jedné paralelní pipeline VLIW procesoru. Jelikož se určité zdroje opakovaně vyskytují ve všech clusterech, můžeme je popsat pomocí bloku uvedeného klíčovým slovem "cluster", a tím znatelně redukovat kód.

```
cluster s0, s1, s2, s3
{
    /********************** DECODE STAGE **********************/
    // Operační kódy instrukcí jsou dekódovány v každém clusteru
    // s0.id_opcode, s1.id_opcode, s2.id_opcode, s3.id_opcode
    signal bit[OPC_W] id_opcode;

    /********************** EXECUTE STAGE **********************/
    register bit[ALU_OP_W] ex_alu_op;

    // Clustery lze libovolně zanořovat
```

```
    cluster s0, s1
    {
        // Vytvoří se pouze s0.ex_mem_op a s1.ex_mem_op
        register bit[MEM_OP_W] ex_mem_op;
    }
    ..
    ..
}
```

Jakmile máme nadefinované všechny mikroarchitekturální registry, zapojíme je do pipeline.

```
pipeline pipe {
    FE: pc;
    ID: id_pc;
    RD: rd_pc, ....;
    EX: ....;
    WB: ....;
};
```

Registry specifikované v konkrétních stupních pipeline je možné hromadně ovládat přes název stupně následujícím způsobem.

```
semantics {
    if (rd_stall)      pipe.RD.stall();
    else if (rd_clear) pipe.RD.clear();
    ...
}
```

V každém stupni lze detekovat nutnost pozastavení pipeline z různých zdrojů (datové hazardy, sběrnice není připravena atd.). Sloučením všech možných zdrojů do jednoho obecného řídicího signálu pro každý stupeň, získáváme velice komfortní možnosti řízení procesoru.

6.2.2 Operandy

Hardwarové zdroje, mezi které patří například jednotlivé registry obecného registrového pole, mohou sloužit jako operandy instrukcí. V tom případě je nutné je modelovat jako element, který je základní stavební jednotka instrukční sady.

V následujícím příkladu předvedu modelování elementů, které reprezentují predikátové registry, jejich modifikátor a tyto stavební bloky použiji pro vytvoření elementu predikátu.

```
// "represents pregs" odkazuje na název hardwarového zdroje
element predreg represents pregs {
```

```
    // Syntax assembleru je "p0" až "p3".
    assembler { "p"~index=unsigned };

    // Konstanta RF_PRED_W určuje minimální počet bitů potřebných pro
    // zakódování všech predikátových registrů.
    binary { index=0b[RF_PRED_W] };

    // Při použití v sekci semantics vrátí index predikátového registru
    return { index; };
};

// Modifikátor určuje, zda je hodnota predikátového registru negovaná
element pmodifier_off { assembler { "" }; binary { 0b0 }; return { 0; }; };
element pmodifier_on  { assembler { "!"}; binary { 0b1 }; return { 1; }; };

// Na hierarchicky vyšší úrovni je set, která reprezentuje všechny v ní
// obsažené elementy.
set pmodifier = pmodifier_off, pmodifier_on;

// Predikáty použité v instrukcích rozhodují, zda se instrukce vykoná nebo
// bude zkonvertována na instrukci NOP.
element pcond
{
    // Reference výše definovaných elementů
    use pmodifier as pneg;
    use predreg;

    assembler { "[" pneg ~ predreg "]" };
    binary { pneg predreg };

    return {
        (pneg == 0 ? pread(predreg) : !pread(predreg));
    };
};
```

6.2.3 Události

Provedení instrukce je úzce spjato s modelováním tří povinných událostí Reset, Main a Halt.

Událost Reset definuje akce, které uvedou procesor do počátečního stavu. Programy pro Codix VLIW počítají s tím, že jsou všechna registrová pole inicializována na nulu.

Pokud je žádané provést nějakou akci pouze při simulaci, je nutné použít speciální pragma příkazy (práce s neinicializovanými hodnotami řídicích sig-

nálů při simulaci způsobuje varovná hlášení, ale v hardware jsou nepřiřazené signály v nule).

```
event reset {
    semantics {
        int i;

        pc = id_pc = rd_pc = 0;
        for (i = 0; i < RF_GPR_SIZE; i++)  regs[i]  = 0;
        for (i = 0; i < RF_PRED_SIZE; i++) pregs[i] = 0;

        int_enabled = 0;

        #pragma simulator
        {
            fe_stall = id_stall = rd_stall = ex_stall = wb_stall = 0;
            id_clear = rd_clear = ex_clear = wb_stall = 0;
        }
    };
};
```

Událost Main je aktivována na začátku každého hodinového cyklu procesoru. V případě modelu na úrovni instrukcí je tato událost místem, kde je nečteno a dekódováno velmi dlouhé instrukční slovo a inkrementována hodnota programového čítače.

```
event main  // ia
{
    // Reference instrukční sady
    // Sloty Codix VLIW podporují jiný okruh instrukcí.
    use slot0_asm;
    use slot1_asm;
    use slot2_asm;
    use slot3_asm;

    // Počáteční místo pro dekódování instrukcí assemblerem
    start {
        { slot0_asm; }
        { slot1_asm; }
        { slot2_asm; }
        { slot3_asm; }
    };

    // Dekodér pro každou pipeline VLIW procesoru.
```

```
    decoders {
        { slot0_asm(fetched_instr_slot0); }
        { slot1_asm(fetched_instr_slot1); }
        { slot2_asm(fetched_instr_slot2); }
        { slot3_asm(fetched_instr_slot3); }
    };

    // Načtení velmi dlouhého instrukčního slova a inkrementace PC
    semantics {
        int pc_offset;
        uint128 vliw;

        vliw = ibus[pc];

        fetched_instr_slot0 = (vliw             ) & INSTR_MASK;
        fetched_instr_slot1 = (vliw >> 1*INSTR_W) & INSTR_MASK;
        fetched_instr_slot2 = (vliw >> 2*INSTR_W) & INSTR_MASK;
        fetched_instr_slot3 = (vliw >> 3*INSTR_W) & INSTR_MASK;

        pc_fetched = pc;
        pc += 4*INSTR_W;
    };
};
```

Poslední instrukcí každého programu je operace halt, která ukončí simulaci a uloží výsledek simulace do souboru.

```
event halt {
    semantics {
        printf("============= Return value =============");
        printf("Result = %d \n", regs[RETVAL_REG]);
        codasip_store_exit_code(regs[RETVAL_REG] & 0xFF);
    }
};
```

V následující kapitole bude demonstrováno implementování instrukce LDBS (Load Byte Signed) patřící do instrukční sady procesoru Codix VLIW.

