# VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

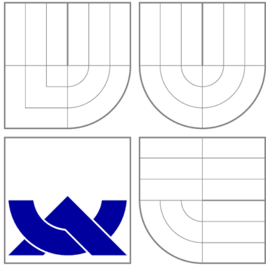

# VYTVOŘENÍ MODELU PROCESORU AVR32

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

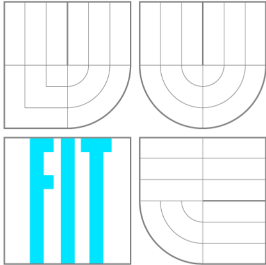

AUTOR PRÁCE — ROSTISLAV SARČÁK
AUTHOR



BRNO 2014




VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY


FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS


# VYTVOŘENÍ MODELU PROCESORU AVR32

AVR32 PROCESSOR MODEL CREATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS


AUTOR PRÁCE — ROSTISLAV SARČÁK
AUTHOR

VEDOUCÍ PRÁCE — Ing. KAREL MASAŘÍK, Ph.D.
SUPERVISOR


BRNO 2014

# Abstrakt

Tato práce se zabývá vytvořením instruction-accurate modelu procesoru AVR32 v jazyce CodAL pro popis architektur. V práci je popsána RISC architektura AVR32, způsob implementace modelu, testování a generování softwarových nástojů. Implementace modelu je realizována pomocí frameworku Codasip. V modelu je popsána instrukční sada procesoru. Výsledkem této práce je instruction-accurate model procesoru Atmel AVR32.



# Abstract

This barchelor's thesis describe creation of AVR32 processor instruction-accurate model using CodAL language. In this thesis RISC AVR32 architecture, approach to implementation of the model, testing and generating of software toolchain is described. Model development is realized in Codasip framework. Model contains implementation of AVR32 instruction set. The result of this work is AVR32 processor instruction-accurate model.



# Klíčová slova

model, ASIP, AVR32, Atmel, CodAL, framework Codasip

# Keywords

model, ASIP, AVR32, Atmel, CodAL, framework Codasip



# Citace

Rostislav Sarčák: Vytvoření modelu procesoru AVR32, bakalářská práce, Brno, FIT VUT v Brně, 2014

# Vytvoření modelu procesoru AVR32

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Karla Masaříka, Ph.D. V práci jsem uvedl veškeré literární zdroje, ze kterých jsem čerpal.

.......................
Rostislav Sarčák
20. května 2014

## Poděkování

Chtěl bych poděkovat vedoucímu mé bakalářské práce panu Ing. Karlu Masaříkovi, Ph.D. za rady a trpělivost při vedení mé práce. Dále bych chtěl poděkovat panu Ing. Adamu Husárovi za rady a pomoc při implementaci modelu.

© Rostislav Sarčák, 2014.
*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.*

# Obsah

**1 Úvod** **2**

**2 Metodologie vývoje ASIP procesorů** **3**
2.1 Tradiční způsob vývoje . . . 3
2.2 Použití jazyků ADL . . . 3
2.2.1 Behaviorální jazyky . . . 4
2.2.2 Strukturální jazyky . . . 4
2.2.3 Smíšené jazyky . . . 5

**3 Codasip Framework** **6**
3.1 Jazyk CodAL . . . 6
3.1.1 Syntaxe jazyka CodAL . . . 7

**4 Mikroprocesor AVR32** **10**
4.1 Vlastnosti architektury AVR32 . . . 10
4.2 Programový model . . . 10
4.3 Instrukční sada . . . 12
4.3.1 Kategorie instrukcí . . . 12
4.3.2 Podmíněné instrukce . . . 14
4.3.3 Ostatní rozšíření . . . 15

**5 Reprezentace AVR32 v jazyce CodAL** **16**
5.1 Definice zdrojů . . . 16
5.2 Popis základních událostí . . . 17
5.3 Instrukční sada . . . 18
5.3.1 Implementace instrukcí . . . 18

**6 Testování** **20**
6.1 Testování instrukcí . . . 20
6.2 Testovací sada programů . . . 20
6.2.1 Automatizace testování . . . 21

**7 Generování softwarové sady nástrojů** **22**
7.1 Porovnání výkonnosti překladače . . . 23

**8 Závěr** **25**

**A Obsah CD** **27**

# Kapitola 1

# Úvod

V dnešní době, kdy elektronika ovlivňuje čím dál více naše životy, vznikají nové produkty, které častěji obsahují *systém na čipu*[1] nebo jednoúčelové věstavné systémy. Ať už se jedná o systémy zpracovávající relativně jednoduché řešení, až po ty komplexní, které vykonávají kritické úlohy v realném čase.

Tato zařízení však musí splňovat základní podmínky - malá velikost, malá spotřeba energie společně s vysokým výkonem. Podmínky můžeme splnit, pokud vyvineme procesor specifický pro danou úlohu. *Aplikačně specifické procesory* umožňují uspokojit tyto podmínky. Tyto procesory poskytují adekvátní poměr mezi flexibilitou standardních procesrů a výkonem specializovaných integrovaných obvodů. Nicméně vývoj je časově i cenově náročný. Čas a cenu můžeme snížit, pokud použijeme specializované vývojové nástroje. Mezi tyto nástroje se řadí i *Codasip framework*[1], jenž je vyvíjen výzkumnou skupinou *Lissom* na Fakultě informačních technlogií Vysokého učení technického v Brně.

Kapitola 2 popisuje základy a způsoby návrhu hardwaru. Zároveň se čtenář seznámí s jazyky ADL pro popis architektur. Následující kapitola 3 přibližuje možnosti vývojového prostředí *Codasip* a jazyku *CodAL* související s touto platformou. V kapitole 4 je čtenář seznámen s architekturou *AVR32*, jeho instrukční sadou, programovým modelem a dalšími vlastnostmi tohoto mikroprocesoru. Kapitola 5 se zaměřuje na implementaci mikroprocesoru v jazyce *CodAL*. V kapitole 6 je popsána metodika testování vytvořené architektury. Jedná se o porovnání modelu s funkčností realného mikroprocesoru *AVR32*. Kapitola 7 popisuje generování sady nástrojů.

[1]SoC - System on a Chip

# Kapitola 2

# Metodologie vývoje ASIP procesorů

## 2.1 Tradiční způsob vývoje

Tradiční způsob vývoje *ASIP procesorů* se skládá ze čtyř etap. Konkrétně se jedná o průzkum cílového řešení, implementaci architektury, návrh aplikace a integrace systému[2][3]. V první etapě je vyhodnocována aplikace, která bude následně používána ve vytvářené architektuře procesoru. Musí se nalézt kritické části aplikace, které vyžadují podporu v hardwaru nebo instrukční sadě. Pro každou iteraci této etapy vývoje je zapotřebí sada vývojových nástrojů podporující vyvíjenou architekturu. Avšak s každou změnou v návrhu je zapotřebí nová sada nástrojů.

Po vytvoření a odladění instrukční sady nastává etapa implementace architektury. Navržený procesor je převeden do hardwarového modelu implementovaného v jazyce popisující hardware (HDL jazyky). Nejčastěji se jedná o jazyk *VHDL*, *Verilog* nebo *SystemC*.

Třetí etapa se zabývá vývojem softwarové části *ASIP procesoru*. Během této etapy se musí řešit rozdíly v požadavcích na vývojový software, jelikož vývojář aplikace klade důraz na rychlost simulace, kdežto vývojář procesoru požaduje podporu *cycle-based* simulace, která je přesná, ale velmi pomalá. Z toho důvodu v etapě navíc často dochází ke stavu, kdy musí být manuálně přepracována cela sada vývojových nástrojů.

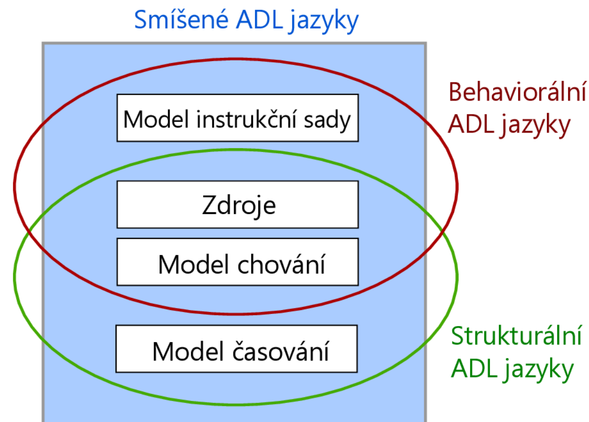
Poslední etapou je integrace do cílového systému a verifikace. Pro procesor musí být vytvořeno rozhraní pro integraci do cílového *SoC*. Následuje samotná integrace. Celý *SoC* je tak připraven pro simulaci a verifikaci. Jednotlivé komponenty cílového systému mohou navíc ovlivnit výsledný procesor, což dává přesnější výsledky než *instruction-accurate* simulátor. Úpravy *SoC*u musí být prováděny manuálně s každou změnou architektury.

