VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

VYTVOŘENÍ MODELU PROCESORU POMOCÍ ADL JAZYKA

PROCESSOR MODEL CREATION USING ADL LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **KRISTIÁN OSTATNÍK**
AUTHOR

VEDOUCÍ PRÁCE **Prof. Ing. TOMÁŠ HRUŠKA, CSc.**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/17841/2016/xostat01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Ostatník Kristián**
Obor: Informační technologie
Téma: **Vytvoření modelu procesoru pomocí ADL jazyka**
Processor Model Creation Using ADL Language
Kategorie: Počítačová architektura

Pokyny:

1. Seznamte se s popisným jazykem CodAL a s integrovaným vývojovým prostředím Codasip Studio, určeným pro souběžný návrh hardwaru a softwaru. Nastudujte způsob generování programovacích a emulačních nástrojů v tomto prostředí.
2. Seznamte se s architekturou vybraného procesoru.
3. Namodelujte vybraný procesor v prostředí Codasip Studia na instrukční úrovni.
4. Po domluvě s vedoucím upravte model tak, aby z něho šel pomocí Codasip Studia vygenerovat překladač jazyka C.
5. Na zvolené sadě testovacích programů odlaďte model procesoru.
6. Zhodnoťte způsob práce s Codasip Studiem, navrhněte vylepšení, která by zjednodušila práci, zhodnoťte problémy nastaly při rozcházení překladače jazyka C.

Literatura:

- Manuál prostředí Codasip Studio a jazyka CodAL
- Dle doporučení vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

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

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% 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: **Hruška Tomáš, prof. Ing., CSc.**, UIFS FIT VUT
Konzultant: Husár Adam, Ing., Ph.D., VCIT FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

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

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Cieľom práce je implementácia modelu procesora ARC v ADL jazyku CodAL na inštrukčnej úrovni. Prvá časť bakalárskej práce sa venuje klasifikácii procesorov a popisu ADL jazykov. Druhá časť práce popisuje priebeh implementácie procesora a generovanie prekladača jazyka C/C++ pre odladenie a verifikáciu vytvoreného modelu. Na záver je popísané porovnanie vytvoreného modelu s existujúcim modelom ARC 700 na sade benchmarkových testov.

Abstract

The goal of this thesis is to create an instruction-accurate model of ARC processor using the CodAL ADL language. The first part is dedicated to classification of processors and ADL languages. The second part describes the implementation process and the generation of C/C++ compiler for debugging and verification of the created model. At the end the created model is compared to an existing model ARC 700 on a set of benchmark tests.

Kľúčové slová

Model, Procesor, ADL, Codasip, CodAL, ARC, Prekladač, Instrukčná sada

Keywords

Model, Processor, ADL, Codasip, CodAL, ARC, Compiler, Instruction set

Citácia

OSTATNÍK, Kristián. *Vytvoření modelu procesoru pomocí ADL jazyka*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Prof. Ing. Tomáš Hruška, CSc.

Vytvoření modelu procesoru pomocí ADL jazyka

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána prof. Ing. Tomáša Hrušku, CSc. Ďalšie informácie mi poskytol v roli konzultanta pán Ing. Adam Husár, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Kristián Ostatník
14. mája 2017

Poďakovanie

Chcel by som poďakovať vedúcemu tejto bakalárskej práce prof. Ing. Tomášovi Hruškovi, CSc. a môjmu konzultantovi Ing. Adamovi Husárovi, Ph.D za odbornú pomoc a usmernenie pri písaní tejto práce, za cenné rady a ochotu.

Obsah

Kapitola 1

Úvod

Každý rok sa vyrába vždy väčšie množstvo produktov, ktoré nejakým spôsobom využívajú procesory. Dané procesory môžu byť využívané na rôzne špeciálne účely, čo znamená, že procesor, ktorý sa používa napríklad v stolnom počítači, nebude múdre použiť napríklad v mobilnom telefóne.

V spojení s výrobou veľkého množstva daných produktov sa momentálne vo veľkom rozbieha internet vecí (IoT - Internet of Things), čo znamená, že všetky elektronické spotrebiče v domácnosti (od žiaroviek po hriankovač) budú môcť byť vybavené vlastným procesorom na najrôznejšie účely. Ďalej sa ročne využívajú miliardy procesorov v automobilovom priemysle, pri výrobe chytrých zariadení ale človek ich nájde napríklad aj v jednoduchých termostatoch. To všetko znamená, že návrh daných procesorov sa potrebuje čo najviac urýchliť a automatizovať aby bolo možné držať krok s vývojom nových produktov.

Rýchlosť návrhu nie je jediná smerodajná vec. Jak bolo spomenuté, procesor ktorý sa použije v stolnom počítači, nie je múdre použiť aj v mobilnom telefóne, pretože mobilný telefón nemôže mať vysokú spotrebu a zároveň je limitovaný so svojou veľkosťou. Daná scenéria sa tiež môže otočiť, stolný počítať potrebuje procesor, ktorý podporuje širokú škálu rôznych aplikácií s veľkou výpočetnou silou, ktoré sú u mobilných telefónoch nepredstaviteľné. Čo znamená, že je nutné navrhnúť dva rôzne procesory s rozlišnými energetickými nárokmi, veľkosťou a rôznou funkčnosťou pre dané produkty. Vyplýva z toho, že pri návrhu procesoru je potrebné zvážiť viaceré aspekty navrhovaného procesora.

Samotná práca sa venuje modelovaniu procesoru ARC v jazyku CodAL. Pre pochopenie a ako úvod do procesorov slúži kapitola 2. Vysvetľuje základné pojmy, rozdelenie procesorov a povie pár informácií o modelovacích jazykoch.

Kapitola 3 priblíži použité vývojové prostredie Codasip Studio. Vysvetlí, ktoré nástroje boli vytvorené pri modelovaní procesoru a oboznámi čitateľa so základnými informáciami o jazyku CodAL.

Kapitola 4 sa venuje popisu procesora ARC, ktorý bol modelovaný v prostredí Codasip Studia. Vysvetľuje podrobnejšie implementované partície procesora a časti, ktoré sú dôležité pre vytvorenie jedného uceleného názoru na daný procesor.

Kapitola 5 sa venuje konkrétnej implementácie modelovaného procesora. Čitateľ tu nájde rôzne ukážky kódov, riešenia a popis niektorých zaujímavých situácií, ku ktorým došlo pri implementácii.

Kapitola 6 vysvetľuje priebeh testovania vytvoreného modelu počas práci a zároveň aj popisuje konečné výsledky práce (porovnanie vytvoreného modelu s existujúcim riešením).

Kapitola 7 sa venuje zhodnotením práce s vývojovým prostredím Codasip Studio a problémom, ktoré nastali pri rozchádzaní prekladača C/C++.

Kapitola 2

Teória

2.1 Klasifikácia procesorov

2.1.1 Inštrukčné sady

Jedným z najdôležitejších charakteristík, ktoré rozlišujú rôzne počítače je povaha ich inštrukcií. V moderných počítačoch sú dva rôzne návrhy inštrukčných sád, ktoré sa fundamentálne líšia. Informácie sú čerpané z [5].

- RISC (*Reduced Instruction Set Computer*) – Počítač s obmedzenou sadou inštrukcií. Princíp RISC-u je založený na predpoklade, že vyšší výkon sa dá docieliť ak každá inštrukcia bude zaberať rovnaký počet slov[1] v pamäti a všetky operandy, potrebné pre inštrukciu v čase vykonávania, budú prítomné v procesorových registroch alebo môžu byť zakódované do inštrukcie. Pevne definovaná šírka inštrukcií dovoluje zreťazené spracovávanie prekladaním aktivít postupnosti inštrukcií, čo znižuje čas vykonania programu. Ideálne sa vykoná každom cykle jedna inštrukcia. Komplikované operácie sa skladajú z viacerých inštrukcií. Toto obmedzenie znižuje zložitosť a množstvo typov inštrukcií, ktoré môžu byť pridané do inštrukčnej sady. Ďalšou charakteristikou RISC-u je, že na prístup k pamäti sa používajú len dve inštrukcie LOAD a STORE.
- CISC (*Complex Instruction Set Computer*) – Počítač s rozsiahlou inštrukčnou sadou. Alternatívou k RISC-u je využitie zložitejších inštrukcií, ktoré môžu zaberať rozlišný počet slov v pamäti a dokážu vykonať komplikovanejšie operácie, čo ale tiež znamená, že rôzne inštrukcie sa môžu vykonať v rozlišnom počte cyklov. Zreťazené spracovávanie je možné ale ťažko dosiahnuteľné.

2.1.2 Architektúra počítačov

Informácie v tomto oddiely sú čerpané zo zdrojov [9] a [7].

Von Neumannova architektúra
Koncepcia vznikla v roku 1945 vývojármi ENIAC[2] počítača a konzultantom matematikom Johnom von Neumannom. Skladá sa z:

[1] Anglický výraz word – označuje pamäťovú jednotku používanú konkrétnym návrhom procesora. Typicky má 32/64 bitov ale rôzne architektúry môžu mať viac aj menej.

[2] *Electronic Numerical Integrator and Computer* – elektronický číslicový integrátor a počítač, bol jedným z prvých vyrobených programovateľných počítačov.

- hlavnej pamäte, kde sa ukladajú dáta aj inštrukcie,
- aritmeticko-logickej jednotky, ktorá vykonáva základné aritmetické a logické operácie s číslami,
- riadiacej jednotky, ktorá načítava inštrukcie z pamäte a stará sa o ich vykonávanie,
- vstupno-výstupných zariadení, ktoré riadi riadiaca jednotka.