6.3 Implementace instrukce LDBS

Instrukce LDBS (Load Byte Signed) je podporována pouze v prvních dvou slotech architektury Codix VLIW, neboť bylo za použití profileru zjištěno, že přítomnost LSU (Load/Store Unit) ve každém slotu nepřináší významné zvýšení výkonu.

Instrukce pro načtení znaménkového bajtu z paměti je v instrukční sadě podporována ve dvou následujících formátech:

Formát	31	30..25	24..21	20..18	17..13	12..8	7..5	4..0
R3	sb:1	opc:10		pred:3	dst	src1	src2_am	src2
LDST_IMM12	sb:1	opc:6	imm:4	pred:3	dst	src	imm:8	

Vysvětlivky:

- sb - Stop Bit
- opc - Operační kód
- pred - Predikát
- dst, src - Indexy registrů
- imm - Operand ve formě přímé hodnoty

Jelikož má instrukce LDBS dva instrukční formáty, je ideálním adeptem na demonstraci rozdílů při implementaci IA a CA modelů.

Instrukce LDBS ve formátu R3 bude prezentována na úrovni instrukcí, ve formátu LDST_IMM12 pak na úrovni cyklů. Čtenář tak získá povědomí o tom, jak se jednotlivé modely liší, a proč je modelování na úrovni cyklů časově náročnější.

6.3.1 Formát R3 - IA model

Formát R3 specifikuje všechny elementy, které je třeba namodelovat a referencovat v popisu instrukce LDBS.

Prvním z nich je stopbit, který označuje poslední instrukci ve velmi dlouhém instrukčním slově. Díky tomu neobsahuje výsledný generovaný program žádné neužitečné instrukce a jsou zmenšeny paměťové nároky aplikace.

Formát R3 pracuje s dlouhým operačním kódem na deseti bitech, neboť je podporován ve velkém množství instrukcí.

Predikáty byly již namodelovány v předcházející sekci.

Formát R3 pracuje se třemi registrovými operandy. Operand dst reprezentuje index cílového registru, src1 a src2 popisují dva zdrojové operandy. Poslední zmíněným operand je možné modifikovat adresným módem.

```
// Stopbit
element stopbit_false { assembler { "" }; binary { 0b0 }; return { 0; }; };
element stopbit_true { assembler { "" }; binary { 0b1 }; return { 1; }; };

set stopbit = stopbit_false, stopbit_true;
```

```
// Dlouhý operační kód
element opc_ldx_bs {
    assembler { "ldx_bs" };
    binary { OPC10_LDX_BS:OPC10_W };
    return { OPC10_LDX_BS; };
};

// Registrové pole
element gpreg represents regs {
    assembler { "r"~index=unsigned };
    binary { index=0b[RF_GPR_W] };
    return { index; };
};

// Adresný mód druhého registrového operandu
DEF_SRC2_AM_REG(AM_SHL0)
DEF_SRC2_AM_SHL(AM_SHL1, "shl1", 1)

set src2_am = src2_am_reg, src2_am_shl1;
```

Nyní, když máme popsané všechny potřebné prvky, můžeme pokračovat samotným popisem instrukce pro načtení znaménkového bajtu z paměti.

```
element instr_r3_ldx_bs_ia {
    use stopbit as sb;
    use opc_ldx_bs as opc;
    use pcond as pred;
    use gpreg as dst, src1;
    use src2_am;

    assembler { pred dst "=" opc "[" src1 "+" src2_am "]" sb };
    binary { sb opcode pred dst src1 src2_am };
    semantics {
        uint32 address;
        uint32 aligned_address;
        uint32 byte_offset;
        uint32 data;

        // Latence instrukce
        #pragma compiler {
            codasip_compiler_schedule_class(cl_ldst_load_r);
        }

        // Instrukce se vykoná pouze pokud je predikovaná
```

```
        if (pred) {
            // Adresu bajtu je třeba rozdělit na bázi a offset
            address = rread(src1) + src2_am;
            base_address = address & ~MASK_BYTE;
            byte_offset = address & MASK_BYTE;

            // Načtená data jsou znaménkově rozšířena podle MSB bajtu
            data = SEXT8TO32(
                       // Přístup na datové rozhranní
                       dbus_s0.read(base_address, byte_offset, 1)
                   );

            // Uložení dat do cílového registru
            regs[dst] = data;
        }
    };
};
```