Jednotlivé etapy jsou prováděny postupně a vývoj je rozdělen do několika týmů, kde figurují zkušení odborníci. Automatizace návrhu je omezena na jednotlivé etapy vývoje. Nicméně tento proces často nevede k nalezení optimálního řešení. Důvody jsou nedostatek času, chybějící nástroje nebo nekonzistence mezi hardwarem a softwarem.

## 2.2 Použití jazyků ADL

Na rozdíl od tradičního způsobu vývoje aplikačně specifických procesorů, jazyky *ADL* poskytují vyšší úroveň abstrakce. Tato vlastnost umožňuje navrhnout vysokoúrovňový model, který obsahuje popis instrukční sady, strukturu a chování architektury.

Jedním z hlavních důvodů použití *ADL* jazyků je schopnost na základě modelu automaticky generovat nástroje (assembler, simulátor, debugger nebo verifikátor). Tyto nástroje umožňují vývojářům přeložit a odsimulovat aplikace, které budou v budoucnu nasazené na aktuálně vyvíjené architektuře. Výsledky simulace tvoří data, která jsou použita jako zpětná vazba pro následující cyklus vývoje. Na základě těchto dat lze upravit instrukční sadu nebo algoritmy aplikací. Pokročilé *ADL* jazyky umožňují generovat syntetizovatelný hardwarový model. Tato schopnost umožňuje odstranit problém tradičního vývoje, který spočívá v nekonzistensci vývojových nástrojů a hardwarového modelu v rozdílných iteracích vývoje.




Obrázek 2.1: Dělení ADL jazyků (Zdroj [4])

### 2.2.1 Behaviorální jazyky

Tato skupina *ADL* jazyků se zaměřuje na instruční sadu. Instrukce jsou popsány binární a textovou formou. Popis instrukce obsahuje sémantickou sekci, která specifikuje chování dané instrukce. Vlastností této skupiny jazyků je schopnost generovat překladače pro jazyky vyšší úrovně. Naopak nepodporují *cycle-accurate* modely a syntézu, jelikož v modelu se nenachází abstrakce pro popis hardwarové mikroarchitektury. Příkladem budiž jazyk *ISDL* nebo *nML*[5].

### 2.2.2 Strukturální jazyky

Druhou skupinou *ADL* jazyků jsou strukturální jazyky, které se zabývají vývojem mikroarchitektury, propojením a strukturou jednotlivých komponent. Je však složité nalézt řešení zachycující vlastnosti různých procesorů. Běžným způsobem řešení tohoto problému je snížení abstrakce popisu modelu. Častou úrovni abstrakce je popis na úrovni přenosů mezi registry. Takto lze detailněji zachytit konktrétní struktury a informace o modelované architektuře.

Strukturální *ADL* jazyky jsou vhodné pro simulaci *cycle-accurate* modelů a hardwarovou syntézu. Jejich nevýhodou je časová náročnost na simulaci a zároveň je omezena podpora generování překladačů pro jazyky vyšší úrovně. Jazyk *MIMOLA*[5] se řadí mezi příklady této skupiny jazyků.

### 2.2.3 Smíšené jazyky

Poslední skupina *ADL* jazyků spojuje výhody behaviorálních a strukturálních *ADL* jazyků. Umožňují automaticky generovat nástroje pro vývoj a simulaci *ASIP* a zároveň dokážou syntézovat model do hardwarové podoby. Mezi reprezentanty těchto jazyků patří *EXPRESSION*[6], *ADL++*[5], *LISA*[7] a také *CodAL*[8].

# Kapitola 3

# Codasip Framework

*Codasip Framework* poskytuje designerům *ASIP* a *SoC* produktů komplexní vývojové prostředí pro vývoj za nejkratší možný čas. Snížení potřebného času je dosaženo automatizací úkolů a generování nástrojů, které by designéři museli vytvořit manuálně[9].

Framework je rozdělen na tři základní vrstvy - prezenční, middleware a simulační vrstva. Tyto vrstvy komunikují prostřednictvím *TCP/IP* protokolu. Toto uspořadání umožňuje běžet každé vrstvě na různých koncových bodech v síti.

**Prezenční** vrstva poskytuje uživatelům rozhraní pro vrstvu Middleware. Reprezentuje ji textová verze ve formě textového terminálu a grafické rozhraní *Codasip Studio*. Textová verze je vhodná k použití pokročilých technik vývoje, jako je skriptování a automatické testování. Grafické rozhraní, které je integrováno do IDE platformy *Eclipse*, přináší vhodnější formu pro vlastní vývoj cílového produktu.

**Middleware** vrstva zpracovává příkazy z prezenční vrstvy, které na jejich základě tyto příkazy vyhodnocuje a následně vykonává. Přikazy mohou obsahovat pokyn pro generování sady nástrojů, začátek simulace nebo vykonání kroku simulace. Zároveň tato vrstva poskytuje prezenční vrstvě informace o stavu dané úlohy a případných chybách. Dále je rovněž zodpovědná za řízení simulační vrstvy a předávání příkazů z vrstvy prezenční.

**Simulační** vrstva má za úkol provádět simulace nad jednotlivými jádry vyvíjeného systému. Při multiprocesorové simulaci může být použito rozdílných typů simulátorů (interpretované nebo kompilované), které mezi sebou komunikují prostřednictvím sdílených prostředků.

## 3.1 Jazyk CodAL

Pro popis *ASIP* modelů je ve frameworku použit jazyk *CodAL*, který je vyvíjen společně s frameworkem. Tento jazyk se řadí mezi smíšené *ADL* jazyky a je inspirován jazykem *LISA*. Jazyk *CodAL* je oproti jazyku *LISA* hiearchický a vysoce strukturovaný *ADL* jazyk, který se používá pro modelování procesorů na vysoké úrovni abstrakce. Společně s generátory nástrojů pro vývoj softwaru a syntézu hardwaru je vysoký stupeň abstrakce klíčem k rychlému a přesnému prototypování *RISC*, *CISC* a *VLIW* architektur. Tato kapitola vychází z manuálnu pro jazyk CodAL[8]. Jazyk *CodAL* podporuje modelování ve dvou stupních:

- **Instruction-accurate model** - Popisuje instrukční sadu vyvíjené architektury. Tento model neobsahuje žádný popis architektury. Používá se v dřívějších stádiích vývoje. Z tohoto typu modelu lze vygenerovat nástroje pro podporu vývoje aplikací a instrukční

sady (assembler, kompilátor, simulátor). Pokud je vývoj instruction-accurate modelu ustálený, může se začít vyvíjet mikroarchitektura cílového procesoru.

- **Cycle-accurate model**- Tento typ modelu umožňuje popsat časování a propojení jednotlivých komponent architektury. V tomto stupni vývoje lze provést syntézu modelu do hardwarové podoby.

### 3.1.1 Syntaxe jazyka CodAL

I přestože architektury dnešních procesoru jsou často odlišné, sdílejí společné základy. Použití instrukcí jako příkaz pro provedení operace nad daty, práce s daty na úrovni registrů nebo na úrovni paměti typu RAM.

Pro úspěšný překlad musí model obsahovat následující části:

- Hlavička modelu,
- popis zdrojů,
- popis instrukcí a událostí.

#### Hlavička modelu

Hlavička modelu obsahuje specifikaci typu modelu (instruction-accurate nebo cycle-accurate). Dále umožňuje konfigurovat vlastnosti generovaného assembleru.

#### Popis zdrojů

V popisu zdrojů jsou definovány základní prvky architektury. Vyskytuje se zde popis registrů, rozhraní nebo definovaní adresového prostoru.

