VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

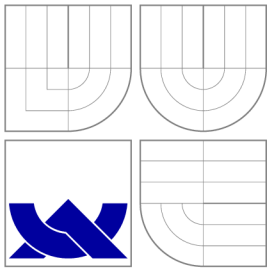
# KNIHOVNA PROCESORŮ PRO NÁVRH VESTAVĚNÝCH SYSTÉMŮ

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

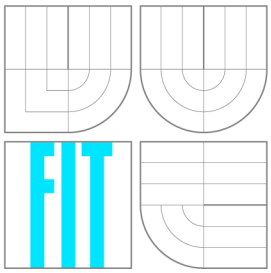
AUTOR PRÁCE
AUTHOR

Bc. RADOVAN ZVONČEK

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# KNIHOVNA PROCESORŮ PRO NÁVRH VESTAVĚNÝCH SYSTÉMŮ

PROCESSORS LIBRARY FOR THE EMBEDDED SYSTEM DESIGN

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADOVAN ZVONČEK

VEDOUCÍ PRÁCE
SUPERVISOR

prof. Ing. TOMÁŠ HRUŠKA, CSc.

BRNO 2011

Sem vložte zadání.

# Abstrakt

Tato práce se zabývá vytvořením knihovny modelů procesorů používaných ve vestavěných systémech. Architektury procesorů jsou popsány pomocí jazyka ISAC, který je jedním z výstupů projektu Lissom běžícím na Fakultě informačních technologií VUT v Brně. První část práce je věnována seznámení se současnými architekturami procesorů vestavěných systémů. Zbytek práce je pak věnován představení zvolených architektur a popisu jejich implementace. Závěr práce tvoří shrnutí získaných poznatků týkající se hlavně vhodnosti jazyka pro popis zvolených architektur a efektivnosti jejich simulace.



# Abstract

This work deals with designing a library of processor models used in embedded systems. Processor architectures are described using the ISAC language. The ISAC language is one of several outcomes of the Lissom project that is taking place at the Faculty of Information Technology, BUT, Brno. The beginning of this work is aimed to provide the introduction to processor architectures used in today's embedded systems. Remaining sections are devoted to presentations of exemplary processor architectures and the description of their implementation. This work is finalized by concluding the gathered experience with emphasis on the suitability of the ISAC language for architecture description and the efficiency of its simulation.



# Klíčová slova

Lissom, ISAC, hardware, software, co-design, architektura, procesor, model, simulace.

# Keywords

Lissom, ISAC, hardware, software, co-design, architecture, processor, model, simulation.

# Citace

Radovan Zvonček: Knihovna procesorů pro návrh vestavěných systémů, diplomová práce, Brno, FIT VUT v Brně, 2011

# Knihovna procesorů pro návrh vestavěných systémů

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Prof. Ing. Tomáše Hrušky, CSc.

.....
Radovan Zvonček
24. května 2011

## Poděkování

Na tomto místě bych rád poděkoval prof. Ing. Tomáši Hruškovi, CSc., vedoucímu mé diplomové práce za jeho cenné rady. Také bych rád poděkoval Ing. Karlu Masaříkovi, Ph.D., Ing. Zdeňku Přikrylovi a Ing. Adamu Husárovi za další odbornou pomoc.

# Obsah

# Kapitola 1

# Úvod

Trh počítačových systémov je jedným z najstabilnejšie rastúcich a najperspektívnejších trhov na svete. Majoritný podiel na tomto úspechu majú vnorené systémy, ktoré nachádzajú uplatnenie v čoraz väčšom počte produktov a zariadení.

Keďže jadrom každého počítačového systému je procesor, podobná situácia panuje aj na trhu procesorov. Svetová produkcia procesorov pozostáva takmer výhradne z procesorov určených do vnorených systémov. V poradí druhá najzastúpenejšia kategória, procesory pre osobné počítače, je zastúpená len v rádoch desatín percent.

Masívna možnosť uplatnenia poskytuje množstvo motivácie pre výrobcov procesorov. Tí sa samozrejme snažia ponúknuť produkty, ktoré sú v niektorom z mnohých aspektov lepšie než tie konkurenčné.

Životný cyklus procesoru je možné (veľmi hrubo) rozdeliť do troch základných častí. Počas prvej fázy je vykonávaný návrh procesoru a jeho verifikácia. Jej súčasťou často býva výskum nových technológií a konceptov, ktorý spotrebuje ohromné množstvo zdrojov. Druhou fázou je výrobný proces, ktorý je, naopak, relatívne lacný. To je zapríčinené hlavne veľkým množstvom produkovaných kusov procesoru. Poslednou fázou je potom vlastné nasadenie procesoru v nejakom vnorenom systéme.

Snahou všetkých výrobcov procesorov je čo najviac zefektívniť návrh a verifikáciu procesoru, pretože to je obdobie, počas ktorého sa dajú najviac znížiť celkové náklady na produkt.

S podobnou myšlienkou bol odštartovaný projekt Lissom, ktorého cieľom je vytvoriť nástroje pre súčasný návrh hardvéru procesoru a softvéru cieleného na navrhovaný procesor. Projekt umožňuje popísať architektúru procesoru na vyššej úrovni abstrakcie pomocou pre tento účel navrhnutého jazyka ISAC (Instruction Set Architecture C, [7]). Popis procesoru potom môže byť automaticky transformovaný do jazyka pre popis hardvéru a pomocou už existujúcich technológií fyzicky realizovaný. Okrem toho je možné na základe popisu automaticky generovať programovacie nástroje pre navrhovaný procesor.

Hlavným prínosom projektu Lissom je, že umožňuje s architektúrou pracovať ešte pred jej realizáciou. Tým je možné súčasne navrhovať hardvérové špecifiká architektúry, vyvíjať softvér optimalizovaný pre navrhovaný procesor a verifikovať niektoré aspekty navrhnutého riešenia. Výsledkom tejto možnosti je okrem zníženie celkových nákladov na vývoj nového procesoru aj skrátenie doby potrebnej na celý proces.

V tejto práci budú s pomocou prostriedkov a nástrojov projektu Lissom vytvorené modely niekoľkých procesorov používaných vo vnorených systémoch. Hlavný prínos práce bude spočívať v prieskume použiteľnosti a vhodnosti jazyka ISAC pre popis týchto architektúr.

Vedľajším efektom bude vytvorenie knižnice modelov procesorov, ktoré budú demonštrovať techniky používané pri vytváraní modelov. Takéto informácie by mohli byť potom použité pre poskytnutie východiskového bodu pre zoznámenie s prostredím projektu Lissom z hľadiska užívateľa.

Prvá kapitola práce je venovaná zoznámeniu s konceptmi architektúr procesorov používaných v súčasných vnorených systémov. Znalosť týchto konceptov poskytne vhodný kontext pre štúdium architektúr reálnych procesorov. Popísané koncepty sú rozdelené do kategórií podľa toho, akým aspektom architektúry sa zaoberajú.

V nasledujúcej kapitole je obsiahnuté zoznámenie s prostredím projektu Lissom. Nosná sekcia je venovaná predstaveniu jazyka ISAC ako jazyka pre popis architektúry procesorov. Ďalej je stručne popísaný proces transformácie popisu architektúry. Mierne väčší dôraz je venovaný predstaveniu užívateľského rozhrania projektu Lissom.

Úlohou ďalšej kapitoly je predstaviť v tejto práci modelované procesory. Dôraz je kladený na tie aspekty architektúr procesorov, ktoré sú dôležité pre vytvorenie modelu procesoru na príslušnej úrovni popisu.

V predposlednej kapitole je pre každý modelovaný procesor popísaný postup implementácie. Pre každý procesor je uvedené aký postup bol počas implementácie zvolený a aké metódy a konštrukcie jazyka ISAC boli pre implementáciu použité.

Posledná kapitola je venovaná zhodnoteniu implementácie a dosiahnutých výsledkov. Je v nej zhodnotená úspešnosť implementácie a vysvetlené prípadné nedostatky jazyka ISAC.

Záver práce je tvorený stručným zhrnutím a zhodnotením celej práce.

# Kapitola 2

# Procesory vnorených systémov a ich architektúry

Úlohou tejto kapitoly je poskytnúť priestor pre zoznámenie sa s architektúrami procesorov používaných vo vnorených systémoch. Kvôli zabezpečeniu zrozumiteľnosti však nie je vhodné začať bezhlavo rozpravu o špecifikách rôznych architektúr, ale je potrebné postupovať systematicky. Preto bude úvod tejto kapitoly venovaný stručnému úvodu do problematiky vnorených systémov. Potom už nebude nič brániť prechodu k podrobnému popisu konkrétnych architektúr.

## 2.1 Vnorené systémy

Vnorené systémy sú trieda počítačových systémov, ktorej formálne definovanie je pomerne zložité. Môže za to hlavne skutočnosť, že vnorené systémy podliehajú intenzívnemu vývoju spôsobeného pokrokom v ich výrobnej technológii a dramatickému zníženiu ceny zahrnutia rôznych doplnkových komponentov [13]. Preto bude mať vysvetlenie pojmu vnorený systém viac neformálny charakter.

Pre vnorené systémy je možné identifikovať niekoľko charakteristík.

Vnorené systémy sú skutočne vnorené. Vnorený systém je možné charakterizovať ako počítačový systém (počítač) zabudovaný do nejakého väčšieho celku, alebo do nejakej inej aplikácie. To znamená, že vnorený systém je iba doplnkom alebo rozšírením nejakého iného produktu a užívateľ často nevie o jeho prítomnosti. Vhodný, zďaleka však nie jediný, príklad je možné nájsť v automobilovom priemysle. Do moderného automobilu býva zabudovaných niekoľko vnorených systémov, ktoré často krát pracujú úplne autonómne.

Vnorené systémy sú v niektorých aspektoch veľmi obmedzené. Hostiteľské zariadenia definujú rôzne obmedzenia pre vnorený systém. Limitujúcim faktorom býva typicky rozmer zariadenia alebo spotreba energie.

Ďalšou charakteristikou je účelovosť. Od začiatku vývoja býva vnorený systém navrhovaný s úmyslom optimalizácie na jedinú úlohu, ktorú bude vykonávať počas svojho života. Vnorený systém sa tak stane veľmi výkonným pri riešení svojej úlohy, avšak ostane takmer nepoužiteľný pre riešenie akýchkoľvek iných úloh.

Pre vnorené systémy je charakteristická tiež konektivita. Zo svojej podstaty je vnorený systém integrovaný do nejakého väčšieho celku, s ktorým musí komunikovať. Vnorený systém teda typicky obsahuje niekoľko rozhraní pre komunikáciu s okolím.

Iné významné vlastnosti sú napríklad spoľahlivosť a vysoká rýchlosť odozvy.

Vhodné zhrnutie pojmu je možné nájsť napríklad v [14], kde je vnoreným systémom chápaný dedikovaný počítač pracujúci v reálnom čase ako súčasť nejakého uceleného systému. Slúži pre riadenie celého systému a poskytuje výpočtové služby ostatným častiam systému.

### 2.1.1 Porovnanie vnorených systémov s univerzálnymi počítačmi

Vlastnosti vnorených systémov je teda možné zhrnúť do týchto bodov:

- vykonávajú jeden program po celý život,
- užívateľ zvyčajne netuší, že pracuje s počítačom,
- hlavná interakcia nemusí byť s človekom.

Oproti tomu, univerzálny počítač by mal:

- umožňovať spúšťanie rôznych programov,
- obsahovať periférie primárne pre komunikáciu s užívateľom,
- obsahovať operačný systém a umožňovať ukladať dáta.

## 2.2 Procesory vnorených systémov

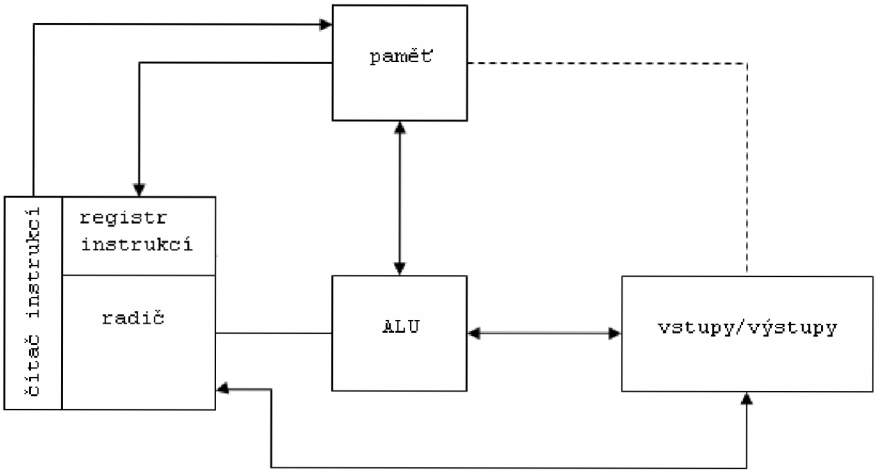
Podobne ako u väčšiny počítačových systémov, aj vnorené systémy obsahujú jeden centrálny prvok - procesor. V [6] je procesor definovaný ako zariadenie, ktoré transformuje vstupné dáta na výstupné podľa určitého predpisu, tj. programu. Obecne ide o centrálnu výpočtovú jednotku počítačového systému, ktorá vykonáva základné výpočty a riadi ostatné komponenty systému. Procesor býva označovaný aj pomocou skratky CPU (angl. central processing unit).

Procesory vnorených systémov je možné klasifikovať do dvoch tried:

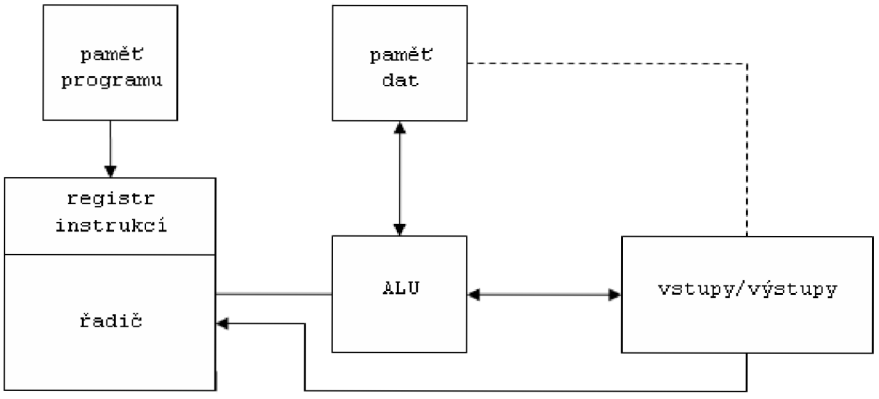
- mikroprocesory - bežné procesory s jednoduchou štruktúrou, malou spotrebou a nízkou cenou,
- mikrokontroléry - procesory, ktoré na svojom čipe obsahujú radu periférií, čím redukujú plochu a cenu výsledného vnoreného systému.

Procesory vnorených systémov bývajú založené na rade architektúr a konceptov. Existujúce a najčastejšie používané procesory väčšinou kombinujú niekoľko prístupov. Situácia je teda úplne odlišná ako v oblasti procesorov pre osobné počítače, kde prakticky dominuje jediná architektúra.

Zvyšok tejto kapitoly bude teda venovaný popisu pôvodných architektúr a konceptov, k torých rysy je možné nájsť v súčasných procesoroch vnorených systémov.




Obrázek 2.1: Von Neumanova architektúra. Zdroj [11].




Obrázek 2.2: Harwardská architektúra. Zdroj [11].

### 2.2.1 Koncept organizácie pamäte

Koncept organizácie pamäte sa okrem samotného procesoru dotýka organizácie celého počítačového systému. Existujú dva prístupy, ktoré sa líšia v rozvrhnutí pamäťového priestoru:

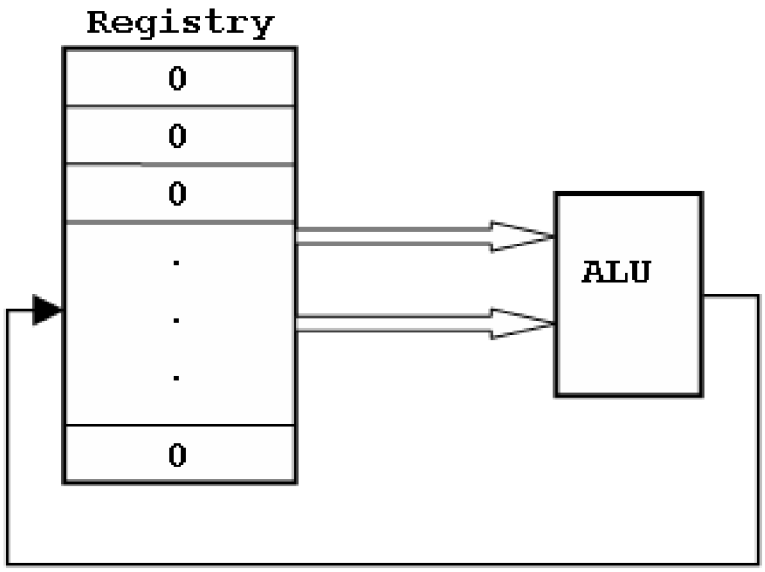
- Von Neumannovo usporiadanie - procesor udržiava jeden adresný priestor, ktorý je zdieľaný programom aj dátami. Z toho vzniká možnosť chápať inštrukcie ako dáta a naopak. Táto skutočnosť sa dá využiť na tvorbu seba-modifikujúceho kódu, avšak vzniká tak aj riziko (ne)úmyselného zneužitia tejto funkcionality. Je zodpovednosťou programátora zaistiť korektné korektnosť vykonávaných inštrukcií. Usporiadanie je znázornené na obrázku 2.1.

- Harwardské usporiadanie - je založené na dvoch oddelených adresných priestoroch. Inštrukcie sú uložené v separátnom pamäťovom bloku a nie je možná ich modifikácia. Usporiadanie je znázornené na obrázku 2.2.

### 2.2.2 Organizácia registrov

Základom väčšiny procesorových architektúr je množina registrov. Registre sú rýchle pamäťové elementy s relatívne malou kapacitou. Slúžia na dočasné uchovávanie vstupných hodnôt a výsledkov operácii procesoru. Podľa [6] sa architektúry procesorov delia na:

- registrové - procesor má k dispozícii konečnú množinu registrov, ku ktorým môže ľubovoľne pristupovať. Počet takto dostupný registrov je daný účelom procesoru. Čím viac registrov je k dispozícii, tým viac dočasných výsledkov dokážu uchovať a tým znížiť počet prístupov do pomalších pamätí nižšieho rádu. Registrová architektúra je zobrazená na obrázku 2.3,

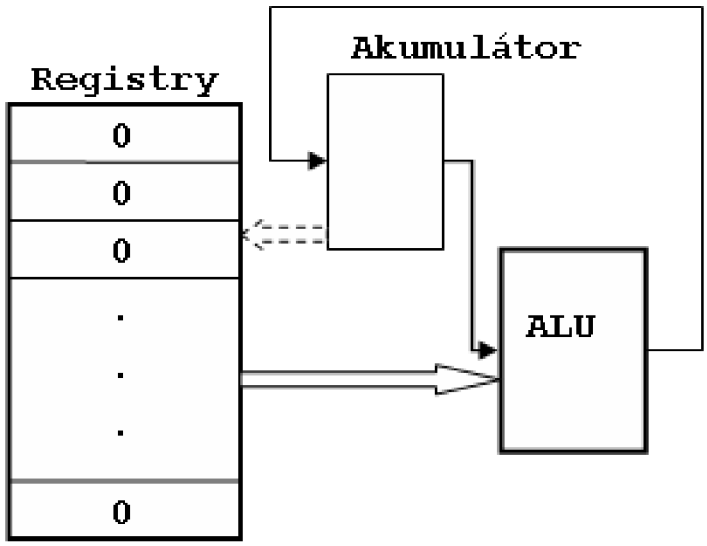



Obrázek 2.3: Registrová architektúra. Zdroj [6].

- zásobníkové - nožina registrov je v tomto prípade organizovaná podľa konceptu zásobníku. Procesor môže pristupovať len na vrchol zásobníku. Tento koncept je efektívny

z hľadiska kódovania programu, pretože odpadá nutnosť explicitne kódovať čísla registrov. Okrem toho, toto usporiadanie registrov kopíruje operačnú pamäť procesoru. Nevýhodou však je neintuitívnosť a zložitosť programovania takejto architektúry človekom,

- akumulátorové - množina registrov je rozšírená o jeden špeciálny register - akumulátor. Ten slúži ako zdrojový a cieľový operand zároveň. Uplatnenie nachádza väčšinou vo výpočtoch, kde sa výsledky operácie používajú nasledujúcej operácii. Takýto prístup šetrí zbytočné prenosy dát medzi registrami a mierne redukuje dĺžku kódových slov inštrukcií. Obrázok 2.4 znázorňuje akumulátorovú architekúru,




Obrázek 2.4: Akumulátorová architektúra. Zdroj [6].

### 2.2.3 Zložitosť inštrukčnej sady

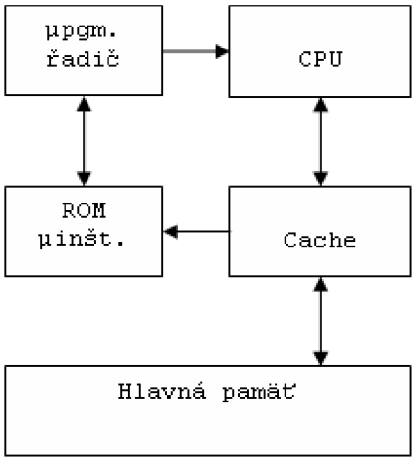
Iným kritériom, podľa ktorého sa dajú klasifikovať procesory je charakter ich inštrukčných sád.

#### CISC procesory

Pôvod CISC (angl. complex instruction set computer) procesorov ma historické opodstatnenie. Prvé procesory narážali na technologické limity. Pamäte prvých mikroprocesorov boli výrazne pomalšie než samotné procesorové jadrá. Preto sa návrhári snažili minimalizovať počet prístupov do pamäte, čo v podstate znamenalo redukovať počet načítaných inštrukcií. Inštrukcie boli teda navrhované veľmi komplexne. Jediná inštrukcia programu bola procesorom rozložená na niekoľko mikro-inštrukcií a až tieto mikro-inštrukcie boli procesorom vykonané. Napríklad, inštrukcia sčítania bola rozložená na mikro-inštrukcie pre načítanie zdrojových operandov, samotné sčítanie a uloženie výsledku. Z tohto princípu vyplývajú základné vlastnosti CISC procesorov:

- rôznorodý, avšak optimalizovaný, formát kódovania inštrukcií,
- menší program znamenajúci menej načítaní inštrukcií,

- veľké množstvo adresných módov,
- inštrukcie priamo odpovedajú konštrukciám vyšších programovacích jazykov.




Obrázek 2.5: Architektúra CISC. Zdroj [5].

Schému typického CISC procesoru je možné vidieť na obrázku 2.5.

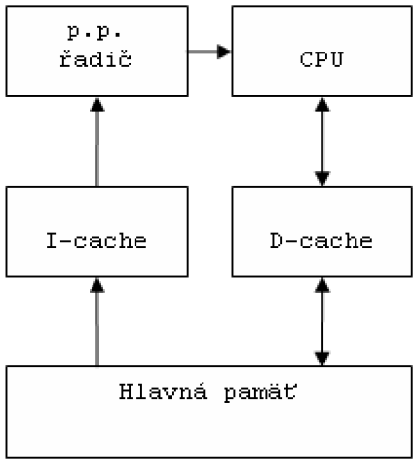
### RISC procesory

Ako protiklad k CISC procesorom bol navrhnutý koncept procesorov RISC (angl. reduced instruction set computer). Motiváciou bolo zistenie, že procesor väčšinu svojho výpočtu vykonáva len malú podmnožinu inštrukcií. Návrhári sa preto snažili presunúť zložité operácie z mikro-programu (tj. postupnosti mikro-inštrukcií) do bežného programu. Výsledkom malo byť ušetrenie plochy čipu, pretože ten už nemusel umožňovať dekódovanie komplexných inštrukcií. Okrem toho, výsledná inštrukčná sada sa stala efektívnejšou a výkonnejšou.

V [5] autor považuje nasledovné vlastnosti RISC za základné:

- malý počet relatívne jednoduchých inštrukcií,
- dokončenie inštrukcie v každom takte,
- pevná dĺžka inštrukčného slova, prípadne malý počet formátov, jednotné kódovanie,
- riadenie mikroprocesoru dané pevnou logikou, nie mikroprogramovaním,
- operácie vykonávané výhradne nad registrami, 2 zdrojové a 1 cieľový register (zdrojové operandy zachované aj po zápise výsledku),
- prístup do pamäte iba s pomocou inštrukcií presunu, malý počet adresných módov,
- veľký počet programovo dostupných registrov,
- nutnosť optimalizujúceho kompilátoru pre naplánovanie inštrukcií kvôli dosiahnutiu optimálneho prekrytia a zreťazenia vykonávania inštrukcií.

Znázornenie generického RISC procesoru je možné vidieť na obrázku 2.6.




Obrázek 2.6: Architektúra RISC. Zdroj [5].

### 2.2.4 Vydávanie a spracovanie inštrukcií

Počas vývoja procesorov bolo navrhnutých niekoľko prístupov, ktoré sa v súčasných procesoroch rôzne kombinujú.

#### Skalárne procesory

Skalárne procesory vydávajú a vykonávajú inštrukcie oddelene - ďalšia inštrukcia vstúpi do procesoru až po tom, ako je predošlá inštrukcia úplne dokončená. Inštrukcie typicky manipulujú s jediným až dvoma operandami. V porovnaní s nižšie uvedenými prístupmi nie je tento typ procesorov extrémne výkonný, avšak jeho jednoduchosť umožňuje veľmi lacnú implementáciu. Tento prístup je tiež vhodný na demonštračné a edukačné účely. Diagram znázorňujúci skalárne vydávanie inštrukcií je možné vidieť na obrázku 2.7.

| Čas → | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $i_1$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ | | | | | | | | |
| $i_2$ | | | | | $S_1$ | $S_2$ | $S_3$ | $S_4$ | | | | |
| $i_3$ | | | | | | | | | $S_1$ | $S_2$ | $S_3$ | $S_4$ |

Obrázek 2.7: Skalárne spracovanie inštrukcií. Zdroj [5].