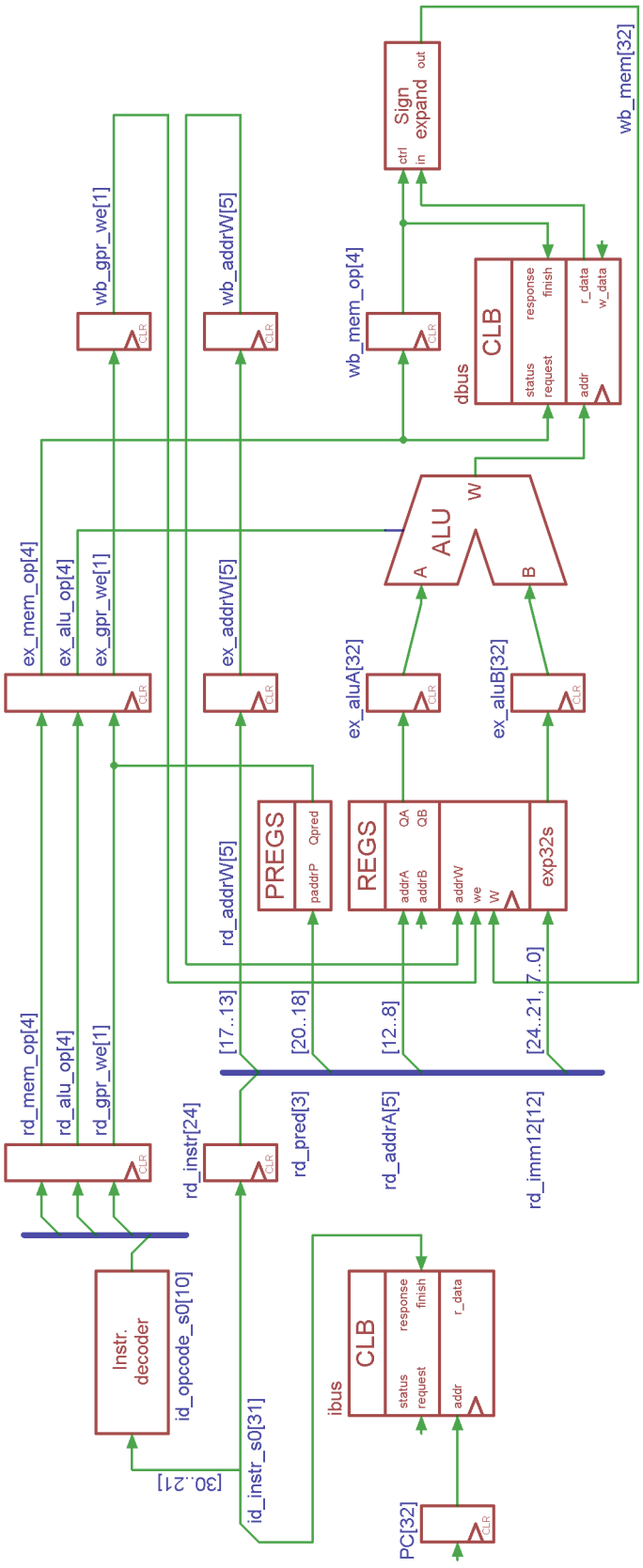
Popis instrukcí v IA modelu je jednoduchá a přímá činnost. Model složitého systému na čipu lze namodelovat v řádu dní. Jakmile je model hotov, je ihned možné generovat nástroje potřebné pro další vývoj procesoru.

6.3.2 Formát LDST_IMM12 - CA model

Implementování instrukce v modelu na úrovni cyklů je řádově složitější proces. V první fázi je vytvořeno schéma mikroarchitekturálních zdrojů s důrazem na sdílení maximálního počtu datových cest instrukcemi.

6.3.2.1 Schéma mikroarchitektury

Za účelem demonstrace jsem připravil zjednodušené schéma mikroarchitektury, na kterém předvedu modelování instrukce LDBS.



Obrázek 6.2: Zjednodušené schéma pro demonstraci instrukce LDBS

6.3.2.2 Zdroje mikroarchitektury

Jakmile máme připravené schéma, můžeme podle něj začít modelovat potřebné zdroje.

```
program_counter bit[32] pc;     // Programový čítač
cluster s0 {
// ID STAGE
    signal bit[32] id_instr;    // Instrukce slotu 0
    signal bit[10] id_opcode;   // Operační kód instrukce

    // RD STAGE
    signal bit[5]  rd_addrA, rd_addrW; // Indexy registrů
    signal bit[3]  rd_pred;     // Index predikátu
    signal bit[12] rd_imm12;    // Přímý operand

    register bit[4]  rd_alu_op, rd_mem_op; // Operace FU
    register bit[1]  rd_gpr_we; // Povolovací signál pro zápis
    register bit[24] rd_instr;

    // EX STAGE
    register bit[4]  ex_alu_op, ex_mem_op;
    register bit[1]  ex_gpr_we;
    register bit[5]  ex_addrW;
    register bit[32] ex_aluA, ex_aluB;

    // WB STAGE
    signal bit[32]   wb_mem;     // Znaménkově rozšířený bajt

    register bit[4]  wb_mem_op,
    register bit[1]  wb_gpr_we;
    register bit[5]  wb_addrW;
};
```

Namodelované zdroje přiřadíme do jednotlivých stupňů pipeline následujícím způsobem.

```
pipeline pipe {
    FE: ;
    ID: ;
    RD: s0.rd_alu_op, s0.rd_mem_op, s0.rd_gpr_we, s0.rd_instr;
    EX: s0.ex_alu_op, s0.ex_mem_op, ex.rd_gpr_we, s0.ex_addrW,
        s0.ex_aluA, s0.ex_aluB;
    WB: s0.wb_mem_op, s0.wb_gpr_we, s0.wb_addrW;
};
```

6.3.2.3 Dekodér

Jednotlivé instrukce instrukční sady jsou rozpoznatelné díky unikátnímu operačnímu kódu. Ten slouží jako vstup instrukčního dekodéru, který generuje řídicí signály, jenž nastaví multiplexory v pipeline a vytvoří tak správnou cestu pro konkrétní instrukce.

Pokud jsou operační kódy dostatečně široké, je možné zakódovat řídicí signály přímo. Výsledkem je minimalistický dekodér, který šetří zdroje a může zkracovat kritickou cestu.

Druhým přístupem je přidělení operačních kódů v řadě za sebou. Takový přístup značně zjednodušuje návrh, má však horší výsledky.