Pre túto architektúru sú všetky súčiastky prepojené jedným systémom troch zberníc:

- dátová – prenáša dáta medzi centrálnou procesorovou jednotkou (CPU - *Central Processor Unit*) a perifériami. Je dvojsmerná.
- adresová – CPU používa adresovú zbernicu aby indikovala, ku ktorému vstupno-výstupnému zariadeniu chce pristúpiť a vrámci zariadenia k špecifickému registru. Adresová zbernica je jednosmerná, CPU vždy zapisuju adresu, ktorá je čítaná perifériami.
- riadiaca – prenáša signály, ktoré sú požívané na synchronizáciu a správu výmien informácií medzi CPU a jej perifériami.

Hlavnou charakteristikou pre Von Neumann architektúru je, že vlastní len jeden adresový priestor. Rovnaká zbernica prenáša všetky výmeny informácií medzi CPU a perifériami vrátane inštrukčných kódov ako aj dát spracovávaných procesorom.

Harvardská architektúra
Harvardská architektúra, ako názov naznačuje, bola vyvinutá na Harvardskej univerzite. Narozdiel od Von Neumann architektúry používa dva oddelené zbernicové systémy na prenos inštrukcií a dát spracovávaných procesorom. Sú to:

- programová – používa sa výhradne na prenos inštrukčných kódov z pamäte programu do CPU počas cyklu načítavania,
- dátová – používa sa výhradne na prenos dát z/do CPU, do/z pamäte alebo periférií.

Porovnanie
Keďže Harvardská architektúra vlastní dva rôzne zbernicové systémy (dva oddelené adresové priestory), umožňuje súčasne čítať inštrukčný kód a čítať/zapisovať dáta/periférie ako súčasť vykonávania predchádzajúcej inštrukcie. Kvôli tejto vymoženosti mala rýchlostnú výhodu nad Von Neumann architektúrou. Je tiež bezpečnejšia, pretože nemôže dojsť k chybnému zápisu kódu procesorom do pamäte programu čo by porušilo vykonávanie programu.

Harvardská architektúra je menej flexibilná, potrebuje aspoň dve pamäťové zdroje (jeden pre dáta a druhý pre inštrukcie, môže ich byť aj viac), ktoré nie sú zameniteľné. Nie je to problém pri vstavaných systémoch, ktoré vždy vykonávajú tú istú úlohu, pretože veľkosti pamätí potrebných na program a dáta sú predvídateľné, čo znamená, že požadované pamäťové zdroje môžu byť optimalizované pre danú aplikáciu. Pri počítačových systémoch, na ktorých beží široká škála aplikácií, kde jedny aplikácie požadujú veľkú pamäť pre program a druhé požadujú veľkú pamäť pre dáta je lepšou voľbou Von Neumann architektúra. Pamäť pre program a dáta sú zameniteľné (programátor má možnosť si vyčleniť potrebné miesto pre inštrukcie/dáta), čo vedie k lepšiemu využitiu pamäťových zdrojov.

2.1.3 Na základe použitia

Univerzálne procesory
Univerzálne procesory sú navrhované tak, aby vyhovovali potrebám mnohých rôznych aplikácií. Využívajú sa hlavne v osobných počítačoch a v pracovných staniciach. Pri návrhu je snaha dosiahnuť čo najväčšiu rýchlosť integrovaním vyrovnávacích pamätí alebo hardvérových súčiastok, napríklad na matematické výpočty, do procesora.

Aplikačno špecifické procesory
Cena a doba návrhu sú nižšie ako pri univerzálnych procesoroch. Delia sa na:

- DSP (*Digital Signal Processor*) – Digitálny signálový procesor je programovateľný mikroprocesor pre rozsiahle numerické výpočty v reálnom čase.
- ASIC (*Application Specific Integrated Circuit*) – Zákaznícky integrovaný obvod. Algoritmus je úplne realizovaný v hardvéri. Medzi ASICy patria napríklad chipy satelitov, chipy v hračkách alebo chip obsahujúci mikroprocesor spolu s ďalšou logikou ako celok. Nepatria medzi ne štandardné súčiastky ako mikroprocesory alebo pamäťové médiá (ROM, DRAM) [8].
- ASIP (*Application-Specific Instruction set Processor*) – Aplikačno špecifický inštrukčný procesor. Hardvér a inštrukčná sada sú navrhované spolu pre 1 špeciálnu aplikáciu. Inštrukčná sada je špeciálne navrhovaná na urýchlenie výpočtovo náročných a najčastejšie využívaných funkcií. Pri návrhu architektúry sa kladie veľký dôraz na minimalizovanie nákladov na hardvér, spotrebu alebo výkon podľa navrhovanej aplikácie [4].

2.2 ADL jazyky

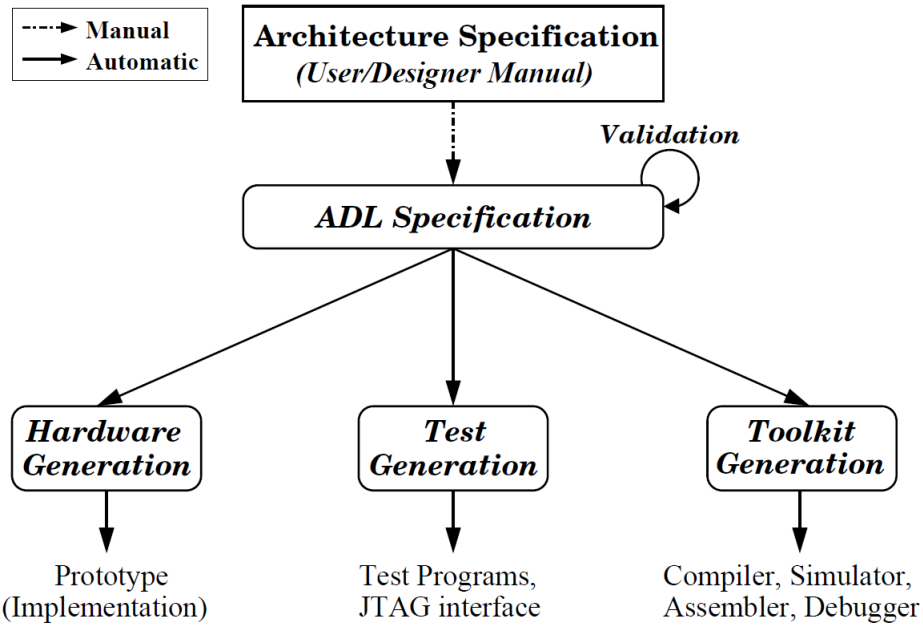
Architecture Description Languages sú jazyky na popis architektúry. Umožňujú automatizáciu návrhu procesora ako to je ukázané na obrázku 2.1. Návrhár vytvára ručne ADL popis na základe špecifikácie architektúry (používateľský/návrhársky manuál alebo programátorská príručka). Z ADL popisu je potom automaticky možné generovať hardvér, vývojové a testovacie nástroje a automaticky vykonať validáciu vytvoreného popisu. Tu nájdené informácie boli čerpané z [6].

Dnes existujúce ADL jazyky sa dajú rozdeliť na dve skupiny na základe orientácie na obsah a účel. ADL jazyky s orientáciou na obsah sú založené na povahe informácií ktoré ADL vie zachytiť, radia do troch kategórií:

- Štrukturálne – Treba zvážiť 2 dôležité aspekty návrhu v ADL a to je level abstrakcie oproti všeobecnosti. Je veľmi ťažké nájsť abstrakciu, ktorá je schopná zachytiť všetky rysy rôznych procesorov. Bežnou metódou na získanie vyššieho stupňa všeobecnosti je zníženie stupňa abstraktnosti, preto je napríklad veľmi obľúbená abstrakcia na úrovni RT, ktorá je dostatočne nízka na detailný popis chovania digitálneho systému ale taktiež dostatočne vysoká aby zakryla implementačné detaily na úrovni hradiel. Zjednodušene sa zameriavajú na štrukturálne komponenty a prepojenie architektúry procesora. Štrukturálny ADL je napr. jazyk MIMOLA.
- Pre popis chovania – Náročnosť extrahovania inštrukčnej sady sa dá obísť abstrahovaním behaviorálnych informácií z štrukturálnych detailov. ADL pre popis chovania výslovne špecifikujú sémantiku inštrukcií a ignorujú detailné hardvérové štruktúry.

Typicky ADL popis a popis inštrukčnej sady v referenčnom manuáli sú vzájomne zhodné. Behaviorálny ADL je napr. nML a ISDL.

- Zmiešané – Zachytávajú štrukturálne aj behaviorálne detaily architektúry. Medzi zmiešané jazyky patria napr. CodAL, LISA, EXPRESSION.