#### Procesory so zreťazeným spracovávaním inštrukcií

Ide o procesory, ktoré zavádzajú koncept zreťazeného spracovania inštrukcií. Využívajú skutočnosť, že každú inštrukciu je možné rozdeliť na fázy (napr. načítanie, dekódovanie, vykonanie a uloženie výsledku). Predpokladom je, že jednotlivé fázy sú realizované oddelenými časťami procesoru a sú na sebe nezávislé. Okrem toho by malo platiť, že jednotlivé časti zreťazenej linky trvajú približne rovnaký čas. Na úrovni inštrukcií sa však závislosti vyskytnúť môžu. Stane sa tak napríklad v prípade, že z dvoch po sebe idúcich inštrukcií, druhá môže potrebovať výsledok tej prvej, ktorý však ešte nemusí byť dostupný. Kvôli

takýmto problémom je nutné spolu so zreťazenou linkou zabezpečiť riešenie takýchto konfliktov. Tento problém sa v praxi rieši dvoma spôsobmi:

- staticky - pred spustením programu, vkladaním prázdnych taktov, napríklad kompilátorom,
- dynamicky - počas vykonávania programu, napríklad premenovávaním registrov.

| Čas → | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $i_1$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ | | | | | | |
| $i_2$ | | $S_1$ | $S_2$ | $S_3$ | $S_4$ | - | | | | |
| $i_3$ | | | $S_1$ | $S_2$ | x | $S_3$ | $S_4$ | | | |
| $i_4$ | | | | $S_1$ | x | $S_2$ | $S_3$ | $S_4$ | | |
| $i_5$ | | | | | x | $S_1$ | $S_2$ | $S_3$ | $S_4$ | |
| $i_6$ | | | | | | | $S_1$ | $S_2$ | $S_3$ | $S_4$ |

Obrázek 2.8: Diagram zreťazeného spracovania inštrukcií. Zdroj [5].

V porovnaní so skalárnymi procesormi zreťazené procesory síce neurýchľujú dobu vykonania jednej inštrukcie, avšak poskytujú vyššiu priepustnosť inštrukcií. Jednoduchá zreťazená linka procesoru je zobrazená na obrázku 2.8.

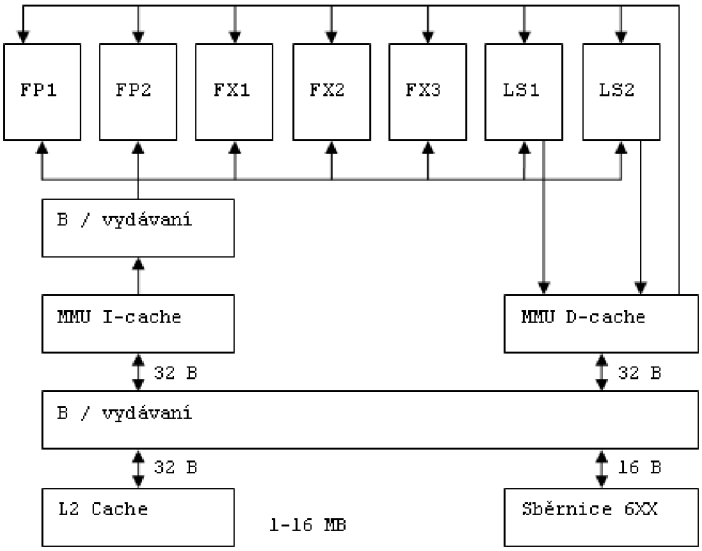
### Superskalárne procesory

Superskalárne procesory vznikli ako výsledok snahy zvýšiť výkonnosť procesoru pomocou zvýšenia počtu funkčných jednotiek. Tým bolo umožnené vydávať a vykonávať niekoľko inštrukcií súčasne (v jednom takte). Spolu s tým však vznikli konflikty a závislosti medzi inštrukciami. Riešenie týchto problémov bolo implementované do samotného procesoru v podobe niekoľkých mechanizmov, napríklad:

- predbežné dekódovanie inštrukcií - inštrukcie sú doplnené o dodatočné informácie o ich type a operandoch,
- zmena poradia vydávania inštrukcií - pomocou zavedenia rôznych vyrovnávajúcich pamätí,
- predikcia skokov a špekulatívne spracovanie inštrukcií - utilizovalo nevyužité funkčné jednotky.

Zabezpečenie stáleho prísunu inštrukcií tak, aby bolo dosiahnuté maximálne využitie funkčných jednotiek, je zložitá úloha. Je to však predpoklad efektívnosti superskalárneho procesoru.

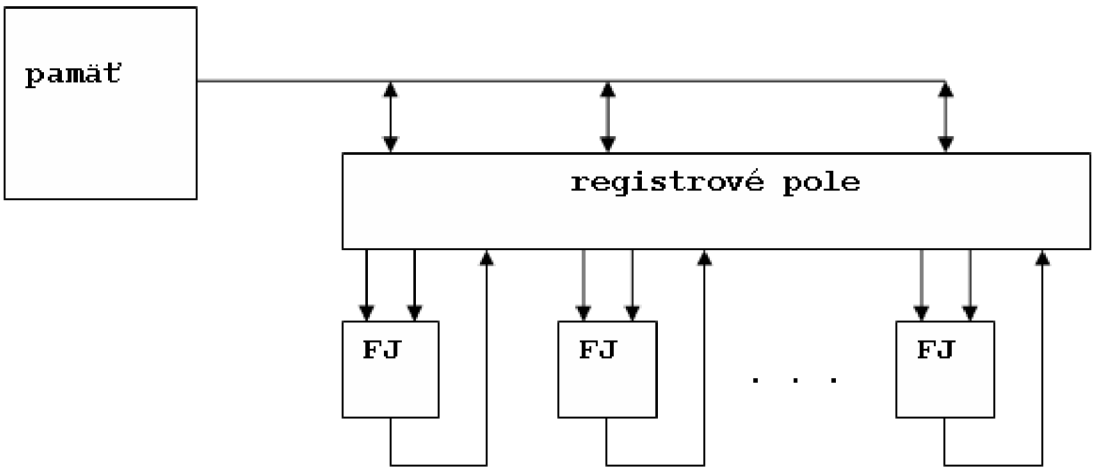
Obrázok 2.9 zobrazuje schému superskalárneho procesoru Power3.




Obrázek 2.9: Blokový diagram porocesoru Power3. Zdroj [5].

### Procesory s veľmi dlhým inštrukčným slovom

Podobne ako superskalárne procesory, sú aj procesory s veľmi dlhým inštrukčným slovom založené na multiplikácii funkčných jednotiek. Odlišujú sa však v spôsobe plánovania a vydávania inštrukcií. V tomto prípade je zodpovednosť za plánovanie inštrukcií delegovaná na prekladač vyššieho programovacieho jazyka. Ten môže analyzovať značne väčší počet inštrukcií než hardvérové riešenie superskalárnych procesorov. Prekladač teda poskladá naplánované inštrukcie do jediného slova, ktoré je dlhé rádovo stovky bitov. Toto inštrukčné slovo je potom jednoducho spracované procesorom, čím odpadá zložitý proces dekódovania. Implementovať prekladač vyššieho jazyka tak, aby dokázal efektívne využiť celé vybavenie VLIW (angl. very long instruction word) procesoru je veľmi náročné. Navyše, inovácia hardvéru typicky znamená zmenu medzi-inštrukčnej latencie a môže sa stať, že algoritmy prekladaču nebudú pre nový hardvér pracovať správne. Obrázok 2.10 znázorňuje všeobecnú schému procesoru s veľmi dlhým inštrukčným slovom.




Obrázek 2.10: Generická schéma VLIW procesoru. Zdroj [3].

#### Viacvláknové procesory

Okrem zreťazenia vykonávania inštrukcií, niektoré architektúry obsahujú aj iný druh paralelizmu - paralelizmus na úrovni vlákien. V [5] je vlákno definované ako „postupnosť inštrukcií videná z hľadiska ich vykonávania, vyžadujúca určitý adresný priestor a čas CPU“. Spolu s inštrukciami je každé vlákno popísané množinou atribútov, ktoré bývajú spravidla implementačne závislé. Procesor môže v danom momente vykonávať práve jedno vlákno. Ostatné vlákna čakajú, kým bežiace vlákno skončí alebo je prerušené a procesor vyberie a spustí nejaké iné vlákno. Viacvláknové procesory obsahujú hardvérovú podporu pre prepínanie vlákien. Najčastejšie sú to rýchle lokálne pamäte obsahujúce informácie o rozpracovaných vláknach. Efektivita tohto typu architektúry sa preukáže v prípade, že o zdroje procesoru súperí viacero vlákien. Kým jedno vlákno čaká na nejakú udalosť, napríklad načítanie dát z nejakej pomalšej pamäte, môže procesor vykonávať vlákno iné. Celková doba vykonávania jediného vlákna však skrátená nie je. Naopak, táto doba sa môže v konečnom dôsledku skôr predĺžiť, a to kvôli prítomnej réžii - aj v prípade, ak na procesore beží iba jediné vlákno.

### 2.2.5 Špecializácia na sadu úloh

Doteraz boli uvažované výhradne univerzálne procesory. Existujú však aj procesory špecializované. Rozdiel medzi univerzálnym a špecializovaným procesorom je v [5] definovaný tzv. kritériom algoritmickej zložitosti, ktoré považuje procesor za špecializovaný ak je štruktúra procesoru ovplyvnená počítaným algoritmom.

#### Vektorové procesory

Vektorové procesory sú prispôsobené práci s homogénnymi dátovými štruktúrami ako sú napríklad polia alebo matice. Obsahujú hardvér schopný jednorázovo načítať, spracovať a uložiť celý vektorový objekt. Benefitov tejto funkcionality je niekoľko:

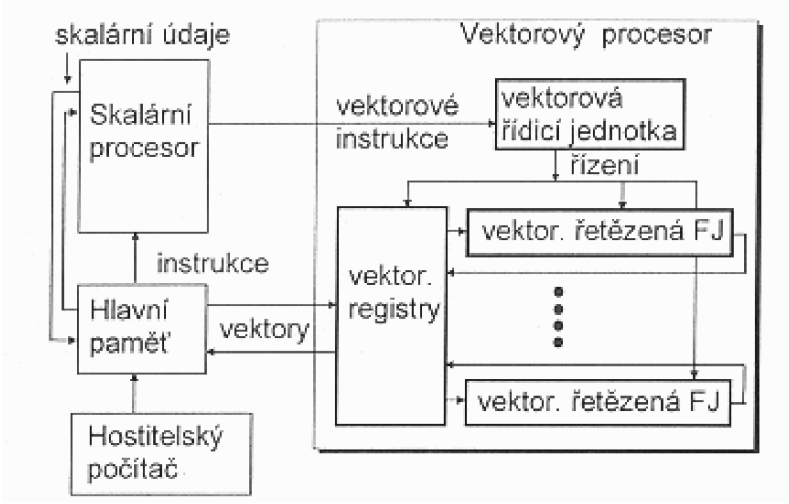
- je odstránená réžia cyklov, ktorá by bola nutná pri sekvenčnom spracovaní,
- latencia pamäte sa prejaví iba jeden krát za celý objekt,
- rovnaké operácie sa dajú ľahko zreťaziť,
- úlohy počítané vektorovými procesormi často bývajú implicitne definované pomocou vektorových objektov.

Nevýhodou vektorových procesorov je, že nie je možné vektorizovať celý program, ale iba jeho časť. Na iné, než vektorové úlohy sú tieto procesory prakticky nepoužiteľné. Obrázok 2.11 zachytáva obecnú schému vektorového procesoru.

#### Grafické procesory

Grafické procesory sa väčšinou používajú ako doplnkové procesory k bežným procesorom a sú na ne delegované výpočty, ktoré nie je efektívne počítať na bežnom procesore. Grafické procesory prešli značným vývojom:

- do prvej generácie grafických procesorov patrili v podstate bežné procesory so silnejšou podporou operácií s plávajúcou desatinnou čiarkou,
- druhá generácia priniesla presun niektorých operácií priamo do hardvéru procesoru,




Obrázek 2.11: Blokový diagram jarda vektorového procesoru. Zdroj [5].

- moderné grafické procesory sú považované za tretiu generáciu procesorov. Sú to masívne paralelné zariadenia, ktoré obsahujú množstvo výpočtových jadier. Okrem toho obsahujú sofistikovanú hierarchiu pamätí optimalizovanú vzhľadom na princípy paralelných výpočtov. V súčasnosti sa výkon dedikovaných grafických procesorov využíva okrem grafických výpočtov aj na množstvo iných, všeobecných úloh paralelného charakteru. Príkladom budiš lámanie hesiel spôsobom hrubej sily, napríklad [9].

### Signálové procesory

Signálové procesory sú procesory optimalizované na číslicové spracovanie signálov. Charakter algoritmov číslicového spracovania signálov sa odzrkadľuje v architektúre signálových procesorov:

- časté prístupy do pamäte - silná podpora priamych prístupov do pamäte, modulové adresovanie, hierarchia vyrovnávajúcich pamätí,
- dedikované funkčné jednotky - implementujú operácie, ktoré by bolo drahé vykonať softvérovo,
- špeciálne registre - napríklad pre počítanie cyklov, čím odpadá softvérová réžia, alebo akumulátory výhodné pre najčastejšie operácie,
- periférie - napríklad AD/DA prevodníky.

Vďaka svojej architektúre dokážu signálové procesory dosahovať pri vykonávaní daných úloh lepšie výsledky ako bežné procesory. Preto bývajú používané v aplikáciách, kde je kladený dôraz na rýchlosť odozvy, nízky príkon alebo malý rozmer výsledného riešenia.

# Kapitola 3

# Projekt Lissom

Úlohou tejto kapitoly je predstaviť projekt Lissom a jeho súčasti. Dôraz bude kladený na zoznámenie s jazykom ISAC, pretože ten bude hlavným vyjadrovacím prostriedkom pri implementácii modelov zvolených architektúr. Okrem toho bude predstavené aj grafické užívateľské rozhranie projektu Lissom, pretože jeho zvládnutie prispeje k efektívnosti procesu implementácie.

Projekt Lissom bol odštartovaný v roku 2003 na Fakulte informačných technológií Vysokého učení technického v Brne. Projekt je zameraný na výskum a vývoj nástrojov pre súbežný návrh softwarového a hardwarového vybavenia počítačov. Jeho ciele je možné zhrnúť do nasledovných bodov:

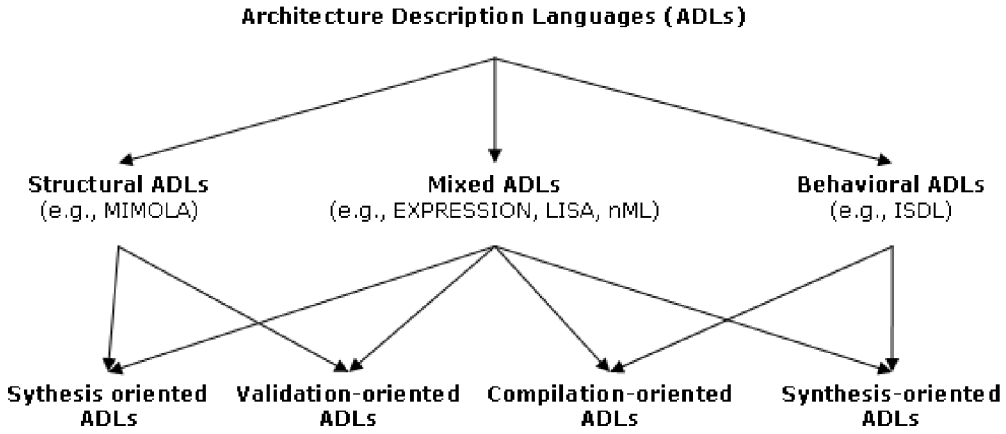
- návrh jazyka pre popis architektúry počítačového systému,
- výskum metód na zjednodušenie popisu architektúry,
- automatizované generovanie programového vybavenia pre popisovanú architektúru,
- získanie syntetizovateľného popisu počítačového systému v jazyku pre popis hardwaru.

V súčasnosti sa projekt nachádza v pokročilom štádiu a boli dosiahnuté určité výsledky. Bol navrhnutý jazyk ISAC slúžiaci na popis architektúr počítačových systémov. Rovnako sú k dispozícii generátory programového vybavenia a popisu hardwaru pre popisovanú architektúru. Výnimkou je prekladač jazyka C, kde je väčšina výskumu smerovaná k vyriešeniu problému extrakcie sémantiky inštrukcií popisovanej architektúry. Ďalším otvoreným problémom je otázka optimalizácie popisu architektúry, pretože súčasné riešenie je prakticky nepoužiteľné pre popis komplikovanejších architektúr.

## 3.1 Jazyky pre popis architektúry

Pred tým, ako sa bude možné zoznámiť s jazykom ISAC je nutné spoznať prostredie jazykov, v ktorom sa nachádza.

Podľa [4], jazyky pre popis architektúry (angl. architecture descritpion languages, ADLs) sú jazyky, ktoré zachytávajú špecifiká softwarových alebo hardwarových architektúr. Špecifikami softwarových architektúr sa rozumie správanie komponentov, z ktorých je architektúra zložená, a tiež ich vzájomné interakcie. U hardwarových architektúr býva zachytená štruktúra komponentov tvoriacich danú architektúru, ich vzájomné prepojenie a správanie




Obrázek 3.1: Taxonómia ADL jazykov. Zdroj [4].

v podobe inštrukčnej sady. V tejto práci budeme pod pojom ADLs chápať práve jazyky pre popis hardwarových architektúr.

Tieto ADLs je ďalej možné klasifikovať podľa viacerých kritérií, avšak pre túto prácu je dostatočné uvažovať klasifikáciu na základe obsahu a povahe informácii obsiahnutých v danom popise architektúry. Potom je možné identifikovať tri druhy jazykov:

- štrukturálne jazyky, ktoré sa sústredia na zachytenie štruktúry architektúry v zmysle použitých komponentov a ich prepojenia. Jazyky tohto typu umožňujú popis na relatívne nízkej úrovni abstrakcie,
- behaviorálne jazyky, ktorá zachytávajú správanie architektúry reprezentované množinou inštrukcií. Tieto jazyky pracujú s vyššou úrovňou abstrakcie, pretože užívateľ nemá typicky možnosť ovplyvniť štruktúru architektúry,
- zmiešané jazyky, ktoré obsahujú prvky oboch vyššie spomenutých prístupov.

Na obrázku 3.1 je zobrazená taxonómia ADLs spolu s najbežnejším použitím jednotlivých typov jazykov.

## 3.2 Jazyk ISAC

V terminológii projektu Lissom sa proces popisu architektúry nazýva modelovanie a samotný popis architektúry sa nazýva model. Ak nebude uvedené ináč, v ďalšom texte bude používaná táto terminológia.

Motiváciou pre vznik jazyka ISAC bola snaha o vytvorenie prostriedku, pomocou ktorého by bolo možné jednotne reprezentovať modelované architektúry. Táto reprezentácia potom môže byť použitá ako vstup generátorov nástrojov pre danú architektúru. Spracovaním reprezentácie modelov sa venuje ďalšia sekcia.

Ak uvážime v predošlej kapitole popísanú klasifikáciu, jazyk ISAC je možné zaradiť do kategórie zmiešané jazyky. Tento jazyk umožňuje popis štrukturálnych detailov architektúr, avšak zároveň je možné špecifikovať behaviorálne vlastnosti pomocou popisu inštrukčnej sady danej architektúry

Táto práca používa práve jazyk ISAC pre popis architektúr tvoriacich základ knižnice pre návrh vnorených systémov. Je teda potrebné v krátkosti predstaviť tento jazyk. Podrobnosti a ďalšie informácie je však možné nájsť napríklad v [7].

### 3.2.1 Úrovne popisu

Vlastnosti architektúry, ktoré je jazyk ISAC dovoľuje zachytiť, je podľa [12] možné rozdeliť do týchto kategórií:

- informácie slúžiace na vystavanie modelu inštrukčnej sady vo forme atribútovej gramatiky,
- informácie o časovaní architektúry a hierarchie inštrukčných analyzátorov,
- informácie zachytávajúce chovanie jednotlivých stavebných blokov modelovanej architektúry,
- štrukturálne informácie popisujúce prepojenie jednotlivých stavebných blokov architektúry.

Prvá z kategórií je výsledkom predošlého výskumu projektu Lissom. Zvyšné tri kategórie boli do jazyka doplnené ako súčasť [12].

Podľa zastúpenia týchto kategórií v rámci modelu je možné identifikovať dva druhy modelov, ktoré je možné pomocou jazyka ISAC vytvoriť. Sú to model architektúry na úrovni jej inštrukcií a model architektúry na úrovni hodinových cyklov.

#### Model na úrovni inštrukčnej sady

Ak model obsahuje prevažne informácie o inštrukčnej sade architektúry, potom hovoríme o modeli ako o modeli inštrukčnej sady. U takýchto modelov nie je kladený dôraz na časovanie a štruktúru architektúry. Modelár je tak oprostený od niektorých problémov spojených s popisom týchto aspektov.

Modely tohto typu sú použiteľné pre generovanie programového vybavenia pre modelovanú architektúru, a teda asembleru, spätného asembleru a simulátoru. Výstup ostatných generátorov je kvôli nedostatku informácii na vstupe zvyčajne nepoužiteľný z dôvodu svojej neoptimálnosti.

Inštrukčná sada ja definovaná dvoma typmi jazykov - jazykom asembleru a jazykom strojového jazyka. Jazyk asembleru popisuje textovú syntax inštrukcií a strojový jazyk ich binárne kódovanie. Model inštrukčnej sady musí obsahovať informácie popisujúce oba tieto jazyky.

Simulácia takéhoto modelu je potom zameraná na analýzu inštrukčnej sady. Počas simulácie je síce zachovaný koncept hodinových cyklov, avšak každá inštrukcia implicitne trvá práve jeden hodinový takt.

#### Model na úrovni hodinových cyklov

Okrem informácii o inštrukčnej sade je do modelu možné zahrnúť aj informácie o časovaní a štruktúre funkčných blokov modelovanej architektúry. Pomocou takýchto informácii je možné definovať správanie architektúry v každom hodinovom takte a pre každú dekódovanú inštrukciu.

Oproti popisu inštrukčnej sady je modelovanie časovania a štruktúry mierne zložitejší proces. Poskytuje však možnosť modelovať komplexnejšie prvky architektúry:

```
RESOURCE {
    /* registre */

    // programový čítač
    PC REGISTER bit[32] prog_cnt;

    // akumulátor
    REGISTER bit[32] acc;

    /* pamäte */

    // RAM pamäť
    RAM bit[8] prog_mem {
        ENDIANNES (BIG); // LSB je v˜pravo
        LAU(8);
        SIZE (256);
        FLAGS (R,W,X); // povolené čítanie, zápis, spúšťanie
    };

    /* organizácia pamäťového priestoru */

    MEMORY_MAP defaultmap {
        RANGE (0,255) -> ram[(8..0)];
    };
}
```

Obrázek 3.2: Príklad popisu zdrojov v jazyku ISAC.

- hierarchia inštrukčných analizátorov - architektúra môže podporovať niekoľko inštrukčných sád, ktoré môžu byť aktívne súčasne, alebo je ich aktivita podmienená stavom procesoru,
- špecifikácia časovania - definuje ktoré štruktúrne bloky procesoru majú byť aktívne v danom hodinovom takte,
- zreťazená linka - prúdové spracovanie inštrukcií je prítomne v takmer každej architektúre, a preto je výhodné umožniť jeho jednoduché modelovanie.

Funkčný blok zodpovedný za riadenie aktivity procesoru sa zvyčajne nazýva *radič* a je jedným z najkomplexnejších funkčných blokov celého procesoru. Jazyk ISAC umožňuje pracovať s architektúrou na vyššej úrovni abstrakcie, čo je výhodné najmä počas návrhu radiča. Modelárovi vďaka tomu mizne nutnosť opakovane riešiť rutinné problémy, a to hlavne vďaka automatickému generovaniu implementácie radiču. Ďalšou výhodou je možnosť simulácie architektúry na úrovni cyklov, ktorá sa blíži skutočnej funkcionalite hardvéru a umožňuje modelárovi pomerne presne analyzovať parametre modelovanej architektúry.

### 3.2.2 Štruktúra modelu

Každý model nejakej architektúry je zložený z dvoch častí - popisu zdrojov a množiny operácií.

#### Popis zdrojov

Popis zdrojov je realizovaný prostredníctvom tzv. RESOURCE sekcie. V tejto sekcii sú deklarované hardwarové prostriedky, ktoré má architektúra k dispozícií. Uvádzajú sa tu registre (vrátane čítača inštrukcií), pamäte, organizácia pamäťového priestoru a iné.

Obrázok 3.2 ukazuje príklad sekcie zdrojov. Ako prvé sú deklarované 32-bitové registre programový čítač *prog_cnt* a akumulátor *acc*. Ďalej je deklarovaná RAM pamäť *prog_mem*, ktorá obsahuje 256 8-bitových adresovateľných jednotiek. Okrem toho sú pre túto pamäť deklarované atribúty špecifikujúce pozície významných (LSB a MSB) bitov a príznaky indikujúce povolené operácie s dátami v pamäti. Na záver je deklarované rozdelenie adresného priestoru, kde sa na prvých 256 adries rozsahu poskytnutého programovým čítačom mapuje práve pamäť *prog_mem*.

### Množina Operácii

Operácie sú základnými stavebnými blokmi modelu. Jedna operácia typicky modeluje nejaký atomický dej alebo element. Atomický dej môže byť napríklad príchod nového hodinového cyklu alebo fáza zreťazenej linky spracovania inštrukcií. Element môže byť napríklad inštrukcia, alebo jej časť.

Operácie sú zložené zo sekcií. Aké sekcie daná operácia obsahuje je ovplyvnené jej účelom. Najdôležitejšie sekcie sú

- sekcia ASSEMBLER - táto sekcia býva prítomná v operáciách, ktoré reprezentujú inštrukcie alebo ich fragmenty. Informácie obsiahnuté v tejto sekcii definujú textovú podobu inštrukcií modelovanej architektúry a sú použité ako hlavný vstup pre generátor asembleru,
- sekcia CODING - ide o symetrickú sekciu k sekcii ASSEMBLER, pretože definuje spôsob binárnej reprezentácie inštrukcií. Berie sa v úvahu hlavne pri generovaní spätného asembleru,
- sekcia BEHAVIOR - v tejto sekcii je popísané faktické správanie operácie. V prípade, že operácia reprezentuje inštrukciu, je tu popísaná sémantika inštrukcie. Ak však operácia reprezentuje nejaký dej, sú tu obsiahnuté všetky akcie, ktoré je treba vykonať. K popisu správania je použitá obmedzená podmnožina jazyka C,
- sekcie CODINGROOT a STRUCTURE - sú zvláštnym sekciami a môžu sa vyskytovať iba v jedinej operácii v rámci modelu. Pomocou týchto operácii je špecifikované, akým spôsobom má architektúra pristupovať ku kódovaniu, dekódovaniu a vykonávaniu inštrukcií,
- sekcia EXPRESSION - v tejto sekcii je možné uviesť výraz jazyka C, ktorý bude použitý v behaviorálnych sekciách inštanciujúcich danú operáciu,
- konštrukcia INSTANCE, ktorá je používaná na sprístupnenie už definovaných operácii a skupín v aktuálne tvorenej operácií.

Operácie môžu obsahovať ešte niekoľko ďalších konštrukcií. Ich podrobný popis je možné nájsť v už spomenutom manuále jazyka ISAC.