Cílem jader procesorů Codix je snadná rozšiřitelnost o nové instrukce. Pokud není instrukční sada neměnná, přináší první přístup přidělování operačních kódů problémy. Chování nové instrukce nemůže být během původního návrhu zohledněno a řídicí signály obsažené v jejím operačním kódu nekorespondují s její funkčností.

```
// Definice operačního kódu
element opcode_ldbs {
    assembler { "ldbs" };
    binary { OPC6_LDBS:OPC6_W };
    return { OPC6_LDBS; };
};

element instr_ldbs_ca {
    use opcode_ldbs as opc;

    assembler { opc };
    binary { opc 0b[4] }; // 6b opc + dontcare 4b
    semantics {
        rd_mem_op = MEM_LDBS;
        rd_alu_op = ALU_ADD;
        rd_gpr_we = TRUE;
    };
};
```

6.3.2.4 Provedení instrukce

Vykonání instrukce LDBS začíná s novým hodinovým taktem, při kterém se aktivuje událost Main. Upozorňuji, že za účelem lepší čitelnosti z modelu abstrahována většina řídicí mechaniky.

```
event main {
    use fe;
    use id;
```

```
    use rd_s0 as rd_s0(s0);
    use ex_s0 as ex_s0(s0);
    use wb_s0 as wb_s0(s0);

    // Pipeline je simulována od konce
    timing {
        wb_s0;
        ex_s0;
        rd_s0;
        id;
        fe;
    };
};
```

Ve stupni Fetch dochází k vystavení hodnoty programového čítače na rozhraní instrukční cache paměti.

```
/**** Fetch Stage ****/
event fe : pipe.FE {
    semantics {
        // Žádost o instrukci
        ibus.request(CP_RQ_READ, pc);
    };
};
```

Ve stupni Decode vrací instrukční paměť instrukci LDBS, jejíž operační kód je poslán na vstupy dekodéru. Ten identifikuje danou instrukci a nastaví řídicí signály.

```
/**** Decode Stage ****/
event id : pipe.ID {
    use slot0_hw;

    semantics {
        uint128 id_vliw;

        // Převzetí velmi dlouhého instrukčního slova
        ibus.ifinish(CP_FI_COMPLETE, vliw);

        // Vymaskování instrukce
        s0.id_instr = id_vliw & INSTR_MASK;

        // Vymaskování operačního kódu
        s0.id_opcode = (s0.id_instr >> OPC_POS) & OPC_MASK;
```

```
    };

    decoders {
        // Operační kód nastavíme na vstup dekodéru
        { slot0_hw(s0.id_opcode); }
        ...
    };
};
```

Hlavním úkolem Read stupně je získat a nastavit správné operandy na vstup aritmeticko-logické jednotky.

```
/**** Read Stage ****/
event rd_s0 : pipe.RD {
    semantics {
        // Instrukci rozbijeme na jednotlivé elementy
        rd_imm12 = (rd_instr >> 13 & 0xF00) | (rd_instr & 0x0FF);
        rd_addrW = (rd_instr >> 13) & RF_GPR_ADDR_MASK;
        rd_addrA = (rd_instr >> 8)  & RF_GPR_ADDR_MASK;
        rd_pred  = (rd_instr >> 18) & RF_PRED_ADDR_MASK;

        // Nastavíme vstupy interstage registrů
        ex_mem_op = rd_mem_op;
        ex_alu_op = rd_alu_op;
        ex_gpr_we = rd_gpr_we;
        ex_addrW  = rd_addrW;

        ex_aluA = regs[rd_addrA];
        ex_aluB = SEXT12TO32(rd_imm12);
    };
};
```

Při simulace Execute stupně je vykonává ALU operaci sčítání, která byla pro instrukci LDBS popsána v dekodéru. Tím je vypočtena adresa, ze které budeme žádat data.

```
/**** Execure Stage ****/
event ex_s0 : pipe.EX {
    semantics {
        uint32 address, base_address, byte_offset;

        // V ALU provedeme operaci sčítání
        switch (ex_alu_op) {
        case ALU_ADD :
```

```
            address = ex_aluA + ex_aluB; break;
        default:
            address = 0;
    }

    // Stejně jako v IA modelu požádáme o jeden byte
    base_address = address & ~MASK_BYTE;
    byte_offset = address & MASK_BYTE;

    dbus_s0.request(ex_mem_op, base_address, byte_offset, BYTE_W);

    // Nastavíme vstupy interstage registrů
    wb_mem_op = ex_mem_op;
    wb_gpr_we = ex_gpr_we;
    wb_addrW = ex_addrW;
    };
};
```

V posledním stupni pipeline načteme požadovaná data, znaménkově je rozšíříme a uložíme do registrového pole.

```
/**** Writeback Stage ****/
event wb_s0 : pipe.WB {
    semantics {
        uint32 data;

        // Načtení dat
        dbus_s0.ifinish(CP_FI_COMPLETE, data);

        // Znaménkové rozšíření
        switch (wb_mem_op) {
            case MEM_LDBS: wb_mem = SEXT8TO32(data);  break;
            case MEM_LDHS: wb_mem = SEXT16TO32(data); break;
            default:       wb_mem = 0;
        }

        // Zápis do registrového pole
        if (wb_gpr_we) {
            regs[wb_addrW] = wb_mem;
        }
    };
};
```

Na příkladu instrukce LDBS byl předveden rozdíl ve složitosti modelování instrukcí v IA a CA modelu. Ve skutečnosti je rozdíl ještě markantnější, neboť

jsem při modelování CA instrukce z důvodu přehlednosti abstrahoval řízení pipeline (když instrukce či data nejsou připraveny, je třeba pipeline pozastavit a korektně implementovat mechanismus načítání instrukcí) a multiplexory na datové cestě.

Kapitola 7

Měření výkonu - benchmarky

Výkon mnou navrženého hardware jsem srovnal vůči druhému nejsilnějšímu rozšiřitelnému procesoru firmy Codasip s názvem Codix RISC na sadě typických benchmarků. Pro demonstraci efektivnosti redukce kódu metodou zmenšování pouze z části zaplněných velmi dlouhých instrukčních slov uvádím hodnoty i pro případ, kdy byla tato vlastnost deaktivována. Všechny benchmarky byly přeloženy automaticky generovaným překladačem z mého IA modelu procesoru Codix VLIW s úrovní optimalizace O3.

