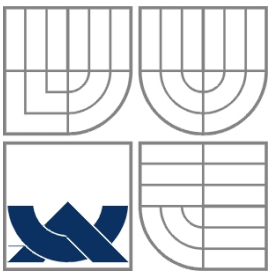
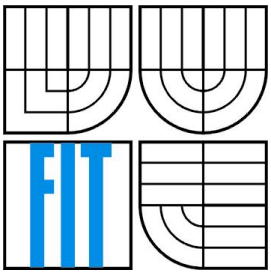


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VYTVOŘENÍ MODELŮ PROCESORŮ POMOCÍ ADL JAZYKA

MODELING OF TWO PROCESSORS IN ADL LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

DOMINIK STEINHAUSER

VEDOUCÍ PRÁCE
SUPERVISOR

PROF. ING. TOMÁŠ HRUŠKA, CSC.

BRNO 2015

Abstrakt

Cílem této bakalářské práce je vytvoření modelů dvou procesoru Tensilica Xtensa a Sparc Leon na instrukční úrovni. Modely byly implementovány v ADL jazyku CodAL. Vývoj simulace a testování probíhalo ve vývojovém prostředí Codasip Studiu.

Aplikačně specifické procesory mohou být vyvíjeny od začátku, nebo je možné pouze upravit již navrhnutý procesor, aby lépe vyhovoval dané aplikaci. Mnou vytvořené modely budou přidány do portfolia firmy Codasip, které budou moci uživatelé Codasip Studia používat či upravovat.

Výsledkem jsou otestované funkční modely těchto procesorů, pro které lze vygenerovat simulátor, assembler i překladač jazyka C. Modely byly porovnány pomocí několika Benchmark testů a výsledky porovnání byly vyhodnoceny.

Abstract

Goal of this bachelor thesis is to create instruction level models of two processors Tensilica Xtensa and Sparc Leon. Models were implemented in CodAL language. Development, simulation and testing took place in Codasip Studio, an IDE developed by Codasip company.

Application Specific Instruction-Set Processors can be implemented from scratch or already implemented processor can be modified to meet needs of specific aplication. My models will be added to portfolio of Codasip company to be used and modified by the user of Codasip Studio.

Result of this work are tested models of these two processors. Simulator, assembler and C language compiler of these processors can be generated. Models were compared by several Benchmark tests and results were analyzed.

Klíčová slova

Procesor, ADL, Codasip, Xtensa, Sparc, modelování, simulace,CodAL

Keywords

Processor, ADL, Codasip, Xtensa, Sparc, modeling, simulation, CodAL

Citace

Dominik Steinhauser: Vytvoření modelů procesorů pomocí ADL jazyka, bakalářská práce, Brno, FIT VUT v Brně, 2015

Vytvoření modelů procesorů pomocí ADL jazyka

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Prof. Ing. Tomáše Hrušky, CSc.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Dominik Steinhauser
20.5.2015

Poděkování

Děkuji profesoru Hruškovi za vedení práce a konzultantovi Adamu Husárovi za čas, který mi věnoval.

Obsah

1 Úvod

Bakalářská práce popisuje vytvořené modely procesorů Xtensa a Sparc LEON na instrukční úrovni v ADL jazyku CodAL.

Druhá kapitola obsahuje stručný úvod do problematiky jazyků pro popis architektur (ADL), ve kterém byly modely vytvořeny.

Třetí kapitola představuje aplikaci Codasip Studio. Jsou představeny jeho nejdůležitější nástroje s důrazem na ty, které byly použity při vývoji mých modelů.

Čtvrtá a šestá kapitola popisuje architekturu procesoru Xtensa respektive Sparc. Zvýšená pozornost je věnována instrukční sadě.

Pátá a sedmá kapitola jsou zaměřeny prakticky. Popisuji zde způsob implementace obou procesorů, zvolené prostředky i problémy, které se během implementace objevily a jak jsem je vyřešil. Pátá kapitola je rozsáhlejší, neboť zde detailněji popisuji zvolený postup implemetace, který byl u obou procesorů podobný a je tedy v sedmé kapitole kratší, abych se vyhnul opakování.

Osmá kapitola obsahuje porovnání architektur obou procesorů a především výsledky simulace na vybraných benchmarcích.

Devátá kapitola obsahuje závěr práce a shrnutí dosažených výsledků.

2 Jazyky pro popis architektury - ADL

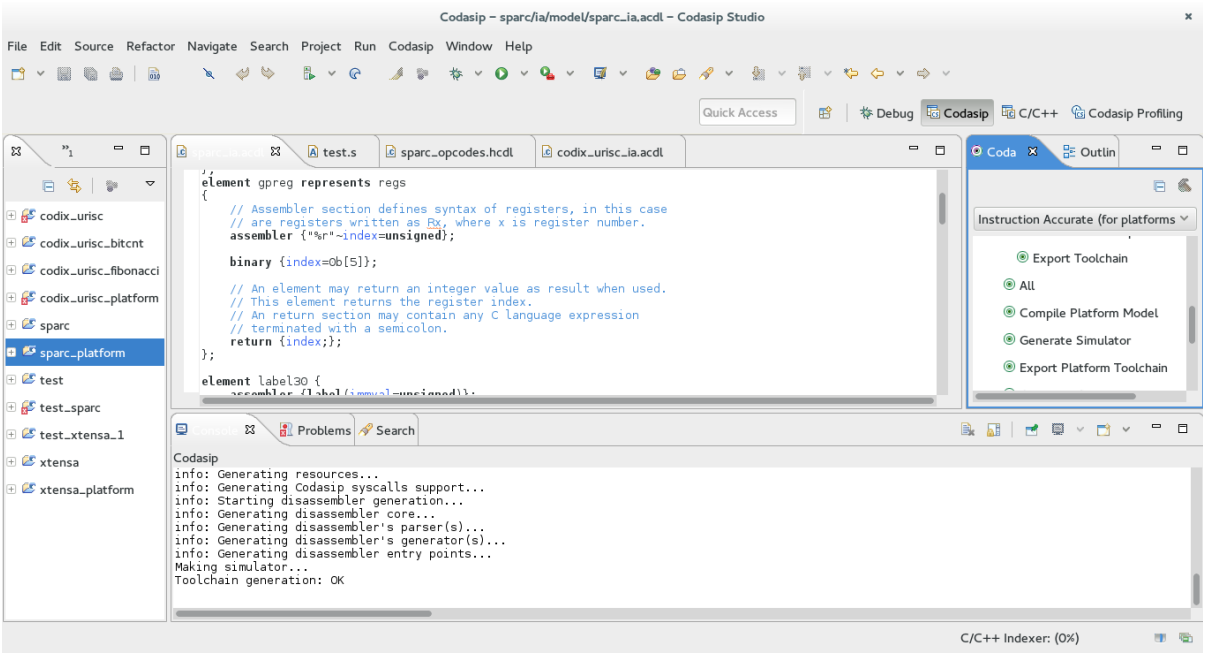
V této kapitole čerpám ze zdroje [2].

Cena procesoru se kromě nákladů na výrobu odvíjí také od ceny vývoje. Aby se snížila doba návrhu procesoru a tedy i jeho cena, používají se jazyky pro popis architektury (ADL), které umožňují automaticky generovat nástroje pro vývoj procesorů s aplikačně specifickou instrukční sadou (ASIP). Jazyky ADL se dělí do tří skupin podle svého zaměření. Jedná se o :

- jazyky zaměřené na instrukční sadu
- jazyky zaměřené na strukturu
- jazyky popisující instrukční sadu a strukturu zároveň

Zadané modely procesorů jsem vytvořil v jazyku CodAL, který patří do poslední zmíněné kategorie. Podrobněji bude popsán v kapitole 3.2.