Na obrázku 3.3 je vidieť príklad operácie reprezentujúcej nejakú inštrukciu. Ako prvé je inštancovaná operácia *value* reprezentujúca priamu hodnotu. Následne je v sekcii ASSEMBLER definovaná textová podoba inštrukcie s využitím inštancovanej operácie. Inštrukcia začína reťazcom *ADD*, ten je nasledovaný reprezentáciou akumulátoru, ktorý je čiarkou oddelený od pripočítavanej hodnoty. Potom je definované binárne kódovanie operácie, ktoré začína operačným kódom inštrukcie, je nasledované kódovaním cieľového registru a na záver obsahuje zakódovanú pripočítavanú hodnotu. Napokon operácia obsahuje behaviorálnu

```
OPERATION inst_add2acc {

    /* inštancia inej operácie */
    INSTANCE value;

    ASSEMBLER { "ADD" "A" "," value };
    CODING { 0b0110 0b0000 value };

    BEHAVIOR {
        acc += value; // pripočíta hodnotu value k~acc
    };
}
```

Obrázek 3.3: Príklad operácie v jazyku ISAC.

```
OPERATION main {

    /* koreňová operácia inštrukčnej sady */
    CODINGROOT {
        inst_add2acc
    };

    /* načítanie inštrukie na adrese denej registrom pc */
    STRUCTURE {
        prog_mem(pc);
    };

    BEHAVIOR {
        printf("Vykonavam cyklus %d\n",pc);
        pc++; // inkrementuje programový čítač
    };
}
```

Obrázek 3.4: Príklad operácie *main* v jazyku ISAC.

sekciu, kde je definované, že inštrukcia má pripočítať hodnotu *value* k aktuálnej hodnote akumulátoru.

Okrem bežných operácii musí každý model obsahovať nasledujúce tri operácie:

- reset - obsahuje informácie o tom ako sa má architektúra správať po resete,
- main - definuje správanie architektúry v každom hodinovom cykle,
- halt - obsahuje správanie architektúry pri skončení jej práce.

Obrázok 3.4 ukazuje príklad operácie main. V sekcii CODINGROOT je definovaná koreňová operácie inštrukčnej sady. Sekcia STRUCTURE definuje, že inštrukcie sa majú načítavať z pamäte *prog_mem* na adrese danej obsahom programového čítača *pc*. Konečne, v sekcii BEHAVIOR je inkrementovaný programový čítač a v prípade simulácie je na štandardný výstup vypísaná informácia o vykonávaní nového hodinového cyklu.

Okrem sekcie zdrojov a jednotlivých operácií je v jazyku ISAC prítomná ešte konštrukcia skupín. Skupiny umožňujú združovať operácie a iné skupiny do väčších celkov. Tým je modelárovi poskytnutá možnosť zaviesť istú formu organizácie operácii a zefektívniť tak celý proces modelovania architektúry.

Na obrázku 3.5 je vidieť príklad skupiny reprezentujúcej skupinu inštrukcií manipulujúcich so zásobníkom nejakej fiktívnej architektúry. V skupine sú prítomné dve inštrukcie.

```
/* definícia operácie */
OPERATION inst_clrAcc { ... }

/* skupina akumulátorových inštrukcií */
GROUP acc_inst = inst_add2acc, inst_clrAcc;
```

Obrázek 3.5: Konštrukcia GROUP jazyka ISAC.

```
#define OP_ADD 0x0
#define OP_SUB 0x1

OPERATION imm4 { ASSEMBLER { imm=#U }; CODING { imm=0bx[4] }; EXPRESSION { imm; };
OPERATION imm8 { ASSEMBLER { imm=#U }; CODING { imm=0bx[8] }; EXPRESSION { imm; };
```

Obrázek 3.6: Definícia konštánt a operácií pre pirame hodnoty.

V obrázku 3.3 definovaná operácia reprezentujúca inštrukciu pre pričítania konštanty k akumulátoru a inštrukcia pre nulovanie akumulátoru.

### 3.2.3 Hierarchická organizácia operacií

Ako už bolo naznačené, jazyk ISAC poskytuje prostriedky pre zavedenie istej štruktúry do organizácie operácií. Táto funkcionalita je kľúčová pre efektívne tvorenie modelu architektúry, a preto je táto sekcia venovaná popisu takéhoto postupu pomocou demonštrácie na jednoduchom príklade.

Ako prvé sú na obrázku 3.6 definované operácie reprezentujúce číselné hodnoty, kde je konkrétna číselná hodnota prítomná v zdrojovom kóde uložená do identifikátoru *imm*, ktorý je prítomný v sekcii EXPRESSION daných operácií.

V druhom kroku je vytvorená operácia *regs* reprezentujúca všetky registre modelovanej architektúry. Operáciu je možné vidieť na obrázku 3.7 Pomocou konštrukcie INSTANCE je do operácie importovaná operácia pre štvorbitové priame hodnoty a s využitím konštrukcie ALIAS premenovaná na identifikátor *id*. Umiestnenie identifikátoru *id* v sekciách operácie regs má za následok vloženie pôvodných sekcií do práve definovanej sekcie. V sekcii ASSEMBLER sa tak za textový literál *„R“* konkatenuje literál reprezentujúci číslo. Kódovanie registru je prakticky totožné s kódovaním samotného čísla. Napokon, identifikátor inštancie čísla registru v sekcii EXPRESSION spôsobí propagáciu hodnoty do ďalšej operácie.

Ďalším krokom je deklarácia operácií, obrázok 3.8 reprezentujúcich operačné kódy inštrukcií pre sčítanie a odčítanie a ich následné zoskupenie do skupiny operations. EXPRESSION sekcie operácií obsahujú hodnoty jedinečne identifikujúce príslušné operácie.

Záverečná operácia *inst_arithm_reg_imm* zobrazená na obrázku 3.9 potom bude reprezentovať všetky aritmetické inštrukcie. Pomocou konštrukcie ALIAS je dva krát inštan-

```
OPERATION regs {
    INSTANCE imm4 ALIAS { id };
    ASSEMBLER { "R"~id };
    CODING { id };
    EXPRESSION { id };
}
```

Obrázek 3.7: Definícia operácie reprezentujúcej skupinu registrov.

```
OPERATION add { ASSEMBLER { "ADD" }; CODING { 0b00000000 }; EXPRESSION { OP_ADD; };
OPERATION sub { ASSEMBLER { "SUB" }; CODING { 0b11111111 }; EXPRESSION { OP_SUB; };
GROUP operations = add, sub;
```

Obrázek 3.8: Operácie reprezentujúce operačné kódy a textovú podobu inštrukcií.

```
OPERATION inst_arithm_reg_imm {
    INSTANCE regs ALIAS { reg_src, reg_dst };
    INSTANCE operations ALIAS { operation };
    INSTANCE imm8 ALIAS { imm };

    ASSEMBLER { operation reg_dst "," reg_src "," imm };
    CODING { operation imm reg_src reg_dst };
    BEHAVIOR {
        switch (operation) {
            case OP_ADD:
                reg_field[reg_dst] = reg_field[reg_src] + imm;
                break;
            case OP_SUB:
                reg_field[reg_dst] = reg_field[reg_src] - imm;
                break;
            default:
                break;
        }
    };
}
```

Obrázek 3.9: Finálna operácia reprezentujúca kompletné inštrukcie.

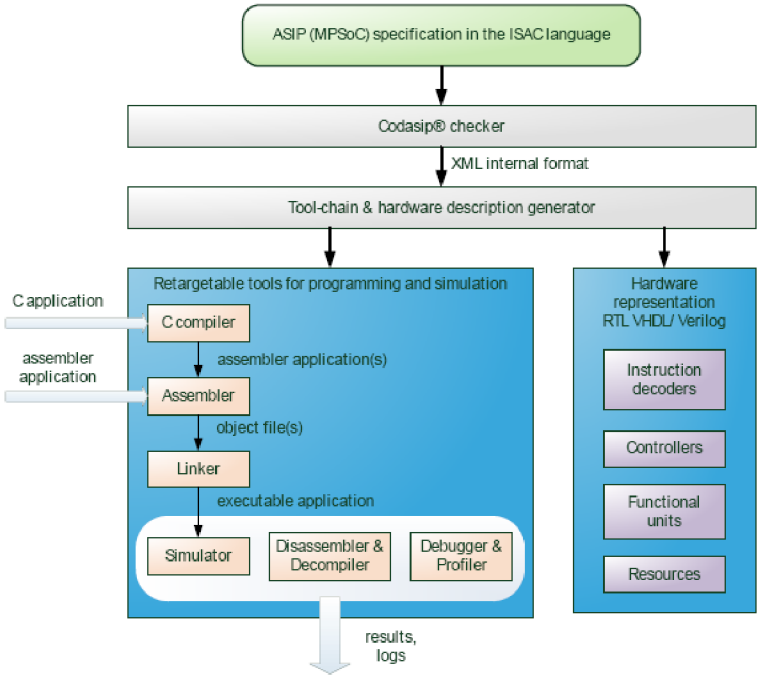
ciovaná operácia *regs*, čím vzniknú dva úplne nezávislé identifikátory reprezentujúce dva samostatné registre. Ďalej je inštanciovaná skupina operácií a operácia pre osembitovú priamu hodnotu. Pomocou sekcie ASSEMBLER je definovaná textová podoba inštrukcie, ktorá je zložená z názvu inštrukcie *ADD* alebo *SUB* a čiarkami oddelených zdrojových a cieľových operandov. Podobne je poskladané aj kódovanie inštrukcie. V behaviorálnej sekcii sú identifikátormi označené hodnoty výrazov v EXPRESSION sekciách inštanciovaných operácií. Je teda možné na základe identifikátoru operation jedinečne určiť o akú operáciu ide a následne túto operáciu vykonať. Samotné vykonanie spočíva v načítaní zdrojového operandu z registrového poľa, pričítaniu priamej hodnoty a uloženie hodnoty naspäť do registrového poľa. Indexy registrov sú získané z expression sekcie operácie regs, ktorá ako už bolo spomenuté je vlastne daná číslom registru.

## 3.3 Spracovanie popisu architektúry

Ďalším z výsledkov projektu Lissom sú postupy transformujúce popis architektúry na iné reprezentácie modelovanej architektúry. Celý systém je možné vidieť na obrázku 3.10. Model architektúry je konvertovaný do interného formátu umožňujúceho efektívnejšie strojové spracovanie. Potom je možné automaticky generovať softwarové nástroje a hardwarový reprezentáciu architektúry.

Softwarové nástroje, ktoré môžeme získať sú dvojitého typu:

- prekladače - jazyka C do jazyka symbolických inštrukcií modelovanej architektúry a asembleru, ktorý transformuje program symbolických inštrukcií do strojového kódu,




Obrázek 3.10: Systém srpacovania popisu architekúry pomocou nástrojov projektu Lissom. Zdroj [7].

- spustiteľné aplikácie - simulátory, reverzné prekladače a iné nástroje pre vývoj aplikácií implicitne optimalizovaných pre modelovanú architektúru.

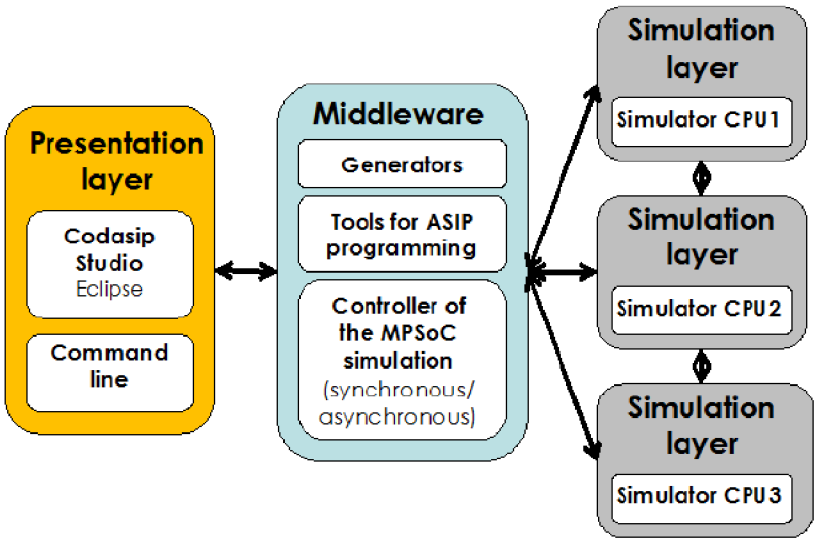
Hardwarová reprezentácia modelovanej architektúry, ktorú automaticky získame, umožňuje simuláciu architektúry nástrojmi typu ModelSim, syntézu architektúry za pomoci technológie FPGA alebo priamo vytvorenie samostatného integrovaného obvodu.

## 3.4 Vývojové prostredie projektu Lissom

Ďalším z výsledkov projektu je vytvorenie integrovaného vývojového prostredia, ktoré sprístupňuje užívateľom kompletnú funkcionalitu systému. Prostredie má vrstvovú architektúru a je znázornené na obrázku 3.11.

Užívateľ prichádza do styku s rozhraním v podobe príkazovej riadky alebo grafického rozhrania postaveného nad modulárnym prostredím Eclipse, ktoré sú označované ako prezentačná vrstva. Jadro systému je sústredene do strednej vrstvy, tzv. Middleware. V tejto vrstve sú obsiahnuté generátory nástrojov popísané v predošlej sekcii. Okrem toho je Middleware zodpovedný za správu simulácii a komunikáciu výsledkov naspäť do prezentačnej vrstvy. Treťou vrstvou je tzv. simulačná vrstva, ktorej jedinou úlohou je vykonávanie simulácie.

Vrstvový koncept umožňuje pomerne efektívnu implementáciu systému, pretože jednotlivé vrstvy môžu byť implementované na rôznych počítačoch. Viacero užívateľov teda môže pristupovať k jedinému Middlewaru, ktorý môže delegovať vykonávanie simulácii na ďalšie počítače. Samotné simulácie sú výpočtovo náročné a použitie dedikovaných zdrojov jednak šetrí výkon Middlewaru a jednak umožňuje zrýchlenie simulácie [15].




Obrázek 3.11: Vrstvová organizácia prostredia Lissom. Zdroj [8].

### 3.4.1 Grafické užívateľské rozhranie

Grafické užívateľské rozhranie je založené na vývojovom prostredí Eclipse a je distribuované ako zásuvný modul do tohto prostredia. Prakticky ide o nadstavbu nad rozhraním príkazovaj riadky.

Hlavnou úlohou grafického rozhrania je poskytnúť užívateľovi komfortný a prehľadný prístup k middleware. To je dosiahnuté prítomnosťou rozšírenia implicitného textového editoru o schopnosť zvýrazňovať syntax jazyka ISAC a funkcionalitou zabezpečujúcou automatické dopĺňanie užívateľských príkazov a prezentáciu odpovedí middlewaru.

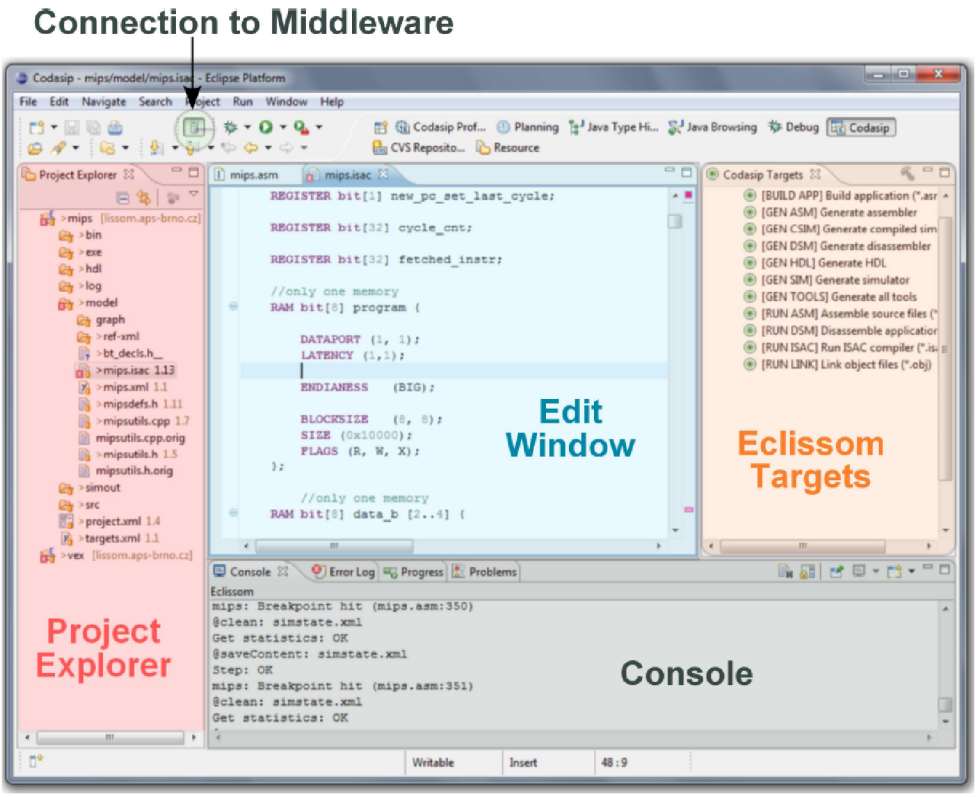
Inštalácia rozhrania je realizovaná štandardným postupom inštalácie zásuvného modulu od prostredia Eclipse. Inštalovaním modulu užívateľ získa možnosť použiť jednu z troch nových perspektív prostredia. Sú to:

- základná perspektíva pre vývoj modelov architektúr a k nim príbuzným softvérovým projektom,
- ladiaca perspektíva pre ladenie aplikácií pre modelované architektúry,
- profilovacia perspektíva určená na prezentáciu štatistických údajov získaných zo simulácie architektúry.

#### Vývojová perspektíva

Vývojová perspektíva je základnou perspektívou vývojového prostredia projektu Lissom. Obrázok 3.12 znázorňuje typické rozvrhnutie perspektívy, ktorá obsahuje

- okno s textovým editorom,
- výstup konzoly,
- prieskumník projektov,
- skupinu tzv. cieľov.




Obrázek 3.12: Základná perspektíva vývojového prostredia projektu Lissom. Zdroj [8].

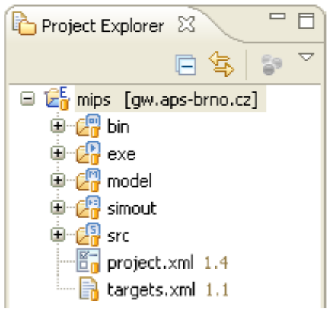
V okne s textovým editorom je možné editovať zdrojové kódy popisu architektúry s použitím už spomenutej podpory syntaxe jazyka ISAC.

Konzola je neinteraktívna a slúži na prezentovanie odpovedí middlewaru na užívateľské príkazy. Okrem toho sú sem presmerované výstupy (štandardný a chybový) aplikácie počas simulácie architektúry.

Ku každému projektu modelu architektúry je možné zobraziť množinu cieľov. V tomto prípade cieľ reprezentuje nejakú akciu alebo príkaz, ktorý má middleware automaticky vykonať. Medzi takéto akcie patrí napríklad kompilácia zdrojového súboru popisujúceho architektúru alebo spustenie generátorov niektorého z nástrojov. Najväčšou výhodou mechanizmu cieľov je, že užívateľovi postačí spustiť vykonávanie cieľu jednoduchým dvojklikom a grafické rozhranie sa samo postará o kompletizáciu príkazu o potrebné parametre, jeho odoslanie middlewaru a prezentáciu výsledku(vrátane uloženia výstupných súborov na ich príslušné miesta).

Posledným nepopísaným segmentom vývojovej perspektívy je prieskumník projektov. Tu sú umiestnené všetky aktuálne projekty modelov architektúr. Zdrojové súbory sú organizované spôsobom zobrazeným na obrázku 3.13. Štruktúra zdrojových súborov každého projektu je nasledovná:

- zložka bin obsahuje objektové súbory softvérových aplikácii pre danú architektúru,
- zložka exe obsahuje spustiteľné súbory softvérových aplikácii,
- zložka model obsahuje zdrojové súbory popisu architektúry, čo sú súbory jazyka ISAC a prípadné externé súbory obsahujúce funkcie alebo hlavičky jazyka C,




Obrázek 3.13: Adresárová štruktúra projektu modelu architektúry. Zdroj [8].

- zložka simout obsahuje súbory s uloženými záznamami simulácií,
- zložka src obsahuje zdrojové súbory softvérových projektov,
- súbor project.xml uchováva informácie o konfigurácii projektu,
- súbor target.xml uchováva konfiguráciu viditeľnosti jednotlivých cieľov projektu v ich príslušnom okne.

Príjemnou funkciou grafického rozhrania je poskytnutie špeciálneho rozhrania pre modifikáciu konfigurácie projektu, čím je užívateľ ušetrený od nutnosti ručne modifikovať príslušný xml súbor.

Ďalej sú ako súčasť zásuvného modulu do prostredia zabudované dialógy na konfigurácii zvyšných častí systému, ako sú napríklad údaje o pripojení k middlewaru alebo ciest k potrebným externým utilitám.

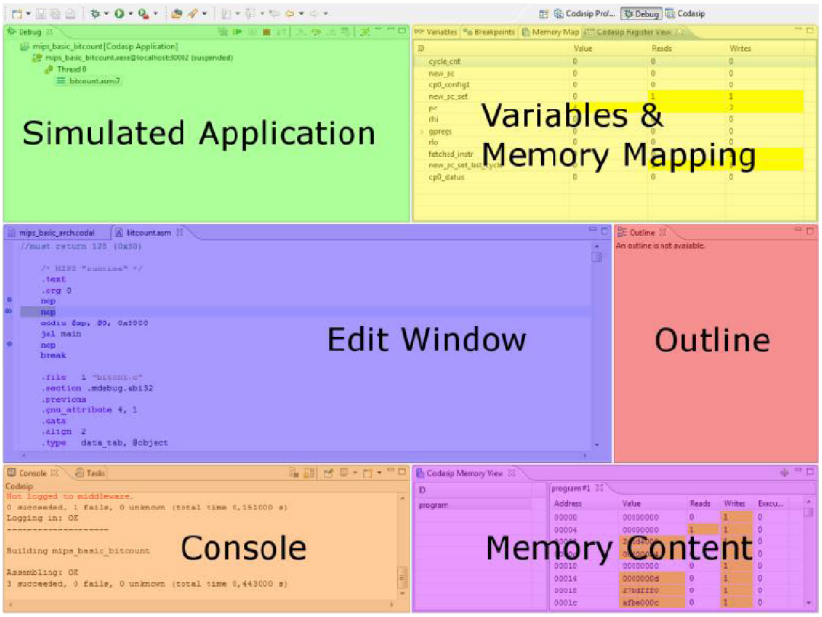
### Ladiaca perspektíva

Ďalšou funkcionalitou prítomnou v zásuvnom modeli projektu Lissom je možnosť simulovať softvérové aplikácie pre modelované architektúry. Podobne ako pre bežné aplikácie, aj v tomto prípade je zachovaná možnosť voľby dvoch typov spustenia:

- bežné spustenie - simulácia prebehne tak, že sú ignorované všetky breakpointy a simulácia sa zastaví až po dobehnutí programu,
- ladiace spustenie - vykonávanie programu rešpektuje breakpointy a simuláciu je možné kedykoľvek ukončiť.

V prípade ladiacej simulácie je možné po narazení na breakpoint prepnúť vývojové prostredie do ladiacej perspektívy, príklad ktorej je možné vidieť na obrázku 3.14. Užívateľovi sú prehľadne prezentované všetky pri ladení aplikácií bežne používané zdroje informácií:

- ovládacie prvky simulácie - umožňujú pokročiť v simulácii alebo prípadne simuláciu predčasne ukončiť,
- aktuálny obsah registrov a pamäte - je možné vytvoriť niekoľko náhľadov na rôzne pamäťové miesta, čím je umožnené pohodlné sledovanie viacerých buniek pamäte,
- zdrojový kód programu so zvýraznením aktuálne vykonávaného príkazu alebo inštrukcie a možnosti modifikovať polohu alebo aktívnosť breakpointov,
- konzola prezentujúca prípadný výstup aplikácie.




Obrázek 3.14: Ladiaca perspektíva prostredia Lissom. Zdroj [8].

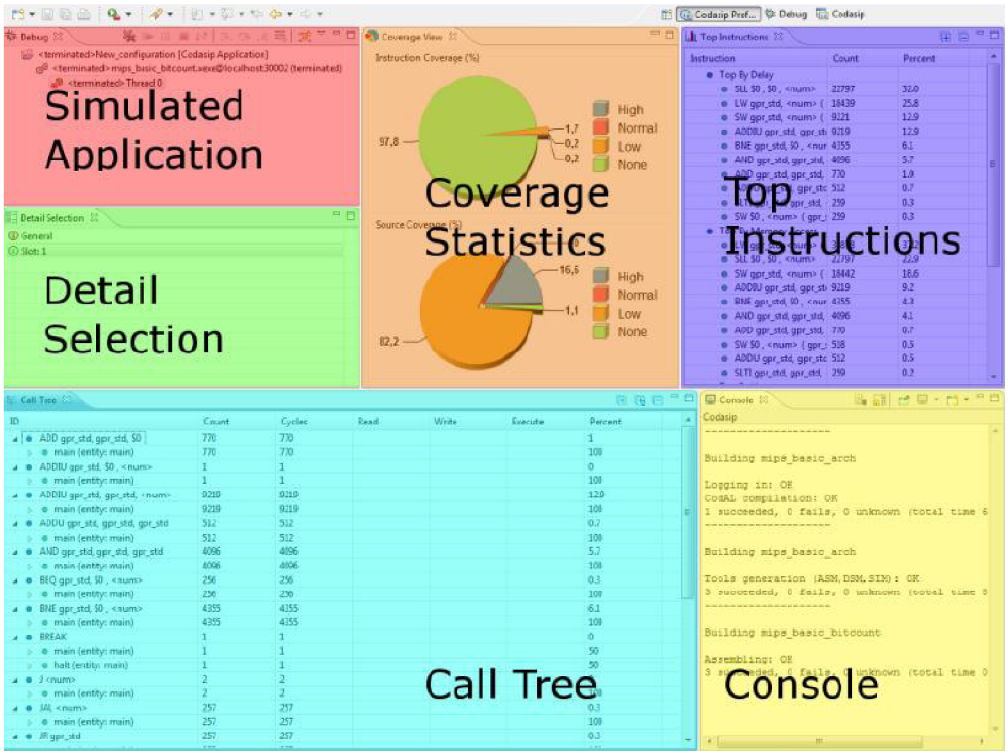
### Profilovacia perspektíva

Po dokončení simulácie je možné nechať zobraziť profilovaciu perspektívu. Príklad takejto perspektívy je na obrázku 3.15. Sú tu zobrazené štatistické údaje zozbierané počas behu simulácie:

- koláčové grafy znázorňujú percentuálne využitie inštrukčnej sady (teda ako často bola používaná aká časť inštrukcií) a ako bol pokrytý zdrojový súbor z hľadiska vykonávania jeho príkazov,
- hierarchiu vykonávania behaviorálnych sekcií operácií reprezentujúcich jednotlivé inštrukcie,
- inštrukcie zoradené podľa parametrov ako počet pamäťových prístupov alebo počtu taktov potrebných pre vykonanie inštrukcie.