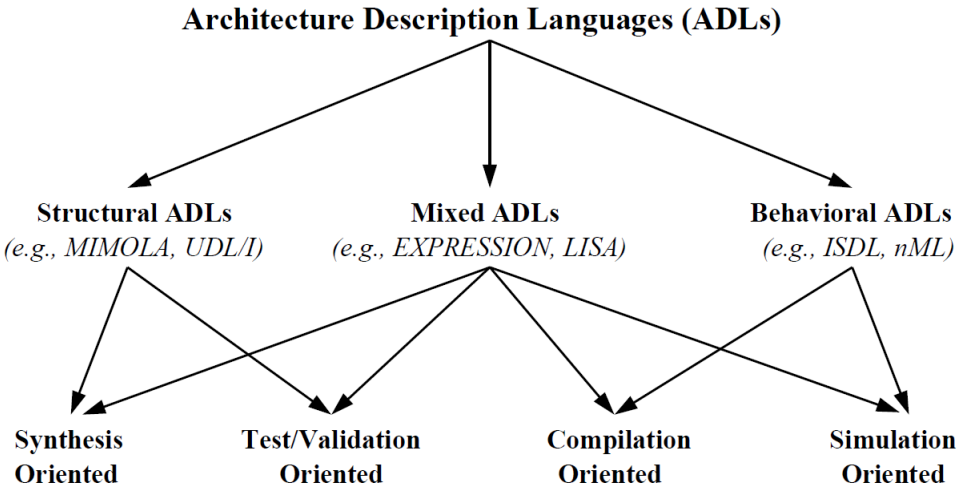
Obr. 2.1: Ukážka automatických a manuálnych krokov pri návrhu

ADL jazyky s orientáciou na účel (objective oriented) sú kategorizované podľa cieľov ADL jazyka. ADL jazyky sa úspešne používajú ako špecifikačné jazyky pre vývoj procesorov. Je potrebné rýchle vyhodnotenie možnej architektúry aby bolo možné vytvoriť čo najlepší možný návrh na základe rôznych obmedzení ako plocha, spotreba alebo výkon procesora, čo sa docieli skúmaním návrhového diagramu. V prvom kroku sa prekladajú a simulujú aplikačné programy, z ktorých sa získava spätná odozva, ktorá môže slúžiť na modifikáciu vytvorenej ADL špecifikácie. Generovaný simulátor vytvára opisné dáta (samotný proces sa nazýva profilovanie), ktoré môžu byť použité na evaluáciu. Vygenerovaný hardvérový model (syntetizovateľný HDL[3]) môže poskytnúť presnejšiu spätnú odozvu spojenú s požadovanou kremíkovou plochou, taktovacou frekvenciou a spotrebou architektúry procesoru. Na obrázku 2.2 na strane 8 je zobrazené zaradenie niektorých konkrétnych jazykov podľa orientácií. Dnešné ADL jazyky s orientáciou na účel sú rozdelené do štyroch kategórií:

- Orientované na prekladač – Cieľom tohto druhu ADL je umožniť automatické generovanie prekladačov, ktoré sa vedia prispôsobiť generovaniu kódu pre rôzne cieľové procesory s znovupoužitím veľkého množstva zdrojového kódu prekladača. Zachytávajú inštrukčnú sadu architektúry spolu s niektorými štrukturálnymi informáciami ako je programový čítač alebo registrové detaily.

[3] *Hardware Description Language* je jazyk na popis hardvéru.

- Orientované na simuláciu – Simulácia môže byť vykonávaná na rôznych úrovniach abstrakcie. Modelovaním inštrukčnej sady na najvyššej úrovni abstrakcie môže byť simulovaná funkčnosť procesoru, čo sa najviac povoľuje pri ADL pre popis chovania. Štrukturálne ADL sú dobrými kandidátmi pre generovanie simulátorov na úrovni cyklov, pretože opisujú viac detailné informácie o časovaní, keďže sú na nižšej úrovni abstrakcie.
- Orientované na syntézu – Štrukturálne centrické ADL jazyky sú vhodné na generovanie hardvéru. Návrhár má voľbu generovať reprezentáciu cieľovej architektúry v jazykoch VHDL, SystemC alebo Verilogu.
- Orientované na validáciu – ADL boli úspešne použité v akademickej i priemyselnej sfére, aby umožnili generovanie testov na funkčné overenie vstavaných procesorov. Na verifikáciu sa používa validačná metodológia zhora dole (top-down), o ktorom sa dá viac dočítať v [6].



Obr. 2.2: Klasifikácia ADL jazykov podľa orientácií na obsah a účel

Na obrázku 2.2 je ukázaný vzťah obsahovo a účelovo orientovaných ADL. O zmiešaných ADL bolo povedané, že zachytáva štrukturálne aj behaviorálne detaily architektúr a ako vidieť na obrázku zároveň zjednocuje aj všetky ciele účelovo orientovaných ADL.

Kapitola 3

Vývojové prostriedky

Táto kapitola predstavuje použité nástroje a jazyk pri modelovaní procesoru. Uvedené poznatky sú čerpané zo zdrojov [3] a [2].

3.1 Codasip Studio

Codasip Studio je plne integrované vývojové prostredie založené na Eclipse. Skladá sa z používateľského rozhrania, príkazového riadku Codasip, nástrojov pre vývoj a generátorov nástrojov ako napríklad generátor prekladača jazyka C/C++. Vývojové prostredie dovoľuje užívateľom vytvárať aplikácie, ktoré môžu púšťať a ladiť pomocou SDK[1], ktorý vygenerovalo Codasip Studio.

- Assembler – Prekladá človekom čitateľný strojový kód na binárny objektový súbor, ktorý je potom linkovaný Codasip Linkerom, čím vznikne binárny kód čitateľný procesorom. Generovaný assembler podporuje kompiláciu programov v jazyku symbolických inštrukcií vo forme bežne používaným GNU[2] assemblerom. Codasip prekladač tak ako aj prekladače GCC a LLVM produkujú výstupy v tomto formáte, takže Codasip assembler je schopný zostaviť ich výstupy.

- Disassembler – Číta spustiteľný súbor vytvorený asseblerom alebo linkerom a transformuje ho na pôvodný strojový jazyk. Výstup z disassemblera je znova zostaviteľný.

- Simulátor – Simulácie Codasip Studia dovolujú návrhárom odladiť a testovať hardvér a softvér vstavaného systému. Je vytváraný z inštrukčne presného popisu. Inštrukčne presný simulátor je založený na konštantnom načítaní, dekódovaní a vykonávaní inštrukcií z pamäte. Simulátor nie je riadený udalosťmi, čo znamená, že chovanie vnútri ASIP-u je simulovaný sekvenčne.

- C/C++ prekladač – Môže sa generovať pre rôzne počítačové platformy. Je založený na populárnom a široko používanom otvorenom softvéri LLVM Framework, ktorý je silno podporovaný veľkými spoločnosťami ako napríklad Google, Apple alebo Intel. Prekladač je dodávaný s štandardnou C knižnicou a runtime (za behu) knižnicou pre softvérovo implementované operácie.

[1] Software development kit – sada vývojových nástrojov.

[2] GNU's Not Unix – GNU nie je Unix, je operačný systém a rozsiahla zbierka počítačového softvéru. GNU sa skladá výlučne zo slobodného softvéru.

- Debugger – Codasip Studio simulátor môže byť generovaný s debugger podporou. Dovoľuje beh simulátora v debug móde a ladenie bežiacich aplikácií alebo rovno CodAL-ovského zdrojového kódu. Umožňuje ladenie na viacerých úrovniach:
  - Úroveň zdrojového kódu – Jeden krok odpovedá vykonávaniu jedného riadku zdrojového kódu. Codasip debugger podporuje krokovanie strojového kódu, kódu C a CodAL.
  - Úroveň inštrukcií – Jeden krok odpovedá vykonávaniu jednej inštrukcie.
  - Úroveň CodAL – Jeden krok odpovedá vykonávaniu jedného CodAL-ovského príkazu.

3.2 Jazyk CodAL

Jazyk CodAL bol vyvinutý pre prototypovanie procesorov s aplikačne špecifickou inštrukčnou sadou a multiprocesorových návrhov. Ako bolo popísané v sekcii 2.2 CodAL je zmiešaný ADL jazyk. Z jedného CodAL-ovského popisu procesora je možné automaticky vygenerovať všetky potrebné nástroje pre programovanie a simuláciu. Rovnaký popis zároveň dovoľuje automatickú generáciu implementácie mikroarchitektúry v hardvéri0, použitím jazykov VHDL alebo Verilogu. Jedným z kľúčových objektových typov je element. Medzi druhy informácií súvisiacich s týmto objektom sú:

- assembler – textové pravidlá spojené s elementom (pre primárne využite assemblerom, ale s viacerými inými nástrojmi tiež),
- binary – informácie o kódovaní (rovnaké využitie ako assembler),
- semantics – chovanie spojené s objektom (pre využitie simulátorom a prekladačom),
- return – referenčné informácie o objekte, typicky použité na spojenie s iným objektom.

```
element i_mov
{
    // Inštancie elementu, správajú sa ako lokálne premenné.
    use rf_gpr as reg_dst, reg_src;
    // Definícia syntaxu inštrukcie v assembleri.
    assembler { "MOV" reg_dst "," reg_src };
    // Binárny tvar danej inštrukcie.
    binary { opc reg_dst reg_src 0:bit[17] };
    // Popis chovania inštrukcie v jazyku C.
    // Hodnota zdrojového registra sa uloží do cieľového
    semantics { rf_gpr[reg_dst] = rf_gpr[reg_src]; };
};
```

Zdrojový kód 3.1: Príklad syntaxu jazyka CodAL

3.2.1 Popis úrovne návrhu

CodAL má dva aspekty. Prvá sa týka návrhu na úrovni prototypov. Táto úroveň popisuje blízke prostredie okolo ASIP-u. Môže obsahovať pamäte, komponenty tretích strán, periférne zariadenia, atď. Druhá sa týka samotného ASIP-u. Toto rozlíšenie ich robí ľahko opakovane použiteľnými na viacerých prípadoch použitia.