Tabulka 7.1: Výsledky benchmark

	RISC		VLIW		VLIW opt	
Benchmark	takty	paměť [B]	takty	paměť [B]	takty	paměť [B]
coremark	40 671 366	20 396	31 035 592	52 864	30 971 364	19 100
dhrystone	5 240 191	10 656	2 870 096	26 864	2 860 096	9 700
bitcnt	18 942 069	7 508	17 920 046	20 416	17 920 046	6 992
crc	2 500 079	7 636	2 200 053	20 816	2 200 053	7 136
dijkstra	686 413	8 140	514 480	21 840	512 421	7 628
quicksort	12 022 007	7 812	11 781 459	21 264	11 755 807	7 332
sha	5 714 099	9 316	2 830 060	24 240	2 830 060	8 756
isqrt	329	7 548	241	20 512	241	7 512
md5	82 623 821	15 376	40 837 433	39 712	40 837 433	15 696
pc1	21 539 126	13 128	7 839 729	30 592	7 429 691	13 456
rc4	19 782 273	11 172	9 295 886	27 808	9 295 886	9 888

Po analýze benchmarků docházíme k několika zajímavým zjištěním. Přestože Codix VLIW dokáže najednou dokončit až čtyři instrukce, je jeho opravdový výkon menší. Velice záleží na paralelizovatelnosti úlohy. V případě benchmarku bitcnt došlo pouze k marginálnímu zvýšení výkonu, naopak u algoritmu md5 došlo k více než dvojnásobnému zrychlení. Je nezbytné upozornit, že při reálných optimalizacích programů dochází k přepisování kódu za účelem

zvýšení potenciálního paralelismu, proto je nutné tyto výsledky považovat za orientační.

Druhým zajímavým jevem je zmenšení celkové výsledné velikosti programu v průměru o úžasných 170% při zapnuté redukci velmi dlouhých instrukčních slov. To v jiném úhlu pohledu také znamená, že na benchmark programech byl schopen automaticky generovaný překladač naplánovat pouze 1,5 z maximálních 4 instrukcí za takt.

V souvislosti se zmenšením paměťové náročnosti došlo v některých případech k mírnému zrychlení při vykonávání programů v důsledku nižších výpadků instrukční cache paměti.

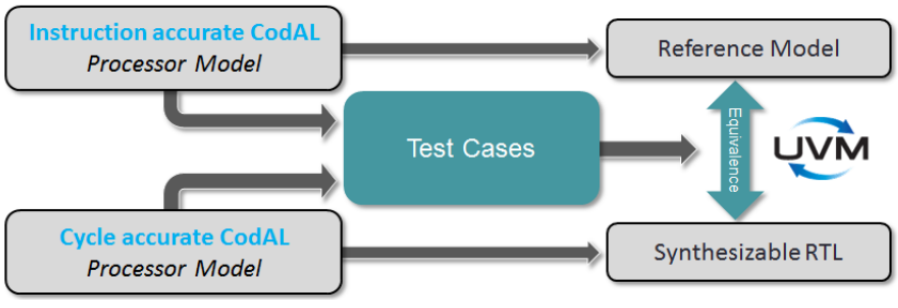
Kapitola 8

Verifikace

Funkční verifikace je metoda postavená na simulaci a použití pokročilých verifikačních technik. Verifikační prostředí je zpravidla popsáno ve specializovaných jazycích pro verifikaci hardware, mezi které patří např. SystemVerilog, SystemC.[17]

Verifikované zařízení DUT bývá popsáno ve VHDL, Verilog či SystemC jazyku. Během funkční verifikace se aplikací náhodných hodnot z omezeného rozsahu kontroluje chování a výstupy DUT vůči referenčnímu modelu.

Procesor Codix VLIW byl verifikován v UVM verifikačním prostředí automaticky generovaném Codasip Studiem. Jako referenční model sloužil simulátor z IA model a DUT byla VHDL reprezentace SoC s procesorem Codix VLIW generovaná z CA modelu.



Obrázek 8.1: Princip funkční verifikace„ převzato z [17]

Funkční verifikaci je možné provést použitím:

- Benchmark aplikací (standardní testovací sada Codasip)
- Vlastních aplikací
- Náhodně generovaných aplikací (generátor random testů integrovaný do Codasip Studia)

Po evaluaci každé aplikace se porovnává obsah registrů a paměti mezi DUT a referenčním modelem. Když se objeví neshoda, postupuje se k lokalizaci chyby. Ta se vykonává buď analýzou wave grafu v simulátoru QuestaSim, pomocnými výpisy ve verifikačním prostředí nebo pomocí sady formálních tvrzení. Ty totiž nahlašují chyby ihned v momentu jejich porušení, a tím je pomáhají lokalizovat.

Druhou nezbytnou částí je pokrytí kódu a funkční pokrytí. Cílem je změřit, do jaké míry byl procesor ověřený: zda se provedly všechny řádky kódu, aplikace vyvolaly každou instrukci instrukční sady, či se provedli všechny příkazy na sběrnici.