V prípade, že architektúra obsahuje viacero inštrukčných dekodérov sú tieto informácie zbierané a prezentované pre každý dekodér zvlášť.

Hlavným prínosom profilovacej perspektívy je, že umožňuje analyzovať aplikácie pre architektúru skôr než je daná architektúra fyzicky realizovaná. Tým je umožnený efektívny vývoj aplikácií, ktorý môže prebiehať skutočne paralelne s vývojom samotnej architektúry. Dôsledkom takéhoto postupu potom môže byť celkové zníženie ceny vyvíjanej aplikácie.




Obrázek 3.15: Profilovacia perspektíva prostredia Lissom. Zdroj [8].

# Kapitola 4

# Modelované architektúry

Táto kapitola je venovaná predstaveniu v rámci tejto práce modelovaných architektúr. Každá architektúra bola zvolená ako reprezentatívny kandidát nejakej skupiny procesorov. Motivácia zvolenia tej ktorej architektúry je však v každom prípade odlišná, podobne ako je odlišné aj plánované použitie modelu.

## 4.1 Texas Instruments MSP430

Triedu univerzálnych mikrokontrolérov reprezentuje mikrokontrolér MSP430 firmy Texas Instruments.

Bol zvolený z dôvodu jeho prítomnosti na výukovom prípravku FITKit. Cieľom je vytvoriť model jadra mikrokontroléru tak, aby bolo možné preniesť binárny kód zostavený v prostredí nástrojov projektu Lissom do samotného mikrokontroléru a prakticky overiť správnosť modelu. Pôjde teda o vytvorenie modelu inštrukčnej sady procesoru.

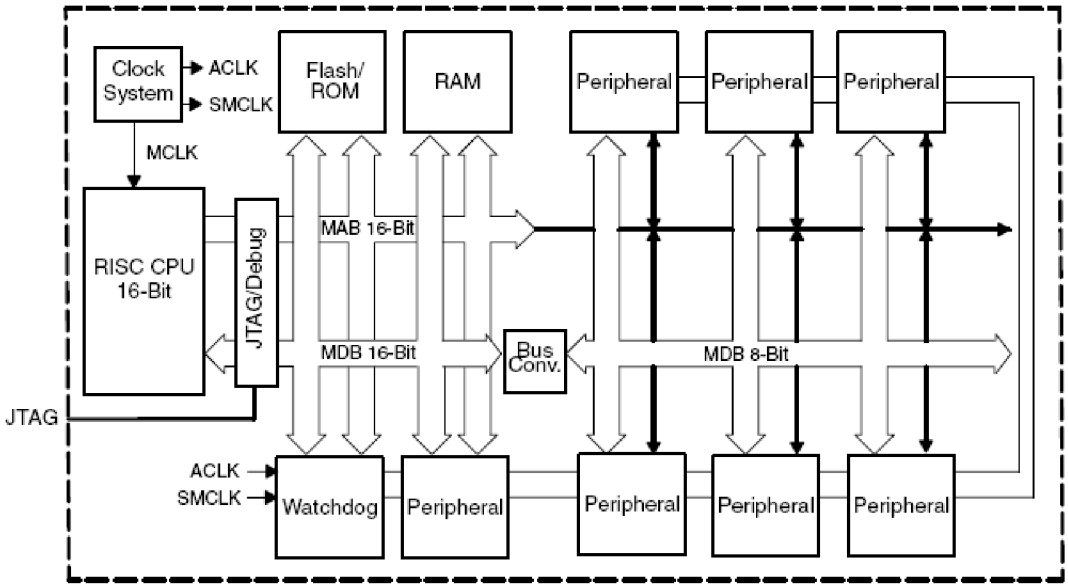
### 4.1.1 Základné vlastnosti

Jadro mikrokontroléru tvorí 16-bitový RISC procesor. Na čipe sú okrem toho prítomné rôzne periférie a hodinový systém. Mikrokontrolér používa von-Neumanovskú organizáciu pamäte a do adresného priestoru sú mapované všetky periférie. Ide o mikrokontrolér malých rozmerov s nízkym príkonom.

### 4.1.2 CPU

Ako už bolo uvedené, ide o 16-bitový RISC procesor, ktorý bol navrhnutý s ohľadom na moderné programovacie techniky a kompatibilitu s vyššími programovacími jazykmi. Základné vlastnosti tohto procesoru sú:

- šestnásť 16-bitových registrov - plne prístupné pre programátora, zahŕňajú aj systémové registre,
- inštrukcií,
- adresných módov,
- ortogonálna architektúra - každá inštrukcia môže byť použitá s každým adresným módom,




Obrázek 4.1: Architektúra mikrokontroléru MSP430. Zdroj [17]

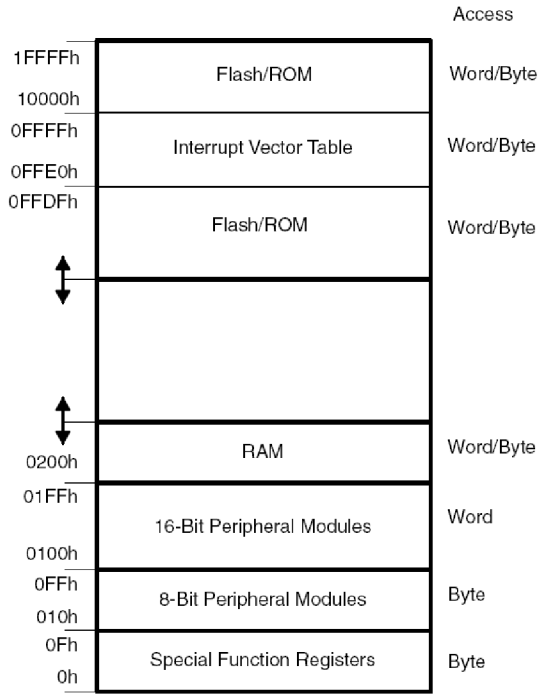
- možnosť adresovať 8 alebo 16 bitové operandy,
- prístup do celého adresného priestoru bez nutnosti stránkovania,
- bajtom sa rozumie 8-bitový element, slovom 16-bitový element.

Adresný priestor procesoru je znázornený na 4.2. Spodné adresy sú rezervované pre špeciálne registre a periférie. Potom nasledujú pamäte na všeobecné použitie a uloženie programu. Vrchné adresy sú vyhradené pre tabuľku prerušení. Adresy 10000h a vyššie sú dostupné v vyšších modeloch procesorov a nie sú v modeli uvažované. K pamäťovým bunkám je možné pristupovať dvomi spôsobmi:

- adresovanie po bajtoch - adresy z celého rozsahu (párne aj nepárne),
- adresovanie po slovách - môžu sa použiť len párne adresy.

Medzi špeciálne registre patria:

- programový čítač - prvý register registrového poľa. Vždy obsahuje párnu hodnotu, pretože inštrukcie sú vždy zarovnané vzhľadom na slovo (16 bitov),
- ukazateľ na zásobník - druhý register registrového poľa, slúži na ukladanie a obnovenie hodnoty programového čítača pri volaní a návrate z podprogramu. Požíva predecrement/postincrement schému a je inicializovaný programátorom,
- status register - tretí register registrového poľa, slúži na uchovanie stavu procesoru, vrátane príznakov,
- generátor konštánt - štvrtý register, slúži na automatické generovanie konštánt, čím odpadá nutnosť explicitne kódovať operand a znižuje sa tak veľkosť kódu programu,




Obrázek 4.2: Organizácia pamäte MSP430. Zdroj [17].

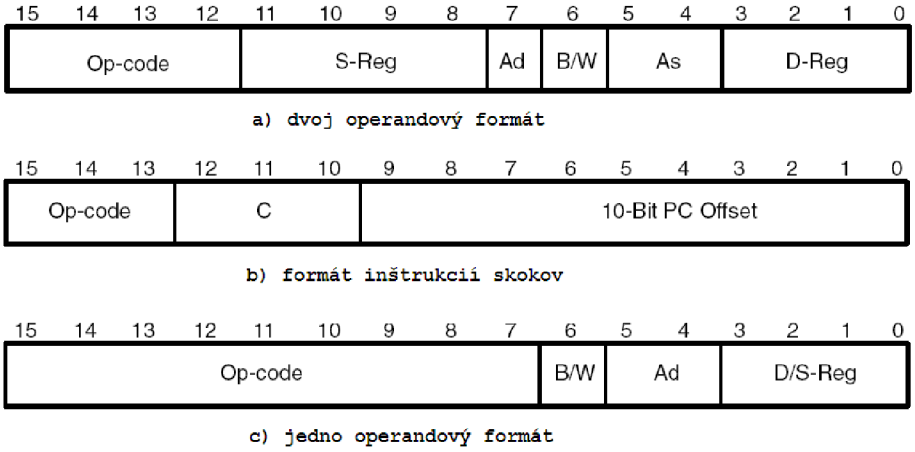
Medzi adresné módy procesoru patria:

- registrový mód - Rx - obsah registru x je považovaný za operand,
- indexový mód X(Rn) - pamäťová bunka s adresou Rn+X obsahuje operand. N je číslo registra, X je hodnota uložená v nasledujúcom slove,
- symbolický mód - ADDR - pamäťová bunka s adresou PC+ADDR obsahuje operand. ADDR je číslo uložené v ďalšom inštrukčnom slove,
- absolútny mód - &ADDR - slovo nasledujúce bezprostredne po inštrukcii obsahuje priamu adresu operandu,
- nepriamy registrový mód - @Rx - obsah registru x je chápaný ako adresa operandu,
- nepriamy autoinkrementačný mód - @Rx+ - obsah registru x je chápaný ako adresa operandu. Obsah registru x je automaticky inkrementovaný,
- priamy mód - #N - Slovo nasledujúce bezprostredne po inštrukcii obsahuje priamu hodnotu operandu.

### 4.1.3 Inštrukčná sada

Inštrukčná sada procesoru je tvorená 27 základnými inštrukciami, ktoré sú kódované jedným z troch možných formátov zobrazených v obrázku 4.3. Význam jednotlivých polí je nasledovný:

- Op-code - operačný kód inštrukcie,




Obrázek 4.3: Formáty inštrukcií proceosru MSP430. Zdroj [17].

| Emulovaná inštrukcia | Skutočná inštrukcia | Popis |
|---|---|---|
| ADC dst | ADDC #0, dst | Inštrukcia pripočítania hodnoty carry príznaku k dst je nahradená pripočítaním nuly k dst pri ktorom je zohľadnení príznak carry |
| NOP | MOV #0, R3 | Inštrukcia prázdnej operácie je nahradená za nahratie hodnoty nula do registru 3. Tento register má však špeciálnu funkciu a zápis do neho je ignorovaný. |

Obrázek 4.4: Príklad emulovanej inštrukcie.

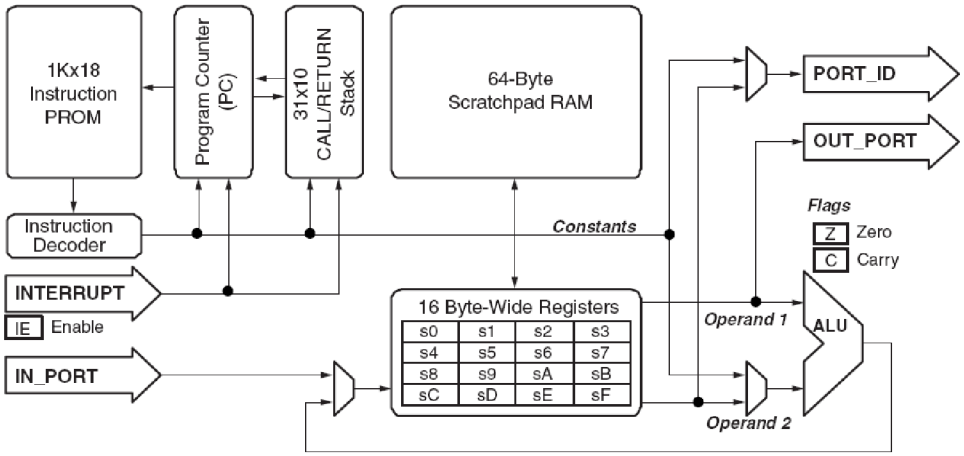
- S-Reg - číslo registru chápaného ako zdrojový operand,
- Ad - adresný mód cieľového operandu,
- B/W - informácia o veľkosti operandov,
- As - adresný mód zdrojového operandu,
- Rd - číslo registru chápaného ako cieľový operand,
- C - kódovanie príznaku, ktorý sa má použiť na rozhodovanie o vykonaní skoku.

Okrem toho do inštrukčnej sady patrí aj 24 emulovaných inštrukcií. Emulované inštrukcie slúžia na zlepšenie čitateľnosti a zjednodušenie tvorby programu. Ide o inštrukcie, ktoré nemajú vlastné kódovanie, ale sú mapované na niektorú zo základných inštrukcií. Príklad emulovanej inštrukcie je možné vidieť na obrázku 4.4.

## 4.2 Xilinx PicoBlaze

Ide o produkt firmy Xilinx, ktorá procesor navrhla ako podporu pre ich FPGA riešenia.

Procesor PicoBlaze reprezentuje skupinu soft procesorov. Softprocesorom je označovaný taký procesor, ktorý sa distribuuje v podobe zdrojových kódov. Rozhodnutie, akú technológiu využiť pre jeho vlastnú realizáciu, je ponechané na koncovom užívateľovi. Motiváciou




Obrázek 4.5: Bloková schéma procesoru PicoBlaze. Zdroj [18].

pre popis takéhoto procesoru v rámci tejto práce je zistiť, ako efektívne môže byť takýto procesor popísaný jazykom ISAC. Zaujímavé tiež bude porovnanie vygenerovaného popisu procesoru v jazyku VHDL s originálnym procesorom a prípadne tiež porovnanie výsledkov syntézy oboch popisov. Procesor sa teda budem snažiť popísať na úrovni jednotlivých hodinových cyklov.

### 4.2.1 Základné vlastnosti

PicoBlaze je 8-bitový RISC procesor, ktorý prináša lacné riešenie funkcionality mikrokontroléru a jednoduchých operácii spracovania dát. Napriek svojej jednoduchosti prináša pomerne vysoký výkon medzi 44 až 100 MIPS [2]. Procesor je možné úplne zabudovať do FPGA čipu a nevyžaduje žiadne externé zdroje.

Najdôležitejšie parametre procesoru sú:

- 16 jedno bajtových registrov,
- programový čítač ako dedikovaný register, ktorý nie je možné programovo modifikovať,
- programová pamäť schopná pojať 1024 inštrukcii, ktorá je naplnená počas inicializácie FPGA,
- 8-bitová aritmeticko-logická jednotka s príznakmi pretečenia a nuly,
- 64 bajtov internej pamäte na ukladanie medzivýsledkov,
- predvídateľný výkon, každá inštrukcia vždy trvá dva hodinové cykly,
- zásobník pre uloženie adries návratu z podprogramu,
- vstupné a výstupné porty pre jednoduchú konektivitu a rozšíriteľnosť.

Bloková schéma celého mikrokontroléru je zobrazená na obrázku 4.5.

Architektúra procesoru pracuje s piatimi adresnými priestormi. Každá inštrukcia implicitne pracuje s niektorým z nich a nie je potrebné ďalej špecifikovať požadovaný priestor. Dostupné adresné priestory sú:

| Instruction | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD sX,kk | opcode | | | | | | x | x | x | x | k | k | k | k | k | k | k | k |
| ADD sX,sY | opcode | | | | | | x | x | x | x | y | y | y | y | 0 | 0 | 0 | 0 |

| Instruction | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FETCH sX, ss | opcode | | | | | | x | x | x | x | 0 | 0 | s | s | s | s | s | s |
| FETCH sX,(sY) | opcode | | | | | | x | x | x | x | y | y | y | y | 0 | 0 | 0 | 0 |

| Instruction | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| INPUT sX,(sY) | opcode | | | | | | x | x | x | x | y | y | y | y | 0 | 0 | 0 | 0 |
| INPUT sX,pp | opcode | | | | | | x | x | x | x | p | p | p | p | p | p | p | p |

Obrázek 4.6: Niektoré formáty inštrukcií procesoru PicoBlaze. Zdroj [18].

- inštrukčný - priestor, kde je uložený program,
- registrové pole,
- scratchpad pamäť,
- vstupno výstupné porty,
- zásobník návratových inštrukcií.

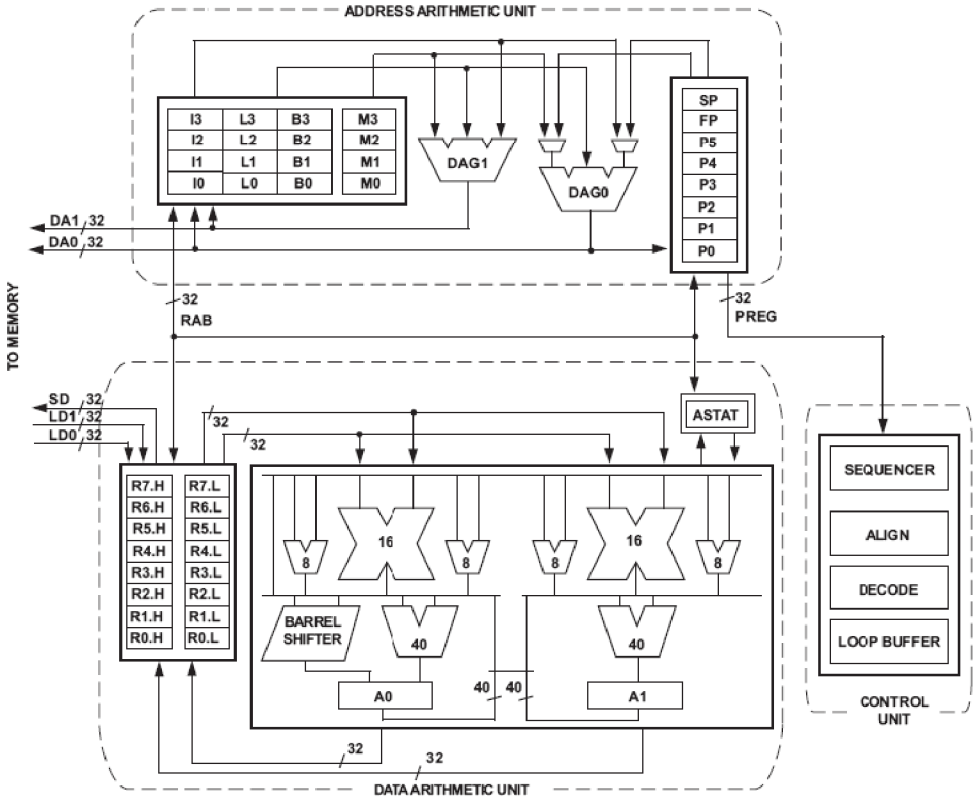
### Inštrukčná sada

Inštrukčná sada procesoru PicoBlaze pozostáva z 48 inštrukcií, ktoré sú v jednom zo šiestich formátov.

- aritmetické a logické inštrukcie,
- inštrukcie zmeny toku programu - skoky, volanie podprogramov a pod.,
- inštrukcie pamäťového prístupu,
- inštrukcie manipulujúce s vstupnými a výstupnými portami,
- inštrukcie rotácii a posunov,
- ostatné inštrukcie.

Formáty niektorých inštrukcií je možné vidieť na obrázku 4.6. Význam jednotlivých polí je nasledovný:

- Opcode - operačný kód inštrukcie,
- X - register sX,
- K - priama, konštantná hodnota,
- Y - register sY,
- A - adresa cieľu skoku alebo začiatku podprogramu,
- S - adresa do scratchpad RAM,
- P - adresa portu.




Obrázek 4.7: Blokový diagram jadra procesoru BlackFin. Zdroj [1].

## 4.3 Analog Devices BlackFin

Trieda signálových procesorov je v tejto práci reprezentovaná procesorom BlackFin spoločnosti Analog Devices. Ide o modernú architektúru, ktorá kombinuje prvky všeobecného mikrokontroléru s prvkami prispôsobenými číslicovému spracovaniu signálov.

Motiváciou vytvorenia modelu tohto procesoru je preskúmať do ako efektívne je možné použiť jazyk ISAC k popisu komplexnej modernej architektúry. Preto bude model vytvorený na čo najnižšej úrovni abstrakcie, a teda na úrovni cyklov.

### 4.3.1 Základné vlastnosti

Procesor kombinuje viacero konceptov. Obsahuje akumulátorovú architektúru umožňujúcu efektívne spracovanie signálov, inštrukčnú sadu založenú na konceptoch RISC, prostriedky na hromadné spracovanie dát a podporu multimediálnych operácií.

Jadro procesoru, zobrazené na 4.7, je zložené z troch blokov.

- aritmetický blok obsahuje osem 32-bitobých registrov, ktoré sú viac portové a môžu byť použité aj ako šestnásť 16-bitových registrov. Ďalej sú tu prítomné 2 aritmeticko-logické jednotky, dve násobičky kombinované s akumulátormi, funkčná jednotka pre bitové posuny a sada aritmeticko-logických jednotiek pre spracovanie videa. Operandy pre všetky operácie sú vždy umiestnené v prítomných registroch,
- kontrolný blok - zaobstaráva načítanie a dekódovanie inštrukcií. Jej úlohou je zabezpečiť, aby v každom takte mohla byť vydaná nejaká inštrukcia. To je dosiahnuté dedi-

| Pipeline Stage | Description |
|---|---|
| Instruction Fetch 1 (IF1) | Issue instruction address to IAB bus, start compare tag of instruction cache |
| Instruction Fetch 2 (IF2) | Wait for instruction data |
| Instruction Fetch 3 (IF3) | Read from IDB bus and align instruction |
| Instruction Decode (DEC) | Decode instructions |
| Address Calculation (AC) | Calculation of data addresses and branch target address |
| Data Fetch 1 (DF1) | Issue data address to DA0 and DA1 bus, start compare tag of data cache |
| Data Fetch 2 (DF2) | Read register files |
| Execute 1 (EX1) | Read data from LD0 and LD1 bus, start multiply and video instructions |
| Execute 2 (EX2) | Execute/Complete instructions (shift, add, logic, etc.) |
| Write Back (WB) | Writes back to register files, SD bus, and pointer updates (also referred to as the "commit" stage) |

Obrázek 4.8: Podrobný popis fáz zrťazenej linky procesoru PicoBlaze. Zdroj [1].

kovanou funkčnou jednotkou program sequencer, ktorá načítava a zarovnáva inštrukcie. Podporované relatívne a nepriame skoky a tiež volanie podprogramov. Prítomný je tiež dedikovaný hardvér podporujúci vykonávanie cyklov s nulovou réžiou,

- blok adresnej aritmetiky obsahuje funkčné jednotky pre výpočet adries operandov a inštrukcií pri zmene toku programu. Poskytuje duálne adresy pre simultánne načítanie dvoch nezávislých blokov dát z pamäte. Okrem toho obsahuje špeciálne registre na prácu s ukazateľmi.

Pamäť procesoru je štruktúrovaná ako jediné 4GB pole adresované 32-bitovými adresami. Interná pamäť procesoru tvorená hierarchiou cache pamätí, externá pamäť a registre vstupov a výstupov sú mapované do separátnych sekcií adresného rozsahu. Programová pamäť a pamäť pre dáta sú teda oddelené.

Procesor umožňuje paralelné vydávanie inštrukcií. V jednom takte môžu byť vydané až 3 inštrukcie - dve 16 bitové a jedna 32 bitová, pričom 32 bitová inštrukcia môže byť NOP. Nie sú však podporované všetky kombinácie inštrukcií. Povolené kombinácie sú uvedené v manuáli procesoru [1]. Pre indikáciu, že dané inštrukcie majú byť vyvolané paralelne sa používa špeciálna notácia na úrovni textovej reprezentácie inštrukcií.

### 4.3.2 Zreťazená linka inštrukcií

Snaha o popis procesoru na úrovni cyklov vyžaduje, aby bolo načítanie, vydávanie a vykonávanie inštrukcií popísané čo najpresnejšie. Procesor obsahuje desať stupňovú linku, ktorá je znázornená na obrázku 5.11.

Proces prechodu jednej inštrukcie procesorom pozostáva z týchto fáz:

- počas IFx fáz je vygenerovaná adresa vystavená na zbernicu, z ktorej sú neskôr prečítané dáta,

- fázy DEC, AC a DFx sú zodpovedné za dekódovanie inštrukcie a sú načítané jej operadnov,
- inštrukcia je vykonaná počas EXx fáz,
- výsledok inštrukcie je uložený počas fázy WB.

K načítaniu inštrukcie z pamäte však nemusí dôjsť v každom cykle, pretože inštrukcie sú načítavané hromadne. V prípade, že sa v jednom prístupe do pamäte načíta viac inštrukcií, program sequencer dokáže inštrukcie postupne vydávať bez nutnosti dodatočných pamäťových prístupov.

Program sequencer tiež zaobstaráva správu celej linky, a to napríklad v prípade neprítomnosti operandov vo vyrovnávajúcej pamäti. Okrem toho sa zabezpečuje, že sa linka nachádza v konzistentnom stave a že všetky hazardy ostanú pred programátorom skryté.

### 4.3.3 Inštrukčná sada

Procesor podporuje dva základné druhy inštrukcií:

- 16 bitové - najbežnejšie inštrukcie, krátky operačný kód vedie na väčšiu hustotu kódu,
- 32 bitové - menej používané, komplexné inštrukcie.

Prvé štyri bity inštrukcie nesú informáciu o dĺžke zvyšku inštrukcie a procesor tak vie veľmi rýchlo zistiť o akú inštrukciu ide.

Kódovanie inštrukcií je pomerné jednotné a na tejto úrovni nie je možné jasne definovať jednotlivé kategórie inštrukcií. Na sémantickej úrovni je však možné hovoriť o inštrukciách špecializovaných na určitý typ úloh.

Ďalším špecifikom tejto inštrukčnej sady je relatívne malý počet implementovaných operácií. Každá operácia však môže mať niekoľko kombinácií operandov, čo zapríčiňuje nárast počtu prípustných inštrukcií.

## 4.4 Architektúra STxP70

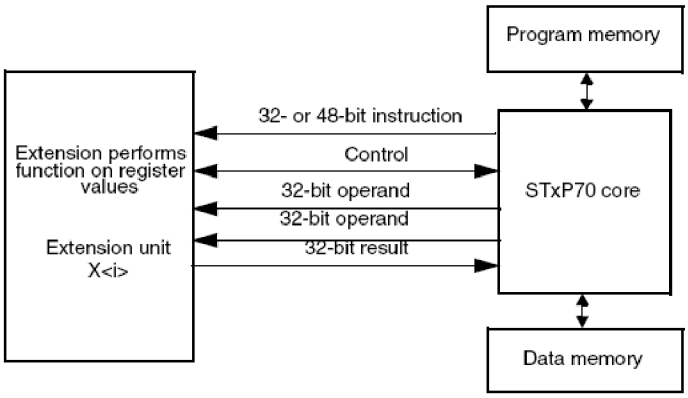
Architektúra STxP70 vzniká ako súčasť projektu spoločnosti STMicroelectronics, ktorého cieľom je navrhnúť a realizovať flexibilnú platformu integrujúcu sieť procesorov na jedinom čipe. Flexibilita architektúry umožňuje vysokú prispôsobiteľnosť a dovoľuje dodať užívateľovi optimálny procesor rozšírený o aplikačne špecifickú funkcionalitu.