Tato sekce však vždy musí obsahovat programový čítač, rozhraní pro přístup k vnější paměti a specifikaci adresového prostoru.

```
program_counter bit[32] pc;

interface dbus {
        endianness = "little";
        flags = {"rw"};
        type = "clb:master";
        bits = {32, 32, 8};
};

address_space as_all {
        type = "all";
        bits = {32, 32, 8};
        interfaces = {fetch, dbus};
}
```

3.1: Ukázka popisu zdrojů

### Popis instrukční sady

Popis instrukční sady se provádí zapomoci bloků `element` a `set`. Blok `set` reprezentuje množinu elementů nebo jiných množin. Blok `element` tvoří zakladní stavební blok pro popis instrukční sady. Vývoj instrukční sady se může odvíjet různými směry, kdy jeden element reprezentuje jednu instrukci. Tento přístup je však časově náročný, jelikož se často musí popisovat více formátů jedné instrukce. Proto se častěji volí přístup, kde instrukce se společnými vlastnostmi (operandy, binární reprezentace) sjednocují do jednoho elementu.

Blok `element` se skládá z dalších sekcí. Sekce `assembler` reprezentuje textovou podobu elementu, `binary` udává číslicovou reprezntaci a `return` udává návratovou hodnotu. V případě chování hraje důležitou roli sekce `semantics`, která udává chování instrukce. V sekci `semantics` je použita redukovaná verze jazyka *C*. Tato verze neumožňuje používat ukazatele, výčtové typy a struktury. Dále je zakázán příkaz goto. Avšak je povoleno vytvářet vlastní funkce a pro potřeby simulace používat funkce ze standardní knihovny jazyka *C*.

```
arch register bit[8] gpreg[2..7]; //registry architektury

element instr_nop       //definice jednoduché instrukce
{
        assembler { "NOP" };
        binary { 0:8 };
}

element instr_add
{
        use gpreg as src, dst;
        assembler { "ADD" dst "," src }; //textová forma
        binary { 0b11 dst src}; //binární reprezentace
        semantics //popis chování instrukce
        {
                signed char result;
                result = gpreg[src] + gpreg[dst];
                gpreg[dst] = result;
                printf("##ADD, result: %d\n", result); //informační výpis
        };
}

set instrset = instr_nop, instr_add;
```

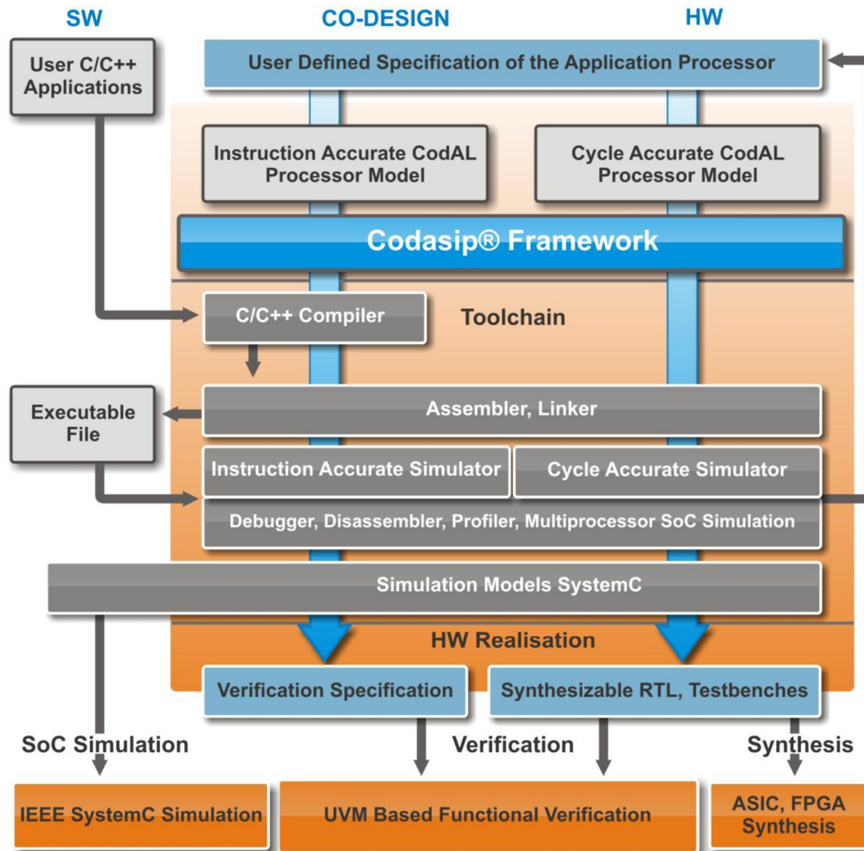
3.2: Ukázka popisu instrukční sady

### Popis událostí

Událost je způsobem, jak popsat základní chování procesoru. Model procesoru musí povinně obsahovat události `Main`, `Reset` a `Halt`. Událost `Main` popisuje co má model vykonávat v každém hodinovém cyklu. Událost `Reset` a `Halt` popisuje chování při resetování nebo zastavení běhu procesoru, jak vyplývá z názvu. Sémantická sekce události `Reset` a `Halt` mohou být prázdné, ale musí být přítomny.

```
event main
{
        use instrset;   // používána množina elementů
        start {{instructions;}};
        decoders (pc) {{instructions(ir);}}; //dekodování instrukce
        semantics { ir = dbus[pc]; }; //načtení instrukce z paměti
}
```

3.3: Ukázka události Main




Obrázek 3.1: Schéma generování nástrojů na platformě Codasip Framework (Zdroj [8])

# Kapitola 4

# Mikroprocesor AVR32

Jádro *AVR32* patří od roku 2006 do rodiny mikrokontrolérů a mikroprocesorů firmy *Atmel*, která zahrnuje i hojně využívané 8-bitové mikrokontroléry *AVR*. Jádro přichází s novou 32-bitovou load/store *RISC* architekturou. Tyto mikroprocesory jsou určeny pro využití ve všech spektrech věstavných systémů.

## 4.1 Vlastnosti architektury AVR32

Architektura *AVR32* obsahuje 16 32-bitových registů pro obecné použití. V závislosti na typu procesoru obsahuje podporu *DSP* a *SIMD* instrukcí. Achitektura se dělí na dvě rozdílné mikroarchitektury. Každá je zaměřena na různé cílové aplikace[10].

Mikroarchitektura *AVR32B* se používá v případech, kde latence aplikací je velice důležitá (real-time systémy). Aplikace využívá registrů pro přerušení, tudíž odpadává nutnost načítat data z paměti a obsluha přerušení se může začít ihned vykonávat. Vývoj a podpora této mikroarchitektury byla ukončena v dubnu roku 2012.

Naopak mikroarchitektura *AVR32A* se zaměřuje na cenu a použití pro low-end aplikace. Cena je snížená odstraněním registů pro přerušení. Tím je dosaženo zmenšení plochy výsledného čipu, ale zároveň zvýšení latence aplikace, jelikož se místo registrů používá paměť. Proto se mikroprocesor založený na této mikroarchitektuře využívá častěji jako 32-bitový mikrokontrolér věstavných systémů. Tuto mikroarchitekturu reprezentují jádra *AVR32 UC3 A(B,C,L)*.

## 4.2 Programový model

Mikroprocesor *AVR32* podporuje operaci s velikostí 8 bitů (byte), 16 bitů (halfword), 32-bitů (word) a 64-bitů (double word). Jestliže jsou data znaménkového typu, tak se pro záporná čísla se používá dvojkový doplněk.

Data jsou ukladána ve formátu big-endian, kdy nejvíce významný bit se uloží na nižší adresu. Nicméně existuje podpora ve formě instrukcí pro konverzi little-endian formátu. Některé instrukce používají dvojslovo jako operátor. Data se pak ukládají do dvou následujících registrů kromě poslední dvojice (R15:R14).

### Soubor registrů

Soubor registrů se skládá z 16 32-bitových registrů. 13 registrů je určeno pro všeobecné použítí. Registry stack pointer, link register a program counter jsou namapovány do tohoto

souboru, tudíž může sloužit jako zdrojový a cílový registr. Registr R12 je navržen pro ukládání návratové hodnoty z volání funkcí.

Program counter obsahuje adresu instrukce, která je zpracována. Nejmenší adresovatelná jednotka je bajt. Všechny instrukce musí být zarovnány k 2 - bajtové hranici. V případě, že je program counter použit jako cílový registr instrukce, chová se daná instrukce jako instrukce skoku.

Link Register obsahuje adresu pro návrat z podprogramu. V době volání podprogramu je do tohoto registru uložena hodnota následující instrukce. Návrat z programu může být potom proveden instrukcí `MOV`. Registr R12 je určen k uchovávání návratové hodnoty.