3 Použité nástroje – Codasip Studio

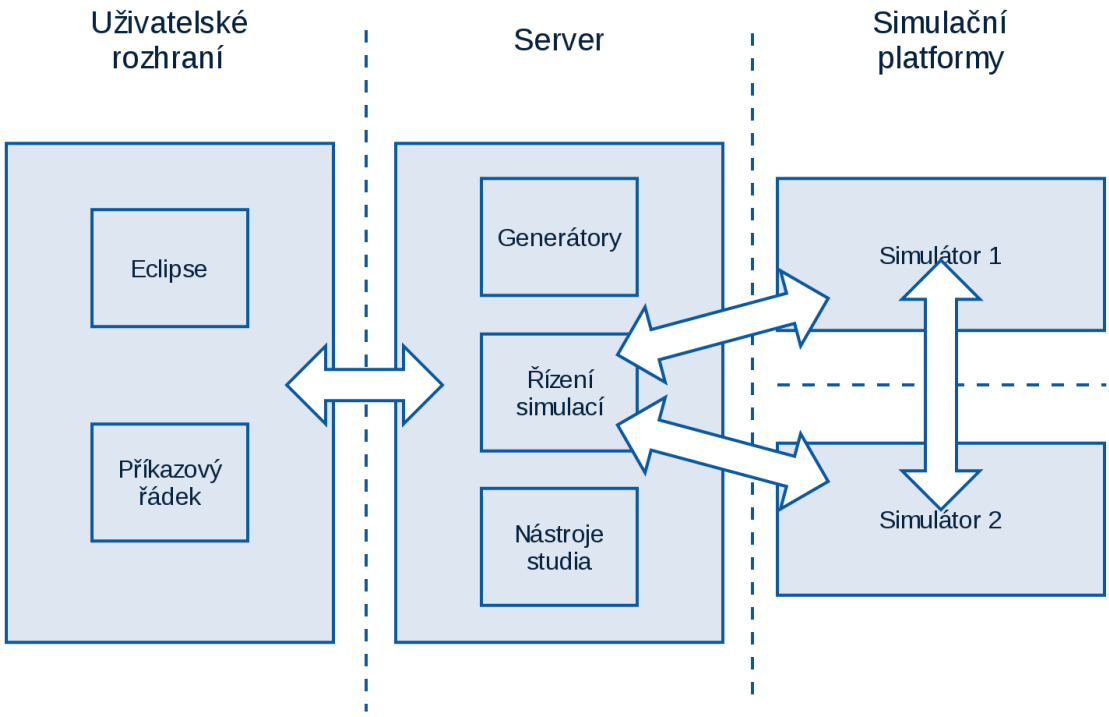


Obrázek 3.1: Úvodní pohled na Codasip Studio

V této kapitole čerpám ze zdroje [3].

Codasip Studio je vývojové prostředí založené na programu Eclipse, které slouží k návrhu procesorů s aplikačně specifickou instrukční sadou. Pohled na aplikaci uvádím na obrázku 3.1. Architektura procesorů je popsána v jazyce CodAL. Z tohoto popisu je možné automaticky vygenerovat popis hardware ve Verilogu či VHDL. K dispozici je mnoho automatických nástrojů, které usnadňují práci programátora a zabraňují vzniku chyb.

Vývojové prostředí pracuje na třech úrovních, jak je znázorněno na obrázku 3.2.



Obrázek 3.2: úrovně Codasip Studia

Na úrovni uživatelského rozhraní je možno používat grafické uživatelské rozhraní ve formě několika pohledů intergrovaných do platformy Eclipse. Pro pokročilé použití je možno používat příkazový řádek, který umožňuje skriptování nebo automatické testování.

Úroveň serveru přijímá pokyny od uživatele, provádí je a vrací uživatelskému rozhraní o úspěchu provedených operací. Server instaluje simulátor a předává mu příkazy z uživatelského rozhraní.

Některé systémy obsahují více než jeden procesor. Pokud chceme takový systém simulovat je vytvořen samostatný simulátor pro každý procesor. Jednotlivé simulátory spolu mohou komunikovat díky přístupu ke sdíleným zdrojům.

Jednotlivé vrstvy spolu komunikují přes TCP/IP protokol a v případě potřeby mohou běžet na různých zařízeních v síti. To umožňuje například rychlou simulaci víceprocesorových systémů, kdy je každý procesor simulován na jiném zařízení.

3.1 Nástroje studia

Codasip Studio poskytuje uživateli automatizované nástroje, které urychlují proces návrhu procesoru. Některé z nich stručně uvedu.

- Assembler

  Assembler provádí překlad z jazyka symbolických instrukcí do objektového souboru, který obsahuje binární zápis instrukcí a tabulku symbolů definovaných ve zdrojovém kódu. Objektové soubory jsou k dispozici v textovém formátu CCOFF a binárním formátu ELF
- Linker

  Linker spojuje objektové soubory vygenerované assemblerem do spustitelného souboru. Pro binární objektové soubory se používá GNU-LD linker, pro soubory ve formátu CCOFF Codasip Linker.
- Simulátor

  Codasip Studio umožňuje simulovat běh procesoru buď na úrovni provádění instrukcí nebo na úrovni cyklů procesoru, podle toho o jaký typ modelu se jedná. Je také možné exportovat popis modelu a simulovat ho pomocí externího simulačního nástroje.
- Debugger

  Debugger je založen na open source platformě GDB. Používá se ve spojení se simulátorem. Umožňuje všechny standardní funkce debbugeru.
- Profiler

  Profiler slouží ke získání detailních informací o běhu programu pro potřeby optimalizace.
- Generátor překladače jazyka C

  Studio umožňuje na základě instrukční sady generovat překladač jazyka C. Pokud instrukce nejsou ve vhodném formátu, je nutné určit, jakými instrukcemi se modelují jaké konstrukce jazyka C.

3.2 Jazyk CodAL

Popis jazyku CodAL vychází ze zdroje [4].

Jazyk CodAL byl vyvinut společností Codasip s.r.o a slouží k návrhu aplikačně specifických instrukčních mikroprocesorů (ASIP). Jazyk umožňuje souběžný návrh hardwaru a softwaru. Popis procesoru je nejprve převeden do formátu XML, který pak zpracovávají generátory nástrojů.

Každému procesoru je namodelována platforma, která obsahuje popis všech rozhraní procesoru jako jsou paměti, koprocesory či jiné komponenty, včetně určení způsobu přenosu dat, adresování atd. Procesor je možné modelovat na instrukční úrovni nebo na úrovni cyklů. Modelují se události[1], kde se určí co se stane např. při restartu procesoru, zdroje, například registry či paměť a instrukce.

1 Povinně musí být implementovány události Halt, Reset a Main.

3.2.1 Struktura modelu

Základním blokem modelu jsou *elementy.* Každý element obsahuje některé ze sekcí *localDeclaration, assembler, binary, semantics, return* a *events.* V sekci *localDeclaration*[2] je možné pomocí klíčového slova *use* povolit použití dříve definovaného elementu, události případně jejich množiny. V sekci *assembler* se popisuje symbolický zápis instrukce jako řetězce a v sekci *binary* jejich strojový formát. Obsahuje-li element jednu z těchto sekcí, musí obsahovat i druhou. Sekce *semantics* určuje operaci, která se při vykonání dané instrukce provede. Sémantika je popsána v podmnožině Jazyka C[3]. Při přistoupení k elementu v sekci *sematics* se vrácí hodnota určená v sekci *return.* Pomocí sekce *events* je možné přiřadit některým instrukcím určité události.[4]

Syntaxi jazyka a příklad jeho použití ilustruje Zdrojový kód 3.1. Jedná se o popis instrukcí *Callx0* a *Jx* procesoru Xtensa. Na řádku 1 – 10 je spárován instrukční kód daných instrukcí s jejich názvem v assembleru. Na řádcích 12 a 15 jsou operační kódy instrukcí vloženy do společné množiny a použity v popisu elementu *instruction_callx_20*, který představuje tyto dvě instrukce. Na řádcích 17 a 18 je určen binární a symbolický formát daných instrukcí. Na řádcích 20 – 31 je popsán sémantický význam daných instrukcí. Povšimněme si použití proměnné *opcode.* Pokud je použita v sekci *assembler*, je použit její symbolikcý zápis. V sekci *binary* je použita její binární podoba. V sekci *semantics* je pak použita její návratová hodnota.