V rámci tejto práce bude vytvorený model inštrukčnej sady takéhoto procesoru. Dôraz bude kladený na čo najpresnejšie modelovanie sémantiky inštrukcií, pretože model bude neskôr použitý pre generovanie prekladaču jazyka C pre túto architektúru. Okrem toho sa predpokladá možnosť simulovať aplikácie spomenutého procesoru pomocou automaticky generovaného simulátoru.

### 4.4.1 Základné vlastnosti

Jadro procesoru je založené na 32-bitovej RISC architektúre používajúcej Harvardskú organizáciu pamäte. Konfigurovateľnosť jadra umožňuje zvoliť:

- počet hardvérovo podporovaných kontextov - 0 až 8,




Obrázek 4.9: Znázornenie funkcionality užívateľských rozšírení. Zdroj [16].

- počet registrových blokov, jeden blok je tvorený 16 registrami,
- počet hardvérovo podporovaných cyklov pre každý kontext,
- prítomnosť dodatočnej aritmeticko-logickej jednotky,
- podpora pre duálne vydávanie inštrukcií.

Ostatné konfigurovateľné parametre je možné dohľadať v [16].

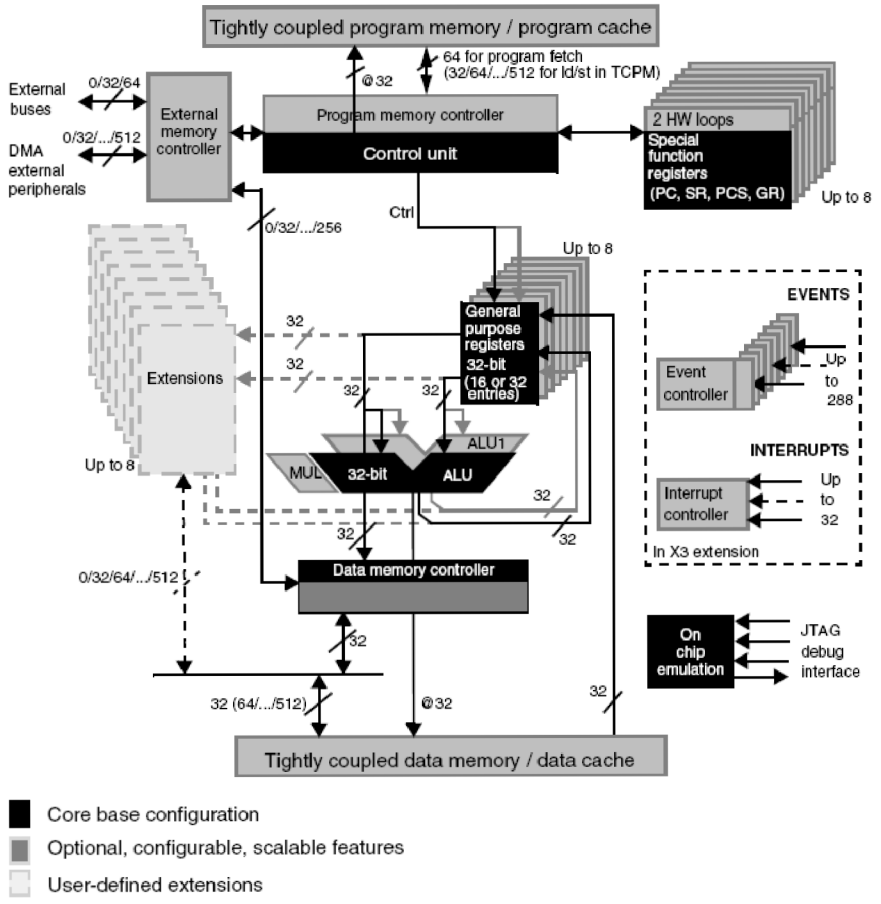
Pre každý kontext procesor obsahuje:

- registrov pre všeobecné použitie,
- špeciálne systémové registre, z ktorých je len 13 využitých a ostatné sú rezervované pre budúce použitie,
- registre pre komunikáciu s prípadnými rozširujúcimi modulmi.

Špeciálne registre obsahujú:

- programový čítač - obsahuje adresu vykonávanej inštrukcie,
- stavový register - obsahuje informácie o stave kontextu,
- guard register - používaný pre predikované vydávanie inštrukcií,
- podporu cyklov.

Harvardská organizácia pamäte znamená, že pamäť pre program je fyzicky oddelená od pamäti pre dáta. Obidve pamäte sú však mapované do jediného lineárneho priestoru schopného adresovať $2^{32}$ pamäťových buniek. Nekonvenčnou funkcialitou je možnosť explicitne modifikovať programovú pamäť, čo umožňuje dynamické inicializovanie obslúh prerušení a zjednodušuje nahrávanie programu. Príklad systému založeného na procesor STxP70-4 je možné vidieť na obrázku 4.10.




Obrázek 4.10: Znázornenie funkcionality užívateľských rozšírení. Zdroj [16]

```
    MAKE32 R0, 1
    MAKE32 R2, 1
G0? ADD R0, R2, 2

// If G0 is true, R0[31:0] is written back with the result of the
// addition: R0[31:0]=3
// If G0 is false, R0[31:0] is not written back with the result of
// the addition: R0[31:0]=1
```

Obrázek 4.11: Príklad predikovaného vykonania inštrukcie. Zdroj [16].

### 4.4.2 Inštrukčná sada

Inštrukčná sada procesoru STxP70-4 je založená na koncepte RISC, avšak je obohatená o niekoľko nekonvenčných konštrukcií.

Nachádzajú sa v nej tieto typy inštrukcií:

- inštrukcie prístupu do pamäte - umožňujú presúvať 8,16 alebo 32 bitové operandy medzi registrami a pamäťou. Tiež sa tu nachádzajú viaccyklové inštrukcie, ktoré systematicky ukladajú alebo obnovujú obsah skupiny registrov na zásobník,
- výpočtové inštrukcie - patria sem aritmetické a logické inštrukcie. Okrem toho tu existujú inštrukcie, ktoré dokážu vykonať danú operáciu nad separátnymi časťami svojich operandov (napr., 32-bitové operand môže byť chápaný ako dva 16-bitové alebo štyri 8-bitové),
- všeobecné inštrukcie - presuny medzi registrami a inicializácia obsahu registrov,
- inštrukcie skokov a zmeny toku programu,
- inštrukcie pre správu behu procesoru - breakpointy alebo manipulácia so systémovými registrami,
- inštrukcie pre rozšírenia - slúžia na riadenie rozširujúcich modulov.

Vykonávanie inštrukcií je predikované. To znamená, že o vykonaní inštrukcie je rozhodnuté bezprostredne pred jej potenciálnym vykonaním na základe hodnôt bitov tzv. guard registru (jedného zo špeciálnych registrov). Index bitu, na ktorom vykonanie inštrukcie je súčasťou textovej reprezentácie inštrukcie, a tiež je zakódované v inštrukčnom slove. Benefitom predikovaného vykonávania inštrukcií je redukcia veľkosti kódu, pretože vetvenie programu bez prítomnosti porovnávacích inštrukcií a podmienených skokov. Obrázok 4.11 obsahuje príklad podmieneného vykonania inštrukcie.

Inštrukcie môžu byť zakódované pomocou 16, 32 alebo 48 inštrukčných slov. Dĺžka inštrukčného slova závisí na niekoľkých faktoroch:

- inštrukcie predikované bitmi 2,3,5 a 6 sú vždy zakódované ako 48 bitové inštrukcie,
- inštrukcie predikované bitmi 0,1,4 a 7 sú zakódované ako 48 bitové inštrukcie len v prípade, že ich operand sa nezmestí do vyhradených bitov 32 bitovej varianty danej inštrukcie,
- inštrukcie sú zakódované pomocou 16 bitov v prípade, že sú predikované 7. bitom a ich operandy sú dostatočné malé aby boli reprezentovateľné určitým (malým) počtom bitov.

```
0000000000000000 <.text>:
   0:   0c 00 01 61 00 c0                        nop ;;
   6:   30 04 fc c7                    G4?  addu    R2, R3, 0xff ;;
   a:   30 04 fd 07 01 c0               G4?  addu    R2, R3, 0x1ff ;;
  10:   c3 c1                                nop ;;
  12:   c3 c1                                nop ;;
  14:   2b 87                          G7&  cmpeq   G0, R2, R3 ;;
  16:   0a 44 0c c0                    G4&  cmpeq   G0, R2, R3 ;;
xzvonc00@pczvoncek:~/ST_2011.1$
```

Obrázek 4.12: Ukážka variabilnej dĺžky inštrukčných slov niektorých inštrukcií.

Obrázok 4.12 obsahuje príklad rôzneho kódovania niektorých inštrukcií. Inštrukcia *addu* bezznamienkovo sčitujúca obsah registra s nejakou hodnotou je zakódovaná rôzne dlhým inštrukčným slovom v závislosti na použitom čísle. Ak nie je možné toto číslo reprezentovať pomocou 8 bitov, je použité dlhšie kódovanie inštrukcie. V druhom prípade je dĺžka inštrukčného slova ovplyvnená indexom použitého predikátového bitu.

# Kapitola 5

# Implementácia modelov

Zvolené procesory sú postavené na rôznorodých architektúrach, ktoré vyžadujú špecifické postupy a techniky modelovania jednotlivých procesorov. Táto kapitola je venovaná popisu implementácie každého z modelov. Sú v nej popísané konkrétne postupy použité pre vytvorenie modelu, prekonávané problémy a špecifické vlastnosti architektúr, ktoré nebolo možné modelovať.

Štruktúra modelov bude v tejto kapitole ilustrovaná okrem ukážok zdrojového kódu aj diagramami, ktoré majú určitú sémantiku:

- obdĺžniky s oválnymi okrajmi reprezentujú skupiny - konštrukcia GROUP jazyku ISAC,
- obdĺžniky s pravouhlými okrajmi reprezentujú operácie,
- pravouhlé spojovacie šípky reprezentujú priradenie operácii alebo skupín do nejakej skupiny,
- priame spojovacie šípky reprezentujú konštrukciu INSTANCE jazyka ISAC.

## 5.1 Model MSP430

Ako súčasť tejto práce bol implementovaný model inštrukčnej sady procesoru MSP430.

### 5.1.1 Zdroje

Obrázok 5.1 obsahuje ukážku popisu niektorých zdrojov mikrokontroléru. V rámci modelu sú obsiahnuté tieto elemetny:

- programový čítač široký 16 bitov modelovaný ako registre nepatriaci do registrového poľa. Tento postup je daný nutnosťou uviesť kľúčové slovo PC pred register a nie pred registrové pole,
- registrové pole 15 registrov, vhodne indexovateľné od 1 do 15,
- pamäťový element reprezentujúci pamäť RAM, ktorý je zložený s 16 bitových buniek, parameter LAU však umožňuje pristupovať aj k 8 bitovým podblokom. Príznaky R, resp. W, umožňujú čítanie, resp. zápis, dát pamäte,

```
RESOURCE {
    PC REGISTER bit[16] pc;
    REGISTER bit[16] gpr[1..15]; // R1 - R15
    bit[16] SP ALIAS { gpr[1] };

    RAM bit[16] ram { // pamäť RAM
        LAU (8); SIZE (0x1780); FLAGS (R,W);
    };

    RAM bit[16] prog_flash { // pamäť pre program
        LAU (8); SIZE (0x3780); FLAGS (R,W,X);
    };

    RAM bit[16] special_registers { // špeciálne registre
        LAU (8); SIZE (0x0100); FLAGS (R,W);
    };

    BUS bit[16] bus { // zbernica
        LAU (8);
    };

    MEMORY_MAP defaultmap { // mapovanie pamäte
        BUS(bus), RANGE (0x0, 0x1ff) -> special_registers[(15..0)];
        BUS(bus), RANGE (0x0200, 0x30ff) -> ram[(15..0)];
        BUS(bus), RANGE (0x3100, 0x9fff) -> prog_flash[(15..0)];
    };
}
```

Obrázek 5.1: Ukážka sekcie zdrojov mikrokontroléru MSP430.

- pamäťový element reprezentujúci pamäť programu, ktorý ma podobné parametre ako element *ram*, príznak X však umožňuje interpretovať v tejto pamäti uložené dáta ako spustiteľný kód,
- špeciálne registre, ktoré sú pre jednoduchosť reprezentované pamäťovým elementom,
- zbernica modelujúca prepojenie procesoru a pamätí.

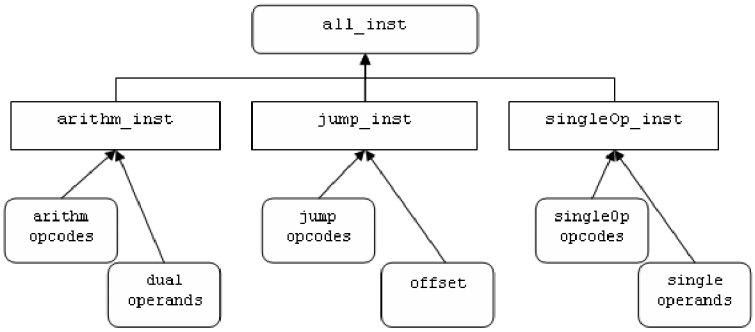
Obrázok 5.1 ďalej ukazuje rozdelenie adresného priestoru na jednotlivé pamäťové elementy. Začiatok adresného rozsahu obsahuje na adresách 0x0-0x1ff špeciále registre, za ktorými sa na adresách 0x200-0x30ff nachádza pamäť *ram*. Pamäť pre program je umiestnená na adresách 0x3100-0x9fff.

### 5.1.2 Štruktúra modelu

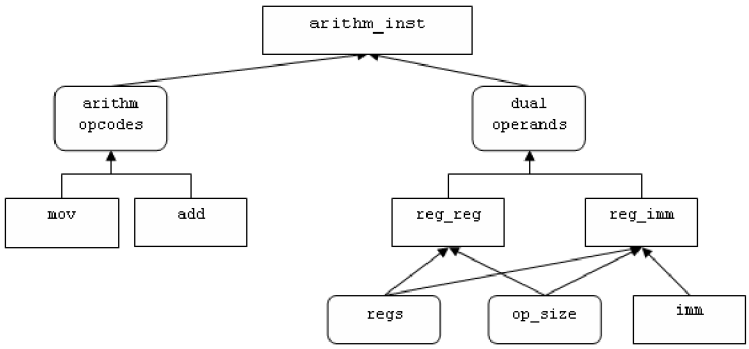
Štruktúra modelu má stromový charakter a kopíruje organizáciu kódovania inštrukcií. Obrázok 5.2 znázorňuje implementovanú hierarchiu operácií reprezentujúcich jednotlivé skupiny inštrukcií.

Koreň je reprezentovaný skupinou *all_inst*, ktorá je použitá v sekcii CODINGROOT ako koreň inštrukčnej sady. V tejto skupine sú spojené tri hlavné skupiny inštrukcií - *arithm_inst*, *jump_inst* a *singleOp_inst* - reprezentujúce jednotlivé skupiny inštrukcií.

Obrázok 5.3 bližšie ilustruje štruktúru operácie *arithm_inst*. V tejto operácii sú inštanciované dve skupiny - skupina operačných kódov *arithm opcodes* a skupina operandov *dual operands*. V skupine *arithm opcodes* sú združené operácie reprezentujúce jednotlivé inštrukcie (sčítanie, presun…). Skupina *dual operands* reprezentuje všetky možné kombinácie operandov pre dvoj-operandové inštrukcie. Ako príklad sú uvedené operácie reprezentujúce




Obrázek 5.2: Základná stromová štruktúra modelu procesoru MSP430.

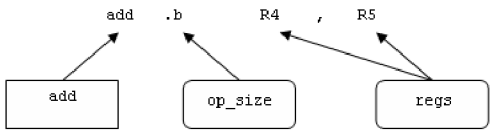



Obrázek 5.3: Detail štruktúry operácie *arithm_inst*.

variantu dvoch registrových operandov a variantu registra s priamou hodnotou. Operácie reprezentujúce dvojice operandov ďalej obsahujú inštancie reprezentujúce jednotlivé registre, priame hodnoty a iné segmenty inštrukcií. Na obrázku 5.4 je zobrazený vzťah medzi reálnou inštrukciou a popísanou hierarchiou operácií a skupín.

Zostručnený zápis konštrukcie operácie *arithm_inst* pomocou jazyka ISAC ukazuje obrázok 5.5. Pre jednoduchosť sú vynechané behaviorálne sekcie operácií, v ktorých je potrebné do príslušných registrov uložiť informácie z inštanciovaných operácií. Tieto registre sú potom použité v behaviorálnej sekcii operácie *arithm_inst* pre identifikovanie a vykonanie danej operácie.

Dôsledkom hlbokej hierarchie operácii vznikla nutnosť rozšíriť model procesoru o sadu registrov určených na uchovávanie hodnôt získaných z EXPRESSION sekcií inštancovaných operácií. Samotná EXPRESSION sekcia na túto úlohu nestačí, pretože jej účelom je výhradne prenos informácie medzi operáciou a jej inštanciou. Operácie *jump_inst* a *singleOp_inst* sú postavené na rovnakých princípoch.




Obrázek 5.4: Reprezentácia segmentov inštrukcí príslušnými operáciami.

```
/* skupina registrov */
OPERATION R0  { ASSEMBLER { "r0" }; CODING { 0b0000 }; EXPRESSION { 0; }; }
GROUP regs = R0;

/* veľkosť inštrukcií */
OPERATION byte_size { ASSEMBLER { ".b" }; CODING { 0b1 }; EXPRESSION { OP_SIZE_BYTE; }; }
OPERATION word_size { ASSEMBLER { ".w" }; CODING { 0b0 }; EXPRESSION { OP_SIZE_WORD; }; }
GROUP opr_size = byte_size, word_size;

/* kombinácia operandov - register, register */
OPERATION operands_r_r {
    INSTANCE regs ALIAS {reg_src, reg_dst };
    INSTANCE opr_size;
    ASSEMBLER { opr_size reg_src "," reg_dst };
    CODING { reg_src 0b0 opr_size 0b00 reg_dst };
    EXPRESSION { OP_R_R; };

    /* propagácia informácie o~indexe registrových operandov */
    BEHAVIOR { r_reg_src = reg_src; r_reg_dst = reg_dst; };
}

/* skupina kombinácií operandov */
GROUP dual_operands = operands_r_r;

/* operačné kódy */
OPERATION move {ASSEMBLER {"mov"}; CODING {0b0100}; EXPRESSION {OP_MOV;};}
OPERATION add {ASSEMBLER {"add"}; CODING {0b0101}; EXPRESSION {OP_ADD;};}
GROUP arithm_opcodes = move,add;

/* finálna operácia */
OPERATION arithm_inst {
    INSTANCE dual_operands ALIAS { operands };
    INSTANCE arithm_opcodes ALIAS { operations };
    ASSEMBLER { operations~operands };
    CODING { operations operands };
}
```

Obrázek 5.5: Konštruckia operácie *arithm_inst* zapísaná v jazyku ISAC.

```
OPERATION arithm_inst {
    BEHAVIOR {
        /* načítanie operandov */
        switch (operands) {
            case OP_R_R:
                op_src = gpr[r_reg_src];
                break;
        }

        /* vykonanie operácie */
        switch (operations) {
            case OP_MOV:
                result = op_src;
                break;
            case OP_ADD:
                res = sum(op_src,op_dst,c,r_opr_size);
                SET_FLAGS; // makrá realizujúce nastavenie príznakov
                SET_RESULT;
                break;
        }

        /* zápis výsledku */
        switch (operands) {
            case OP_R_R:
                gpr[r_reg_dst] = result;
                break;
        }
    };
}
```

Obrázek 5.6: Popis správania inštrukcií modelovaných operáciou *arithm_inst* pomocou jazyka ISAC.

### 5.1.3 Správanie inštrukcií

Popis správania inštrukcií je sústredený do spomenutých troch hlavných operácií, jedna operácia pre každú skupinu inštrukcií. Pomocou mechanizmu sekcie EXPRESSION je možné v týchto operáciách získať informáciu o operačnom kóde aktuálnej inštrukcie a vykonať príslušnú akciu. Keďže je sémantika inštrukcií popisovaná pomocou jazyka C, je možné využiť volanie funkcií, pričom telá funkcií sú odsunuté do externého súboru, čím sa sprehľadní zdrojový kód modelu.

Behaviorálna sekcia demonštračnej operácie *arithm_inst* má potom podobu ukázanú na obrázku 5.6. V príklade neuvažované kombinácie operandov a operácií by boli do popisu jednoducho pridané pomocou rozšírenia vetiev príkazov *switch*. Registrové pole je indexované pomocou registrov, ktorých hodnoty boli nastavené v behaviorálnej sekcii operácie *operands_r_r*.

### 5.1.4 Emulované inštrukcie

Súčasťou implementácie je aj model emulovaných inštrukcií. Konštrukcia ALIAS v kontexte priraďovania operácií do skupín umožnila zachytenie tejto funkcionality v rámci tvoreného modelu.

Podobne ako pri reálnych inštrukciách, aj emulované inštrukcie bolo možné rozdeliť do skupín podľa formátu kódovania, kde ako kritérium opäť slúžila použitá kombinácia operandov.

Okrem toho, niektoré emulované inštrukcie vyžadovali rôzne (nie však všetky) kom-

```
/* operácia pre emulovanú inštrukciu */
OPERATION nop {
    /* emulované ako mov #0, R3 */
    ASSEMBLER { "nop" };
    CODING { 0b0100 0b0000 0b0011 0b0011 0b00000000 0b00000000 };
}

/* skupina všetkých inštrukcií modelu, použitá v˜sekcii CODINGROOT */
GROUP all_inst =
    arithm_inst ALIAS {mov};
```

Obrázek 5.7: Ukážka implementácie emulovanej inštrukcie.

binácie operandov. To však nebolo z implementačného hľadiska prekážkou, pretože vyššie popísané princípy skladania operácií boli použiteľné aj v tomto prípade.

Obrázok 5.7 demonštruje mechanizmus na inštrukcii *nop*, ktorá je zakódovaná ako inštrukcia *mov*, ktorá nahráva hodnotu nula do registru s indexom 3. Pre emulovanú inštrukciu je vytvorená samostatná operácia. Textová podoba je teda odlišná od originálnej inštrukcie iná, kódovanie inštrukcie je však potrebné uviesť úplné, a to presne tak, aby zodpovedalo originálnej inštrukcií. Na obrázku je tiež ukázané priradenie oboch inštrukcií do koreňovej skupiny inštrukcií tak, aby inštrukcia *nop* bola aliasom inštrukcie *mov*.

## 5.2 Model PicoBlaze

Implementácia modelu bola vykonaná v dvoch fázach. V prvej fáze bol implementovaný model jej inštrukčnej sady. Druhým krokom bolo prispôsobenie popisu generátorom hardvérového popisu architektúry.

Spôsob implementácie bol veľmi podobný ako v prípade mikrokontroléru MSP430. Preto nebude ďalej podrobne uvedená ich štruktúra zapísaná pomocou jazyku ISAC. Vznikne tak priestor pre nových, efektívnych alebo zatiaľ nepoužitých konštrukcií.

### 5.2.1 Zdroje

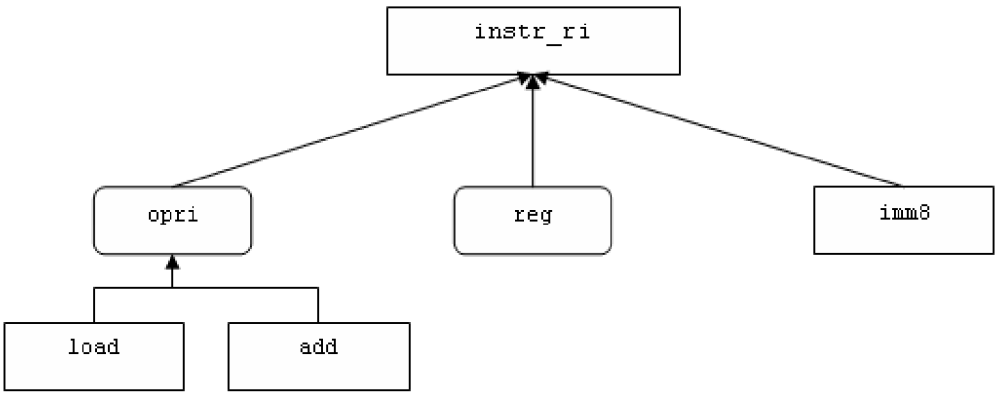
Dokumentácia modelu pomerne jasne a jednoznačne popisuje hardvérové vybavenie procesoru, ktoré je možné presne reprezentovať v sekcii RESOURCE jazyka ISAC. Medzi implementované zdroje teda patria:

- registre - programový čítač, ukazateľ na vrch zásobníku, príznaky, registre pre porty,
- pamäte - inštrukčná pamäť (neumožňuje zápis), *scratchpad* a zásobník.

Odlišnosťou od originálu je zavedenie dodatočného pamäťového prvku reprezentujúceho priestor externých portov, ktorý je potrebný pre simuláciu operácií s externými portami. Mapovanie pamäte je priamočiare. Celý rozsah pamäte je mapovaný do pamäťového prvku iprom modelujúceho pamäť inštrukcií.

### 5.2.2 Inštrukčná sada

Štruktúra modelu inštrukčnej sady je opäť daná formátom kódovania inštrukcií. V koreňovej skupine inštrukcií sú zahrnuté tieto typy inštrukcií:




Obrázek 5.8: Detail inštrukcie s registovým a priamym operandom procesoru PicoBlaze.

- inštrukcie s dvoma registrovými operandami,
- inštrukcie s registrovým operandom a priamou hodnotou,
- inštrukcie skokov,
- inštrukcie logických posunov,
- inštrukcie prísptupu do pamäte,
- inštrukcie vyvolávania podprogramov,
- inštrukcie podmieneného návratu do pamäte,
- inštrukcie prerušení

Každý typ je reprezentovaný jednou operáciou, v ktorej je (podobne ako v modeli procesoru MSP 430) situované správanie jednotlivých inštrukcií daného typu. Operácie pre jednotlivé typy boli implementované pomocou rovnakých princípov a postupov ako tie, ktoré boli použité pre modelovanie tried inštrukcií procesoru MSP430.