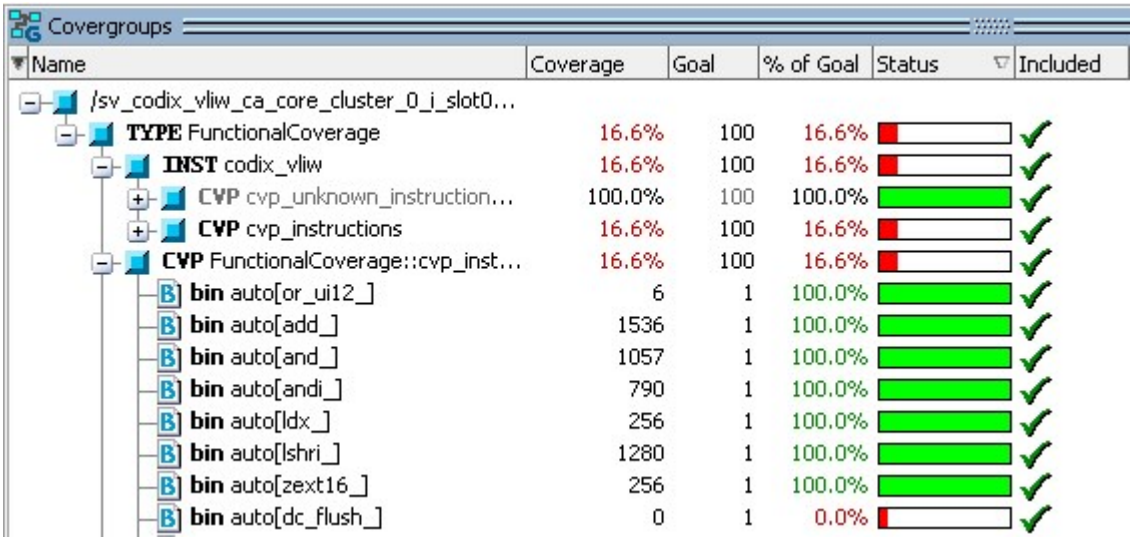
V případě funkční verifikace se zapnutou kontrolou formálních tvrzení na příkladě testu bitcount, který byl použit již v předchozí kapitole při analýze doby výpočtu a velikosti programu, získáme následující výstupy.

```
# UVM_ERROR codix_vliw_ca_t_abv_probe.sv(43) @ 52570 ns:
    reporter [ASSERTION_ERROR_UNKNOWN] An X value occurred on
    top.dut.HDL_DUT_U.codix_vliw.p_error.
    This may cause an unexpected behavior.
# UVM_INFO halt_detection.svh(78) @ 52580 ns:
    ASIP 'codix_vliw' finished after 5183 cycles.

# UVM_INFO codix_vliw_ca_t_env//scoreboard.svh(108) @ 52580 ns:
    The result for 'codix_vliw' is OK, 0/0 miscompares!
# UVM_INFO codix_vliw_platform_ca_t_env//scoreboard.svh(129) @ 52580 ns:
    The result for 'codix_vliw_platform_ca_t' is OK, 0/0 miscompares!
# UVM_INFO codix_vliw_platform_ca_t_env//scoreboard.svh(135) @ 52580 ns:
    The result for 'codix_vliw.rf_gpr' is OK, 0/32 miscompares!
# UVM_INFO codix_vliw_platform_ca_t_env//scoreboard.svh(141) @ 52580 ns:
    The result for 'codix_vliw.rf_pred' is OK, 0/8 miscompares!
# UVM_INFO codix_vliw_platform_ca_t_env//scoreboard.svh(147) @ 52580 ns:
    The result for 'mem' is OK, 0/8192 miscompares!
```

Na prvním řádku výpisu nás kontrola formálních tvrzení upozorňuje na otevřený port p_error. Všechna formální tvrzení jsou vyhodnocována každý takt o nesplněné podmínce je verifikační inženýr uvědoměn.

Po skončení simulace jsou porovnány výsledky registrů a paměti. V případě testu bitcount jsme dospěli k totožným výsledkům.

Dále budeme pokračovat s analýzou pokrytí instrukcí.



Obrázek 8.2: Pokrytí instrukcí v prvním slotu

Automatizovaně generovaný překladač jazyka C/C++ naplánoval pro první ze čtyř slotů pouze 7 instrukcí. Důsledkem je nedostatečné 16% pokrytí instrukcemi. Za účelem testování celé instrukční sady se zpravidla používají náhodně generované testy.

Při reálných verifikacích se slučují výsledky skupin testů dohromady za účelem vyššího pokrytí. V případě, že procesor nebyl ještě nikdy vyroben jako křemíkový ASIP, požaduje se kvůli vysokým cenám za masky pokrytí vyšší než 99%.

Kapitola 9

Syntéza do FPGA

Výstupem mé diplomové práce je systém na čipu s procesorem Codix VLIW ve formě bitstreamu pro platformu Xilinx ZYNQ.

Procesor Codix VLIW popsaný v jazyce Codal je Codasip frameworkem přeložen do interního formálního modelu, ze kterého je možné provést překlad do HDL jazyka VHDL či Verilog.

Na níže uvedeném příkladu demonstruji překlad události v jazyce Codal, která je zodpovědná za výpočet cílové adresy skoku do ekvivalentní jednotky v jazyce VHDL.

```
event ex_adder : pipeline(pipe.EX)
{
    semantics
    {
        // Jump address
        switch (r_ex_adder_op)
        {
            case ADDER_ABS:
                s_ex_adder_output = r_ex_aluB;
                break;
            case ADDER_REL:
                s_ex_adder_output = r_ex_aluA + r_ex_aluB;
                break;
            default:
                break;
        }
    };
};
```