2 Sekce *localDeclaration* není uvedena žádným klíčovým slovem, nýbrž následuje ihned po uvedení názvu elementu. Viz zdrojový kód 3.1.

3 Zakázány jsou struktury a výčtové typy,ukazatele, příkaz goto, deklarace a inicializace v jednom příkazu a použití příkazu switch s jinými sekcemi než jsou case nebo default.

4 Například instrukci reset můžeme přiřadit událost reset, která se pak provede při každém zapnutí simulátoru.

```
element opcode_callx0 {
      assembler {"CALLX0"};
      binary {OPC_CALLX0:20};
      return {OPC_CALLX0;};
};
element opcode_jx {
      assembler {"JX"};
      binary {OPC_JX:20};
      return {OPC_JX;};
};

set opcode_callx_20=opcode_callx0,opcode_jx;
//element implementuje obě uvedené instrukce
element instruction_callx_20 {
      use opcode_callx_20 as opcode;
      use gpreg as s;//Proměnná s představuje registr popsaný jinde.
      assembler {opcode s};
      binary  {opcode[19..8]  s  opcode[7..0]};//Operační  kód  je
rozdělen do dvou částí
      semantics {
            switch(opcode) {//Operace se vykoná na základě operačního
kódu instrukce
                  case OPC_CALLX0:
                        ar[0]=pc+3;
                        pc=ar[s];
                  break;
                  case OPC_JX:
                        pc=ar[s];
                  break;
            }
      };
};
```

Zdrojový kód 3.1: Příklad popisu dvou instrukcí.

4 Popis procesoru Xtensa

Informace uváděné v této kapitole čerpám z dokumentace uvedené jako zdroj [5].

Procesory pro obecné použití nejsou pro danou aplikaci vždy optimální. Návrh procesoru pro určitou aplikaci je zase zdlouhavý a nákladný. Procesor Tensilica Xtensa umožňuje použít procesor pro

obecné použití a nastavit jej tak, aby co nejlépe vyhovoval potřebám dané aplikace. Při jeho návrhu byly sledovány 3 cíle: rozšiřitelnost, nastavitelnost a znovupoužitelnost[5].

Rozšiřitelnost dovoluje vývojářům navrhnout své vlastní instrukce, zatímco nastavitelnost umožňuje rozhodnout, zda se v aplikaci využijí předem navržené sady instrukcí[6]. Navržený procesor je poté možné mapovat do různého hardwaru bez nutnosti provádět rozsáhlejší změny v návrhu procesoru.

4.1 Proces návrhu procesoru

Proces návrhu procesoru je znázorněn ve schématu 4.1. Programátor má možnost definovat své vlastní instrukce popsané v jazyku TIE a dále zvolí, která rozšíření budou v daném procesoru použita. Automaticky je vytvořena sada nástrojů, která umožňuje simulovat běh procesoru a analizovat jeho výkon. Na základě výsledků je možné procesor optimalizovat. Po optimalizaci procesoru je možné vygenerovat výsledný procesor.

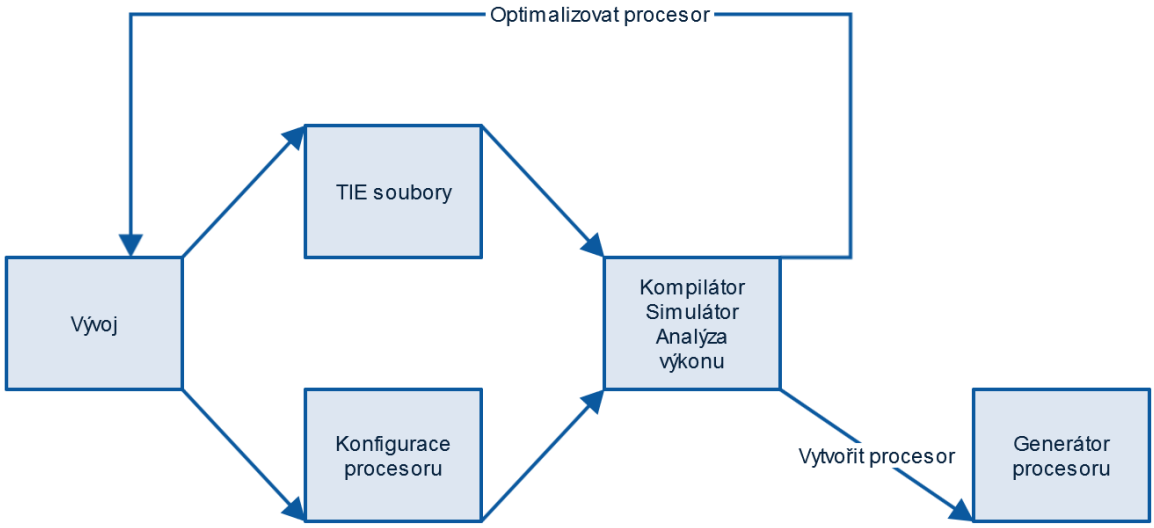


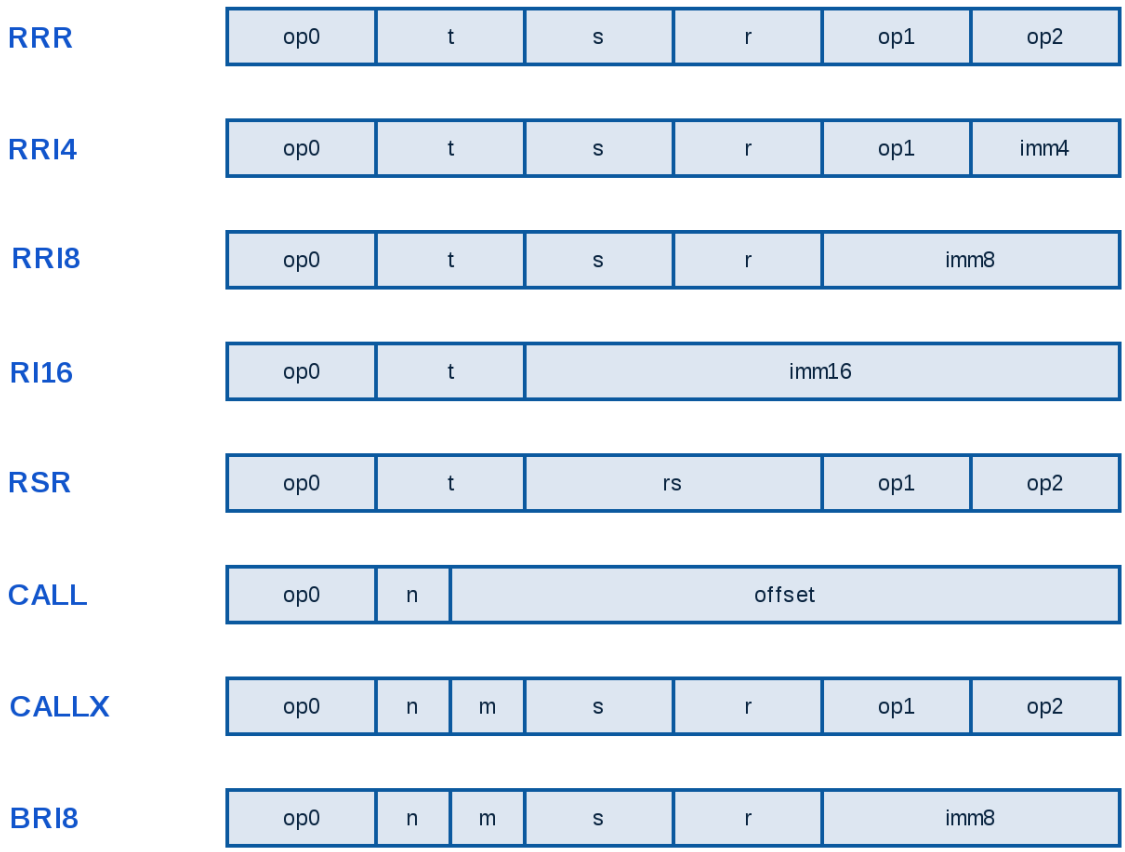
Schéma 4.1: Proces návrhu procesoru Xtensa