Obrázok 5.8 ilustruje štruktúru operácie reprezentujúcej inštrukcie s registrovým a priamym operandom. Inštrukcia je zložená z troch častí:

- operačný kód - skupina operačných kódov všetkých inštrukcií daného typu,
- registrový operand - skupina operácii reprezentujúcich jednotlivé registre,
- priama hodnota - 8-bitové číslo.

Do inštrukčnej sady bola doplnená inštrukcia *halt*, ktorá slúži počas simulácie procesoru a jej úlohou je zastaviť vykonávanie programu a ukončiť simuláciu. Jej behaviorálna sekcia poskytuje vhodné miesto na umiestnenie výpisu informácií o obsahu registrov a stave procesoru ako takom.

V porovnaní s popisom správania inštrukcií mikrokontroléru MSP430 bola vyvinutá snaha o maximálne využitie externých funkcií. Obrázok 5.9 ukazuje časť správania operácie modelujúcej inštrukcie s dvoma registrovými operandami. Výpočet výsledku a určenie hodnoty príznaku je odsunuté do externých funkcií. Externou funkciou je riešené aj rozhodnutie o tom, či daná inštrukcia vôbec s príznakmi manipuluje.

```
BEHAVIOR {
     /* načítanie operandov */

     /* vypočíta výsledok */
     result = exec_operation(operation,op_x,op_y,c);

     /* niektoré inštrukcie neovplyvňujú príznaky */
     if ( affects_flags(operation) ) {
         c = set_carry (operation,result);
         result &= 0xFF;
         z~= set_zero (operation,result);
     };

     /* uloženie výsledku */
 }
```

Obrázek 5.9: Využitie funkcií pre sprehľadnenie zdrojového kódu modelu.

```
/* kaskáda premenných pre propagovanie výsledku */
result = 0;
result_1 = ( operation == OP_RR_AND ) ? (op_x & op_y) : ( result );
result_2 = ( operation == OP_RR_OR ) ? (op_x | op_y) : ( result_1 );

/* uloženie výsledku do registra */
regs[reg_x] = result_2;
```

Obrázek 5.10: Ukážka syntetizovateľného popisu správania operácií.

### 5.2.3 Syntetizovanie popisu architektúry

Aby bolo možné vygenerovať popis architektúry v nejakom jazyku pre popis hardvéru, musela byť implementácia inštrukčnej sady výrazne modifikovaná.

Generátory hardvéru požadujú striktné dodržanie pravidiel, ktoré pochádzajú zo zásad programovania v jazykoch pre popis architektúr. K implicitným obmedzeniam jazyka ISAC tak pribudli ešte:

- nemožnosť používať volanie externých funkcií,
- zákaz použitia konštrukcií pre podmienky if a switch,
- vyžadovanie dodržania pravidla o priradení hodnoty premennej - do každej premennej môže byť hodnota priradená iba raz.

Absenciu vetvení programu bolo čiastočne možné odstrániť pomocou ternárnych operátorov. Aby však bolo dodržané pravidlo o jedinom priradení hodnoty do premennej, muselo byť zavedený niekoľko nových premenných, ktoré boli použité na prenos výsledku medzi jednotlivými operátormi. Tento mechanizmus je znázornený na obrázku 5.10. Premenná *result* je najskôr inicializovoaná na nulu. Ak je splnená podmienka prvého operátoru, je do prechodnej premennej *result_1* uložená hodnota nesúca výsledok operácie, inak je propagovaná predošlá hodnota. Podobne je tomu aj u druhého operátoru, odlišná je však prenosná premenná. Na záver je do registru uložený výsledok prítomný v poslednej prenosnej premennej.

Modifikovať popis správania inštrukcií, ktorý hojne využíval práve zakázané konštrukcie, sa ukázalo byť neúnosné. Preto bolo rozhodnuté model redukovať a implementovať syntetizovateľný popis len podmnožiny jeho inštrukcií. V modeli ostali prítomné:

- aritmetické inštrukcie pracujúce s registrovými operandami,
- inštrukcia pre nahratie priamej hodnoty do registru,
- inštrukcie pre presuny medzi registrami a pamäťou scratchpad.

Týmto spôsobom bolo možné zachovať základnú funkcionalitu procesoru, tak aby bola výsledná implementácia modelu prakticky syntetizovateľná.

## 5.3 Model BlackFin

V rámci tejto práce bola vykonaná aj experimentálna implementácia modelu sofistikovanej architektúry procesoru BlackFin. Snahou bolo popísať linku zreťazených inštrukcií a pomocou simulácie overiť jej činnosť.

Implementácia prebiehala v niekoľkých fázach. V prvej fáze boli modelované počiatočné stupne zreťazenej linky vydávania inštrukcií. Druhou fázou bolo nevyhnutné vytvorenie modelu aspoň zlomku inštrukčnej sady tak, aby bolo možné pomocou simulácie prakticky overiť funkčnosť a správnosť modelu. V tretej fáze bolo plánované postupne dopĺňať ostatné inštrukcie.

Implementácie tretej fázy nebola v rámci tejto práce vykonaná. Kapitola 6, v ktorej je zhodnotená implementácia z hľadiska použiteľnosti jazyka ISAC, vysvetľuje na aké prekážky bolo narazené a čo zabránilo pokračovaní v implementácii.

### 5.3.1 Fáza 1 - zreťazená linka

Podrobnosti štruktúry a spôsobu práce zreťazenej linky sú zrejme proprietárnym tajomstvom spoločnosti Analog Devices a nie sú zahrnuté vo verejnej dokumentácii procesoru. Implementácia modelu zreťazenej linky je teda iba približná (napokon, ide o model) a intuitívna.

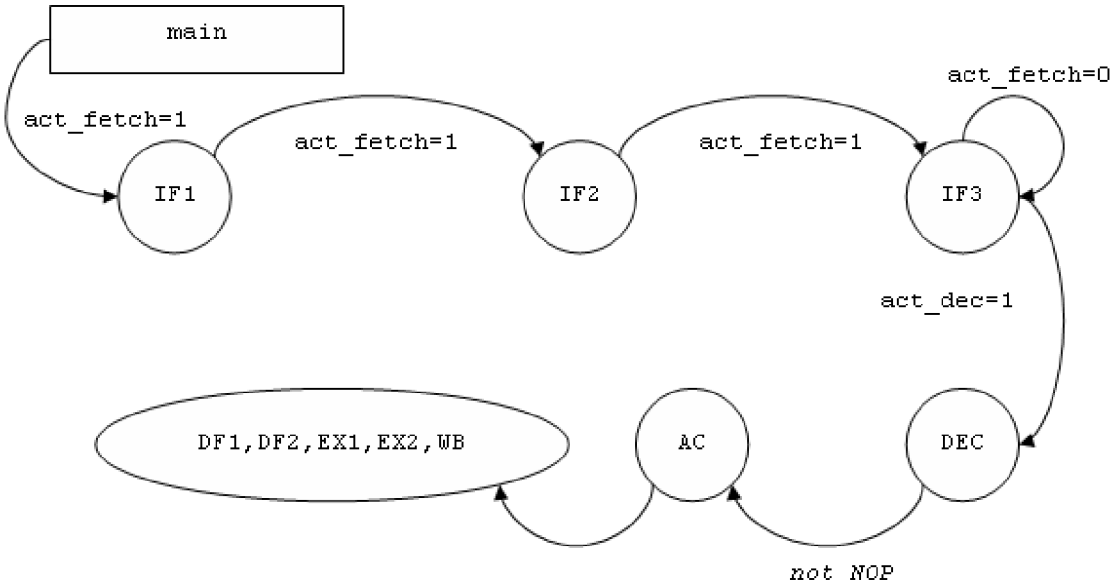
Nevyhnutnosťou bolo rozšíriť model o sadu interných registrov, ktoré sú používané na uchovávanie informácii o stave linky a prenosu dát medzi jednotlivými stupňami.

Obrázok 5.11 znázorňuje jednotlivé stupne, sekvenciu ich aktivácie a podmienky kedy k nej skutočne dochádza. V každom hodinovom takte sa podľa hodnoty stavového registru *act_fetch* rozhodne, či má byť aktivované načítanie nového kusu dát pomocou sekvencie stavov *IFx*. Finálny stav tejto trojice, *IF3*, reprezentuje jednotku zarovnávajúcu inštrukcie. Jeho úlohou je udržiavať vyrovnávajúcu pamäť inštrukcií tak, aby mohla byť plynulo vydávané. To je dosiahnuté náhľadom na operačný kód inštrukcie a príslušné nastavenie signálov povoľujúcich aktivovanie jednotlivých stavov linky. Ak je k dispozícií dostatok dát na vydanie inštrukcií, je nulovaný signál *act_fetch*, čím sa pozastaví načítanie ďalších dát. Zároveň je nastavený signál *act_dec*, ktorý umožní dekódovanie inštrukcie. Fáza *DEC*, reprezentujúca inštrukčný dekodér, spracuje inštrukciu a v prípade, že to nie je prázdna inštrukcia, tzv. *NOP*, odošle ju na ďalšie spracovanie.

Na obrázku 5.12 je zobrazená ukážka implementácie fázy IF3 realizujúcej riadenie procesu načítania a vydávania inštrukcií.

### 5.3.2 Fáza 2 - inštrukčná sada

Funkčnosť inštrukčnej sady mala byť overená s pomocou inštrukcií presunov, ktoré boli zvolené ako reprezentatívna vzorka inštrukčnej sady. Hoci ide o jedinú operáciu, inštrukcia




Obrázek 5.11: Znázornenie modelu inštrukšnej linky procesoru BlackFin.

```
OPERATION instruction_fetch_3 IN instruction_pipeline.IF3 {
    BEHAVIOR {
        /* ak nie je v˜buferi dostatok dát, načíta nové */
        if ( seq_should_fetch == 1) {
            fetch_inst(currently_fetched);
        }

        /* uprav bufer inštrukcií a nastav potrebné režíjne registre */
        if ( inst_buf[0] >= 0xC000 && (int)seq_unused_blocks >= 2 ) {
            /* pre 32 bitovú inštrukciu */
            act_dec = 1; // signál pre aktiváciu inštrukčného dekodéru
        }
        else if (inst_buf[0] <= 0xBFFF && (int)seq_unused_blocks >= 1 ) {
            /* pre 16 bitovú inštrukciu */
            act_dec = 1;
        }

        /* podľa výsledku zmeny stavu buffer aktivuj ďalšie načítanie inštrukcií */
        if ( should_act_fetch(0) == 1 ) {
            seq_should_fetch = 1;
            act_fetch = 1;
        }
    };

    /* aktivácia ďalších operácií */
    ACTIVATION {
        /* ak je možné aktivovať dekodér */
        IF ( act_dec == 1)
            { instruction_decode; }
        /* ak je k˜dispozícii dostatok dát a nie je potrebné čakať na ich doplnenie */
        IF ( act_fetch == 0 )
            { %1; instruction_fetch_3; }
    };
}
```

Obrázek 5.12: Operácia riadiaca načítanie a vydávanie inštrukcií modelu BlackFin.

podporuje relatívne vysoký druhov operandov. Podľa typu operandu sa rozhoduje, či má byť inštrukcia zakódovaná pomocou 16 alebo 32 bitov.

Prvým krokom bolo namodelovať 16 bitové inštrukcie. Tieto inštrukcie umožňujú ľubovoľný presun medzi:

- všeobecnými registrami,
- registrami ukazateľov,
- časťami akumulátorov,
- systémovými registrami,
- registrami pre generovanie adries,
- a podmienený presun medzi všeobecnými registrami a registrami ukazateľov.

Pre každú kombináciu zdrojového a cieľového typu registru bolo nutné vytvoriť zvlášť operáciu, pretože v kódovanie inštrukcií neobsahuje žiaden priamočiary systém, ktorý by vyhovoval stromovému konceptu stavania inštrukcií. Inou nepravidelnosťou a nepohodlnosťou boli registre pre generovanie adries, ktoré sú kódované inak ako ostatné registre v 16 bitových inštrukciách.

Ako druhý krok boli modelované 32 bitové inštrukcie, ktoré sú obecne určené na poskytovanie nadštandardnej a málo používanej funkcionality. Tieto inštrukcie umožňujú presuny medzi registrami tak, že je presun istým spôsobom obmedzený alebo konkretizovaný. Obmedzenia pre operandy sú napríklad:

- použitie výhradne operandov s párnym indexom,
- prístup len k polovici registru pre čítanie alebo zápis,
- špecifikácia nejakého parametru operácie, definujúceho spôsob zarovnania operandov v prípade ich rozdielnej veľkosti,
- účasné načítanie z dvoch rôznych registrov a súčasné uloženie ich hodnôt do dvoch registrov iného typu.

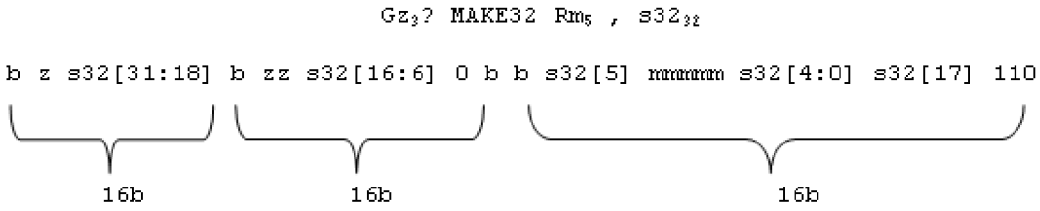
Každá z týchto špeciálnych kombinácii bola implementovaná samostatnou operáciou, pričom často krát bolo potrebné znovu implementovať aj operandy špecifické pre túto operáciu.

Operácie modelujúce inštrukcie prakticky reprezentujú inštrukčný dekodér. Ich behaviorálne sekcie sú vykonávané počas dekódovacej fázy *DEC* zreťazenej linky vydávania inštrukcií. Preto muselo byť v každej operácií, ktorá reprezentuje nejaký segment inštrukcie, zabezpečení uloženie informácií o týchto dátach do príslušných registrov tak, aby mohli byť propagované do ďalších stavov linky.

## 5.4 Model STxP70-4

Práce na implementácii modelu inštrukčnej sady architektúry STxP70-4 začali až po skúsenosti s procesorom BlackFin. Preto bol zvolený opatrný postup.

Pred implementáciou tohto modelu bol venovaný čas analýze kompletnej inštrukčnej sady. Okrem toho, že bol takto získaný prehľad o celej inštrukčnej sade, boli všetky inštrukcie zoskupené do skupín podľa svojho kódovania. Práve táto organizácia potom umožnila




Obrázek 5.13: Inštrukcia zložená z troch 16b segmentov.

modelovať jednorázovo väčšiu množinu inštrukcie a nevychádzať len z jej malej a nedostatočnej vzorky.

V rámci tejto práce bol vytvorený model 48 bitových inštrukcií inštrukčnej sady.

Samotná implementácia bola opäť rozdelená na fázy. V prvej fáze sa modelovala výhradne syntax a kódovanie inštrukcií aby sa zistilo, či jazyk ISAC dokáže zachytiť komplexné črty tejto architektúry. V prípade neúspechu by sa tak predišlo zbytočnej práci na zvyšku modelu. V druhej fáze bolo potom dopĺňané správanie inštrukcií tak, aby bolo pochopené extraktorom sémantiky. V dobe písania tejto kapitoly stále prebieha tretia fáza, a to simulácia inštrukcií a kontrola správnosti správania.

### 5.4.1 Štruktúra modelu

Koreňová skupina inštrukcií je podľa typu a prítomnosti predikátu rozdelená na skupiny predikované:

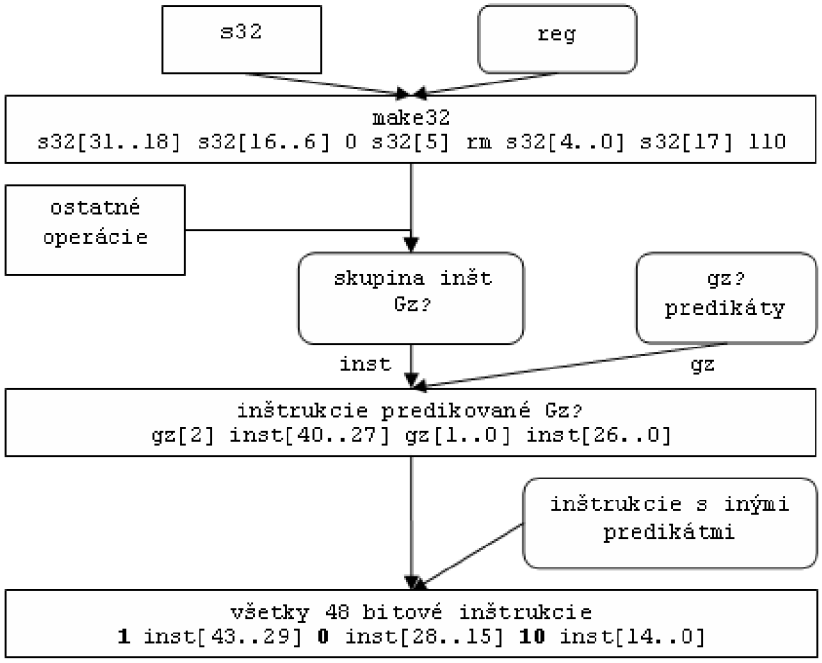
- Gx? Predikátom - obsahuje bežné inštrukcie,
- Gx& predikátom - obsahuje inštrukcie manipulujúce so samotnými predikátovými registrami,
- Gx&? Predikátom - inštrukcie obsahujúce dva druhy predikovaných akcií,
- žiadnym predikátom - špeciálne systémové inštrukcie.

Každá z týchto skupín je potom zložená zo skupín inštrukcií, ktoré boli identifikované počas prvotnej analýzy inštrukčnej sady.

Z implementačného hľadiska je ďalej zaujímavá organizácia každej 48, resp. 32, bitovej inštrukcie, v ktorej je vidieť, že aj takto dlhá inštrukcia je zložená z troch, resp. dvoch, menších segmentov. Tieto segmenty sú oddelené istými režijnými bitmi. Viac segmentové inštrukcie však typicky obsahujú operadny, ktoré sú zakódované do viacerých segmentov. Názorný príklad je možné vidieť na obrázku 5.13. Inštrukcia *MAKE32*, ktorá do registru $m$ nahrá 32 bitovú hodnotu *s32*. Číslo predikátu je zakódované troma bitmi $z$, číslo registru je zakódované piatimi bitmi $m$ a samotná konštanta je rozdelená do celého inštrukčného slova. Priame hodnoty bitov, ktoré je možné nazvať operačným kódom inštrukcie, sú tiež roztrúsené po celom inštrukčnom slove.

Zatiaľ nepopísane $b$ bity sú takzvané *bundle* bity oddeľujúce jednotlivé segmenty. Je možné odhadnúť, že práve podľa nich dokáže inštrukčný dekodér interpretovať prúd bitov a správne identifikovať inštrukciu. V rámci modelu je táto konštrukcia modelovaná hierarchicky. Pre vysvetlenie je opäť použitý obrázok 5.14.

Ako prvé je poskladaná samotná inštrukcia. Tá je potom zoskupená s ostatnými inštrukciami predikovanými tým istým typom predikátu do jednej skupiny inštrukcií. Táto skupina




Obrázek 5.14: Postup modelovania inštrukcie pomocou postupného vkladania bitov na ich príslušné miesta.

je potom inštancovaná v operácií, ktorej úlohou je do syntaxe a kódovania inštrukcií pridať daný predikát. Kódovanie pôvodnej inštrukcie musí byť roztrhnuté, čím vznikne správne miesto pre kód predikátového registru. Roztrhnutie je potom zopakované ešte na jednej úrovni, pretože do kódovania inštrukcie musia byť vložené hrubo vyznačené *bundle* bity.

Implementácia postupu v jazyku ISAC je ukázaná na obrázku 5.15.

### 5.4.2 Správanie inštrukcií

Kódovanie inštrukcií si vynútilo zhromaždenie všetkých modelovaných inštrukcií v jedinej operácii. Táto skutočnosť je príjemná pre implementáciu správania inštrukcií, pretože umožňuje správanie jednoducho zastrešiť volaním jedinej externej funkcie, tak ako je to ukázané na obrázk 5.16. Hierarchia operácii modelujúca inštrukcie zabezpečuje nastavenie pomocných registrov (identifikátory s prefixom *reg_*) informáciami potrebnými pre vykonanie sémantiky inštrukcie, a tak môže byť korektne vykonaná.

Popísanie sémantiky predikovateľnosti tiež nie je problém, pretože je s výhodou možné využiť konštrukcie podmienky jazyka ISAC skúmajúcu sa.

```
/* operácia pre register */
OPERATION reg { ASSEMBLER { "R0" }; CODING { 0b00000 }; EXPRESSION {0;}; }

/* operácia pre priamu hodnotu */
OPERATION simm32 { ASSEMBLER { imm=#S31 }; CODING { imm=0bsx[32] }; EXPRESSION { imm; }; }

/* inštrukcia make32 */
OPERATION make32_48 {
    /* inštancia reg a simm32 */
    ASSEMBLER { "MAKE32" rm "," s32 };
    CODING { s32[31..18] s32[16..6] 0b0 s32[5] rm s32[4..0] s32[17] 0b110 };
}

/* skupina inštrukcií predikovaných ako Gx? */
GROUP inst_48b_zQ_group = make32_48;

/* doplnenie predikátov */
OPERATION inst_48b_0_7Q {
    INSTANCE inst_48b_zQ_group ALIAS { inst };
    INSTANCE g23567q ALIAS { gz };
    ASSEMBLER { gz inst };
    CODING { gz[2] inst[40..27] gz[1..0] inst[26..0] };
}

/* skupina všetkých inštrukcií */
GROUP inst_48b_group = inst_48b_zQ;

/* korenňová operácia 48 bitových inštrukcií, doplnenie bundle bitov */
OPERATION inst_48b {
    INSTANCE inst_48b_group ALIAS { inst };
    ASSEMBLER { inst };
    CODING { 0b1 inst[43..29] 0b0 inst[28..15] 0b10 inst[14..0] };  // bundle bity
}
```

Obrázek 5.15: Proces dopĺňania predikátov a bundel bitov do inštrukčných slov 48 bitových inštrukcií architektúry STxP70.

```
OPERATION inst_48b {
    BEHAVIOR {
        /* nepodmienené modifikovanie adresy pred jej použitím */
        exec_am_pre_modif(reg_op, reg_src1, reg_src2, reg_src3, reg_src4, reg_res);

        /* rohzhodnutie o~(ne)vykonaní inštrukcie podľa hodnoty predikátového bitu */
        if ( GREAD(GUARD,reg_guard) == 1 ) {
            /* akcie, ktoré majú byť vykonané ak je podmienka splnená */
            exec_op_cond(reg_op, reg_src1, reg_src2, reg_src3, reg_src4, reg_res);
        } else {
            /* akcie, ktoré majú byť splnené ak podmienka nie je splnená */
            exec_op_NOTcond(reg_op, reg_src1, reg_src2, reg_src3, reg_src4, reg_res);
        }

        /* vykonanie nepodmieneného správania niektorých inštrukcií */
        exec_op_UNcond(reg_op, reg_src1, reg_src2, reg_src3, reg_src4, reg_res);

        /* nepodmienené modifikovanie adresy po jej použití */
        exec_am_post_modif(reg_op, reg_src1, reg_src2, reg_src3, reg_src4, reg_res);
    };
}
```

Obrázek 5.16: Centralizácia vykonávania inštrukcií.

# Kapitola 6

# Zhodnotenie implementácie

V tejto kapitole je zhodnotená implementácia modelov zvolených procesorov. Pre každý procesor je popísané v akom stave sa model nachádzal po dokončení implementácie aké problémy sa pri implementácii podarilo vyriešiť a aké problémy ostali nevyriešené. V prípade praktického použitia modelu je uvedené, akým spôsobom bol model použitý. Záver kapitoly tvorí celkové zhodnotenie jazyka ISAC z pohľadu použiteľnosti na popis zvolených architektúr.

Implementácia modelov zvolených architektúr sa ukázala byť masívny projekt, počas ktorého boli dôkladne preverené všetky aspekty jazyka ISAC. Počas modelovania bolo odhalených niekoľko prístupov a možností jazyka, ktoré značne uľahčili a zefektívnili celý tvorbu modelu. Zároveň však bolo možné identifikovať situácie, kde sa ukázalo, že jazyk má problém popísať niektoré špecifiká modelovaných architektúr.

Niektoré modely sa po implementácii dočkali aj praktického použitia. Bolo tak možné demonštrovať prínos projektu Lissom ako celku.

V tejto kapitole budú pre každý model zhrnuté poznatky nadobudnuté počas jeho implementácie, ktoré sa týkajú hlavne vhodnosti jazyka ISAC pre popis zvolenej architektúry. Ďalej bude podrobne prezentované prípadné použitie modelu a zhodnotené niektoré parametre modelu.

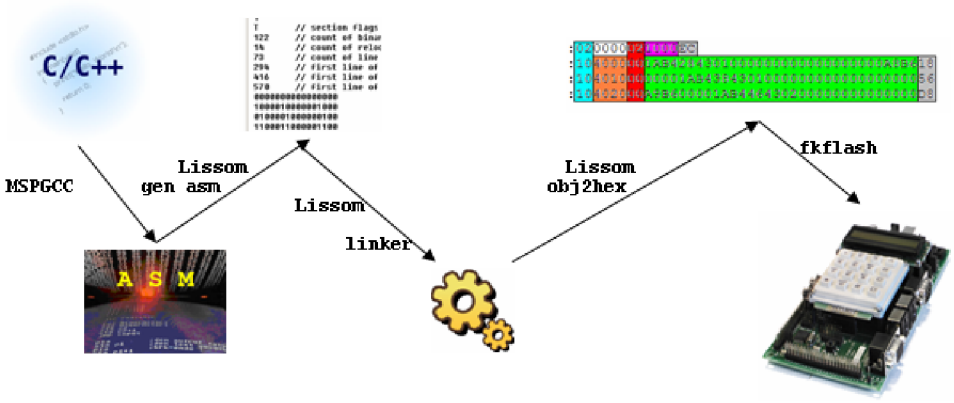
## 6.1 Model MSP430

Model procesoru MSP430 bol tvorený ako prvý. Počas jeho implementácie bolo objavených niekoľko vlastností jazyka ISAC, ktoré sa potom opakovane prejavili aj počas implementácie ostatných modelov. Tieto postrehy budú uvedené na záver tejto kapitoly.

### 6.1.1 Použitie modelu

Hlavnou ambíciou vytvorenia modelu procesoru bolo dosiahnuť spustenie kódu zostaveného automaticky generovanými nástrojmi na reálnom procesore. Jedným z generovaných nástrojov je aj simulátor, ktorý by umožnil simuláciu zostaveného kódu a v kombinácii s grafickým užívateľským rozhraním by tak prakticky zadarmo vzniklo vývojové prostredie aplikácií pre daný procesor. Takéto prostredie je pomerne atraktívne, pretože v súčasnom balíku softvéru v projekte FITKit zatiaľ chýba.