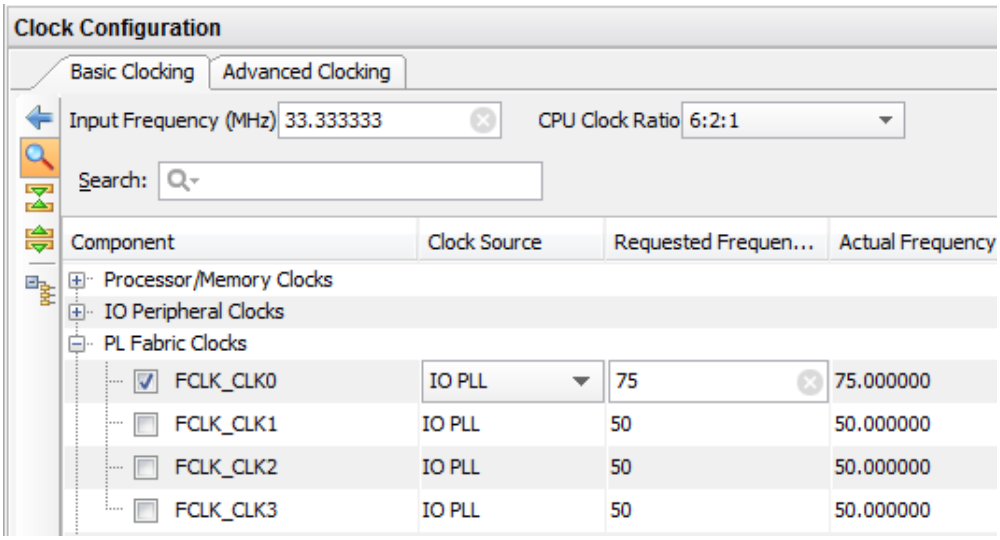
Funkčně ekvivalentní kód exportovaný do jazyka VHDL vypadá následovně:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity codix_vliw_ca_core_cluster_3_ex_adder_ex_adder_fu_semantics_t is
port (
ACT : in std_logic;
cluster_3_r_ex_adder_op_Q0 : in std_logic;
cluster_3_r_ex_aluA_Q0 : in std_logic_vector(31 downto 0);
cluster_3_r_ex_aluB_Q0 : in std_logic_vector(31 downto 0);
cluster_3_s_ex_adder_output_D0 : out std_logic_vector(31 downto 0)
);
end entity codix_vliw_ca_core_cluster_3_ex_adder_ex_adder_fu_semantics_t;


architecture RTL of
codix_vliw_ca_core_cluster_3_ex_adder_ex_adder_fu_semantics_t is
-- signal (inner)
signal codasip_tmp_var_1 : unsigned(0 downto 0);
-- signal (inner)
signal codasip_tmp_var_1_STATEMENT_AST_25574 : unsigned(0 downto 0);
-- signal (inner)
signal cluster_3_s_ex_adder_output_STATEMENT_AST_25583 :
unsigned(31 downto 0);
-- signal (inner)
signal cluster_3_s_ex_adder_output_STATEMENT_AST_25598 :
unsigned(31 downto 0);

-- datapath signals and constants
constant CONSTANT_0_25578: unsigned(0 downto 0) := to_unsigned(0, 1);
constant CONSTANT_0_25591: unsigned(31 downto 0) := to_unsigned(0, 32);
constant CONSTANT_1_25581: unsigned(0 downto 0) := to_unsigned(1, 1);
constant CONSTANT_1_25602: unsigned(1 downto 0) := to_unsigned(1, 2);

begin
-- Datapath code
codasip_tmp_var_1=cluster_3_ ...
codasip_tmp_var_1_STATEMENT_AST_25574(0) <=
cluster_3_r_ex_adder_op_Q0;
cluster_3_s_ex_adder_output. ...
cluster_3_s_ex_adder_output_STATEMENT_AST_25583 <=
unsigned(cluster_3_r_ex_aluB_Q0);
cluster_3_s_ex_adder_output. ...
cluster_3_s_ex_adder_output_STATEMENT_AST_25598 <=
```

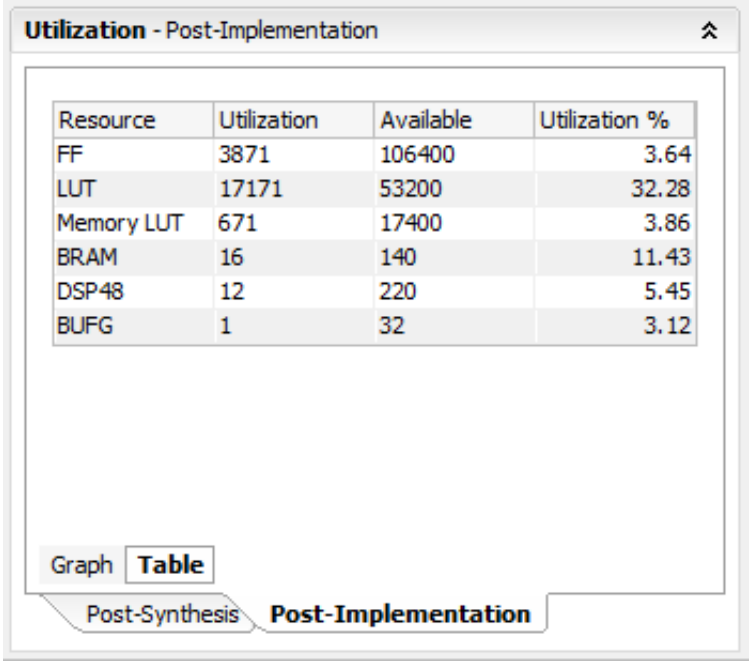
```
(unsigned(cluster_3_r_ex_aluA_Q0) + unsigned(cluster_3_r_ex_aluB_Q0));
codasip_tmp_var_1 <= codasip_tmp_var_1_STATEMENT_AST_25574 when
((ACT = CONSTANT_1_25581(0))) else CONSTANT_0_25578;
cluster_3_s_ex_adder_output_D0 <=
std_logic_vector(cluster_3_s_ex_adder_output_STATEMENT_AST_25583)
when (((ACT = CONSTANT_1_25581(0)) and
(CONSTANT_0_25578 = codasip_tmp_var_1)))
else
std_logic_vector(cluster_3_s_ex_adder_output_STATEMENT_AST_25598)
when (((ACT = CONSTANT_1_25581(0))
and (resize(CONSTANT_1_25602, 1) = codasip_tmp_var_1)))
else std_logic_vector(CONSTANT_0_25591);