5 Možnost použít procesor v různém hardwaru

6 Seznam těchto sad je v příloze 1.

4.2 Instrukční sada

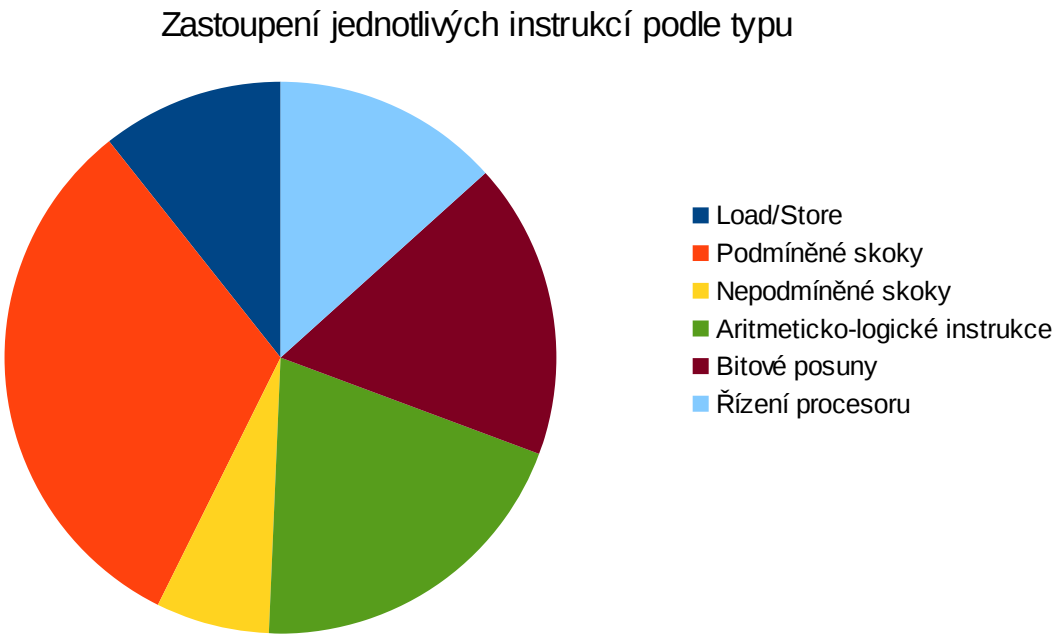
Instrukční sada vychází z RISC a zahrnuje 80 instrukcí. Specifická je délka instrukcí, které mají 24 bitů, v případě rozšíření *density option* mají některé nejpoužívanější instrukce pouze 16 bitů. V základní verzi procesoru je definováno 8 formátů instrukcí. Formáty uvedené na obrázku 4.2 jsou převzaty z dokumentace procesoru, přesto však nejsou v návrhu striktně dodržovány. Některé instrukce například využívají pole určená k adresám registrů, k uložení hodnoty operandu[7].



Obrázek 4.2: Formáty instrukcí procesoru Xtensa

Zastoupení instrukcí v procesoru Xtensa je znázorněno na grafu 4.3. Graf potvrzuje, že tvůrci procesoru věnovali podmíněným skokům zvláštní pozornost. Podmíněně lze skákat na 7bitovou, 8bitovou nebo 12bitovou relativní adresu.

7 Extrémním příkladem je instrukce NOP, která je zařazena do formátu RRR, ačkoliv obsahuje pouze operační kód.



Graf 4.3: Zastoupení instrukcí v procesoru Xtensa

4.3 Registry

Procesor obsahuje 16 registrů AR, registr PC a 6bitový Shift-amount register (SAR).
Registry AR jsou označeny jako adresové registry, aby se odlišili od registrů koprocesorů. které nesou v mnoha aplikacích data. Nejsou však určeny výlučně k uchovávání adresy, mohou uchovávat i data.

SAR je jediný speciální registr, obsažený v základní architektuře. Do speciálních registrů se přistupuje pomocí instrukcí WSR.*, RSR.*, kde * je označení daného speciálního registru. V případě registru SAR se tedy jedná o instrukce WSR.SAR atd.

4.4 Použití

Procesor je modifikovatelný a může proto být použit mnoha způsoby. Je používán například při zpracování grafiky nebo audiovizuálních dat[8].

8 Tensilica HiFi: Audio/Voice DSP IP. SoC Design IP and Verification IP Solutions: Cadence IP [online]. [cit. 2015-05-19]. Dostupné z: http://ip.cadence.com/ipportfolio/tensilica-ip/audio

5 Model procesoru Xtensa

Procesor xtensa byl modelován na instrukční úrovni. Model záměrně zanedbává vnitřní uspořádání procesoru a simulovat lze tedy s nejmenším krokem jedné instrukce. Struktura modelu odpovídá jazyku CodAL, popsaném v kapitole 3.2.

Model je rozdělen do čtyř souboru *xtensa_ia.acdl*, *xtensa_opcodes.hcdl*, *xtensa_platform.pcdl* a *user_semantics.sem*[9]. První soubor popisuje model samotného procesoru. Truhý soubor je hlavičkový a obsahuje binární operační kódy instrukcí. Třetí soubor modeluje platformu a rozhraní procesoru.

5.1 Platforma a rozhraní

Popis platfromy je v souboru *xtensa_platform.pcdl.* Obsahuje Paměť *mem* popsanou v jazyku CodAL, která je propojena s odpovídajícím rozhraním procesoru. Implementaci paměti mem v jazyku CodAl vidíme na zdrojovém kódu 5.1.

Používá se jeden adresový prostror, kde jsou uložena data i program. Procesor Xtensa je možné používat jak pro uspořádání paměti *little endian*, tak i *big endian.* Pro svůj model jsem zvolil uspořádání *big endian.*

Velikost paměti byla redukována na 100 MB, aby se zrychlilo načítání simulátoru, což bylo zvláště potřebné při provádění automatických testů.

```
memory mem {
                bits = {32, 32, 8};// 32bitová adresa, 32bitová slova, 8bitové bajty
                size = 0x0100000000;//Velikost udávaná v bajtech
                endianness = "big";// pořadí bajtů v paměti
                unaligned = "yes";//povolení nezarovaného přístupu

                interface read_write
                {
                      type = "memory:slave";
                      flag = "rw";
                };
          };
          //Propojení rozhraní paměti s rozhraním procesoru.
          connect xtensa.mem => mem.read_write;
```

Zdrojový kód 5.1: Rozhraní procesoru Xtensa

9 Pár definic konstant obsahuje i soubor *config.hcdl*,

5.2 Registry

32bitový programový čítač *pc* obsahuje adresu právě prováděné instrukce.

Dále model obsahuje 16 obecných registrů *ar,* které jsou modelovány elementem *gpreg*. Registry jsou adresovány 4bitovou adresou a v assembleru jsou značeny a[číslo registru].

Speciální registr *sar* nemá určenou adresu, přistupuje se k němu pouze ve speciálních instrukcích.