Dokončením prvej verzie implementácie modelu bol odštartovaný inkrementálny proces




Obrázek 6.1: Postup programovania mikrokontroléru FITKitu s využitím nástrojov projektu Lissom.

opravovania a prispôsobovania jak modelu (pretože nie všetka funkcionalita bola zrozumiteľne a jasne popísaná v dokumentácii procesoru) tak generovaných nástrojov.

Prvým krokom bolo zaistenie binárnej kompatibility programov zostavených generovaným asemblerom a linkerom s referenčnými výstupmi prekladaču MSPGCC, čo je voľne šíriteľný prekladač jazyka C do strojového kódu rodiny mikrokontrolérov MSP.

Po dosiahnutí binárnej kompatibility boli zahájené experimenty s programovaním mikrokontroléru pomocou programu zostaveného automaticky generovaným asemblerom. Keďže formát spustiteľných súborov interne používaný v projekte Lissom nezodpovedá formátu, v ktorom sú uložené programy mikrokontroléru, bolo potrebné doprogramovať konvertor medzi týmito formátmi. Implementácia konvertoru bola vykonaná Ing. Adamom Husárom, členom týmu projektu Lissom. Samotné programovanie procesoru, teda prenos dát reprezentujúcich inštrukcie programu, bolo vykonané pomocou utility *fkflash*, ktorá je súčasťou softvérového balíku projektu FITKit.

Správnosť nahraného programu bolo následne nutné overiť. K tomuto účelu bol implementovaný výpočet kódu CRC, ktorý po dosiahnutí správneho výsledku rozsvieti jednu z diód prístupných na prípravku.

Ďalším krokom bolo overenie funkčnosti zložitejších konceptov ako napríklad volanie podprogramov a práca so zásobníkom.

Finálnou fázou celého procesu bolo integrovanie funkcionality periférií prítomných na prípravku do programu. Ako východiskové sa použili demonštračné aplikácie dostupné v nástroji *qdevkit*. Postupne sa podarilo spustiť programy, ktoré komunikovali s LCD displejom a klávesnicou. Vrcholom snaženia bolo dosiahnutie správneho behu hry Had, ktorá okrem komunikácie s VGA adaptérom vyžíva aj koncept prerušení.

Obrázok 6.1 ilustruje finálnu schému procesu programovania mikrokontroléru pomocou automaticky generovaných nástrojov. Na vstupe celého procesu je nejaký program zapísaný v jazyku C. Ten je pomocou už spomenutého prekladača MSPGCC preložený do jazyka symbolických inštrukcií procesoru MSP430. Následne je pomocou automaticky generovaného asembleru vytvorený objektový súbor, ktorý je vzápätí zlinkovaný linkerom objektových súborov projektu Lissom. Spustiteľný súbor je potom transformovaný do formátu HEX [10] pomocou špeciálneho konvertoru. Na záver je skonvertovaný program odoslaný do samotného mikrokontroléru pomocou utility fkflash.

| Program pre MSP430 | Originál | Lissom | Rozdiel |
|---|---|---|---|
| CRC | 748 B | 720 B | 28 B |
| Displej | 6 646 B | 5 764 B | 882 B |
| Had | 12 506 B | 10 842 kB | 1 664 B |

Obrázek 6.2: Porovnanie veľkostí programov pre procesor MSP430 zostavených originálnym asemblerom a automaticky generovaným asemblerom. Z porovnávaných súborov boli ručne vystrihnuté len relevantné sekcie.

### 6.1.2 Pozorované vlastnosti jazyka

Jedným z hlavných princípov projektu Lissom je automatizované generovanie programovacích nástrojov, ktoré nevyžaduje žiaden zásah človeka. V zmysle tohto princípu bol jazyk ISAC navrhnutý tak, aby umožňoval popis štandardných konštrukcií a princípov inštrukčných sád. Preto nie je nikdy vopred možné určiť, či sa nejaké špecifikum alebo optimalizácia inštrukčnej sady podarí úspešne modelovať.

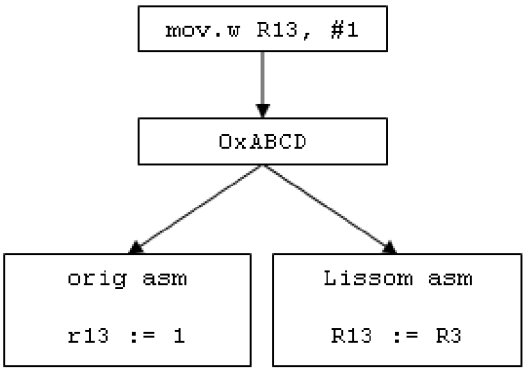
Jednou z takýchto špecialít je koncept emulovaných inštrukcií. V tomto prípade bolo možné využiť niektoré konštrukcie jazyka a emulované inštrukcie úspešne modelovať. Hoci bol proces ich implementácie pomerne prácny, neboli zaznamenané zásadné problémy z tohto konceptu prameniace. Jediným možným nedostatkom je skutočnosť, že emulované inštrukcie sa po reverznom asemblovaní spustiteľného súboru príslušným generovaným nástrojom objavia vo výstupe ako ich obrazy. Tým pádom dôjde k zníženiu čitateľnosti disasembolvaného programu.

Kvôli spomenutému dôrazu na všeobecnosť a autonómnosť bol však počas implementácie modelu objavený mechanizmus, ktorý nebolo možné zachytiť. Týmto mechanizmom je využitie generátoru konštánt. Prostriedky jazyka ISAC neposkytujú možnosť popísať netriviálnu funkcionalitu ako je táto. Dôsledky tejto situácie sú dvojaké:

- v prvom rade sú inštrukcie, ktoré by normálne boli originálnym asemblerom kódované pomocou kratších inštrukčných slov, v kóde zostavenom automaticky generovaným prekladačom dlhšie. Tým dôjde k miernemu nárastu celkovej veľkosti spustiteľných programov. Tabuľka 6.2 zobrazuje porovnanie veľkosti niekoľkých programov preložených originálnym asemblerom a nástrojmi projektu Lissom,

- druhým dôsledkom je nutnosť explicitnej interpretácie inštrukcií v prípade simulácie kódu generovaného originálnym asemblerom. Optimalizácia spočíva v použití špeciálneho registru namiesto jedného z operandov. Použitie špeciálneho registru je rozpoznané inštrukčným dekodérom v procesore a je zabezpečené neštandardné správanie. Generovaný simulátor však nemá žiadne informácie o takejto funkcionalite a preto sa snaží inštrukciu vykonať tak ako je dané jej kódom - vykonať príslušnú operáciu a použiť špeciálny register ako operand. Na obrázku 6.3 je príklad takejto situácie. Z tohto dôvodu bolo nutné do popisu správania inštrukcií doplniť kód, ktorý stráži použitie tejto optimalizácie a eventuálne zaisťuje potrebné správanie. Tým je umožnené simulovanie optimalizovaného kódu hoci nie je možné jeho generovanie. Reálny procesor však nemá problém s vykonávaním neoptimalizovaných inštrukcií, pretože tieto inštrukcie nijak neporušujú syntaktické pravidlá inštrukcií.

Ďalším limitom, na ktorý bolo počas riešenia práce narazené je nemožnosť simulácie




Obrázek 6.3: Príklad rozdielnej interpretácie optimalizovaného kódovania inštrukcie.

všetkých periférií procesoru.

Medzi periférie, ktorých funkcionalitu sa podarilo odsimulovať, patria napríklad LED diódy. Tie sú ovládané pomocou nastavovania bitov špeciálnych registrov mapovaných na určité adresy adresného priestoru. Modelovanie adresného priestoru nie je problematické a počas simulácie bolo možné priamo sledovať zmeny bitov v pamäťových bunkách obsahujúce obrazy daných registrov.

Problematickou sa ukázala byť snaha o simuláciu periférií, ktoré môžu procesoru poskytnúť nejaké vstupné dáta, napríklad klávesnice. Prostredie simulácie projektu Lissom zatiaľ neumožňuje emuláciu takýchto komponentov. Preto nie je možné do simulovaného programu doručiť potrebnú informáciu a program teda nemôže na túto informáciu reagovať.

Implementácia modelu môže byť označená ako úspešná, pretože sa podarilo splniť určený cieľ. Jazyk ISAC sa ukázal byť dostatočne silný na to, aby zachytil všetky potrebné špecifiká architektúry a umožnil vytvorenie prakticky použiteľných programovacích nástrojov, ktorých správna funkcionalita bola následne overená s použitím skutočného procesoru. Vygenerované nástroje dokázali korektne pracovať so všetkými konštrukciami jazyka C a utilizovať (hoci v niektorých prípadoch len jednosmerne) všetky komponenty a periférie mikrokontroléru.

### 6.1.3 Simulácia modelu

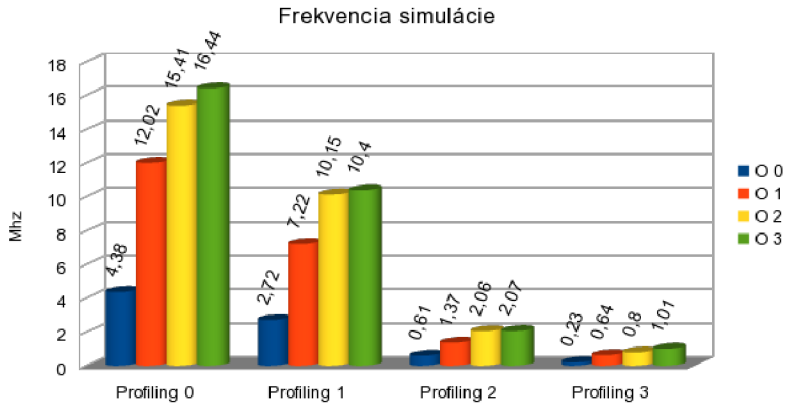
Stupeň implementácie modelu umožnil simulovať prakticky bežne používané programy. Z toho dôvodu bol tento model použitý na prieskum vplyvu parametrov procesu generovania simulátoru, ale aj samotnej simulácie na jej rýchlosť.

Ako simulovaný program bol zvolený výpočet kódu CRC, ktorý bol cyklicky počítaný. Dôvodom bolo potrebné predĺženie doby behu programu. Pre získanie relevantných dát bolo nevyhnutné, aby simulácia bežala aspoň 10 sekúnd.

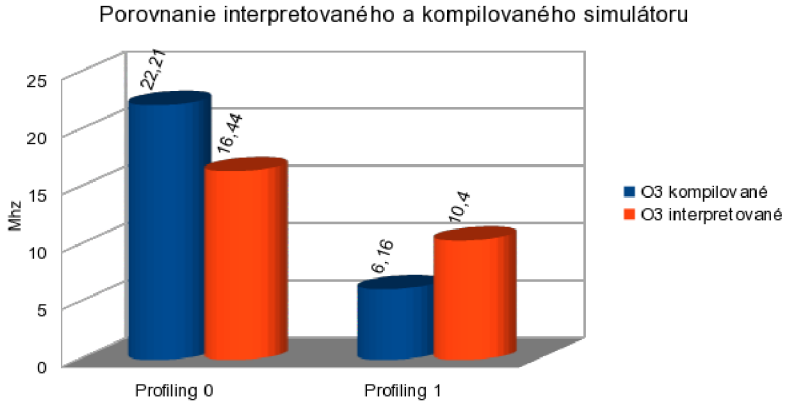
Skúmaný bol výkon simulácie pri zmene dvoch parametrov:

- optimalizácia prekladača - simulátor je v tomto smere bežná aplikácia, ktorá vznikla prekladom zdrojových súborov. Preklad môže byť optimalizovaný jednou zo 4 úrovní (0, vôbec, 3 maximálne). Maximálna optimalizácia síce sľubuje väčší výkon, cenou je však predĺženie doby kompilácie,

- úroveň profilovania - profilovanie je proces zberu štatistických dát v priebehu simulácie. Teoreticky, čím viac dát je zbieraných, tým nižšia je rýchlosť simulácie.




Obrázek 6.4: Priemerné rýchlosti simulácie pre jednotilvé konfigurácie simulátoru. Zdroj B.1.




Obrázek 6.5: Porovnanie staticky zostaveného a kompilovaného simulátoru. Zdroj B.1 a B.2.

Obrázok 6.4 graficky znázorňuje pozorované výsledky. Ukázalo sa, že úroveň profilovania má zásadný vplyv na rýchlosť simulácie. Vplyv optimalizácie kompilácie je najviac vidieť ak je profilovanie úplne vypnuté, alebo je vykonávané vo svojej najjemnejšej podobe.

Základný variant simulátoru je tzv. interpretovaný simulátor, kedy samotný kód realizujúci simuláciu nie je kompilovaný, ale len interpretovaný. Projekt Lissom však poskytuje možnosť simulačný kód predkompilovať a zostaviť tzv. kompilovaný simulátor. Od kompilovaného simulátoru sa očakáva zvýšenie rýchlosti simulácie. Obrázok 6.5 zobrazuje porovnanie výsledkov meraní rýchlostí kompilovaného a interpretovaného simulátoru. Anomáliou je, že po povolení profilovania interpretovaný simulátor prekonal kompilovaný simulátor.

### 6.1.4 Možnosti ďalšej práce

Priestor pre nadviazanie práce na modeli je niekoľko, nie všetky však priamo súvisia so samotným modelom.

Po dokončení automatického generátoru prekladaču jazyka C do asembleru modelovanej architektúry bude pravdepodobne potrebné revidovať popis správania inštrukcií tak, aby bolo možné ich sémantiku korektne extrahovať. V dobe písania tejto práce podlieha nástroj

| Device Utilization Summary (estimated values) | | | | [-] |
|---|---|---|---|---|
| **Logic Utilization** | **Used** | **Available** | **Utilization** | |
| Number of Slices | 376 | 768 | | 48% |
| Number of Slice Flip Flops | 152 | 1536 | | 9% |
| Number of 4 input LUTs | 714 | 1536 | | 46% |
| Number of bonded IOBs | 175 | 124 | 141% | |
| Number of GCLKs | 1 | 8 | | 12% |

Obrázek 6.6: Výsledky syntézy procesoru picoBlaze. Cieľová platforma je rodina obvodov Spartan3.

pre generovanie prekladaču jazyka C intenzívnemu vývoju, ktorý znamená časté zmeny požiadaviek na spôsob zápisu sémantiky inštrukcií, a preto nebolo možné model tomuto účelu definitívne prispôsobiť.

Ďalšou možnosťou rozšírenia použiteľnosti modelu je umožnenie simulácie periférií, čo je však úloha spadajúca viac do oblasti grafického užívateľského rozhrania.

## 6.2 Model PicoBlaze

Implementácia modelu in3truk4nej sady procesoru picoBlaze bola vykonaná v relatívne krátkom čase. Počas celého procesu neboli zaznamenané žiadne komplikácie. Je však nutné konštatovať, že v tomto aspekte je procesor veľmi jednoduchý a takáto jednoduchosť nie je pravidlom medzi dnešnými procesormi.

Táto architektúra je ideálnym príkladom efektívnosti aplikácie projektu Lissom. Jednoduchosť architektúr tohto typu umožňuje plné využitie konštrukcií jazyka ISAC pre vytvorenie relatívne presného modelu skutočného procesoru. Absencia optimalizácií a nadštandardnej funkcionality v inštrukčnej sade alebo samotnej architektúre zároveň odstraňuje nutnosť riešiť tieto problémy pomocou konvenčných konštrukcií.

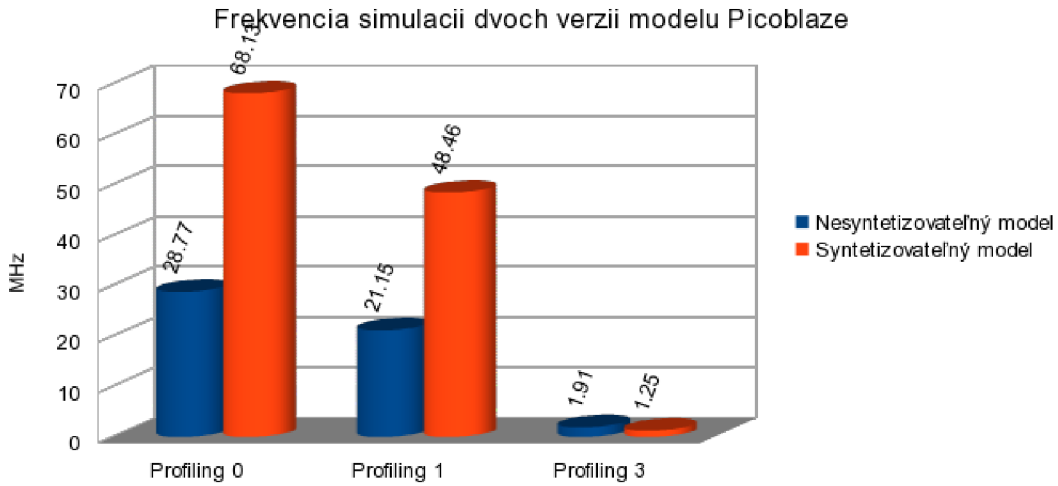
### 6.2.1 Syntéza procesoru

Jednoduchosť tvorby modelu inštrukčnej sady má pôvod vo využití konštrukcií jazyka C pre popis správania inštrukcií pomocou minimálneho množstva kódu. Požiadavky na spôsob zápisu programu kladené generátormi popisu hardvéru však znemožnili použitie týchto konštrukcií.

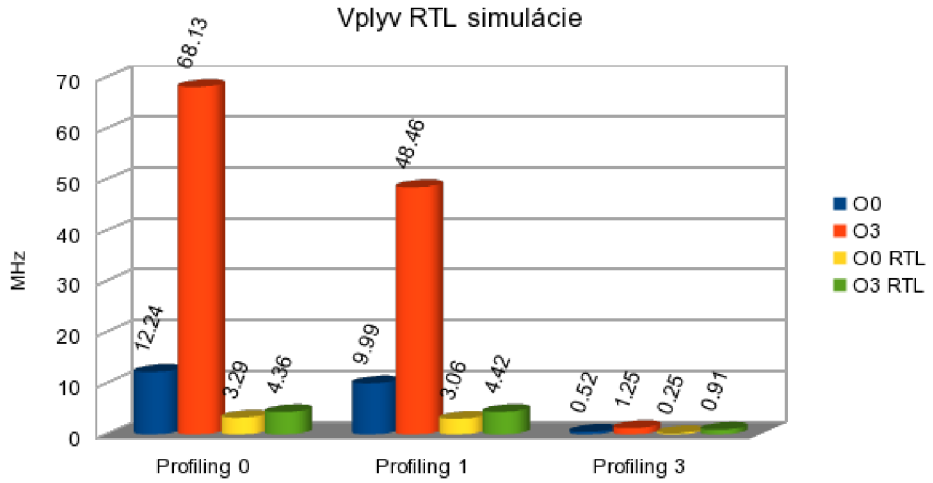
Ukázalo sa, že pre syntetizovateľný popis procesoru je potrebné na tieto obmedzenia pamätať od počiatočných fáz návrhu a implementácie modelu.

V tejto práci bol zvolený iný postup - bola vyvinutá snaha konvertovať model, ktorý nedbal na tieto obmedzenia. Ako už bolo vysvetlené, bol dosiahnutý iba čiastočný úspech. Obrázok 6.6 zobrazuje parametre syntetizovaného obvodu.

Prínosom snaženia o implementáciu syntetizovateľného popisu procesoru však je poukázanie na možnosti vylepšenia nástrojov pre generovanie popisu hardvéru. Súčasná situácia, kedy je pri tvorbe modelu brať ohľad na mechanizmus prepojenia funkčných blokov a nastavovania signálov v hardvéry, znižuje úroveň abstrakcie popisu architektúry.




Obrázek 6.7: Výsledky simulácie dvoch verzií modelu PicoBlaze. Zdroj B.4 a B.5.




Obrázek 6.8: Rozdiel medzi obyčajnou simuláciou a simuláciou na úrovni RTL. Zdroj B.4.

### 6.2.2 Simulácia modelu

Ako už bolo spomenuté, model vznikol v dvoch verziách, ktoré sa líšia spôsobom popisu správania inštrukcií. Zaujímavé bude sledovať, aký dopad bude mať tento rozdiel na rôzne typy simulácie.

Obrázok 6.7 prezentuje výsledky porovnania rýchlosti simulácie syntetizovateľnej a nesyntetizovateľnej varianty. S výnimkou maximálnej úrovne profilácie dosahuje syntetizovateľný model zhruba dvojnásobne lepšie výsledky. Dôvodom je zrejme optimálny zápis správania inštrukcií. Cenou je však sťažená čitateľnosť tohto popisu. Simulátory boli kompilované s použitím maximálnych optimalizácií.

Druhým pokusom bol prieskum vplyvu RTL (angl. register transfer level)+ simulácie. Ukázalo sa 6.8, že tento spôsob simulácie síce výrazne znižuje dosahované frekvencie, avšak eliminuje vplyv prítomnosti najjemnejšej úrovne profilovania. Optimalizácia kompilácie taktiež stráca váhu, pretože vďaka prítomnosti RTL simulácie nie je možné využiť efektívny kód simulátoru.

### 6.2.3 Možnosti ďalšej práce

Zamýšľané použitie procesoru picoBlaze spočíva v paralelnom zapojení viacerých takýto procesorov pod správou iného, väčšieho a komplikovanejšieho procesoru. Takouto topológiou by vznikol systém procesorov integrovaných na jedinom čipe.

Jedným z perspektívnych cieľov projektu Lissom je poskytnúť vývojové prostredie a nástroje aj pre takéto systémy. Vhodným pokračovaním práce by teda mohla byť implementácia modelu riadiaceho procesoru. Ďalšou možnosťou by bola implementácia podpory takýchto systémov do generátorov programovacích nástrojov projektu Lissom. V súčasnosti síce jazyk ISAC obsahuje niekoľko kľúčových slov a konštrukcií pre tieto účely, avšak ich sémantika je zatiaľ ignorovaná.

Zrejmá je aj možnosť ďalšieho výskumu v oblasti transformácie popisu správania operácií do jazyku pre popis hardvéru, pretože umožnenie používania aspoň niektorých konštrukcií jazyka C tak aby nebola znemožnená syntéza je potenciálne veľmi užitočné.

## 6.3 Model BlackFin

Implementácia modelu procesoru BlackFin bola poznamenaná opakovaným narážaním na limity jazyku ISAC, pretože išlo o vôbec prvý pokus popísať modernú architektúru s komplexnou štruktúrou a radou špecifických konštrukcií. Výhodou tejto situácie však bolo, že vznikol priestor na identifikáciu nedostatkov jazyka ISAC, a tiež príležitostí pre ich odstránenie.

Modelovanie zreťazenej linky inštrukcií prinieslo nutnosť udržiavať pre každú fázu linky samostatnú skupinu registrov. Nutnosť takýchto registrov je zrejmá. Rovnako je vítaná možnosť individuálne takéto registre definovať.

Z užívateľského hľadiska by však bolo príjemné mať k dispozícii prostriedok, ktorý by priniesol istú formu abstrakcie pre daný problém. Mechanizmus takéhoto riešenia by mohol byť inšpirovaný napríklad konceptom dátových štruktúr jazyka C. Užívateľ by potom mohol definovať a používať skupinu registrov jednorázovo, pričom transformáciu notácie do súčasnej podoby by bolo realizované až v dobe transformácie popisu.

Kódovanie inštrukcií procesoru je heterogénne. Formáty kódovania sú zdieľané len malým počtom inštrukcií. Aj napriek tomu, že boli využité všetky príslušné mechanizmy jazyka ISAC, nepodarilo sa dosiahnuť výraznú redukciu počtu operácií potrebných na reprezentáciu inštrukčnej sady. V tomto prípade však nie je vhodné hľadať príčinu vzniknutej situácie v nedostatkoch jazyka ISAC, avšak skôr v návrhu inštrukčnej sady.

Zásadným problémom, na ktorý bolo narazené a nepodarilo sa ho vyriešiť, sa ukázalo byť vytvorenie operácie reprezentujúcej jednu špecifickú skupinu inštrukcií. Sú to inštrukcie presunu, ktoré v rámci jedinej inštrukcie presunú obsahy akumulátorov A0, resp. A1, do spodnej, resp. vrchnej, časti nejakého všeobecného registru. Príklad takejto inštrukcie a znázornenie problému je možné vidieť na obrázku 6.9.

Problém spočíva v duplicitnej prítomnosti textovej reprezentácie všeobecného registra v syntaxi inštrukcie, ktorá je v obrázku 6.9 pomenovaná *dreg*. Použitý register je potom zakódovaný na jedinom mieste v operačnom kóde inštrukcie. Toto je konštrukcia, ktorú nie je možné zachytiť jazykom ISAC, pretože nezodpovedá vnútornému formálnemu modelu generátorov projektu Lissom (ktoré sú založené na dvojcestných párových automatoch [12]. Musí platiť, že žiaden inštanciovaný element ASSEMBLER sekcie sa musí vyskytovať v príslušnej

```
OPERATION move_dreg_lohi_a_01 {
    INSTANCE group_of_dregs ALIAS { dreg };

    // "R2_lo = A0, R2_hi = A1"
    ASSEMBLER { dreg~"_lo" "=" "A0" "," dreg~"_hi" "=" "A1" };

    CODING { 0b11000000000001110011100 dreg 0b000000 };
}
```

Obrázek 6.9: Ukážka kódovania inštrukce s dvojitým výskytom inštancie *dreg* v sekcii ASSEMBLER.

```
// dreg = sysreg_100
OPERATION move_dreg_stysreg_100 {
    CODING { 0b0011 0b000 0b100 dreg sysReg };
}

// dreg = sysreg_110
OPERATION move_dreg__sysreg_110 {
    CODING { 0b0011 0b000 0b110 dreg sysReg };
}

// dreg = sysreg_111
OPERATION move_dreg__sysreg_111 {
    CODING { 0b0011 0b000 0b111 dreg sysReg };
}
```

Obrázek 6.10: Tri operácie potrebné na modelovanie troch variánt systémových registrov.

CODING sekcii práve raz.

Snaha o implementáciu modelu inštrukčnej sady priniesla prvotný impulz pre doplnenie jazyka ISAC o mechanizmus roztrhnutia kódovania inštanciovaných operácií.

Kódovanie systémových registrov je zložené z dvoch častí - kódu typu registru a kódu identifikujúceho samotný register. Systémové registre sú z pohľadu kódovania rozdelené do troch skupín. Každú z týchto skupín je nutne modelovať samostatne (obrázok 6.10), pretože kód skupiny a kód registra sa nenachádzajú bezprostredne vedľa seba.