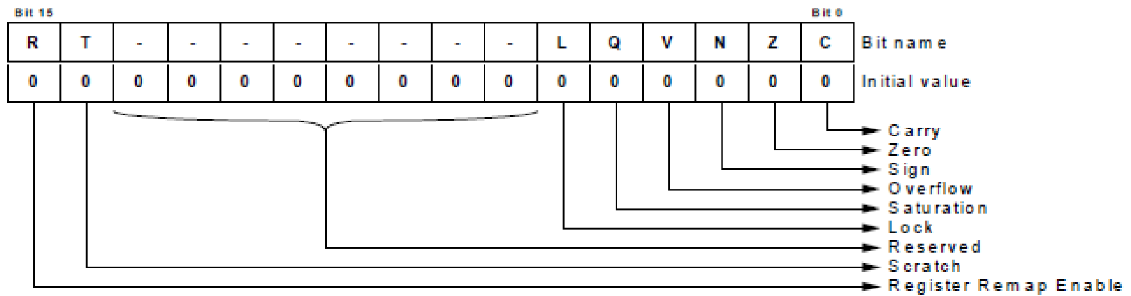
| Application | Supervisor | INT0 | INT1 | INT2 | INT3 | Exception | NMI |
|---|---|---|---|---|---|---|---|
| Bit 31 Bit 0 | Bit 31 Bit 0 | Bit 31 Bit 0 | Bit 31 Bit 0 | Bit 31 Bit 0 | Bit 31 Bit 0 | Bit 31 Bit 0 | Bit 31 Bit 0 |
| PC | PC | PC | PC | PC | PC | PC | PC |
| LR | LR | LR / LR_INT0 | LR / LR_INT1 | LR / LR_INT2 | LR / LR_INT3 | LR | LR |
| SP_APP | SP_SYS | SP_SYS | SP_SYS | SP_SYS | SP_SYS | SP_SYS | SP_SYS |
| R12 | R12 | banked registers (implementation defined) | banked registers (implementation defined) | banked registers (implementation defined) | banked registers (implementation defined) | R12 | R12 |
| R11 | R11 | | | | | R11 | R11 |
| R10 | R10 | | | | | R10 | R10 |
| R9 | R9 | | | | | R9 | R9 |
| R8 | R8 | | | | | R8 | R8 |
| R7 | R7 | | | | | R7 | R7 |
| R6 | R6 | | | | | R6 | R6 |
| R5 | R5 | | | | | R5 | R5 |
| R4 | R4 | | | | | R4 | R4 |
| R3 | R3 | | | | | R3 | R3 |
| R2 | R2 | | | | | R2 | R2 |
| R1 | R1 | | | | | R1 | R1 |
| R0 | R0 | | | | | R0 | R0 |
| SR | SR | SR | SR | SR | SR | SR | SR |
| | RSR_SUP | RSR_INT0 | RSR_INT1 | RSR_INT2 | RSR_INT3 | RSR_EX | RSR_NMI |
| | RAR_SUP | RAR_INT0 | RAR_INT1 | RAR_INT2 | RAR_INT3 | RAR_EX | RAR_NMI |

Obrázek 4.1: Registrový soubor procesoru AVR32 (zdroj [10])

### Status registr

Status registr se dělí dvě půlslova. Bity 31-16 informují o aktuáním nastavení a stavu procesoru, ve kterém se nachází. Přístup k těmto bitům lze provádět pouze v provilegovaném režimu. Spodní bity obsahují následující příznaky:

- **C** - Carry - udává přenos vrchního bitu.
- **Z** - Zero - pokud je výsledek roven nule, je tento příznak nastaven.
- **N** - Sign - příznak je nastaven, pokud je výsledek záporný.
- **V** - Overflow - příznak je nastaven, pokud při operaci dojde k přetečení
- **Q** - Saturation - příznak je nastaven, pokud při použití saturační aritmetiky dojde k přetečení. Tento příznak lze vynulovat pouze manuálně.




Obrázek 4.2: Bity 15 - 0 stavového registru (zdroj [10])

## 4.3 Instrukční sada

Instrukční sada obsahuje dva základní formáty instrukcí. Kompaktní 16-bitové nebo rozšířené 32-bitové. Zatímco kompaktní instrukce pokrývají většinu instrukcí, rozšířené popisují instrukce s více nebo složitějšími operátory. Například instrukce s konstantou mívají kompaktní i rozšířenou verzi, kdy na základě velikosti konstanty lze vybrat vhodný formát.

Často využívané instrukce (`ADD`, `SUB`) mohou mít dva nebo tři operandy. Tyto formáty instrukcí se mohou v kódu volně kombinovat. Výběrem vhodných instrukcí lze dosáhnout optimalizace na velikost nebo rychlost výsledného strojového kódu.

Nevýhoda této instrukční sady spočívá ve velikosti celé instrukční sady. Pro většinu procesoru je instrukce reprezentována operačním kódem, který tvoří část binární reprezentace instrukce. Procesor *TI MSP430* dovoluje používat pro jeden operační kód více typů operandů (registr, ukazatel do paměti)[11]. U mikroprocesoru *AVR32* je typ operandů pevně svázán s operačním kódem. Tudíž pět formátů tvoři pět rozdílných operačních kódů.

Instrukční sada pokrývá pouze oblast výpočtu nad celými čísly. Operace nad desetinými čísly je emulovaná pomocí knihoven. Hardwarovou podporu přináší až třetí revize této architektury[12]. Konkrétně se jedná o architekturu *AVR32 UC3C*.

### 4.3.1 Kategorie instrukcí

Celá instrukční seda se dá rozdělit do následujících základních kategorií, které jsou popsány následovně.

#### Aritmetické instrukce

Kategorie zahrnuje základní aritmetické instrukce jako je sčítání, odčítání, maximum a minimum. Kromě toho tato kategorie obsahuje i instrukce porovnání nebo absolutní hodnoty.

Do skupiny aritmetických instrukcí se dále řadí operace dělení. Operandy dělení jsou vždy 32-bitové. Výsledek je však 64-bitové číslo, kdy v horním slově je uložen zbytek po dělení a v dolním výsledek operace dělení.

#### Instrukce pro podporu násobení

Procesor podporuje znaménkové a bezznámenkové násobení, násobení se střadáním. Výsledek násobení může být 32 nebo 64-bitové číslo.

Pro operace násobení je vyhrazena MUL jednotka. Tato jednotka umožňuje násobení v 1 cyklu procesoru. V případě, kdy výsledek má být 64-bitový je zapotřebí 2 cyklů.

### Bitové operace

Bitové instrukce obsahují podporu pro extrakci bitů (`BFEXTS`, `BFEXTU`). Tyto instrukce umožňují vyjmout určitý počet bitů, které následně rozšíří na slovo. Instrukce `CAST` provádí znaménkové i bezznaménkové rozšíření bajtu popř. půlslova na slovo. Výměnu znaků a půlslova uvnitř slova je podporována instrukcí `SWAP`. Dále lze v této množině nalézt instrukce podporující operace na úrovni jednotlivých bitů (`BST`, `CBR`).

### Logické operace

Logický součin (`AND`, `TST`), logický součet (`OR`) a exkluzivní součet (`EOR`) se i zde řadí mezi standardně podporované operace. Tyto instrukce existují ve formě, která pracuje s horním nebo spodním půlslovem.

### Řízení programu

Pro řízení běhu programu existují v instrukční sadě instrukce skoku, volání podprogramu a návrat z podprogramu. Skoky jsou prováděny prostřednictvím instrukcí `RJMP` a `BR`. Instrukce `RJMP` provádí nepodmíněně relativní skok. Instrukce `BRcond` vykoná skok pouze, pokud je splněna podmínka cond.

Při volání podprogramu je adresa programového čítače uložena do registru LR. Obsluhu volání provádí instrukce `ACALL, ICALL, MCALL` a `RCALL` v závislosti na typu volání. Návrat z podprogramu je vykonáván instrukcí `RETcond`. Jak vyplývá z formátu instrukce, návrat může být podmíněný.

Pro řízení programu mohou být použity i ostatní instrukce, které umožňují modifikovat obsah programového čítače. Pro návrat z podprogramu může být například použita instrukce `MOV` ve tvaru `MOV pc, lr`.

### Přesun dat

Přesun dat je prováděn instrukcí `MOV`. Cílovým operandem této instrukce je vždy registr. Zdrojový operand může být registr nebo konstanta. V instrukční sadě existuje i podmíněná podoba této instrukce (`MOVcond`).

### Instrukce pro práci s pamětí

Přesun hodnot mezi pamětí a registry lze provádět pouze touto skupinou instrukcí. Načítání z paměti je prováděno instrukcí `LD`, ukládání instrukcí `ST`. Velikost dat, se kterými chceme pracovat lze specifikovat příponou. Instrukce podporuje zpracovávání všech velikostí dat (.B - byte, .H - halfword, .W - word, .D - doubleword).