V jazyku CodAL jsou hodnoty v registrech interpretovány jako neznaménkové. Je-li potřeba provádět například znaménková porovnání, je nutné registr explicitně přetypovat.

5.3 Modelování instrukcí

Každá instrukce má svůj operační kód v souboru *xtensa_opcodes.hcdl.* Kód instrukce v assembleru je poté s tímto kódem spárován ve zvláštním elementu jazyka CodAL.

Kvůli zabránění opakování kódu jsou nstrukce modelovány v množinách, které mají stejný binární i symbolický formát. Tím se sníží počet elementů jazyku CodAL a dosáhne se větší čitelnosti kódu. Vycházel jsem z množin instrukcí stejného formátu určených výrobcem, popsaných v kapitole 3.2. Bohužel mnoho instrukcí se od tohoto formátu odlišuje, takže bylo nutné rozdělit výrobcem určené množiny na podmnožiny, které bylo možné modelovat jako jednotlivé elementy jazyka CodAL. Struktura modelu instrukcí je znázorněna na schématu 5.1. Implementace množiny instrukcí *instruction_callx_20*, byla už uvedena jako ilustrační příklad v kapitole 2.2.1 o jazyku Codal a může sloužit i jako příklad implementace jednotlivých instrukcí procesoru Xtensa. Pro potřeby simulátoru je do modelu přidána instrukce *halt.*

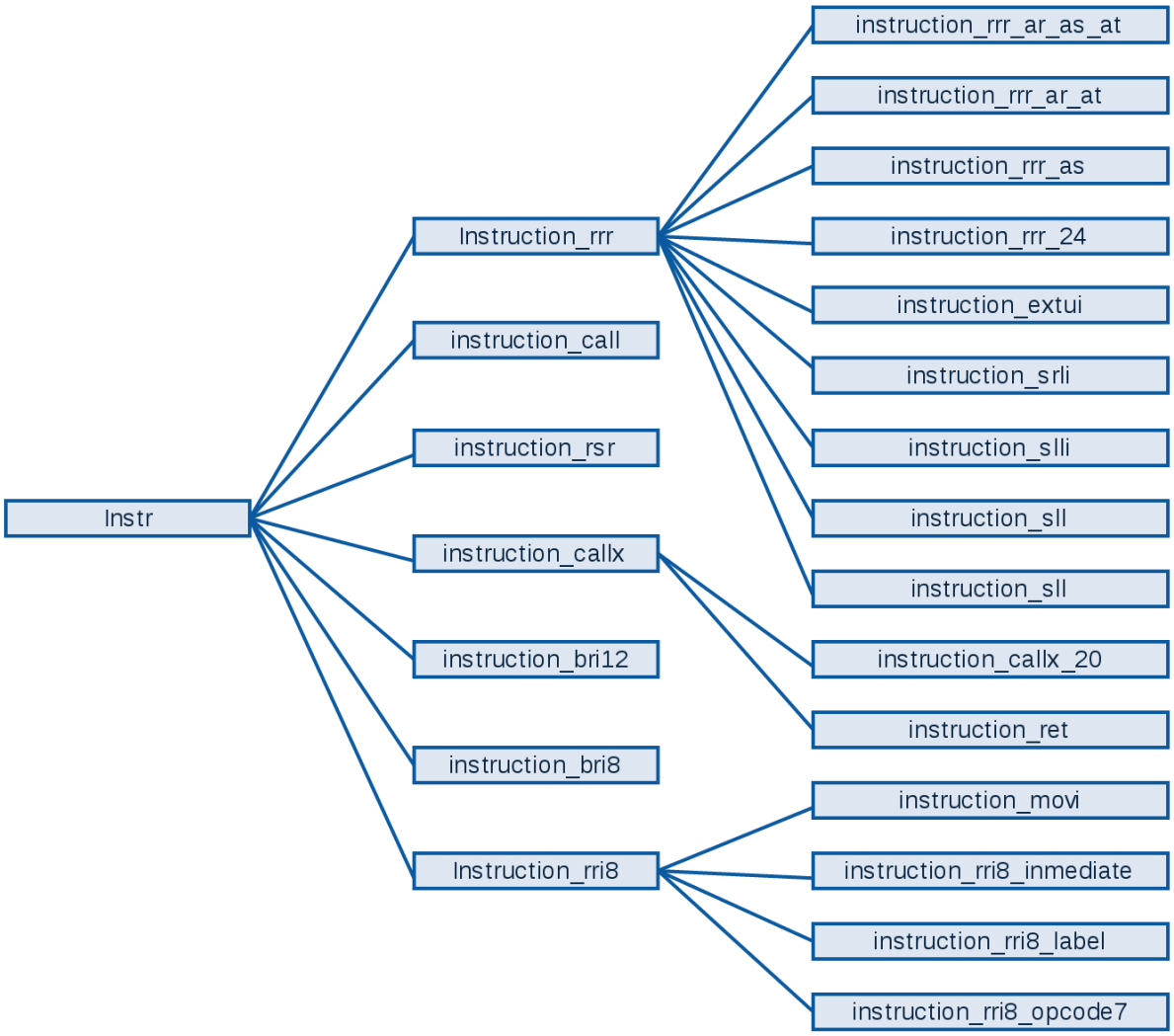


Schéma 5.1: Rozdělení instrukcí do množin se stejným formátem

5.4 Rozsah modelu

Byla namodelována základní verze[10] procesoru Xtensa a rozšíření zkrácených instrukcí, kde jsou nejpoužívanější instrukce zkráceny na 16 bitů. Nebyly implementovány instrukce zabývající se synchronizací paměti s procesorem, protože v modelu na instrukční úrovni nejsou tyto instrukce použitelné.

5.5 Překladač

Aby bylo možné vygenerovat překladač bylo nutné model procesoru upravit.

10 Tzv. *Core architecture*

Instrukční sada procesoru neobsahuje instrukce pro znaménková i neznaménková porovnání dvou registrů, kde by se výsledek ukládal do registru. Bez těchto instrukcí nebylo možné vygenerovat překladač. Po dohodě s vedoucím práce jsem využil skutečnosti, že návrh procesoru umožňuje přidávat rozšíření či uživatelské instrukce a na místo instrukcí rozšíření *Miscellaneous Operations Option* jsem přidal instrukce[11] provádějící požadované porovnání.

Procesor navíc neumožňuje načtení konstanty do registru a v seznamu doporučených praktik[12] se doporučuje nahrát konstantu z paměti, což generátor překladače neumožňuje. Do souboru *user_semantics.sem* jsem tedy doplnil emulaci této operace pomocí tří načítání 12bitové konstanty a standardních bitových operací.

Dále jsem pomocí seznamu doporučených praktik doplnil emulaci oprací pro bitové posuny, kde jsou všechny operandy registry. A určil které registry budou sloužit jako ukazatel zásobníku a ukazatel na začátek zásobníkového rámce.

Způsob emulace instrukcí v souboru *user_semantics.sem* prezentuji na jednoduchém příkladu bitového posunu vlevo. Instrukční sada procesoru používá k tomuto účelu speciální registr, překladač však vyžaduje použití obecného registru. Instrukci posunu vlevo za použití obecných registrů budu emulovat dvojicí instrukcí, z nichž první přesune obsahu obecného registru do speciálního registru a druhá provede samotný posun.

Implementaci vidíme ve zdrojovém kódu 5.2. Na řádcích 1 a 2 je nastaveno označení emulované instrukce, informace o použitelnosti instrukce pro překladač a nastavení operandů, což jsou v našem případě tři registry. Na řádcích 3 – 5 je určena sémantika instrukce v mezikódu Codasip Studia. Do proměnných *%5* a *%7* jsou načteny obsahy registrů *gpreg_1* a *gpreg_2*. Na řádku 5 je proveden samotný bitový posun a výsledek je vložen do proměnné *%4*, která je poté uložena do registru *gpreg0*. Na řádcích 8 a 9 je pak určeno pomocí jakých instrukcí se požadovaná operace dosáhne. Na řádcích 10 a 11 je pak příklad konečné podoby instrukcí, které překladač vygeneruje vždy, když potřebuje provést bitový posun vlevo.