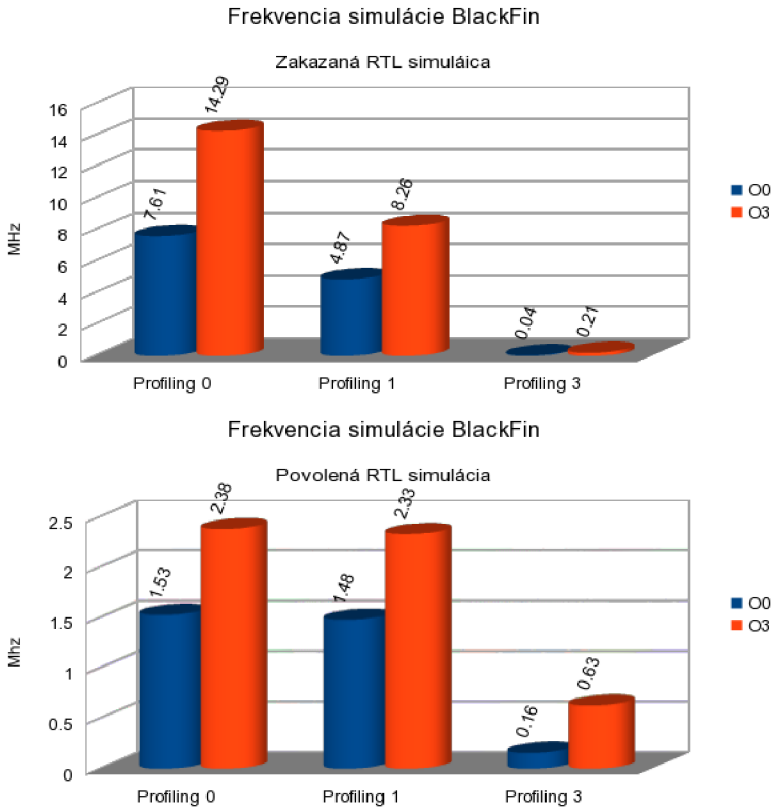
Ak by bolo kódovanie typu registra presunuté do operácie reprezentujúcej samotný register, potom by bolo možné pomocou roztrhnutia inštancie zlúčiť operácie do jednej, tak ako je to znázornené na obrázku 6.11, kde notácia [x..y] značí x-tý až y-tý bit a platí, že $x > y$.

Podobný princíp je aplikovateľný na viacerých miestach inštrukčnej sady a prispel by k zefektívneniu jej popisu.

Komplexná inštrukčná sada teda spôsobila zastavenie práce na implementácií modelu. Boli však identifikované niektoré slabosti a nedostatky jazyka, ktorých odstránenie bude rozhodne prínosom.

```
OPERATION move_dreg_sysreg {
    CODING { 0b0011 0b000 sysReg[5..3] 0b111 dreg sysReg[2..0] }
}
```

Obrázek 6.11: Reprezentácia všetkých variánt inštrukcií pomocou jedinej operácie.




Obrázek 6.12: Dopad povolenia RTL simulácie na frekvenciu. Zdroj B.3.

### 6.3.1 Simulácia modelu

Neúplnosť inštrukčnej sady znemožnila tvorbu nejakého zmysluplného programu. Z hľadiska simulácie to však problém nie je, pretože každá inštrukcia je chápaná rovnako. Preto bolo možné vytvoriť umelý program, ktorý opakoval vykonávanie nejakej inštrukcie.

Snahou simulácií bolo odhaliť vplyv povolenia RTL simulácie na celkovú frekvenciu simulácie. Dopad bol skúmaný s optimalizáciou kompilácie na nultej a tretej úrovni, pričom boli uvažované krajné úrovne profilácie. Výsledky simulácií sú znázornené na obrázku 6.12.

Z výsledkov je možné usúdiť, že povolenie RTL má zásadný vplyv na frekvenciu v prípade neaktivovania profilovania. Po povolení profilovaní sa váha RTL simulácie mierne vytráca.

### 6.3.2 Možnosti ďalšej práce

Od zastavenia prác na modeli procesoru BlackFin už v dobe písania tejto práce ubehol čas, počas ktorého bola vývojovým týmom projektu Lissom implementovaná nová funkcionalita do generátorov programovacích nástrojov. Jednou z nich je aj popísaný koncept roztrhnutia. Nadviazaním na túto prácu by mohlo byť navrátenie sa k tejto architektúre a prieskum možností, ktoré nová funkcionalita priniesla, s úmyslom vytvoriť model aspoň časti inštrukčnej sady.

## 6.4 Model STxP70

Pred implementáciou modelu procesoru STxP70 bolo venované zvýšené úsilie analýze inštrukčnej sady. Jej úmyslom bolo zoskupiť inštrukcie podľa ich dĺžok a formátov kódovania. Z predošlých skúseností s popisom inštrukčných sád bolo totiž jasné, že kódovanie inštrukcií je základným faktorom, ktorý ovplyvňuje spôsob akým sa bude dať navrhnúť štruktúra modelu a maximálne využiť dostupné konštrukcie jazyka ISAC.

Ukázalo sa, že toto úsilie bolo na mieste a prinieslo prehľadný náhľad do štruktúry samotnej inštrukčnej sady a tak umožnilo síce prácny ale priamočiary postup pri popise jednotlivých skupín inštrukcií.

Problematickým sa ukázala byť variabilná dĺžka inštrukcií. Faktory, ktoré ovplyvňujú dĺžku inštrukčného slova sú síce experimentálne identifikovateľné, avšak ich popis pomocou jazyka ISAC nie je možný. Mimo iné, v jazyku neexistuje spôsob akým zapísať, že inštrukcia má byť kódovaná 32-bitovým inštrukčným slovom ak je hodnota jej operandu menšia než nejaká iná hodnota.

Z dôvodu snahy o zachytenie čo najväčšej funkcionality boli teda modelované iba najdlhšie inštrukcie.

Doplnenie kratších inštrukcií do modelu však nebude triviálna úloha. Príčinou je hlavne zdieľanie syntaxe medzi krátkou a dlhou verziu danej inštrukcie. Navyše, model procesoru BlackFin ako jediný obsahuje implemntáciu zreťazenej linky spracovania inštrukcií. Zaujímavé bude porovnanie s rýchlosťami ostatnými simuláciami. Tento prístup nevyhovuje požiadavkám formálnych modelov generátorov nástrojov projektu Lissom. Jednotlivé verzie inštrukcií by museli byť odlíšené na syntaktickej úrovni. Tým by však odpadla požadovaná možnosť prekladu zdrojového kódu pomocou originálneho a generovaného prekladaču bez nutnosti jeho modifikácie.

Súčasný stav je teda taký, že modelované najdlhšie inštrukcie dokážu vyjadriť rovnakú funkcionalitu ako ich kratšie verzie. Generované programy sú však väčšie. Teoreticky je výstup generovaného asembleru kompatibilný s výstupom reálneho asembleru. Keďže krátke inštrukcie ostali neimplementované, pokus o simuláciu optimalizovaného kód pomocou generovaného simulátoru by stroskotal.

Súčasne s implementáciou modelu prebiehala implementácia mechanizmu roztrhnutia v jazyku ISAC. Model slúžil ako testovací príklad, na ktorom sa verifikovala správnosť implementácie. Bez spomenutého mechanizmu by vôbec nebolo možné modelovať niektoré inštrukcie.

Relatívne veľký počet inštrukcií spôsobil, že model sa zaradil medzi najrozsiahlejšie modely, ktoré boli kedy vytvorené. Novozavedený mechanizmus roztrhnutia zároveň priniesol zvýšené požiadavky na proces generovania nástrojov, ktorý sa stal prakticky nerealizovateľný, pretože sa narážalo na nedostatok operačnej pamäte a neúnosné dlhé trvanie celého procesu.

V neskorších fázach implementácie potom muselo byť postupované tak, že v modeli bola vždy aktívnych iba minimum inštrukcií. Tak bolo možné v prijateľnom čase vygenerovať príslušné nástroje a overiť správnosť modelu. Prácnou bola hlavne nutnosť modifikovať model pri každej zmene testovanej inštrukcie.

Počas písania tejto práce sú však finalizované práce na novom koncepte niektorých algoritmov spojených s generovaním nástrojov, ktoré by mali znížiť celkové nároky a model by sa tak mal stať prakticky použiteľným.

| Veľkosť zdrojových kódov | Zahrnuté inštrukice |
|---|---|
| 488 kB | bez predikátov |
| 1.2 MB | bez predikátov, pluspredikované G?& |
| 24 MB | bez predikátov, plus preikované G?& a G& |
| 572 MB | preedikovaná G?, ale bez inšt. s adresný mód |

Obrázek 6.13: Nárast veľkosti zdrojových kódov generovaných nástrojov projektu Lissom pre model STxP70.

Rozpracovaný je tiež systém pre extrahovanie sémantiky inštrukcií z modelu za účelom generovania cieleného prekladaču jazyka C. Správanie inštrukcií bolo popísané tak, aby maximálne vyhovovalo požiadavkám vtedajším požiadavkám extraktoru. Požiadavky extraktoru sa vo finálnej verzii pravdepodobne zmenia a pravdepodobne bude nutné revidovanie niektorých inštrukcií.

### 6.4.1 Simulácia modelu

Vytvorený model je špecifický v tom, že jeho veľkosť naráža na limity súčasných možností generátorov projektu Lissom. Súčasné generátory nezvládli vygenerovať také nástroje, aby bolo možné získať relevantné výsledky.

V tabuľke 6.13 sú uvedené veľkosti zdrojových kódov generovaných nástrojov. Snaha o kompilácie takýchto obrovských kódoch jednak trvala neúnosne dlho (rády hodín), narážala však aj na limity dostupnej operačnej pamäte.

### 6.4.2 Možnosti ďalšej práce

Ďalšia práca týkajúca sa tohto modelu by mala byť smerovaná k vytvoreniu podpory kratších verzii inštrukcií. Riešenie nemusí mať nutne podobu plného popisu týchto inštrukcií jazykom ISAC. Možným prístupom je doplnenie generátorov projektu Lissom, konkrétne asembleru, o schopnosť generovať optimalizované inštrukcie, a to napríklad za pomoci nejakých doplnkových informácii na úrovni modelu.

Iná možnosť pokračovania práce na tomto modeli smeruje k úspešnému vygenerovaniu prekladaču jazyka C. V tomto prípade je možné, že prispôsobenie popisu sémantiky inštrukcií nebude jediným krokom, ktorý bude treba vykonať.

## 6.5 Zhrnutie vlastností jazyka

Počas implementácie modelov vybraných procesorov boli nadobudnuté praktické skúsenosti s jazykom, ktoré sa týkajú hlavne vhodnosti jazyka pre popis danej architektúry. Nasledujúce charakteristické vlastnosti je možné považovať za kľúčové:

- možnosť tvoriť modely skupín inštrukcií pomocou kompozitného prístupu je esenciálna. Odstraňuje sa tak nutnosť vytvárať samostatnú operáciu pre každú inštrukciu s každou kombináciu operandov, prípadne iných parametrizovaných fragmentov,
- jedným z týchto mechanizmov je sekcia EXPRESSION. Tá však umožňuje prenos iba jednej jednoduchej hodnoty, prípadne hodnoty poskladanej logickými operáciami

na úrovni bitov. Užívateľsky príjemnejšie by bolo, ak by bolo možné touto cestou predávať štruktúrované dátové typy,

- podpora volania externých funkcií jazyka C z externých súborov odľahčuje zdrojový kód popisu architektúry. Existujú však situácie, kedy by bolo praktické pomocou návratovej hodnoty funkcie vrátiť štruktúrovaný dátový typ alebo použiť parametre predávané odkazom. Táto funkcionalita je však zakázaná,

- snaha o vygenerovanie popisu procesoru v jazyku pre popis hardvéru tak, aby bol tento popis reálne syntetizovateľný, v súčasnosti striktne definuje formu akou musí byť model navrhnutý a implementovaný.

Je však nutné skonštatovať, že samotná použiteľnosť jazyka ISAC na popis nejakej existujúcej architektúry však závisí na vlastnostiach danej architektúry:

- jednoduché architektúry je možné efektívne popísať v jazyku ISAC,

- zložitejšie architektúry občas obsahujú funkcionalitu, ktorá nie je opísateľná jazykom ISAC, stále je však možné dosiahnuť prakticky použiteľné výsledky,

- komplexné architektúry narážajú na limity jazyka a zvyčajne je možné popísať len určitú časť architektúry.

Samostatnú skupinu tvoria architektúry, ktoré sa nachádzajú vo fáze návrhu a hľadajú sa prostriedky pre ich realizáciu.

V tomto prípade je jazyk ISAC takmer ideálnym nástrojom, pretože umožňuje súčasný návrh a realizáciu jak hardvérovej časti architektúry a softvéru pre danú architektúru.

Opäť však platí, že hlavným meradlom vhodnosti je zložitosť architektúry. Odlišnosťou od tvorby modelov existujúcich architektúr je však možnosť lacnej zmeny parametrov architektúry v akejkoľvek fáze vývoja.

V neposlednom rade je tu možnosť použiť jazyk ISAC pre realizáciu len základu architektúry, ktorý bude slúžiť ako východisko pre konvenčný návrhový proces.

# Kapitola 7

# Záver

V rámci tejto práce bolo implementovaných niekoľko modelov procesorov používaných vo vnorených systémoch, čím bol vytvorený základ knižnice procesorov pre návrh vnorených systémov.

Prvým krokom bolo zoznámenie sa s konceptmi architektúr procesorov používaných vo vnorených systémov. Výstupom tejto kapitoly bolo zistenie, že reálne procesory často kombinujú viaceré prístupu a zvyčajne nie je možné presne zaradiť danú architektúru.

Nasledovalo zoznámenie s projektom Lissom a jazykom ISAC pre popis architektúr. Tento krok bol nevyhnutný kvôli získaniu zručnosti v používaní vývojového prostredia projektu Lissom.

Ďalším krokom bolo zoznámenie sa so zástupcami niektorých typov procesorov používaných vo vnorených systémoch. Bola analyzovaná štrukturálna stránka procesorov týkajúca sa hlavne prítomných registrov a organizácie pamäte. Dôraz bol kladený na identifikáciu tých aspektov architektúr, ktoré boli dôležité z hľadiska implementácie modelu na príslušnej úrovni popisu. Dôsledkom bolo získanie znalostí potrebných pre implementáciu modelov.

Potom nasledovala implementácia samotných modelov. Popis implementácie bol zameraný na zachovanie chronologického priebehu jednotlivých fáz implementácie a vysvetlenie postupov použitých pre vytvorenie modelu daných architektúr. Ďalším úmyslom tejto časti práce bolo vytvoriť súbor postupov použiteľných ako úvod do problematiky modelovania procesorov pomocou jazyka ISAC.

Po dokončení implementácie bolo učinené jej zhodnotenie, kde bolo zhodnotený jej dosiahnutý stav a prípadné praktické použitie modelu. Boli identifikované a objasnené vlastnosti jazyka, ktoré ho robia vhodným pre popis architektúr. Zároveň však boli objavené špecifiká architektúr, ktoré narážajú na limity jazyka a ich popis je neoptimálny alebo nemožný. Taktiež bolo vykonané celkové zhodnotenie vhodnosti jazyka a jeho vhodnosť na popis zvolených architektúr.

Prácu je možné zhrnúť ako úspešnú, pretože sa v jej rámci podarilo dosiahnuť praktické použitie modelov, poskytnúť niekoľko návrhov na rozšírenie jazyka ISAC a vytvoriť súbor modelov, ktoré sú kvalitne zdokumentované a môžu byť použité pre demonštračné a edukačné účely.

Priestor pre nadviazanie v práci je možné nájsť v rozšírení modelov o zatiaľ nemodelované elementy alebo doplniť o implementáciu modelov ďalších architektúr.

# Literatura


[1] Analog Devices: *Blackfin©Processor Programming Reference*. Analog Devices, Inc., Norwood, 2008.

[2] Chapman, K.: *KCPSM3 8-bit Micro Controller for Spartan-3, Virtex-II and Virtex-IIPRO*. Xilinx Ltd, 2003.

[3] Colwell, R.; Nix, R.; O'Donnell, J.; aj.: A VLIW architecture for a trace scheduling compiler. *Computers, IEEE Transactions on*, ročník 37, č. 8, aug 1988: s. 967 –979, ISSN 0018-9340, doi:10.1109/12.2247.

[4] Dutt, N.; Mishra, P.: *Processor description languages : applications and methodologies*. Boston: Morgan Kaufmann, 2008, iSBN 978-012-3742-872.

[5] Dvořák, V.; Drábek, V.: *Architektura procesorů*. Brno: VUTIUM, 1999, iSBN 80-214-1458-8.

[6] Georgiadis, P.: *Simulátor procesoru jako Java applet*. Diplomová práce, FIT VUT v Brně, Brno, 2006.

[7] Hruška, T.: *Manuál jazyka ISAC*. FIT VUT v Brně, Brno, 2004.

[8] Hruška, T.: *Manuál vývojového prostředí pro HW/SW Co-design*. FIT VUT v Brně, Brno, 2009.

[9] Hu, G.; Ma, J.; Huang, B.: Password Recovery for RAR Files Using CUDA. *Dependable, Autonomic and Secure Computing, IEEE International Symposium on*, ročník 0, 2009: s. 486–490, doi:http://doi.ieeecomputersociety.org/10.1109/DASC.2009.123.

[10] Intel Corporation: Hexadecimal Object File Format Specification. `http://microsym.com/editor/assets/intelhex.pdf`, 1988.

[11] Ličev, L.; Morkes, D.: *Procesory, architektura, funkce, použití. Kompletní průvodce procesory a souvisejícími hardwarovými komponentami*. Brno: Computer Press, první vydání, 1999, iSBN 80-722-6172-X.

[12] Masařík, K.: *Systém pro souběžný návrh technického a programového vybavení počítačů*. VUTIUM, Faculty of Information Technology BUT, první vydání, 2008, ISBN 978-80-214-3863-7, 156 s.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=8915

[13] Noergaard, T.: *Embedded systems architecture : a comprehensive guide for engineers and programmers*. Newnes, 2005, iSBN 978-075-0677-929.

[14] Novotný, T.: *Transformace popisného jazyka mikroprocesoru do jazyka pro popis hardware*. Diplomová práce, FIT VUT v Brně, Brno, 2007.

[15] Přikryl, Z.; Křoustek, J.; Hruška, T.; aj.: Fast Just-In-Time Translated Simulation for ASIP Design. In *14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, IEEE Computer Society, 2011, ISBN 978-1-4244-9753-9, s. 279–282.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9567

[16] STMicroelectronics: *STxP70-4 core and instruction set architecture*. 2010, interný dokument.

[17] Texas Instruments: *MSP430x2xx Family User's Guide*. Texas Instruments Incorporated, Dallas, 2008.

[18] Xilinx Inc.: *PicoBlaze 8-bit Embedded Microcontroller User Guide*. 2010.

# Dodatek A

# Obsah CD a návod na použitie

Na elektronickom nosiči sú uložené nasledujúce dáta. Prítomné sú nasledujúce zložky:

- *doc* - dodatočné dokumenty spojené s touto prácou prácou.
- *lissom@fitkit* - obsahuje kompletné a automatizované riešenie použitia nástrojov projektu Lissom s platformou FITKit. Balík požaduje prostredie operačného systému Windows. V balíku je obsiahnutá aj podrobná dokumentácia a návod na použitie,
- *models* - modely vytvorené c rámci tejto práce. Každý model obsahuje príklad spustiteľného kódu,
- *src* - zdrojové kódy textovej správy.

Použitie práce by malo prebiehať v týchto fázach:

- oboznámiť sa s prostredím projektu Lissom s pomocou dostupnej dokumentácie,
- získanie prostredia Eclipse z bežne dosutpných zdrojov,
- inštalácia zásuvných modulov prostredia Lissom podľa pokynov v relevanntnej dokumentácii,
- pripojenie k middlewaru a práca s modelmi.

# Dodatek B

# Merania frekvencie simulácií

| Optimalizácia | Profiling | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Priemer |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 4.38 | 4.54 | 4.21 | 4.6 | 4.19 | 4.22 | 4.55 | 4.38 |
| 0 | 2 | 2.81 | 2.69 | 2.54 | 2.94 | 2.71 | 2.69 | 2.69 | 2.72 |
| 0 | 3 | 0.61 | 0.73 | 0.57 | 0.64 | 0.58 | 0.55 | 0.61 | 0.61 |
| 0 | 4 | 0.21 | 0.33 | 0.25 | 0.17 | 0.23 | 0.29 | 0.16 | 0.23 |
| 1 | 0 | 12.01 | 11.89 | 12.32 | 11.94 | 12.01 | 11.94 | 12.02 | 12.02 |
| 1 | 1 | 7.6 | 7.31 | 7.02 | 6.84 | 7.31 | 7.22 | 7.22 | 7.22 |
| 1 | 2 | 1.4 | 1.22 | 1.39 | 1.43 | 1.37 | 1.32 | 1.48 | 1.37 |
| 1 | 3 | 0.67 | 0.61 | 0.64 | 0.64 | 0.69 | 0.64 | 0.59 | 0.64 |
| 2 | 0 | 15.41 | 15.5 | 15.43 | 15.29 | 15.42 | 15.4 | 15.41 | 15.41 |
| 2 | 1 | 10.23 | 9.94 | 10.15 | 10.23 | 10.15 | 10.19 | 10.16 | 10.15 |
| 2 | 2 | 2.23 | 2.6 | 2.17 | 1.94 | 2.09 | 2.1 | 1.84 | 2.14 |
| 2 | 3 | 1.2 | 0.7 | 0.79 | 0.8 | 0.6 | 0.72 | 0.8 | 0.80 |
| 3 | 0 | 16.74 | 16.43 | 16.21 | 16.33 | 16.33 | 16.49 | 16.55 | 16.44 |
| 3 | 1 | 10.83 | 10.27 | 10.24 | 10.12 | 10.56 | 10.38 | 10.42 | 10.40 |
| 3 | 2 | 2.24 | 1.98 | 2.19 | 2.03 | 1.89 | 2.15 | 2.03 | 2.07 |
| 3 | 3 | 1.14 | 0.94 | 1.02 | 0.99 | 1.01 | 0.94 | 1.01 | 1.01 |

Tabulka B.1: Namerané frekvencie interpretovaných simulácií modelu MSP430. Hodnoty sú v MHz.

| Optimalizácia | Profiling | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Priemer |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 0 | 22.34 | 22.1 | 22.21 | 22.03 | 22.41 | 22.19 | 22.21 | 22.21 |
| 3 | 1 | 6.03 | 5.95 | 6.01 | 6.24 | 6.41 | 6.22 | 6.23 | 6.16 |

Tabulka B.2: Namerané frekvencie kompilovaných simulácií modelu MSP430. Hodnoty sú v MHz.

| RLT | Optimalizácia | Profiling | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Priem. |
|---|---|---|---|---|---|---|---|---|---|---|
| nie | 0 | 0 | 7.64 | 7.59 | 7.41 | 7.89 | 7.57 | 7.5 | 7.64 | 7.61 |
| nie | 0 | 1 | 4.92 | 4.83 | 4.77 | 4.9 | 4.88 | 4.89 | 4.87 | 4.87 |
| nie | 0 | 3 | 0.04 | 0.06 | 0.05 | 0.04 | 0.03 | 0.04 | 0.04 | 0.04 |
| nie | 3 | 0 | 14.24 | 14.39 | 14.17 | 14.19 | 14.32 | 14.29 | 14.4 | 14.29 |
| nie | 3 | 1 | 8.26 | 8.27 | 8.24 | 8.28 | 8.28 | 8.32 | 8.19 | 8.26 |
| nie | 3 | 3 | 0.22 | 0.21 | 0.21 | 0.29 | 0.17 | 0.19 | 0.2 | 0.21 |
| áno | 0 | 0 | 1.53 | 1.55 | 1.52 | 1.53 | 1.54 | 1.54 | 1.52 | 1.53 |
| áno | 0 | 1 | 1.48 | 1.48 | 1.41 | 1.42 | 1.53 | 1.52 | 1.49 | 1.48 |
| áno | 0 | 3 | 0.14 | 0.15 | 0.16 | 0.16 | 0.17 | 0.17 | 0.16 | 0.16 |
| áno | 3 | 0 | 2.37 | 2.48 | 2.38 | 2.23 | 2.41 | 2.39 | 2.38 | 2.38 |
| áno | 3 | 1 | 2.54 | 2.33 | 2.35 | 2.21 | 2.27 | 2.33 | 2.28 | 2.33 |
| áno | 3 | 3 | 0.64 | 0.63 | 0.59 | 0.61 | 0.62 | 0.64 | 0.69 | 0.63 |

Tabulka B.3: Namerané frekvencie simulácií modelu BlackFin. Hodnoty sú v MHz.

| RLT | Optimalizácia | Profiling | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Priem. |
|---|---|---|---|---|---|---|---|---|---|---|
| nie | 0 | 0 | 12.35 | 12.46 | 12.04 | 11.97 | 12.33 | 12.25 | 12.27 | 12.24 |
| nie | 0 | 1 | 10.23 | 9.78 | 9.84 | 10.12 | 10.17 | 9.92 | 9.85 | 9.99 |
| nie | 0 | 3 | 0.53 | 0.51 | 0.55 | 0.49 | 0.52 | 0.51 | 0.53 | 0.52 |
| nie | 3 | 0 | 68.19 | 68.39 | 67.44 | 68.15 | 68.75 | 68.44 | 67.54 | 68.13 |
| nie | 3 | 1 | 48.65 | 48.12 | 48.79 | 48.13 | 48.61 | 48.43 | 48.46 | 48.46 |
| nie | 3 | 3 | 1.26 | 1.42 | 1.22 | 1.23 | 1.22 | 1.22 | 1.21 | 1.25 |
| áno | 0 | 0 | 3.32 | 3.24 | 3.29 | 3.31 | 3.29 | 3.29 | 3.27 | 3.29 |
| áno | 0 | 1 | 3.04 | 3.07 | 3.12 | 2.98 | 3.07 | 3.06 | 3.06 | 3.06 |
| áno | 0 | 3 | 0.34 | 0.27 | 0.21 | 0.19 | 0.27 | 0.24 | 0.25 | 0.25 |
| áno | 3 | 0 | 4.25 | 4.49 | 4.21 | 4.47 | 4.38 | 4.2 | 4.49 | 4.36 |
| áno | 3 | 1 | 4.43 | 4.47 | 4.35 | 4.42 | 4.49 | 4.43 | 4.38 | 4.42 |
| áno | 3 | 3 | 0.95 | 0.94 | 0.84 | 0.91 | 0.97 | 0.91 | 0.87 | 0.91 |

Tabulka B.4: Namerané frekvencie syntetizovateľného modelu PicoBlaze. Hodnoty sú v MHz.

| RLT | Optimalizácia | Profiling | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Priem. |
|---|---|---|---|---|---|---|---|---|---|---|
| nie | 3 | 0 | 27.94 | 28.94 | 28.45 | 28.44 | 29.47 | 28.99 | 29.13 | 28.77 |
| nie | 3 | 1 | 20.97 | 21.48 | 21.47 | 20.06 | 21.14 | 21.25 | 21.69 | 21.15 |
| nie | 3 | 3 | 1.91 | 1.94 | 1.87 | 1.91 | 1.92 | 1.94 | 1.87 | 1.91 |

Tabulka B.5: Namerané frekvencie nesyntetizovateľného modelu (model inštrukčnej sady) PicoBlaze. Hodnoty sú v MHz.