Vyššia (top) úroveň návrhu je typicky použitá na popis blízkeho prostredia okolo ASIP-u, pričom navrhovaný ASIP je inštanciovaný na určitej úrovni návrhu a spojený k jeho vonkajšiemu pamäťovému subsystému.

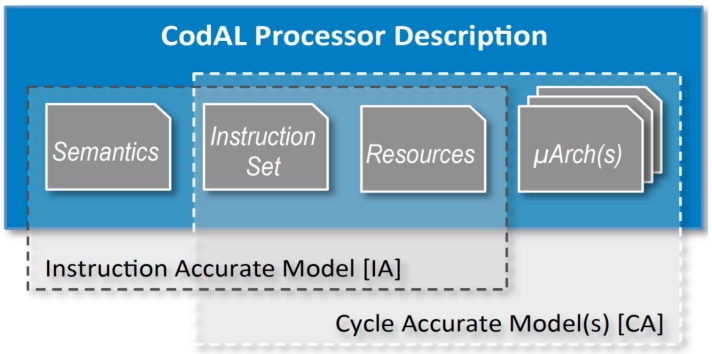
Popis ASIP-u je riadený oddelene od vyššej úrovni. Aj keď sa popisy ASIP-ov môžu rozlyšovať v zložitosti architektúr a na úrovni zachytených detailov, všetky z nich zdieľajú rovnakú štruktúru. ASIP popisy majú štyri časti:

- Zdroje architektúry: Napríklad programový čítač a registre.
- Inštrukčná sada: Mená inštrukcií, ich operandov a ich binárna forma (opkódy).
- Sémantika: Chovanie každej inštrukcie a výnimky - ako ovplyvnia zdroje viditeľné architektúrov.
- Implementácia: Zdroje a chovanie (hlavne časovanie), ktoré nie sú viditeľné v ASIP architektúre ale definujú špecifické realizácie mikroarchitektúry.

Každá mikroarchitektúra dedí zdroje a inštrukčnú sadu od jej ASIP architektúry a definuje svoju konkrétnu implementáciu. Čo znamená, že jadro procesora môže byť popísaný na dvoch úrovniach abstrakcie:

- Model na úrovni inštrukčnej sady – nie je náročný na počítačové zdroje a dovoľuje generovanie veľmi rýchleho funkčného simulátora. Pri pridávaní nových inštrukcií nie je potrebné brať do úvahy mikroarchitektúru.
- Model na úrovni cyklov – popisuje zreťazené spracovanie procesoru. Používa sa na syntézu procesora na jazyk VHDL alebo Verilog a môže obsahovať špecifické optimalizácie pre hardvérovú implementáciu.

Inštrukčná sada, popis binárneho kódovania a zdroje procesora môžu byť zdieľané medzi týmito dvoma modelmi ako je vidieť na obrázku 3.1.



Obr. 3.1: Popis CodAL procesoru

Kapitola 4

Procesor ARC

4.1 Aktuálny stav

Argonaut RISC Core je vstavaný 32-bitový procesor navrhnutý firmou ARC international. Od roku 2010 ju vlastní firma Synopsys. Procesory ARC sú široko používané v SoC[1] zariadeniach (mobilné zariadenia, automobily, IoT[2]). Informácie v tejto kapitole sú čerpané z webovej stránky firmy Synopsys[3] a zo zdroja [1]. ARC procesor je vysoko konfigurovateľný, možnosť nastavenia rôznych možností pre:

- šírky inštrukcií, programového čítača,
- veľkosť registrového pola (register file),
- typ pamäte, veľkosť, rozdelenie, bázová adresa,
- riadenie spotreby, hradlovanie hodín,
- násobičky, deličky a iné hardvérové súčiastky,
- licencované komponenty, napr. MPU, FPU,
- pridávanie/odstraňovanie inštrukcií.

Dovoľuje používateľovi pridávať rôzne rozšírenia:

- vlastné inštrukcie a hardvér,
- základné a pomocné registre,
- definovanie/pridávanie vlastných príznakov.

4.2 Dátové formáty

Všetky procesory založené na špecifikácii ARCompact predvolene podporujú little-endian[4] architektúru ale niektoré konfigurácie môžu byť aj big-endian[4]. Procesor môže pracovať na

[1] System on a Chip – Systém na čipe je integrovaný obvod, ktorý integruje všetky komponenty počítača alebo iného elektrického systému na 1 substrát.

[2] Internet of Things – internet vecí je prepojenie vstavaných zariadení s internetom

[3] https://www.synopsys.com/designware-ip/processor-solutions/arc-processors.html

[4] Endianita je v informatike spôsob uloženia čísel v pamäti počítača, ktorý definuje, v akom poradí sa ukladajú jednotlivé jednotky informácie príslušného dátového typu. Označuje sa tiež ako poradie bajtov (anglicky byte order).

dátach rôznej veľkosti. Pamäťové operácie (načítavanie a ukladanie) môžu mať dáta šírky 32 bitov (dlhé slovo), 16 bitov (slovo) a osem bitov (bajt). Operácie použité na osem alebo 16 bitových dátach načítavajú spodných 8/16 bitov a môžu rozšíriť znamienkový bit po celom dlhom slove závisiac na inštrukcii. Pamäť dát je prístupné používajúc bajtové adresy čo znamená, že pri prístupe k dlhým slovám a slovám môže nastať prístup k nezarovnaným dátam. Táto situácia sa rieši rôzne pri rôznych variantách procesoru, kým procesor ARC 700 generuje výnimku, pri procesore ARC 600 kontrola nad nezarovnanými dátami závisí na konfigurácii pamäťovej jednotky.

4.3 Inštrukčná sada ARCompact

Inštrukčná sada ARCompact je navrhnutá tak aby zredukovala veľkosť kódu a maximalizovala dostupný priestor operačného kódu pre rozšíriteľné inštrukcie. V inštrukčnej sade sú definované 16-bitové verzie inštrukcií, ktoré sa štatisticky najčastejšie vyskytujú v kóde. Tieto inštrukcie sa môžu voľne miešať s 32-bitovými inštrukciami.

4.4 Registrové sady

Základná registrová sada sa skladá z 64 32 bitových registrov.

- r0-r25 – Základné registre dostupné pre programátora.
- r26-r28 – Registre s ukazovateľmi. GP ukazuje na malú sadu zdieľaných dát počas vykonávania programu. SP ukazuje na najnižšiu použitú adresu zásobníka. FP ukazuje na dátovú štruktúru, ktorá môže byť použitá na spätné vyhľadávanie cez volania funkcií.
- r29-r30 – Poskytujú spätné adresy na miesta, kde nastalo prerušenie alebo vetvenie. Môžu byť tiež použité ako registre so všeobecným použitím ale ak nastane prerušenie alebo skok tak musia byť rezervované na tento účel.

Meno	Použitie
r0-r25	Všeobecné použitie
r26	Globálny ukazovateľ (GP)
r27	Ukazovateľ na rámec (FP)
r28	Ukazovateľ zásobníka (SP)
r29	ILINK1
r30	ILINK2
r31	BLINK
r32-r59	Rozšíriteľné registre
r60	LP_COUNT[31:0]
r61	Rezervované
r62	Indikátor 32-bit operandu
r63	Programový čítač

Tabuľka 4.1: Základná registrová sada

- r32-r59 – Rozšíriteľné registre. Používateľ môže rozšíriť svoju registrovú sadu o ďalšie registre so všeobecným účelom alebo ak je potrebné využiť registre na uchovanie výsledkov 32-bitového násobenia (r57-r59).
- r60 – Čítač cyklov je používaný pre cykly s nulovým oneskorením. Pretože LP_COUNT je dekrementovaný ak sa hodnota programového čítača rovná adrese konca cyklu, nie je odporúčané ju používať ako register so všeobecným použitím.
- r62 – Indikátor dlhých okamžitých dát je rezervovaný pre zakódovanie 32-bitového adresovacieho módu do inštrukčných slov. Môže sa použiť len na tento účel. Nie je dostupný pre programátora ako všeobecný register.
- r63 – Uschováva hodnotu programového čítača, ktorú môžu využiť inštrukcie dovolujúce PC relatívne adresovanie. Zapisovanie do tohto registra nie je povolené.
- Prídavné registre – Základný procesor založený na ARCompact-e používa malú sadu stavových, kontrolných a rezervovaných registrov, ponechávajúc zostávajúcich 2^{32} registrov na účely rozširovania. Patrí sem napríklad register STATUS32, ktorý uchováva príznaky.

4.5 Adresovacie módy

Architektúra podporuje šesť základných adresovacích režimov.

- Priamy – operácie sú vykonávané na hodnotách uložených v registroch.
- Nepriamy – operácie sú vykonávané na miestach špecifikovaných obsahom registrov.
- Nepriamy s offsetom – operácie sú vykonávané na miestach špecifikované s obsahom registrov a pridaním hodnoty offsetu (uloženej v inom registri alebo ako okamžitá hodnota).
- Okamžité – operácie sú vykonávané použitím konštantných dát, ktoré sú zakódované do operačného kódu.
- PC relatívne – operácie sú vykonávané relatívne k súčastnej hodnote v programovom čítači (skoky alebo PC relatívne načítavania).
- Absolútne – operácie sú vykonávané na dátach na mieste v pamäti špecifikovanom konštantnou hodnotou zakódovanou v operačnom kóde.