K adresaci dat v paměti lze využít jeden z následujících čtyř adresových módů:

- **Post-increment** - Po provedení instrukce je ukazatel inkrementován o velikost dat.
- **Pre-decrement** - Před provedení instrukce je ukazatel dekrementován o velikost dat.
- **Displacement** - Výsledná adresa je dána součtem obsahu registru a hodnotou posunu.

- **Indexováný** - Hodnota ukazatele je dána bázovou adresou a indexem.

#### Instrukce pro práci s více registry

Kromě instrukcí pro práci s pamětí se v instrukční sadě nacházejí instrukce podporující práci s více registry najednou. Konkrétně se jedná o instrukce `POPM` a `PUSHM`, které jsou určeny slouží pro práci se zásobníkem. Běžný přístup k paměti zastupuje dvojice `LDM` a `STM`. Použití těchto instrukcí napomáhá vytvářet efektivnější pro zpracovánání většího objemu dat. Jako příklad lze uvést blokové kopírování paměti.

#### Read-modify-write instrukce

Že se jedná o *load/store* architekturu lze poznat z velkého počtu instrukcí patřící do skupiny pro práci s pamětí. Architektura AVR32 obsahuje podporu pro práci s paměti na úrovni jednotlivých bitů. Instrukce `MEMC` nastavuje daný bit v paměti na hodnotu 0, `MEMS` na hodnotu 1 a `MEMT` provádí negaci hodnoty bitu. Během provádění těchto instrukcí nelze vyvolat přerušení, proto jsou vhodné jako základ atomických operací.

#### Řízení procesoru

Tyto instrukce jsou používány hlavně pro modifikování obsahu systémových registru, pomocí kterých je nastavováno chování procesoru. Instrukce zároveň přístup k hornímu půlslovu status registru. Instrukce jsou přístupné pouze v privilegovaném režimu.

#### DSP instrukce

Rozšíření DSP poskytuje vývojáři podporu saturační aritmetiky. Tento druh aritmetiky je hojně využíván v oblasti zpracování digitálního signálu, multimediálních aplikací.

#### Instrukce pro práci s plovoucí desetinnou čárkou

Jak již bylo zmíněno v kapitole 4.3, třetí verze architektury přináší hardwarovou podporu pro operace nad čísly s plovoucí desetinnou čárkou[1]. Operace však lze provádět pouze nad jednoduchou přesností, tj. datovým typem float. Pro typ double je stále nutné operace emulovat zapomoci softwarových knihoven.

### 4.3.2 Podmíněné instrukce

Kromě instrukcí pro řízení programu obsahující příponu, která udává typ podmínky, existuje v instrukční sadě podpora podmíněného provedení instrukce pro nejčastěji využívané výpočetní operace. Zapomocí těchto instrukcí lze snížit velikost výsledného kódu a zároveň zvýšit efektivitu programu. Mezi těmito instrukcemi lze nalézt podmíněné sčítání, odčítání, základní logické operace (`AND, OR, XOR`) a podmíněné operace s pamětí. Seznam použitelných podmínek lze nalézt v následující tabulce 4.1.

[1] Dle standardu IEEE 754.

| Význam | Použita přípona | Výraz | B/Z |
|---|---|---|---|
| Rovnost | eq | $Z$ | B/Z |
| Nerovnost | ne | $\overline{Z}$ | B/Z |
| Menší než | cs, lo | $C$ | B |
| Menší nebo rovno | ls | $C \vee Z$ | B |
| Větší než | hi | $\overline{C \vee Z}$ | B |
| Větší nebo rovno | cc, hs | $\overline{C}$ | B |
| Menší než | lr | $N \oplus V$ | Z |
| Menší nebo rovno | le | $Z \vee (N \oplus V)$ | Z |
| Větší než | gt | $\overline{Z} \wedge (N = V)$ | Z |
| Větší nebo rovno | ge | $N = V$ | Z |
| Záporné číslo | mi | $N$ | Z |
| Kladné číslo | pl | $\overline{N}$ | Z |
| Došlo k přetečení | cs | $V$ | B/Z |
| Nedošlo k přetečení | vc | $\overline{V}$ | B/Z |
| Došlo k saturaci | qs | $Q$ | B/Z |
| Vždy | al | $True$ | |

B - bezznaménkové, Z - znaménkové

Tabulka 4.1: Seznam podporovaných podmínek

### 4.3.3 Ostatní rozšíření

Jádra *AVR32* v závislosti na konkrétním typu procesoru mohou obsahovat další rozšíření. Rozšíření *JAVA* umožňuje vyvojáři vytvořit *Java Virtual Machine*, na kterém lze spouštět *Java Bytecode*. Vše je řešeno převodem *Java Bytecodu* do *RISC* instrukcí na hardwarové úrovni[13].

Dalším rozšířením je podpora *SIMD* instrukcí. Tyto instrukce umožňují vykonávat paralélní operace nad daty pomocí jedné instrukce. Opět se jedná o rozšíření pro multimediální aplikace. Data lze zpracovávát pouze nad registrem o velikosti 32 bitů. Jsou tak podporovány datové typy 4×8 bitů a 2×16 bitů.

# Kapitola 5

# Reprezentace AVR32 v jazyce CodAL

Cílem této kapitoly je popsat postup implementace modelu *AVR32* v jazyce *CodAL*. Implementace vychází z vlastností architektury popsané v kapitole 4.

## 5.1 Definice zdrojů

Prvním krokem implementace bylo popsání registrů pro všeobecné použití. Registry `r0-r14` byly definovány jako soubor registrů pod názvem `gpreg`[1]. Registr s indexem 13 odpovídá registru `SP` a index 14 odpovídá registru `LR`. Jelikož se jedná o 32-bitovou architekturu, všechny registry jsou implementovány jako 32-bitové, pokud není uvedeno jinak.

Z vlastnosti jazyka *CodAL* vyplývá, že programový čítač musí být definován jako samostatný registr za pomoci klíčového slova `program_counter`. Programový čítač byl dle zvyklosti pojmenován `pc`. Narozdíl od reálné implementace je programový čítač logicky oddělen od souborového registru a tudíž musí být řešen způsob, jak spojit tyto registry. Z toho důvodu byly vytvořeny funkce zapouzdřující čtení a zápis do registrů. Funkce `RWRITE` příjmá zapisovanou hodnotu a index registru. V případě, že index registru je roven 15, dojde k zápisu do programového čítače, v opačném případě do registru pro obecné použití dle indexu. Na shodném principu pracuje funkce `RREAD`, která ovšem místo hodnoty programového čítače vrací hodnotu registru `fetched_pc` popsaného níže.

Pro každý ze 16 registrů musí být následovně vytvořen blok `element`, který slouží pro textovou a binární reprezentaci v jazyku assembler. V případě registru `SP, LR` a `PC` musí být vytvořeny tzv. alias elementy, jelikož tyto registry lze reprezentovat buď jejich názvem nebo řetězcem `rN`. `N` označuje index registru.

V ukázce 5.1 se nachází množiny (set) `regs` a `regs_dw`. Tyto množiny slouží k sjednocení registrů do jednotného logického celku. Narozdíl od `regs`, `regs_dw` slouží k reprezentaci dvojregistrů, proto zde nebyly vloženy určité registry. Dvojregistr je totiž reprezentován vždy spodním registrem, ať už se jedná textovou nebo binární reprezentaci.

[1] General purpose registers

```
program_counter bit[32] pc;
arch register bit[32] gpreg[14];

element r0 { assembler { "r0" }; binary { 0:4 }; return{ 0; }; };
element r15 { assembler { "r15" }; binary { 15:4 }; return{ 15; }; };
element r15_alias { assembler { "pc" }; binary { 15:4 }; return{ 15; }; };

set reg_nopc represents gpreg = r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10,
     r11, r12, r13_alias alias { r13 }, r14_alias alias { r14 };

set regs = reg_nopc, r15 alias { r15_alias };
set regs_dw represents gpreg = r0, r2, r4, r6, r8, r10, r12,
r14_alias alias { r14 };
```

5.1: Ukázka popisu registrů procesoru AVR32

Důležitou součast ve vytvářeném modelu tvoří rovněž stavový registr. V modelu tento registr nese název `sr`. Pro operace s jednotlivými bity, z důvodů zjednodušení, byla navržena makra `GETx` a `SETx`. `x` označuje název jednotlivých bitů stavového registru (CARRY, SIGN, ...). Pro extrakci sémantiky a generování softwarových nástrojů modelované platformy jsou jednotlivé bity reprezentovány jako 1-bitové registry. Jelikož přístup k těmto registrům je řešen pomocí maker, nemusí docházet k složitému upravování výsledného modelu. Problematika generování softwarových nástrojů je podrobněji popsána v kapitole 7.