end architecture RTL;
```

Výsledný kód je z důvodu automatizovaného generování méně přehledný a delší než ručně psaný. V konečném důsledku to však není žádný problém, neboť uživatel pracuje pouze na vyšší úrovni v jazyce Codal a syntézní nástroje provádí řadu optimalizací za účelem zjednodušení a zvýšení výkonu, při kterých se neužitečné prvky odstraní.

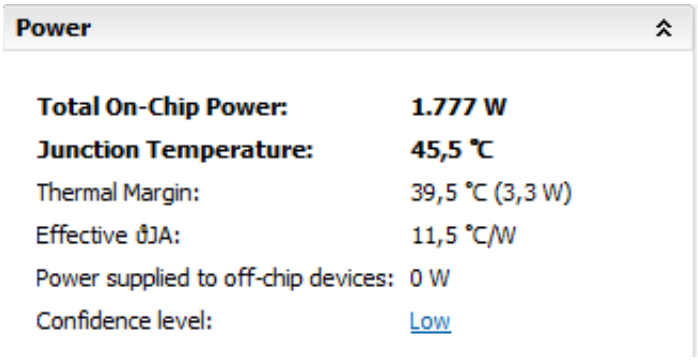
Stejným způsobem jsem exportoval kompletní systém na čipu do jazyka VHDL. Následně jsem provedl syntézu do platformy Xilinx ZYNQ pomocí návrhářského programu Xilinx Vivado s implicitním nastavením. Výsledky časování, využití zdrojů a odhadu spotřeby energie jsou demonstrovány níže.



Obrázek 9.1: Výsledek syntézy: frekvence



Obrázek 9.2: Výsledek syntézy: zdroje



Obrázek 9.3: Výsledek syntézy: spotřeba

Závěr

Navrhl jsem mikroarchitekturu procesoru s velmi dlouhým instrukčním slovem Codix VLIW a vytvořil IA a CA model, ze kterých lze automatizovaně generovat kompletní toolchain obsahující např. překladač jazyka C/C++, assembler, různé typy simulátorů a profiler.

Tyto modely se dle předpokladu zařadily do portfolia produktů společnosti Codasip a jsou nabízeny ke komerčnímu užití.

Procesor jsem nejprve odladil na komerční testovací sadě společnosti ACE, následně byl verifikován verifikačním týmem. Exportovaný kód v jazyce VHDL jsem v návrhářském studiu Xilinx Vivado syntetizoval do platformy Xilinx ZYNQ.

Při návrhu mikroarchitektury jsem bral v potaz, že se jedná o tzv. rozšiřitelné procesorové jádro. Důsledkem toho je možné jednoduše přidávat nové instrukce i architekturální vlastnosti.

V současné době probíhá na Codix VLIW jádře, vzniklém v rámci této diplomové práce a rozšířeném mimo jiné o SIMD operace, evaluace největší korejskou firmou v elektrotechnickém průmyslu. Během ní bude mnou navržený procesor analyzován syntézními nástroji v aktuálně nejmodernější 14nm technologii.

Mým cílem je udělat Codix VLIW procesor jednoduše rozšiřitelný o širokou sadu operací a volitelných architekturálních vlastností, aby se mohl stát jedním ze světově rozšířených procesorů v nastupujícím věku internetu věcí.

Literatura

[1] Intel Keynote. 2015. Dostupné z: https://www.youtube.com/watch?v=x6a3tKOwN7A

[2] Shen, J.; Lipasti, M.: *Modern Processor Design: Fundamentals of Superscalar Processors*. Electrical and Computer Engineering, McGraw-Hill Companies,Incorporated, 2005, ISBN 9780070570641. Dostupné z: https://books.google.cz/books?id=Nibfj2aXwLYC

[3] Schwarzmann, M.: E-svet.cz. 2014. Dostupné z: http://e-svet.e15.cz/technika/internet-veci-aneb-svet-chytry-a-jeste-chytrejsi-1145228

[4] Olivka, P.: Procesory CISC a RISC. 2010. Dostupné z: http://poli.cs.vsb.cz/edu/arp/down/procrisc.pdf

[5] Complex instruction set computing. 2001-. Dostupné z: https://en.wikipedia.org/wiki/Complex_instruction_set_computing

[6] Complex Instruction Set Computer. 2001-. Dostupné z: http://cs.wikipedia.org/wiki/CISC

[7] Tanenbaum, A. S.: *Structured Computer Organization 6th Edition*. New Jersey: Pearson, 2015, ISBN 0131485210.

[8] Maurice V. Wilkes - Short Biography. 2010. Dostupné z: http://www.cl.cam.ac.uk/archive/mvw1/short-biography.html

[9] DSP. 2001-. Dostupné z: https://cs.wikipedia.org/wiki/digitalni_signalovy_procesor

[10] Digital signal processor. 2001-. Dostupné z: https://en.wikipedia.org/wiki/Digital_signal_processor

[11] Hennessy, J.; Patterson, D.: *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design, Elsevier Science, 2006, ISBN 9780080475028. Dostupné z: https://books.google.cz/books?id=57UIPoLt3tkC

[12] Fisher, J.; Faraboschi, P.; Young, C.: *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Electronics & Electrical, Morgan Kaufmann, 2005, ISBN 9781558607668. Dostupné z: https://books.google.cz/books?id=R5UXl6JoOXYC

[13] Application specific instruction set processor. 2001-. Dostupné z: http://en.wikipedia.org/wiki/Application-specific_instruction_set_processor

[14] Masarik, K.: Jazyky pro popis architektur. 2013.

[15] Codasip: Codal reference language manual. 2015. Dostupné z: https://en.wikipedia.org/wiki/Complex_instruction_set_computing

[16] Codasip: Codasip framework manual. 2015. Dostupné z: https://www.codasip.com/download/?filename=Codasip_Framework_Manual.pdf

[17] Codasip: Codasip studio user guide. 2015. Dostupné z: https://www.codasip.com/download/?filename=Codasip_Studio_User_Guide.pdf

Příloha **A**

Seznam použitých zkratek

ASIP Aplikačně specifický instrukční procesor

UVM Universal Verification Methodology

RISC Reduced Instruction Set Computer

CISC Complex Instruction Set Computer

VLIW Very Long Instruction Word

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `bitstream` ..... bitstream Codix VLIW pro Xilinx ZYNQ
- `toolchain` ..... toolchain Codix VLIW pro Win
- `src`
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text`
  - `thesis.pdf` ..... text práce ve formátu PDF