4.6 Kódovanie inštrukcií

Počítač kóduje inštrukcie procesoru ako numerické hodnoty a ukladá ich do pamäte. Kódovanie týchto inštrukcií je jeden z najdôležitejších úloh pri návrhu inštrukčnej sady a vyžaduje veľmi starostlivé zváženie. ARC definuje viaceré kódovania podľa druhu a operandov inštrukcií. Inštrukčný kód je rozdelený na viacero polí, kde každé pole má svôj definovaný význam. Pre pochopenie formátu kódovania je potrebné rozumieť konvenciám v tabuľke 4.3 na strane 16.

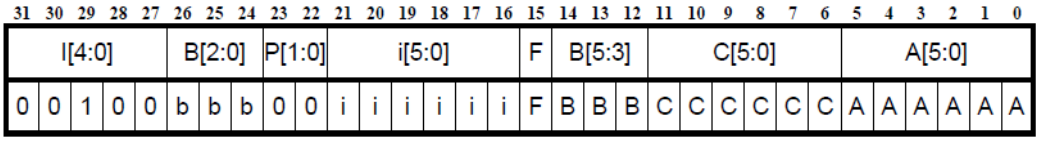
4.6.1 Opkódy

Kódovanie inštrukcií sa delí na rôzne typy podľa 5-bitových veľkých opkódov[5], rozdelenie je znázornené v tabuľke 4.2.

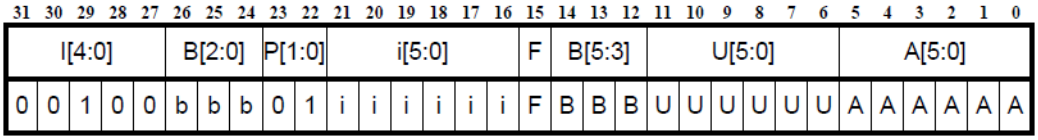
Veľký opkód	Inštrukcia a/alebo typ	Poznámky
0x00	Bcc	Vetvenie
0x01	BLcc, BRcc	Podmienené vetvenie so zápisom spätnej adresy Podmienené vetvenie na základe porovnania
0x02	LD reg + offset	Načítanie
0x03	ST reg + offset	Ukladanie
0x04	op a, b, c	Základné ARC inštrukcie
0x05-0x06	op a, b, c	Rozšíriteľné ARC inštrukcie
0x07-0x08	op a, b, c	Používateľské inštrukcie
0x09-0x0B	op <špecifikované trhom>	Trhom špecifické inštrukcie

Tabuľka 4.2: 32 bitové veľké opkódy

Veľký opkód môže určovať druh alebo samotnú inštrukciu (BR, ST, LD). Ak ide o druh, tak sa samotná inštrukcia definuje podľa šesťbitového subopkódu (aritmeticko-logické inštrukcie). Špeciálnymi prípadmi subopkódov sú SOP[6] (inštrukcie pre posuv a rotáciu) a ZOP[6] (BRK, NOP), ktoré sa kódujú na päťbitové pole cieľového registra A v prípade SOP a na pole cieľového/zdrojové registra B v prípade ZOP. Na obrázkoch 4.1, 4.3 a 4.2 sú znázornené rôzne použité kódovania inštrukcií.



Obr. 4.1: Kódovanie inštrukcie s tromi registrami (cieľový a dva zdrojové)



Obr. 4.2: Kódovanie inštrukcie s 12-bitovou okamžitou konštantnou hodnotou

[5]Opkód - Operačný kód je časť inštrukčného poľa, ktorý určuje typ inštrukcie.
[6]Single/Zero Operand Instructions - inštrukcie s jedným operandom alebo ani s jedným

Symbol	Pole	Syntax
I	I[4:0]	veľký opkód
i	i[n:0]	subopkód
A	A[5:0]	cieľový register
b	B[2:0]	spodné bity zdrojového/cieľového registra
B	B[5:3]	horné bity zdrojového/cieľového registra
C	C[5:0]	zdrojový/cieľový register
Q	Q[4:0]	<.cc> stavový kód
u	U[n:0]	bezznamienková konštanta (n je veľkosti poľa)
s	S[n:0]	spodné bity znamienkovej konštanty
S	S[m:n+1]	horné bity znamienkovej konštanty
T	S[24:21]	horné bity znamienkovej konštanty (nepodmienený skok do diaľky)
P	P[1:0]	formát operandov
M	M	conditional instruction operand mode
F	F	<.f> nastavovanie príznakov
Z	Z	<.zz> veľkosť dát
X	X	<.x> znamienkové rozšírenie

Tabuľka 4.3: Kódovacia konvencia

4.6.2 Príznaky

Procesor založený na ARCompact-e má rozsiahlu inštrukčnú sadu, z ktorej väčšina inštrukcií môže sama nastavovať príznaky.

Obnovenie príznakov nastáva len ak je použitá príznaková direktíva .F. Pre niektoré inštrukcie je jediným účinkom nastavenie príznakov a nevykonanie žiadnych zmien v registroch, takýmito inštrukciami sú napr. CMP, RCMP a TST.

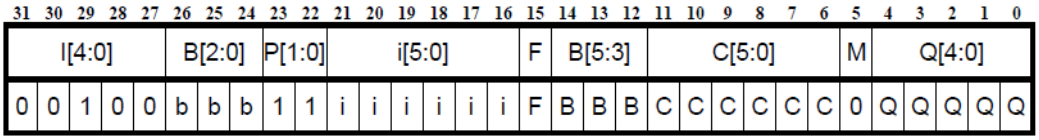
4.6.3 Podmienené vykonávanie

Rada 32-bitových inštrukcií podporuje podmienené vykonávanie. Rovnako ako pri príznakoch podmienené vykonávanie je povolené len ak je prítomná direktíva .cc, kde sa dané symboly nahradzujú konkrétnymi stavovými kódmi. Päťbitové pole Q[4:0] dovoľuje testovanie 32 nezávislých podmienok pred vykonaním inštrukcie, pričom ARC definuje 16 vlastných podmienok a zbytok necháva k dispozícii používateľovi.

4.6.4 Príklad kódovania

Je veľmi dôležité pochopenie binárneho kódovania inštrukcií pred samotnou implementáciou. Na obrázku 4.3 je vidieť pseudopríklad zakódovanej inštrukcie s dvomi registrami a podmienkou. Podľa tabuľky 4.3 sa dajú rozšifrovať jednotlivé bitové polia. Prvých päť bitov značí veľký opkód. Na obrázku je uvedený opkód s číslom štyri. Z tabuľky 4.6.1 sa dá zistiť, že ide o základnú ARC inštrukciu. Ďalšie pole značí spodné bity registru B. Dvojbitové pole P značí formát operandov. Podľa jednotlivých hodnôt môžu byť oba operandy registre (00), jeden operand register a druhý konštanta (01-šesť alebo 10 - 12 bitová) alebo podmienená inštrukcia (11). Ďalších šesť bitov značí subopkód (malý opkód). Kebyže nadobúdala hodnotu nula, tak by šlo o inštrukciu ADD. Bit F nastavuje príznaky, u tejto inštrukcie nie je napevno preddefinovaná hodnota (napríklad u inštrukcii CMP je vždy 1), čo znamená, že jej hodnota bude závisieť od príznakovej direktívy. Trojbitové pole B značí

horné bity registra B, zároveň ďalších šesť bitov poľa C značí zakódovaný register C. Pole M je špeciálny M formát, ktorý značí pri podmienených inštrukcií, či je druhý operand register (0) alebo konštanta (1). Posledných päť bitov Q značí jednu z definovaných podmienok, ak bola použitá direktíva .cc.



Obr. 4.3: Kódovanie inštrukcie s dvomi registrami a podmienkou

Toto je len jeden z mnoha formátov inštrukcií, ale princíp stavby daných inštrukcií je rovnaký aj pri ostatných formátoch. S nejastnostiami jednotlivých symbolov použitých v kódovaní pomôže tabuľka 4.3 na strane 16.

Kapitola 5

Implementácia

Táto kapitola sa zaoberá samotnou implementáciou procesoru ARC. Popisuje priebeh práce a vytvorené inštrukcie. Ďalej sa zaoberá popisom súboru `crt0.s` a rozchodením prekladača C/C++.

5.1 Udalosti

Main
Hlavná udalosť má na starosti načítavanie a dekódovanie inštrukcií. V každom inštrukčnom cykle sa načítava 1 inštrukcia do špeciálneho registra (instruction buffer),ktorý by mal mať šírku jednej inštrukcie. Kvôli využívaniu 32 bitových operandov sa musela šírka registra zdvojnásobiť aby sa do nej zmestila inštrukcia a operand, ktorý za ňou nasleduje.

Inštrukcie sa načítavajú po 32 bitoch a po ich načítaní je potrebné načítané informácie posunúť tak aby inštrukcia začínala na mieste najvýznamnejšieho bitu. Po posunutí nasleduje načítanie ďalších 32 bitov, ktoré môžu obsahovať operand. Po naplnení registra sa informácie predávajú ako argument pre dekóder. O inkrementovanie programového čítača sa stará každá inštrukcia sama a zvyšuje programový čítač o štyri alebo osem v závislosti či používa 32 bitové operandy.

Reset
Stará sa o inicializáciu všetkých registrov a programového čítača.

5.2 Registre

Registre sú vytvárané pomocou makra DEF_REG, ktorý berie 2 argumenty, názov registrovej sady a číslo registra. Tento spôsob dovoľuje jednoduché rozširovanie sady registrov na všeobecné účely ak je to potrebné, poprípade rýchlu definíciu prídavných špeciálnych registrov. Implementované boli registre: r0-r30 (registre všeobecného použitia), BLINK, STATUS32, indikátor 32-bitových operandov a programový čítač. Registre sú podrobnejšie popísané v sekcii 4.4. Tieto registre sa podľa použitia zaraďujú do dvoch registrových sád:

- gpreg_src – zdrojové registre,
- gpreg_dst – cieľové registre (napríklad nepatrí sem r63).