Pro základní funkčnost modelu muselo být definováno několik dalších registrů. Tyto registry jsou skryty a nejsou přístupné pro programatorské účely.

Registr `fetched_pc` uchovává adresu aktulně prováděné instrukce, oproti registru `pc`, který v momentě provádění instrukce uchovává hodnotu následující instrukce.

Událost `main` požaduje existenci registru, který bude uchovávat aktuálně načtenou instrukci. Pro tyto účely byl definován registr `fetched_instr`. Tento registr je 32-bitový, jelikož maximální délka instrukce může být také 32-bitů.

Registr `cycle_cnt` byl navržen, jak lze z názvu odvodit, k uchovávání hodnoty provedených cyklů. V *instruction-accurate* se jeden cyklus rovná jedné provedené instrukci. Tento registr slouží k poskytování informace o výkonosti generováných nástrojů. Výkonnost lze tak jednoduše porovnat počtem provedených instrukcí aplikací. Porovnává počet vykonaných cyklů programem, který byl přeložen dodávanou sadou nástrojů oproti vykonaných cyklů programem přeloženým nástroji v generované sadě.

Poslední důležitou součástí modelu je paměť typu ram. Pro testovací účely byla implementována jednoduchá paměť, kde adresa i data mají velikost 32 bitů. Nejmenší adresovatelnou jednotkou je bajt. Jak již bylo zmíněno, data jsou v paměti uspořadána ve formátu big-endian.

## 5.2 Popis základních událostí

Dalším krokem bylo vytvoření povinných událostí (event). V sémantické sekci události `main` bylo zapotřebí navrhnout logiku načítání instrukcí a inkrementování programového čítače. Tato logika je však jednoduchá. Programový čítač je vždy inkrementován o 2 bajty, což odpovídá kompaktním instrukcím. Zda se jedná o rozšířenou variantu instrukce lze zjistit otestováním tří nejvíce významných bitů. V případě, že jsou všechny tyto bity nastaveny

na hodnotu 1, jedná se o rozšířenou variantu a programový čítač je inkrementován o další 2 bajty.

Událost `reset` je vykonávaná při každém startu nebo resetu procesoru. Proto sémantika této události obsahuje příkazy pro inicilizaci všech registrů včetně stavového registru.

Poslední povinná událost `halt` obsahuje příkaz `exit` pro ukončení činnosti simulace. V reálné instrukční sadě AVR32 však nelze nalézt instrukci `HALT` nebo jinou, která by vykonávala ukončení činnosti procesoru. Proto byla vybrána instrukce `BREAKPOINT`, která v modelu zastupuje ukončení činnosti procesoru. Tato instrukce je v reálném procesoru určena pro nastavení ladících operací. V modelované instrukční sadě byla tato instrukce bezpředmětná a mohla být použita jako náhrada. K aktivaci události `halt` byla využita sekce `timing`, která spouští danou událost.

## 5.3 Instrukční sada

Implementace instrukční sady vychází z popisu kategorií v kapitole 4.3. Avšak pro implementaci instrukční sady bylo zapotřebí instrukce rozdělit do kategorií dle shodnosti syntaxe assemblerovského zápisu a binárního formátu instrukce. Některé instrukce tak nemohly být zařazeny do žádné skupiny a musely být implementovány jako samostatné elementy. Vznikly tak skupiny instrukcí, které byly rozděleny dle shodné binární reprezence. Tyto skupiny byly následovně sjednoceny do větších skupin dle prováděné operace.

### 5.3.1 Implementace instrukcí

Jakmile byly instrukce správně rozděleny do jednotlivých skupin, byla zahájena implementace instrukční sady.

Pokud instrukce nepatřila do žádné ze skupin, byl pro ní vytvořen samostatný blok `element`. Následoval popis syntaxe assembleru, binární reprezentace a sémantiky instrukce. Kromě popisu prováděné operace instrukce v sémantické sekci, muselo být implementováno i nastavování bitů stavového registu, pokud to operace dané instrukce vyžadovala. Instrukce byla následně otestována. Testování instrukcí je popsáno v kapitole 6.1. Poté, co byla instrukce otestována, byla zařazena do množiny instrukcí zahrnující shodnou kategorii operaci (násobení, práce s pamětí, . . . ).

Jestliže instrukce patřila do skupiny se společnými vlastnostmi, byla abstrahována společná syntaxe zápisu a binarní reprezentace daných instrukcí. Následovalo vytvoření elementů pro jednotlivé operační kódy instrukční sady. Tyto elementy se následovně spojily do jednoho bloku `set`, který tyto operační kódy sjednocoval. Na této úrovni abstrakce byl vytvořen blok `element` pro celou skupinu. Vytvořený blok tak obsahuje navíc pouze logiku, která rozhoduje, jaká operace se vykoná. Rozhodovací logiku nejčastěji zastupuje příkaz switch, který na základě operačního kódu vykoná danou operaci.

```
element op_add {
        assembler { "ADD" }; binary { OP_ADD:8 }; return { OP_ADD; };
};
element op_sub {
        assembler { "SUB" }; binary { OP_SUB:8 }; return { OP_SUB; };
};
set op_i_rr = op_add, op_rsub, op_sub;
```

5.2: Ukázka popisu operačních kódů instrukcí

Implementace instrukční sady se však neskládala pouze z popisu instrukcí nebo celých skupin. Bylo zapotřebí nadefinovat elementy, které jsou použity jako základní prvky jednotlivých instrukcí.

Elementy `simmN`, `uimmN` byly implementovány kvůli nutnosti reprezentace čísel v jazyku assembler. V případě elementu `simmN` se jedná o reprezentaci čísel se znaménkem, `uimmN` je používán pro bezznámenková čísla nebo absolutní adresu v aplikaci. Pro popis relativnino adresování byly vytvořeny elementy `dispN`. Elementy obsahují klíčové slovo `actual_address`, které zastupuje adresu (pozici) ve výsledném programu. Hodnota N zastupuje bitovou šířku.

```
element uimm16 {
 assembler { val = unsigned };  binary { val = 0b[16] }; return { val; };
};

element disp21 {
 assembler {disp=signed {disp = disp  - actual_address  >> 1;}};
 binary {disp=0bs[21] {disp = (disp << 1) + actual_address ;}};
 return {disp;};
};
```

5.3: Blok element popisující immediate

Překladač od firmy Atmel využívá k volání podprogramů a návratu z nich instrukce `LDM` a `STM`, popřípadě `PUSHM` a `POPM`. Tyto instrukce tak musely být zahrnuty do instrukční sady. První problém nastal v návrhu implementace těchto instrukcí. Tyto instrukce používají tzv. seznamy, kdy jednotlivé registry mohou nebo nemusí být uvedeny v seznamu. Textový formát instrukce tak není pevně stanoven. Jelikož tyto instrukce používá minimum procesorů a vždy existuje způsob, jak tyto instrukce nahradit, neexistuje ve frameworku pro tyto konstrukce podpora. Existuje však způsob, jak tyto seznamy registrů vytvořit. Musel být vytvořen element `reglist`, který obsahuje elementy zastupující zahrnutí jednotlivých registrů v seznamu. `Reglist` neobsahuje žádnou sémantiku, obsahuje však sekci `return`, která vrací číslo reprezentující zahrnutí jednotlivých registrů.

Jedním z mála důležitých kroků byla implementace podmínek. Podmínky reprezentují elementy `cond*`. V sekci `return` těchto elementů se nachází výraz vyhodnocující podmínku. Výrazy byly zkonstruovány dle tabulky 4.1. Podmínky jsou implementovány ve dvou formátech. V prvním případě se jedná o podmínky, které jsou používány v kompaktních verzích instrukcí. Výsledná binární reprezentace má šířku 3 bity a umožňuje používat pouze prvních sedm podmínek. V druhém případě se jedná o 4-bitovou verzi, která je využívána v rozšířeném formátu instrukcí a umožňuje používat všechny typy podmínek.

```
element cond3_ne
{ assembler { "ne" }; binary { COND_NE:3 }; return { !GETZERO; }; };
element cond4_al
{ assembler { "al" }; binary { COND_AL:4 }; return { 1; }; };
```

5.4: Ukázka popisu podmínek

Pro přehlednost a zjednodušení modelu byly shodné operace zapouzdřeny do funkcí, které vykonávají podobné operace. V sémantické sekci se tak pouze nachází volání funkce, které se předají parametry společně s typem operace. Dále byly vytvořeny funkce `load` a `store`, které zapouzdřují operace s pamětí. Nemusí se tak popisovat konstrukce pro práci s pamětí, stačí pouze zavolat danou operaci a předat ji adresu společně s velikostí adresované jednotky. V případě ukládání dat do paměti je navíc předána nové hodnota.