11 Jde o instrukce SETEQ, SETNEQ, SETSL, SETULT, SETSLE a SETULE.

12 Jde o kapitolu *Instrucion idioms.*

```
instr emul_sll, ok (0),
{ gpreg_0 = regop(gpreg), gpreg_1 = regop(gpreg),gpreg_2 = regop(gpreg) },
%5 = i32 gpreg_1;
%7 = i32 gpreg_2;
%4 = shl(%5,%7);
gpreg_0 = %4;
,
"SSL" gpreg_2 "\n\t"
"SLL" gpreg_0~"," gpreg_1 "\n\t"
----------------------------------------
SSL a6
SLL a3, a4

```

Zdrojový kód 5.2: Emulace bitového posunu vlevo, kdy jsou všemi operandy obecné registry

Pro správnou kompilaci je nutné pro každou architekturu vytvořit spouštěcí kód, který inicializuje zásobník, definuje funkce *exit* a *abort* a zavolá funkci *main.*

6 Popis procesoru sparc

SPARC je instrukční sada. Její první verze byla vytvořena firmou Sun Microsystems roku 1985. V současnosti. Architektura je otevřená a neproprietární.

Procesor se skládá z celočiselné jednotky (UI), jednotky pro operace s pohyblivou řádovou čárkou (FPU) a volitelně z koprocesoru (CP). Každá z částí má vlastní registry.

Procesor se může nacházet ve dvou stavech: *user* nebo *supervisor*. Procesor ve stavu *supervisor* může provádět i privilegované instrukce. Pokud se o provedení privilegované instrukce pokusí procesor ve stavu *user*, dojde k výjimce.

6.1 Adresování

Architektura je čistě typu big endian. Paměťový prostor je lineární 32bitový. Adresa je dána buď jako součet dvou registrů nebo jako součet registru s konstantou. Instrukce pro přístup do paměti obsahují identifikátor adresového prostoru (ASI), který určuje, zda je procesor spuštěn v módu *user* nebo *supervisor*, zda se přistupuje do prostoru pro instrukce nebo pro data.

6.2 Instrukční sada

Jedná se o instrukční sadu typu RISC. Instrukce jsou rozděleny do několika kategorií

- Instrukce pro načítání a ukládání

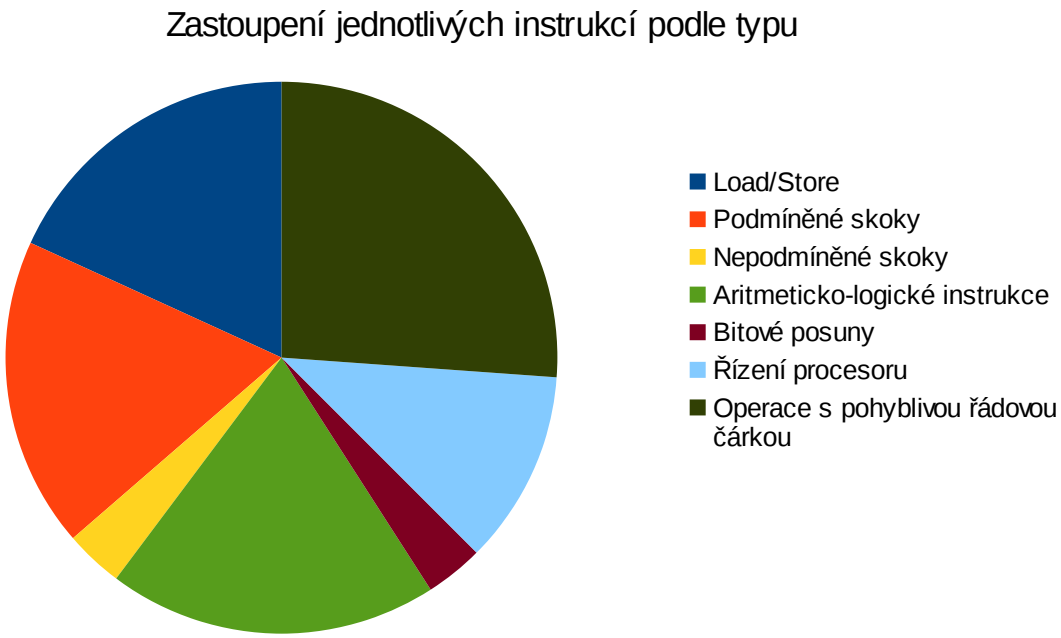
  Do paměti přistupují pouze k tomu určené instrukce. Je možné načítat a ukládat data po bytech (8 bitů), půlslovech (16 bitů), slovech (32 bitů) a dvouslovech (64 bitů), kdy je výsledek zapsán do dvou po sobě jsoucích registrů. Při načítání je možné zvolit, zda budou data načtena jako znaménková či bezznaménková.
- Aritmetické a logické instrukce s celými čisly

  Tyto instrukce pracují obvykle se třemi registry tak, že se výsledek operace prováděné na dvou registrech ukládá do třetího registru. Většina těchto instrukcí tvoří páry například *ADD* a *ADDcc*, přípona cc na konci instrukce znamená, že podle výdledku operace budou nastaveny příznaky v registru *PSR*, podle kterých se pak řídí běh programu.
- Instrukce pro řízení běhu

  Instrukční sada obsahuje instrukce pro přímý či nepřímý skok jako i pro podmíněné skoky. Přítomné jsou i skoky závislé na porovnání čísel v pohyblivé řádové čárce. Dále jsou zde instrukce pro volání a návrat z přerušení.
- *Instrukce pro operace s pohyblivou řádovou čárkou*

  Procesor obsahuje zvláštní jednotku pro operace s pohyblivou řádovou čárkou. Instrukce mají podobně jako Aritmetické a logické instrukce tři operandy. Jsou zde instrukce, které převádějí číslo z pohyblivé řádové čárky na celé číslo a naopak.
- Instrukce koprocesoru

  Tyto instrukce jsou vykonány na připojeném koprocesoru. Není-li koprocesor přítomen, vede jejich použití k vyvolání přerušení.