Zdrojový kód 5.1 znázorňuje vytvorenie arch registra STATUS32, čo znamená, že daný register bude viditeľný pre programátora. Taktiež je potrebná identifikácia architektonic-

kých registrov pre generovanie prekladača. Ďalej je ukázané vytvorenie registrového aliasu, ktorý pridáva ďalší logický pohľad a prístup k prekrývajúcemu sa zdroju. Daný alias bol vytvorený pre prístup k deviatemu bitu registra STATUS32 cez iné meno (carry). Podľa špecifikácie boli vytvorené aj ďalšie aliasy pre príznaky overflow, negative a zero.

```
arch register bit[WORD_W] STATUS32;

alias register bit[1] carry
{
    overlap = STATUS32[9..9];
};
```

Zdrojový kód 5.1: Definícia registra a vytvorenie aliasu registra

O zápis do registrov sa starajú funkcie rf_gpr_read a rf_gpr_write 5.2. Aj keď sa zdá, že je zbytočné vytvoriť funkcie ak sa zápis/čítanie do/z registrov môže rovno naimplementovať spôsobom ako je v príklade 3.1 ale pri rôznych kritériách zápisov alebo dodatočných kontrolách, ktoré môžu byť neskôr pridané sa takto ušetrí čas a minimalizuje riziko chýb pri prepisu kódu.

```
uint32 rf_gpr_read(const uint6 i)
{
    return (i != REG_GPR_PC) ? rf_gpr[i] : r_pc;
    // ak sa inštrukcia pokúsi čítať reg 63,
    // vráť aktuálnu hodnotu programového čítača
}

void rf_gpr_write(const uint6 i, const uint32 val)
{
    if (i != REG_GPR_PC)
    {
        rf_gpr[i] = val;
    }
    else
    {
        r_pc = val;
        // ak sa inštrukcia pokúsi zapisovať do reg 63,
        // vráť aktuálnu hodnotu programového čítača
    }
}
```

Zdrojový kód 5.2: Funkcie na zápis a čítanie do/z registrov

5.3 Inštrukčná sada

Inštrukčná sada sa delí na viacero typov inštrukcií, najlepšie charakterizovateľných podľa rozdelenia na veľké opkódy v sekcii 4.6.1. Bolo implementovaných 44 inštrukcií, ktoré sa môžu rozdeliť do troch väčších kategórií: riadiace, na premiestnenie dát a spracovávanie dát.

Inštrukcie sa tvoria skladaním menších celkov do komplexnejších skladieb. Ako najmenšie stavebné jednotky sú naimplementované operačné kódy a ďalšie menšie časti inštrukcií ako napríklad direktívy alebo podmienky. Na ich definíciu slúži makro `DEF_OPC`, zdrojový kód s použitím 5.3. Ako vidieť vytvára element s vlastnou assemblerovskou syntaxou, bi-

nárnym kódovaním (šírka je nastaviteľná, pretože direktívy/podmienky/opkódy majú rôznu veľkosť) a návratovou hodnotou, ktorá je použitá na spojenie s iným objektom.

```
#define DEF_OPC(name, syntax, opc, width)
    element opc_##name \
    {\
        assembler { syntax }; \
        binary { opc:bit[width] }; \
        return { opc }; \
    };
```

Zdrojový kód 5.3: Makro na definovanie opkódov

5.3.1 Riadiace inštrukcie

Riadiace inštrukcie slúžia na riadenie toku vykonávania programu. Sú zodpovedné za vykonávanie inštrukcií v správnom poradí. Patria sem:

- BRK – Zastavuje procesor, pri implementácii mikroarchitektúry vyprázdňuje zreťazenú linku.
- NOP – Žiadna operácia. Je implementovaná ako alias inštrukcie MOV s assemblerskou syntaxou MOV 0, 0. Zdrojový kód 5.4 znázorňuje zápis aliasu. Ako vidieť chýba definícia sémantiky, pretože aj keď používateľ použije inštrukciu NOP v skutočnosti sa bude vykonávať inštrukcia MOV so sémantikou definovanou v elemente i_mov, ale s binárnym kódovaním definovanou v aliasi. V binárnej sekcii je použitý ZOP (Zero OPerand) na mieste, kde by mala byť konštanta alebo register, čo značí, že ide o inštrukciu bez operandov. Na mieste registrov A a B sú nuly vyplývajúc zo syntaxi MOV 0, 0.
- Bcc – Podmienené vetvenie, ak je špecifikovaná podmienka (cc = pravda), tak vykonávanie programu pokračuje od relatívnej adresy (hodnota PC + posunutie). Podmienené vetvenie má maximálne rozpätie skoku +/- 1MB. Pri nepodmienenom vetvení (inštrukcia B) je maximálne rozpätie +/- 16 MB.
- BRcc – porovnaj a vetvi. Používa 6 základných podmienok (=, !=, >, <, >=, <=).
- Jcc – Podmienený skok. ak je špecifikovaná podmienka (cc = pravda), tak vykonávanie programu pokračuje od absolútnej adresy. Inštrukcie skokov nemajú limitované rozpätie. Môžu skočiť na ľubovolnú adresu v adresovanej pamäti.
- JLcc/BLcc – Podmienené vetvenie/skok s ukladaním spätnej adresy do registra BLINK. Spätná adresa ukazuje na nasledujúcu inštrukciu za JLcc/BLcc.

```
element i_nop: assembler_alias(i_mov)
{
    use opc_nop as opc;
    assembler{ opc };
    binary
    {
        OPC_G_ALU:bit[OPC_G_W] 0:bit[3] OPC_COMPR:bit[OPC_IND_W]
        opc 0:bit[1] 0:bit[3] ZOP:bit[REG_W] 0:bit[REG_W]
    };
};
```

Zdrojový kód 5.4: NOP ako alias inštrukcie MOV

5.3.2 Inštrukcie na premiestnenie dát

Inštrukcie na premiestnenie dát sú zodpovedné za presuv dát vo vnútri procesora. Manipulujú s dátami uloženými v registroch alebo v pamäti počítača. Implementované boli:

- MOV – inštrukcia pre posuv dát do/z registrov alebo medzi registrami. Zaujímavosťou pri tejto inštrukcii je, že ak je zdrojový alebo cieľový register programový čítač (r63), tak pri zápisu hodnoty do registra priraďuje danú hodnotu programovému čítaču a pri čítaní sa vracia aktuálna hodnota programového čítača. Je to potrebné na uchovanie spätnej hodnoty pri skokoch, inak by nebolo možné návratu z funkcií.

  Pri implementácii bola pridaná jedna verzia tejto inštrukcie, ktorá nie je v súlade so špecifikáciou. MOV má dve varianty, kde jedna pracuje len s 12-bitovou neznamienkovou hodnotou a nepovoluje podmienené vykonávanie (nie je miesto na zakódovanie podmienky do binárneho kódu) a druhá využíva šesť a 32-bitový operand plus povoluje podmienené vykonávanie. Dané varianty nebolo možné zlúčiť kvôli rôznej assemblerovskej syntaxi ale zároveň pri implementácii oboch verzií nastalo ku konfliktu, pretože nie je jednoznačné, ktorá inštrukcia by bola použitá, napríklad ak je operand 11-bitová hodnota. Riešením bola zmena syntaxu pri použití 12-bitovej varianty výmenou operandov.

- LD – inštrukcia na načítanie dát z pamäti. O načítanie dát sa stará funkcia `load_data`, ktorá berie za argumenty adresu, šírku, cieľový register a znamienkový príznak. Keďže nezarovnaný prístup k dátam nie je dovolený tak sa adresa musí rozdeliť na dve časti, na adresu základu (zarovnaná na 32 bitov) a na návestie vo vnútri slova. Po získaní základu a návestia sa vyberá správna verzia inštrukcie podľa počtu bitov (šírky), ktoré sa bubú načítavať (8/16/32) a ak je znamienkový príznak pravdivý, tak sa načítané dáta rozširujú o hodnotu najvýznamnejšieho bitu po celom slove pomocou definovaných makier `SEXT8TO32`/`SEXT16TO32` a zapíšu do cieľového registra.

- ST – inštrukcia na uloženie dát do pamäti. Rovnako ako inštrukcia LD, dokáže zapisovať 8 (STB), 16 (STW) a 32 (ST) bitov.

5.3.3 Inštrukcie na spracovanie dát

Medzi inštrukcie na spracovanie dát patria aritmetické, logické a inštrukcie na porovnávanie. Väčšina inštrukcií pracuje s tromi registrami alebo s dvoma registrami a konštantnou hodnotou, ktorá môže byť zakódovaná na 6, 12 alebo 32 bitov. Tak ako pri inštrukcii MOV, tiež bola prizmenená syntax pri 12-bitovej konštante prehodením registra za konštantu.