# Kapitola 6

# Testování

Správnost implementace modelovaného procesoru a jeho instrukční sady popsané v kapitole 5 musí být řádně otestována. Samotné testování spočívá v porovnání binární reprezentace a kontrole sémantiky instrukcí.

## 6.1 Testování instrukcí

První fáze testování byla prováděna během vývoje instrukční sady. Jakmile byla implementována skupina instrukcí, byl vytvořen jednoduchý zdrojový kód v jazyce assembler, který zachycuje extrémy a kombinace operandů instrukcí. Následně byl tento kód přeložen do dvou různých verzí. První verze sadou nástrojů od firmy Atmel, která slouží jako referenční. Druhá verze byla přeložena kompilátorem vygenerovaným frameworkem Codasip.

Následujcím krokem byla kontrola sématiky implementovaných instrukcí. V každé iteraci testování byla pomoci vygenerovaného simulátoru a prostředí debug v Codasip studiu provedena simulace programu. V jednotlivých krocích na úrovni instrukce bylo kontrolováno chování instrukce, čtení a zápis operandů a výpočet příznaků stavového registru. Pokud instrukce prováděla nesprávné operace, byl zkontrolován popis sémantiky, popřípadě bylo chování porovnáno vůči simulátoru integrovaném v Atmel Studiu. Po opravě chyby byly vygenerovány nástoroje pro následující iteraci testování. Ve chvíli, kdy skupina instrukcí vykonávala vše v rozsahu testování správně, bylo přikročeno k ověření binární reprezentace.

Binární reprezentace byla ověřována prostřednictvím nástroje *objdump*. Výstupem tohoto nástroje je souhrn informací v čitelné podobě o objektovém souboru nebo výsledném programu. Mezi všemi těmito informacemi lze nalézt i binární formu instrukcí (disassemblovaný objekt), která je podstatná pro tento postup testování. Testování binární reprezentace pak spočívalo v jednoduchém porovnání výstupu generovaného assembler kompilátoru vůči výstupu referenčního kompilátoru od firmy Atmel.

## 6.2 Testovací sada programů

I přestože byla otestována každá instrukce zvlášť, mohl nastat stav, ve kterém instrukce provádí chybnou operaci. Pro tento případ byla poskytnuta rozsáhlejší sada testovacích programů. Sada obsahuje programy v jazyce *C*, které pokrývají většinu používaných programových struktur a lze tak celkově ověřit implementovanou instrukční sadu procesoru. Testovací programy jsou navrženy tak, aby byly ukončeny návratovou hodnotou, která musí

souhlasit s hodnotou výpočítanou testovacím programem. Tato vlastnost umožňuje použití automatizovaného testování, které je výrazně rychlejší.

Struktura testovacích programů však neobsahuje základní inicializaci procesoru. Nalezneme zde pouze základní vstupní funkci `main`. Pro ukončení jsou volány funkce `exit(int)`, které lze předat návratovou hodnotu. Funkce `abort()` ukončuje vždy program s chybovou hodnotou. Z toho důvodu bylo nutné vytvořit inicializační soubor `crt0.s`. Soubor obsahuje vstupní bod `_start`. V tomto bodě je nastaven zásobník a zavolána funkce `main`. Následovně byly v tomto souboru implementovány již zmiňované funkce `exit()` a `abort()`.

Testovací programy kromě základních operací vyžadují podporu funkcí, které se nacházejí ve standardní knihovně jazyka *C*. Všechny potřebné funkce byly z knihovny vyextrahovány a následně z nich byl vytvořen archiv `call.a`.

### 6.2.1 Automatizace testování

Pro zjednodušení testování byla vytvořena sada skriptů, která automatizuje testování nad poskytnutou sadou programů. Skript provádí následující operace:

1. **Překlad programu do jazyku symbolických adres** - V tomto kroku je pomoci kompilátoru *avr32-gcc* vygenerován assemblér kod.

2. **Odstranění nekompatibilních direktiv** - Skriptu typu filtr odstraní nebo převede direktivy, které používá sada nástrojů AVR32.

3. **Překlad do objektového souboru** - Vygenerovaný nástroj assembler přeloží zdrojový soubor do objektového souboru.

4. **Linkování programu** - Pomocí nástroje linker je objektový soubor spojen s archívem podporující základní funkce knihovny C a souborem obsahující inicializační funkce spojen do programu určeného pro simulaci. Podpůrné soubory jsou popsány následovně.

5. **Simulace programu** - Výsledný program je odsimulován a návratová hodnota je porovnána s očekávanou hodnotou.

Průchodem všemi testy bylo možné jednoduše určit, které testy skončily s chybou, na tyto programy se zaměřit a určit příčinu chyby. V tomto kroku se přechází opět k manuálnímu testovaní, ve kterém je potřeba provést simulaci na úrovni instrukcí, nalézt a chybu opravit. První testovácí iterace probíhaly bez optimalizace překladače (s parametrem -O0).

Následoválo testování s optimalizací na druhé úrovni (použití parametru -O2). Překlad z jazyka *C* do jazyka assembler obsahoval komplexnější instrukce a bylo možné ověřit téměr celou instrukční sadu. Iterace testování se opakovaly, dokud nedošlo k ustálení vývoje instrukční sady.

# Kapitola 7

# Generování softwarové sady nástrojů

Poslední krokem vývoje tohoto modelu, v rámci mé práce, bylo vygenerování softwarové sady nástrojů obsahující kompilátor jazyka *C*, assembler, linker a další nástroje potřebné pro vývoj aplikací.

Před samostaným vygenerováním nástojů bylo zapotřebí z modelu vyextrahovat sémantiku. Získaná sémantika slouží jako základ pro generování sady nástrojů. Protože framework zatím nepodporuje určité syntaktické a sémantické konstrukce, musel být model upraven.

Úpravy se týkaly:

- Vyloučení instrukcí pro práci s více registry z extrakce.
- Vyloučení instrukcí podporující plovoucí destinnou čárku z extrakce.
- Vyloučení instrukcí vykonávající operace s dvojslovem z extrakce.
- Vyloučení DSP instrukcí z extrakce.
- Úpravy použití datových typů v sémantické části.
- Způsobu výpočtu hodnot bitů pro stavový registr.

Instrukce pro práci s více registry byly vyloučeny z extrakce, jelikož množství generovaných instrukcí činilo přibližně 262 tisíc. Toto množství velice objemné a zpracovávání by vyžadovalo velký výpočetní výkon. Proto bylo rozhodnuto zachovat tyto instrukce pouze pro assembler a simulátor.

Z extrakce byla dále vyloučena podpora datového typu float a dvojslova. Podpora těchto datových typů je zatím ve stádiu vývoje a není zaručena spolehlivost těchto operací. Nicméně tyto datové typy jsou kompilátorem podporovány a to za pomocí knihovny *Compiler-rt*.

V případech, kdy sémantika používala speciální datové typy (uint2, int5), bylo zapotřebí provést úpravy na datové typy poskytující větší rozsah. Zároveň byla převedena explicitní typová konverze na implicitní.

Extraktor sémantiky požaduje, aby každý bit stavového registru byl reprezentován jako samostatný 1-bitový registr. Následovala tak úprava stavového registru a maker, které jsou určeny pro práci s ním. Aby byl extraktor sémantiky schopný rozpoznat prováděné operace, musely být pro výpočet příznaku přenesení (carry) nebo přetečení (overflow) použity funkce z knihovny *libcodasip*.

Po těchto úpravách byla úspěšně vygenerována sémantika modelu. Z této sémantiky tak byla vygenerována první sada nástrojů. Následovalo testování překladače. Testování probíhalo podobně jako testování instrukční sady popsané v kapitole 6.1. Rozdíl spočíval v programech, které byly kompilovány prostřednictvím vygenerovaného překladače.