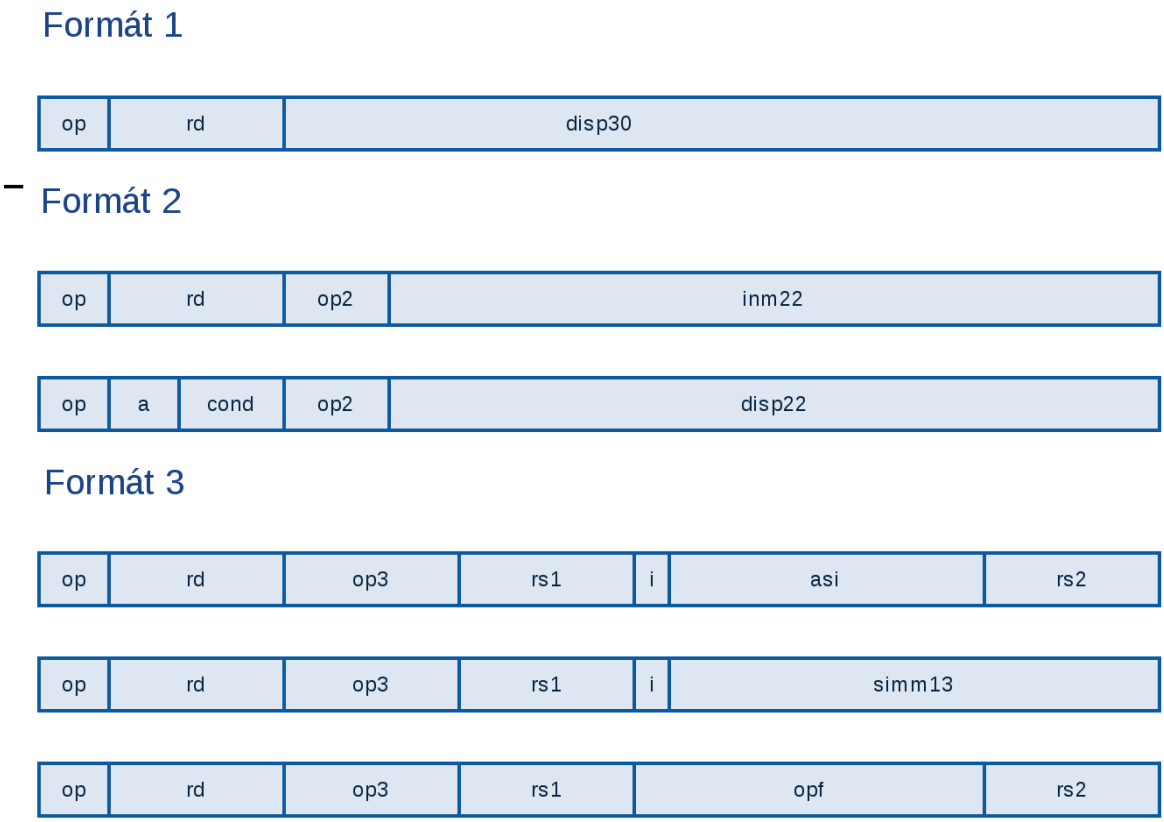


Graf 6.1 Rozložení instrukcí v procesoru Sparc

Počet instrukcí v těchto kategoriích znázorňuje graf 6.1.

Oproti procesoru Xtensa jsou zde navíc instrukce s pohyblivou řádovou čárkou, mezi které jsem zařadil i podmíněné skoky závisející na podmínce s čísly pohyblivou řádouvou čárkou i načítání a ukládání těchto čísel.

Je zde také výrazně více instrukcí typu *load/store*, protože je možné načítat a ukládat data také do koprocesoru.



Obrázek 6.2: Formáty instrukcí procesoru Sparc

Formát instrukcí je znázorněn na obrázku 6.2. Instrukce se dělí do tří hlavních instrukčních formátů podle hodnoty 2bitového operačního kódu op. Hodnota *00* určuje instrukci formátu 1, která existuje pouze jedna a to CALL, instrukce přímého skoku.

Hodnotou *01* jsou určeny instrukce formátu 2. Podle hodnoty 3bitového operačního kódu *op2* jsou rozlišeny dva typy instrukcí. První typ je odpovídá obsahuje adresu cílového registru *rd* a konstantu délky 22 bitů. Tento typ má například instrukce *SETHI*, která nastavuje horních 22 bitů registru *rd* na hodnotu *inm22*, případně instrukce *NOP* Druhý typ obsahuje *annul* bit a, který určuje zda se použije zpožděné vyhodnocení instrukce, 4bitový kód podmínky, který určuje podmínku skoku[13] na základě hodnot flagů *n*, *z*, *v* a *c* v registru *psr*.

Hodnoty *10* a *11* určují instrukce formátu 3, kam se řadí všechny ostatní instrukce. Podle operačního kódu *op3* se určí o jakou instrukci se jedná, *rs1* a *rs2* jsou adresu zdrojového registru. Podle bitu *i* se posledních 13 bitů interpretuje buď jako konstanta nebo jako indentifikátor adresového prostoru následovaný odresou druhého registru. Třetí typ instrukcí je určen hodnotou *op3* a obsahuje určení operace s pohyblivou řádovou čárkou nebo určení operace koprocesoru.

13 Např. „Branch Always“, „Branch on Greater“ atd.

6.3 Registry

Všechny registry jsou 32bitové. Procesor používá dva typy registrů: speciální registry a obecně účelné registry. Oběma typům bude věnována samostatná podkapitola.

6.3.1 Obecně účelné registry

Obecně účelných registrů **r** je v *IU* podle implementace 40 až 520. Jsou rozděleny na 8 globálních a volitelný počet sad po 16 restrech. V daném čase může instrukce přistupovat k 8 globálním registrům a 24 registrům, které obsahuje registrové okno. Jaké registry jsou přístupné v daném čase v registrovém okně závisí na hodnotě 5bitového pole CWR v PSR. Čtyři registry mají v architektuře určeno použití.

- První registr (r[0]) má konstantní hodnotu 0[14].
- Instrukce CALL zapisuje do registru r[15]
- Pokud nastane výjimka jsou hodnoty PC a nPC uloženy do registrů r[17] a r[18].

FPU obsahuje navíc 32 registrů **f** pro operace s pohyblivou řádovou čárkou. Instrukce má na rozdíl od r registrů přístup do všech 32 registrů.

6.3.2 Speciální registry

6.3.2.1 *Processor State Register* (PSR)

*Processor State Register o*bsahuje několik polí, kde jsou uloženy infromace o stavu procesoru.

Patří sem *integer_cond_codes*, kde jsou uloženy informace o poslední instrukci, která toto pole modifikovala. Pole se skládá ze čtyř bitů *n*, *z*, *v* a *c*. První dva určují, zda byl výsledek poslední instrukce záporný respektive nulový. Další dva bity určují, zda v instrukci došlo k přetečení respektive přenosu.

Dalšími poli jsou *enable_coprocessor* a *enable_floating-point*, které povolují koprocesor a operace s pohyblivou řádovou čárkou a *current_working_pointer CWP*, který určuje polohu registrového okna.

14 Při čtení vrací hodnotu 0, při pokusu o zápis se data zahodí.

6.3.2.2 Program Counters (PC, nPC)

Procesor obsahuje dva registry programového čítače. Registr PC obsahuje adresu aktuálně prováděné instrukce a nPC adresu instrukce, která má následovat. Pokud nastane výjimka jsou obsahy těchto registrů uloženy do dvou lokálních registrů.

6.3.2.3 Ostatní speciální registry

Procesor obsahuje další speciální registry. Například *Window Invalid Mask* (*WIM*) , který je ovládán procesorem ve stavu *supervisor*, který kontroluje, zda nedošlo k neočekávanému stavu registrového okna, který by měl být ošetřen výjimkou, *Trap Base Register (TBR)* , který uchovává adresu, ze které se bude pokračovat v případě výjimky a *Multiply/Divide Register (Y)* , který se používá pro rozšíření jednoho z operandů při násobení a dělení.

6.4 Použití

Procesor Sparc byl vyvinut firmou Sun a byl využíván v pracovních stanicích této firmy, kde nahradil procesor Motorola. V současnosti je používán v *aerospace* technologiích[15].

7 Model procesoru Sparc

Popis procesoru Sparc vychází z dokumentace[6].
Model má podobnou strukturu jako model procesoru Xtensa popsaný v kapitole 5. Zaměřím se tedy na záležitosti, ve kterých je odlišný od procesoru Xtensa.

7.1 Platforma a rozhraní

Modeloval jsem pouze jádro procesoru, nikoliv koprocesor či jednotku pro operace s pohyblivou řádovou čárkou, takže se problematika rozhraní redukuje na přístup do paměti. Paměť jsem oproti 4GB udávaných v dokumentaci zmenšil na cca 200 MB, aby bylo možné program rychleji ladit či automaticky testovat.