Aritmetické inštrukcie
Aritmetické inštrukcie vykonávajú základné matematické operácie. Patria sem:

- ABD – absolútna hodnota,
- ADD – sčítanie,
- ADC – sčítanie s bitom prenosu (carry flag),
- SUB – odčítanie,
- SBC – odčítanie s bitom prenosu,
- MAX – výber väčšej znamienkovej hodnoty,
- MIN – výber menšej znamienkovej hodnoty,
- RSUB – prevrátené odčítanie (závisí na poradí operandov),
- SEXB, SEXW – znamienkové rozšírenie najvýznamnejšieho bitu (ôsmy alebo 16. bit) až po 32. bit,
- EXTB, EXTW – rozšírenie o nuly od ôsmeho/16. bitu až po 32,
- ADD1/2/3 – sčítanie s ľavým posunom bitov o jeden až tri bitov druhého operandu,
- SUB1/2/3 – odčítanie s ľavým posunom bitov o jeden až tri bitov druhého operandu,
- ASL – aritmetický posun bitov do ľava,
- ASR – aritmetický posun bitov do prava,
- ROR – rotácia bitov do prava.

Logické inštrukcie

- AND, OR, XOR – základné bitové operácie,
- BIC – bitový AND s invertovaním bitov druhého operandu,
- BSET – nastavenie (na jednotku) individuálneho bitu v prvom operande, druhý operand obsahuje hodnotu, ktorá definuje pozíciu daného bitu (používa sa len spodných päť bitov operandu na určenie pozície),
- BCLR – vynulovanie individuálneho bitu,
- LSR – logický posun bitov do prava,
- RLC – rotácia do ľava s príznakom prenosu (carry),
- RRC – rotácia do prava s príznakom prenosu,
- NOT – inverzia jednotlivých bitov.

Inštrukcie porovnávania Tieto inštrukcie zahadzujú výsledok operácií, používajú sa len na nastavovanie príznakov. Bitové pole príznakovej direktívy je vždy nastavená na jednotku aj keď sa nepoužíva prípona .F. Používajú len zdrojové registre, pričom cieľový nie je vôbec prítomný:

- TST – logický bitový AND,
- CMP – porovnanie je vykonané odčítaním druhého operandu od prvého s nasledovným nastavením príznakov,
- RCMP – obrátené porovnanie je vykonané rovnako ako u CMP výmenou operandov.

5.3.4 Inštrukcie s podmieneným vykonávaním

Inštrukcie s podmieneným vykonávaním sú napríklad MOV, TST, CMP, J. Funkcia `condition_test` kontroluje podľa opkódu podmienky či bola podmienka splnená alebo nie. Ak podmienke bolo vyhovené, funkcia vracia jednotku a daná inštrukcia môže vykonať svoju operáciu. Aby sa predošlo implementácii ďalších verzií týchto inštrukcií tak bol pridaný prázdny podmienečný opkód CC_UNCOND s prázdnym assemblerovským telom, čo znamená ak nebude prítomná za inštrukciou podmienená direktíva tak sa daná inštrukcia vždy vykoná.

5.3.5 Inštrukcie s nastavovaním príznakov

Podobne ako pri inštrukciách s podmieneným vykonávaním, nastavovanie príznakov je povolené len ak je prítomná príznaková direktíva. Nie je možné pri každej sade inštrukcií automaticky kontrolovať príznaky. Aj keď inštrukcie patria do jednej skupiny, tak podľa špecifikácie, napríklad pri inštrukciách bitových posuvov a rotácií, sa nastavované príznaky líšia. Preto bolo nutné vo väčšine prípadov implementovať nastavovanie príznakov jednotlivo pri každej inštrukcii.

5.4 Súbor crt0.s a newlib

Súbor crt0.s obsahuje štartovací kód pre programy napísané v jazyku C. Jej hlavnou úlohou je inicializácia ukazovateľa vrcholu zásobníka a volanie hlavnej funkcie. Taktiež obsahuje telá funkcií abortu a exitu používaných v testoch prekladača.

Newlib je štandardná C knihovňa určená na použitie v vstavaných systémoch, ktoré nemajú operačný systém. Je kolekciou niekoľkých častí knižníc. Obsahuje komplexnejšiu verziu súboru crt0.s (napr. ukladanie argumentov).

Súbor crt0.s a assemblerovské časti newlibu sú napísané v assemblerovskom jazyku modelovaného procesora. V konfiguračnom súbore sa nastavuje generovanie daných súborov.

Kapitola 6

Testovanie

Táto kapitola sa zaoberá popisom jednotlivých fáz testovanie počas implementácie. Na konci sa venuje porovnaniu vytvoreného modelu s voľne dostupným modelom ARC 700[1].

6.1 Assembler

Codasip Studio dovoluje vygenerovanie assembleru aj keď je implementovaných len pár inštrukcií. Presnejšie hneď jak boli implementované inštrukcie NOP a BRK bolo možné testovať správnosť chovania týchto inštrukcií. Takže prvá fáza testovania prebiehala súbežne s implementovaním jednotlivých inštrukcií. Bol vytvorený jeden assemblerovský súbor, do ktorého sa postupne pridávali nové inštrukcie pričom sa netestovala len základná funkčnosť (napríklad pri ADD sčítanie) ale aj správnosť nastavovania príznakov a podmieneného vykonávania. Vytvorený súbor pomáhal odstaňovaniu chýb po väčších úpravách ako napríklad po premiestnení funkcie inkrementovania programového čítača z hlavnej udalosti do každej inštrukcie (popísané v sekcii 5.1).

6.2 Testy prekladača

Testovanie prekladača prebiehalo pomocou sady testov firmy Codasip. Daná sada obsahuje štyri druhy testov: základné, testy celých čísiel, reálnych čísiel a reálnych čísiel s dvojitou presnosťou. Na spustenie testov je potrebná exportovaná sada nástrojov. Prekladač musí byť generovaný s základnou C knižnicou a runtime knižnicou na úspešné vykonanie všetkých testov.

Testy prekladača boli nápomocné hlavne pri generovaní runtime knižnice. Prekladač sa vygeneroval ale niektoré programy končili pri preklade s chybou (napríklad násobenie dvoch premenných) lebo chýbali funkcie z danej knižnice.

Jednotlivé testy mohli končiť s rôznym výsledkom, tie najčastejšie sa vyskytujúce boli:

- OK – test skončil bez chýb,
- CC – chyba pri prekladaní zdrojového súboru,
- LNK – chyba pri linkovaní,
- TO – vypršal čas na dokončenie testu,

[1] Dostupné z stránky http://www.ovpworld.org/ip-vendor-synopsys-arc

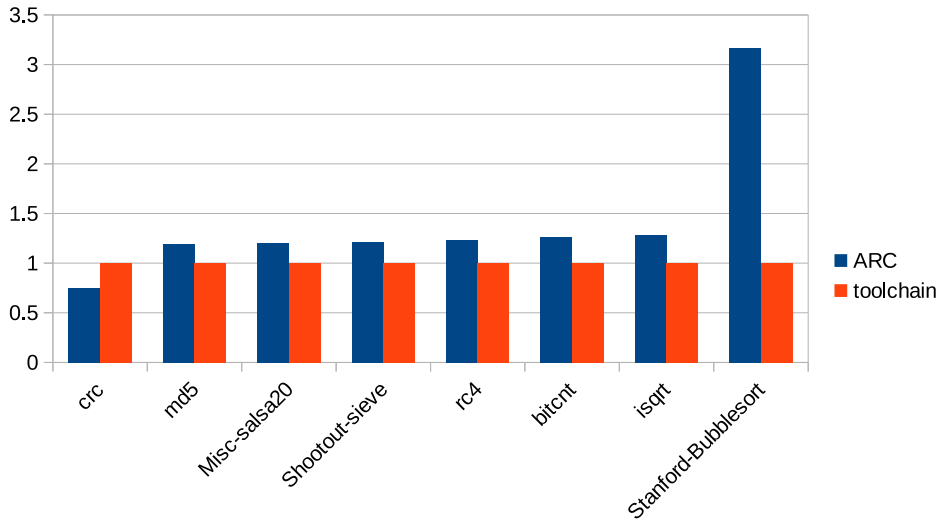
- EC – nesprávny návratový kód programu,
- SIM – chyba simulátora.

Výsledky testov s optimalizáciou o3:

- OK: 910,
- CC: 1 - padá pri prekladu kvôli runtime knižniciam,
- ostatné: 0.

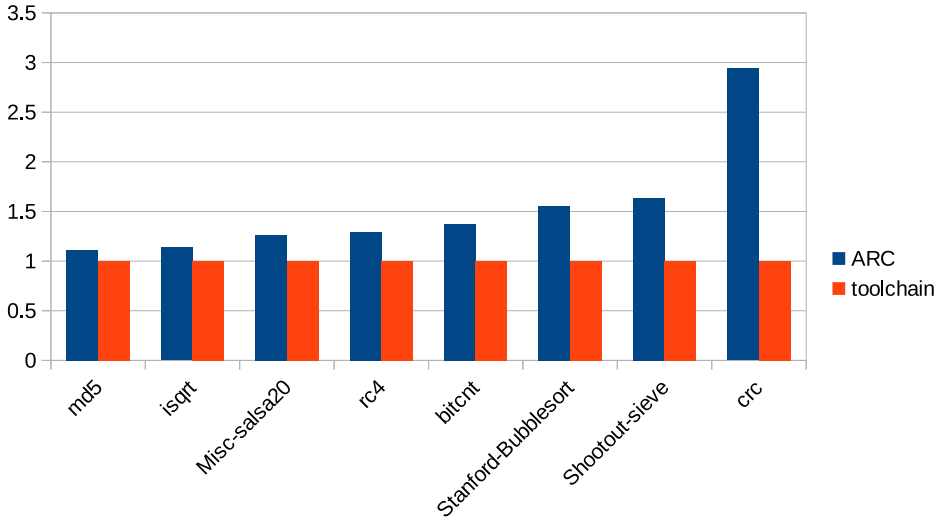
6.3 Porovnanie modelu s existujúcim riešením

Naimplementovaný model bol porovnaný s procesorom ARC 700 na sade benchmarkov poskytnutých firmou Codasip. Dokopy bolo osem rôznych testov na rôzne algoritmi (Eratosthenovo sito, arc4), triedenie alebo odmocninu. Na grafoch sú hodnoty udávané v percentách a odvíjajú sa od existujúceho riešenia (toolchain). Takže vyššie hodnoty modelu ARC znamenajú dosiahnutie horších výsledkov.