## 7.1 Porovnání výkonnosti překladače

Důležitou roli hraje výsledná rychlost programů. Jedním z cílů této práce bylo vytvořit kompilátor, který se přibližuje rychlosti dodávaného překladače. Bylo tak potřebné vyhodnotit rychlost pomocí různých benchmarků. Vyhodnocování probíhalo na všech úrovních optimalizace. V následujících kapitolách jsou zobrazeny výsledky vybraných testů. K vyhodnocení výkonosti překladače byla v případě sady nástrojů od firmy Atmel použita verze *AVR32_GNU_Toolchain_3.4.2_435 (gcc 4.4.7)*. Generovaná sada nástrojů byla vytvořena pomocí Codasip Frameworku ve verzi *2.0.1-2.jenkins.332.nightly.140505*.

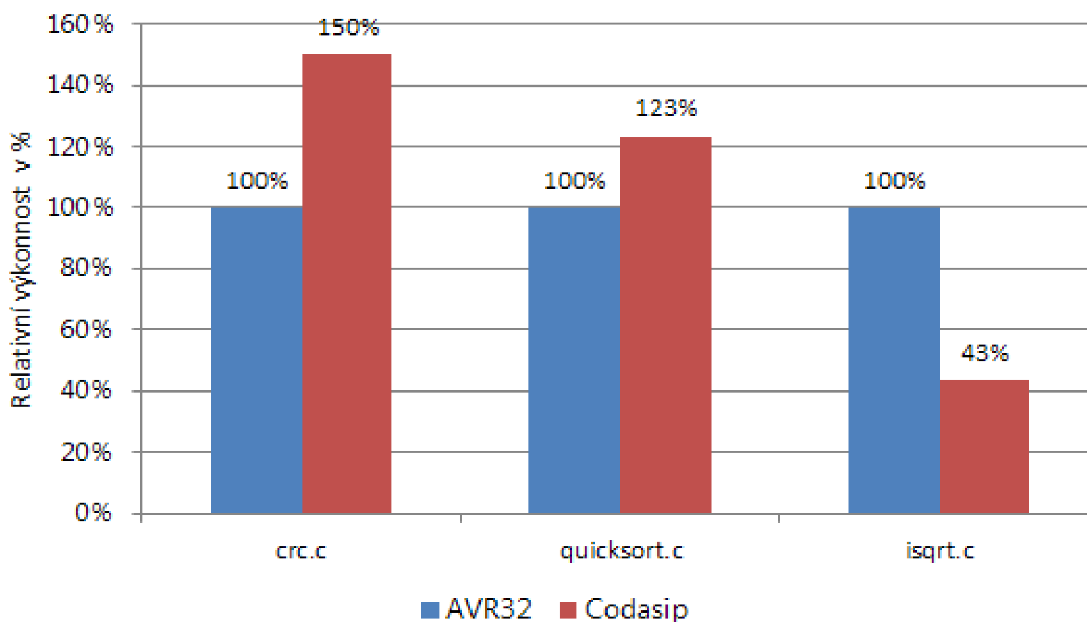
Program crc.c provádí výpočet kontrolního součtu nad daným polynomem. Hodnoty polynomu jsou pevně stanoveny. Tento program provádí většinou bitové operace.

Program quicksort.c provádí seřazení 256 hodnot za pomocí řadicího algoritmu quicksort.

Program isqrt.c provádí výpočet odmocniny z daného čísla. Jelikož se jedná program vykonávající poměrně malý úsek kódu, byl tento program upraven tak, aby prováděl více iterací nad různými čísly.

V grafu 7.1 je zobrazena relativní výkonnost překladače od firmy Atmel vůči generovanému Codasip překladači. Graf zobrazuje rychlost jednotlivých programů, kde 100% udává výsledek dodavaného překladače. Vyšší hodnoty značí lépší výkon.

V případě dodávaného překladače byly programy zkompilovány pomocí příkazu `"avr32-gcc -O2 -mno-asm-addr-pseudos"`. Pro překlad programů vygenerovaným překládačem byl použit příkaz `"avr32-codasip-gcc -O2"`.




Obrázek 7.1: Výsledky benchmarků

| Program | Překladač | |
|---|---|---|
| | AVR32 | Codasip |
| crc.c | 3800022 | 1900030 |
| quicksort.c | 38667 | 29664 |
| isqrt.c | 57404 | 89916 |

Tabulka 7.1: Počet vykonaných instrukcí na úrovni optimalizace O2

Z výsledků vyplývá, že z modelu byl vygenerován překládač, který umožňuje překládat na úrovni, která je lepší oproti dodávanému překladači. V případě programu `isqrt.c` je pokles způsoben softwarovým emulovaním operací nad desetinnými typy, které nejsou v překládači zatím podporovány a dochází tak k nárustu počtu prováděných instrukcí. Rozdíl mezi úrovni optimalizace O2 a O3 není pro porovnávané programy žádný, jelikož na daných programech bylo dosaženo maximální možné optimalizace již na úrovni O2. Výrazný rozdíl v programu `crc.c` je způsoben inline optimalizací.

# Kapitola 8

# Závěr

Cílem této bakalářské práce bylo vyvoření *instruction-accurate* modelu procesoru AVR32. V prvním kroku bylo zapotřebí seznámit se s instrukční sadou architektury AVR32. Následovalo nastudování principů modelování procesorů na platformě Codasip. Ze získaných znalostí byl úspěšně implementována instrukční sada architektury AVR32. Následně byla z tohoto modelu vygenerována sada softwarových nástrojů pro vývoj aplikací běžících na modelovaném procesoru AVR32. Výkonnost programů přeložených pomocí vygenrovaného překladače je vyšší oproti referenčnímu překladači od firmy Atmel. Jelikož Codasip Framework je neustálé ve vývoji, může se výkon překládaných programů lišit. Zároveň vytvoření celé sady softwarových nástrojů trvalo kratší dobu, než vytvoření této sady softwarovými vývojáři. Model byl zařazen mezi modely platformy Codasip a rozšířil tak již početnou skupinu modelů.

V budoucnu by se model dal rozšířit o *cycle-accurate* model, ze kterého by byl následně vysyntétizován reálný procesor. Dále pro tento model může být naportována knihovna *newlib*, která slouží jako standardní knihovna jazyka *C* pro věstavné systémy. Současně s vývojem frameworku Codasip by mohla být následně přidána podpora datového typu float, dvojslova a podpora instrukcí pro práci s více registry v generované sadě nástrojů.

# Literatura


[1] *Codasip - ASIP Development Automation* [online]. 2014 [cit. 2014-05-10]. Html. Dostupné na: <http://www.codasip.com>.

[2] Hohenauer, M. a Leupers, R. *C Compilers for ASIPs: Automatic Compiler Generation with LISA*. New York: Springer, 2010. 223 s. ISBN 978-1-44-191175-9.

[3] Hoffmann, A., Kogel, T., Nohl, A. et al. A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*. Nov 2001, roč. 20, č. 11. S. 1338–1354. ISSN 0278-0070.

[4] Masařík, K. *Prezentace k předmětu Principy programovacích jazyků a OOP*. 2012.

[5] Mishra, P. a Dutt, N. *Processor description languages: applications and methodologies*. [b.m.]: Morgan Kaufmann Publishers, 2008. 432 s. ISBN 978-0-12-374287-2.

[6] *Language EXPRESSION* [online]. 2003 [cit. 2014-05-10]. Html. Dostupné na: <http://www.ics.uci.edu/%7Eexpress/>.

[7] *LISA: Description Language for Hardware and Software* [online]. 2014 [cit. 2014-05-10]. Html. Dostupné na: <http://www.ice.rwth-aachen.de/research/tools-projects/lisa/lisa/>.

[8] *CodAL Manual: reference guide*. 2014.

[9] *Codasip Framework Manual: User's guide*. 2014.

[10] *AVR32 Architecture Document* [online]. 2011 [cit. 2014-05-10]. Dostupné na: <http://www.atmel.com/images/doc32000.pdf>.

[11] *MSP430 family instruction set* [online]. [cit. 2014-05-10]. Dostupné na: <http://www.physics.mcmaster.ca/phys3b06/MSP430/Instruction_Set.pdf>.

[12] *AVR32UC Technical Reference Manual* [online]. 2010 [cit. 2014-05-10]. Dostupné na: <http://www.atmel.com/images/doc32002.pdf>.

[13] *Java Technical Reference* [online]. 2009 [cit. 2014-05-10]. Dostupné na: <http://www.atmel.com/images/doc32049.pdf>.

# Příloha A

# Obsah CD

Přiložené CD obsahuje tyto složky:

- `/avr32-export` - Vygenerovaná sada nástrojů pro systém Windows 7 64-bit.
- `/doc` - Elektronická verze této práce.
- `/model` - Zdrojové soubory modelu AVR32.