Obr. 6.1: Porovnanie na základe vykonaných cyklov

Na obrázku 6.1 je vidieť porovnanie spomínaných dvoch procesorov na základe vykonaných cyklov. Vytvorený model dosiahol lepšie výsledky len pri teste cyklickej redundantnej kontroly a to cca o 25%. V ostatných prípadoch vykonával o 20-28% viac cyklov, okrem extrémneho prípadu pri bublinkovom triedení (bubblesort), kde vytvorený model vykonal trojnásobný počet cyklov.



Obr. 6.2: Porovnanie veľkostí kódu

Na obrázku 6.2 je vidieť porovnanie na základe veľkosti vygenerovaného kódu. Ani v jednom prípade nedosiahol vytvorený model lepšie výsledky ako existujúci. Väčšina výsledkov je horšia o 11-64%. Rovnako ako pri cykloch sa v jednom prípade objavila väčšia odchýlka. Pri cyklickej redundantnej kontroly vytvorený model dosiahol skoro trojnásobne horší výsledok, čo je prekvapivé na základe toho, že rovnaký test pri cykloch bol jediný, ktorý dosiahol lepšie výsledky ako existujúce riešenie.

Kapitola 7

Zhodnotenie práce s Codasip Studiom

Práca s integrovaným vývojovým prostredím Codasip Studio je samo intuitívne a pri problémoch sa dá spoľahnúť na používateľskú alebo technickú príručku. Prácu veľmi urýchluje možnosť prepínania medzi rôznymi náhľadmi. Najviac použité boli náhľady Codasip a Debug. Naďalej pri každom náhľade je možné si otvoriť ďalšie pohľady napríklad do registrov. Tieto pomocné nástroje výhodne doplňujú pracovnú plochu a zvyšujú efektivitu práce. Za nevýhodu považujem nedostatočné odlíšenie Codasip nástrojov v nastaveniach, čo často zapríčinilo prepisovanie nastavení iných nástrojov.

Pri práci s Codasip Studiom bolo odhalených viacero vnútorných chýb a niekoľko malých grafických nezrovnalostí napríklad pri zobrazení výstupu z disassembleru, kde sa názov inštrukcie v niektorých prípadoch zlučuje s prvým operandom. Chyba kvôli ktorej neprechádzalo viacero testov bola pri inštrukcii ADD1, ktorá má za úlohu spočítať dve hodnoty pričom druhú hodnotu ešte posunie o jeden bit do ľava. Prekladač generoval kód, kde danú hodnotu posúval v opačnom smere, čo zapríčinilo stratu najnižšieho bitu, kvôli čomu sa načítavali hodnoty z pamäti z nesprávnych miest. Dané chyby boli nahlásené a väčšina z nich je aj odstránená.

Pri rozchádzaní prekladača na začiatku nastali problémy s chýbajúcimi inštrukciami, ktoré vedia pracovať s 32-bitovými operandmi. Špeciálnym prípadom chýbajúcich inštrukcií bolo, keď implementované inštrukcie pre porovnanie nerozoznal prekladač. Problémom bolo, že sa v podmienke nachádzali príliš komplikované operácie nad príznakmi, ktoré nevyhovovali pravidlám prekladača. Riešením bolo nájdenie ekvivalentných zjednodušených výrazov.

Ďalšie problémy nastali pri prekladaní runtime knižníc. Prekladač potrebuje priamy skok s dostatočne veľkým rozpätím ale v modeli sa nachádzal len relatívny skok, ktorý vyhovuje tejto podmienke (inštrukcia B). Takže bolo potrebné pridať alias tejto inštrukcie, ktorý vyzerá ako priamy skok pre prekladač. Prekladač jazyka C/C++ potom generuje syntax tohto aliasu ale assembler zostavuje syntax z originálnej inštrukcie s nepriamym skokom.

Najčastejšou chybou pri implementácii bola zhoda assemblerovského zápisu pri rôznych variantách (6,12 alebo 32-bitová konštanta) inštrukcií. Na odstránenie tohto problému bol zmenený assemblerovský zápis pri jednej z variant, čo nesúhlasí so špecifikáciou daného procesora. Preto by som navrhoval zavedenie podpory pre definovanie inštrukcií s identickým asseblerovským zápisom ale s rôznym binárnym kódovaním a chovaním. Dané vylepšenie je

dôležité len v prípade ak tento problém nastáva často a nie len pri procesoroch podobných ARC-u.

Počas implementácii vyšlo viacero verzií Codasip Studia, čo niekedy komplikovalo priebeh práce. S novými verziami prichádzali nové chyby, ktoré niekedy zapríčinili, že predtým funkčný model sa nedal preložiť. V daných situáciách sa buď hľadala chyba, ktorá to zapríčinila alebo sa naďalej pracovalo na staršej verzii Codasip Studia. Rovnaká situácia nastala tiež pri aktualizácii operačného systému. Preto je veľkou výhodou Codasip Studia jednoduchosť inštalácie a fakt, že Codasip Studio sa neaktualizuje na novú verziu ale je možné si nainštalovať viaceré verzie štúdia vedľa seba.

Kapitola 8

Záver

Cieľom tejto bakalárskej práce bolo vytvorenie modelu procesora pomocou ADL jazyka. K práci bolo potrebné zoznámenie sa s vývojovým prostredím Codasip Studio a jazykom CodAL na popis architektúry. Práca sa v prvej časti venuje zasväteniu čitateľa do danej problematiky a poskytuje teoretický základ pre pochopenie aktuálneho stavu procesorov a poznatkov spojených s ich návrhom. V druhej časti sa venuje predstaveniu procesora ARC, s jej implementáciou a testovaním. Daný procesor bol úspešne implementovaný na inštrukčnej úrovni. Po úprave a pridaní ďalších inštrukcií bol vygenerovaný prekladač jazyka C/C++ a otestovaný na sade testov.

Pri prvotnom návrhu modelu a modelovaných inštrukcií nebol braný zreteľ na prácu s 32-bitovými konštantami, čo spôsobilo, že pred generáciou prekladača bolo nutné prepísať značnú časť kódu. Daná nepozornosť ukazuje aký dôležitý je počiatočný návrh modelu.

Vytvorený model bol následne porovnaný s existujúcim riešením. Výsledky porovnania dokazujú, že daný model je použiteľný. Aj keď je výkonnosť vytvoreného modelu horšia než existujúceho riešenia, rozdiely v nameraných hodnotách nie sú veľmi veľké okrem niektorých extrémnych prípadov.

Počas práce som získal nové znalosti zo sveta procesorov a podrobnejšie sa zoznámil s jednotlivými úrovňami návrhu. Ďalej som sa bližšie zoznámil s inštrukčnou sadou ARCompact, z ktorej vychádza vytvorený model. Pri implementácii som narazil na viacero chýb Codasip Studia, ktoré boli nahlásené. Taktiež som hlbšie porozumel ADL jazyku CodAL. Vytvorený model je ďalej rozšíriteľný na špeciálne účely doimplementovaním rozšíriteľných inštrukcií a registrov.

Literatúra

[1] ARC International: *ARCompact^{TM} ISA Programmer's Reference*. 2008, [Online; navštívené 14.05.2017].
URL http://me.bios.io/images/d/dd/ARCompactISA_ProgrammersReference.pdf

[2] Codasip Ltd.: *Codal Language Reference Manual*. Dokumentácia firmy Codasip s.r.o., 14.05.2017.

[3] Codasip Ltd.: *Codasip Studio User Guide*. Dokumentácia firmy Codasip s.r.o., 14.05.2017.

[4] Liu, D.: *Application Specific Instruction Set DSP Processors volume 2*. Morgan Kaufmann, 2008, ISBN 978-0-12-374123-3.

[5] Manjikian, N.; Hamacher, C.; Vranesic, Z.; aj.: *Computer Organization and Embedded Systems 6th*. McGraw-Hill Higher Education, 2011, ISBN 978-0-07-338065-0.

[6] Mishra, P.; Dutt, N.: *Architecture Description Languages*. [Online; navštívené 14.05.2017].
URL https://pdfs.semanticscholar.org/a04e/059882df0de21c6dc6c640289cf79feb4f84.pdf

[7] Paillard, B.: *An Introduction To Digital Signal Processors*. 2002, [Online; navštívené 14.05.2017].
URL http://dsp-book.narod.ru/InDSPrcs.pdf

[8] Smith, M. J. S.: *Application-Specific Integrated Circuits*. Addison-Wesley Professional, 1997, ISBN 978-0321602756.

[9] Stallings, W.: *Computer Organization and Architecture 8th edition*. Prentice Hall, Inc., 2010, ISBN 978-0136073734.

Príloha A

Obsah CD

Priložené CD obsahuje:

- Doc – Obsahuje vypracovanú dokumentáciu technickaSprava.pdf.
- Model – Obsahuje vytvorený model ARC.
- Tex – Obsahuje zdrojové súbory a ďalšie materiály použité pri tvorbe technickej správy.