7.2 Registry

Codasip Studio neumožňuje existenci více programových čítačů, proto byly procesory nPC a PC modelovány jako jediný programový čítač PC. Registr nPC sloužil pro podporu zpožděného

15 Viz např. GR712RC Dual-Core LEON3FT SPARC V8 Processor. Aeroflex Gaisler: IP Cores [online]. [cit. 2015-05-18]. Dostupné z: http://www.gaisler.com/index.php/products/components/gr712rc

vykonávání instrukcí, která není na instrukční úrovni modelována, takže tímto zásahem není dotčena správnost modelu.

Stavový registr PSR byl redukován na čtyři příznaky *r_z*, *r_n*, *r_c* a *r_o*, které jsou modelovány jako čtyři 1bitové registry. Tyto přiznaky jsou využívany k řízení podmíněných skoků.

Dále model obsahuje 32 obecných registrů *regs* a registr *y* používaný při dělení a násobení.

7.3 Modelování instrukcí

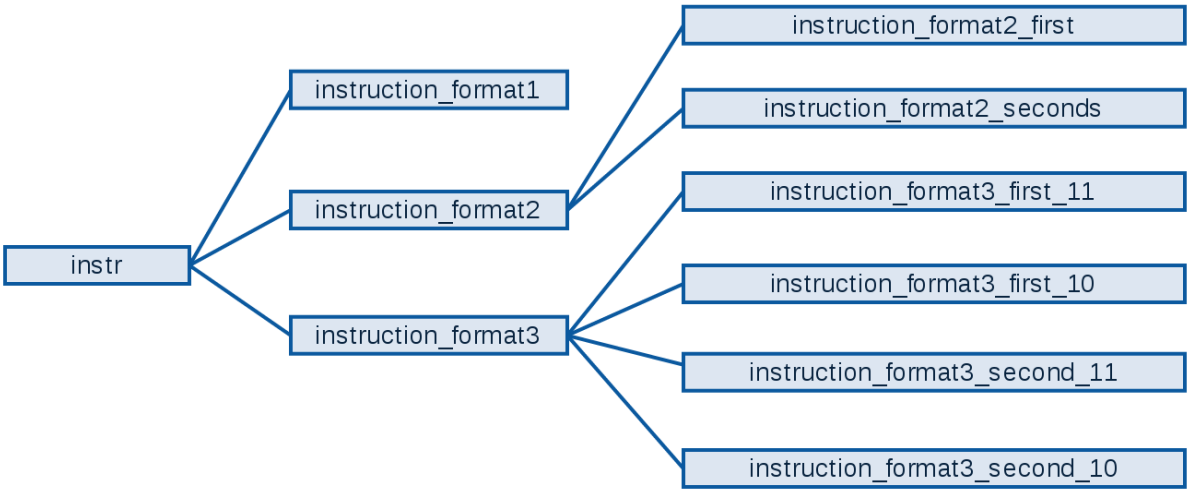


Schéma 7.1: Rozdělení instrukcí do množin na základě jejich formátu

Formát instrukcí procesoru Sparc popsaný v kapitole 5.2 je striktně dodržován, proto je i v mém modelu méně množin instrukcí (viz schéma 7.1), které jsou modelovány dohromady. Tím je výrazně zmenšena velikost kódu. Pro potřeby simulátoru je do modelu přidána instrukce *halt*.

7.4 Rozsah modelu

Model procesoru jsem implementoval podle dokumentace autora. Model je vytvořen na instrukční úrovni, proto není modelováno zpožděné vykonávání instrukcí. Dále není implementováno registrové okno.

7.5 Překladač

Také model procesoru Sparc bylo nutné upravit, aby mohl být vygenerován překladač jazyka C. Bylo nutné nastavit sémantiku v souboru *user_semantic.sem*.

Architektura sice obsahuje instrukce pro nepřímý skok a návrat z funkce, avšak generátor překladače jejich sémantiku nerozeznal. Takže jsem musel jejich sémantiku v souboru *user_semantic.sem* zjednosušit.

Dále bylo třeba nastavit způsob načtení 32bitové hodnoty do registru, což jsem učinil způsobem popsaným v dokumentaci procesoru pomocí kombinace instrukcí *SETHI*, která nastavuje horních 22 bitů registru s instrukcí *OR*, která provádí bitový součet.

Porovnání jsem emuloval pomocí instrukce *SUBcc*, která nastaví hodnoty příznaků podle rozdílu porovnávaných registrů. Výsledek je uložen do rezervovaného registru, aby operace neměla vedlejší efekt. Oproti reálnému procesoru se jedná o zhoršení, protože tam se příznaky nastavují po určených aritmetických či logických operacích automaticky a není nutné pro porovnávání provádět další operaci nebo rezervovat speciální registr.

Po aplikaci těchto úprav bylo možné překladač vygenerovat.

8 Srovnání procesorů

Oba procesory jsou 32bitové, instrukce procesoru Sparc jsou 32bitové, zatímco instrukce procesoru Xtensa jsou 24bitové, volitelně některé i 16bitové. Zatímco procesor Sprac je určen pro okamžité nasazení, procesor Xtensa je dodávám pouze v základní verzi a očekává se, že bude před použitím přizpůsobem vhodným výběrem rozšíření a uživatelských instrukcí.

V instrukční sadě je zásadní rozdíl v řízení běhu programu. Zatímco Xtensa používá podmíněné skoky, které závisí na porovnání dvou registrů, procesor Sparc obsahuje instrukce, které nastavují příznaky, kterými jsou pak skoky řízeny.

8.1 Srovnání pomocí Benchmarků

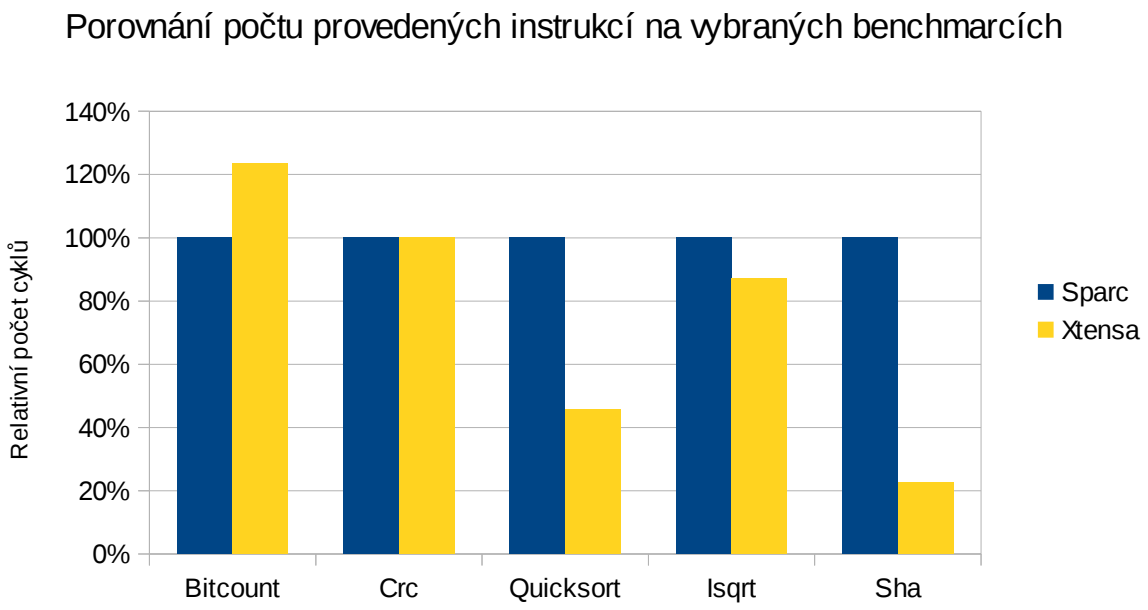
Procesory jsem srovnal pomocí několika benchmarkových testů. Použil jsem testy Mibench Vytvořené na Michiganské universitě. Autoři testy rozdělili do několika kategorií. Srovnání jsem prováděl v Codasip Studiu 3.4 se zapnutou optimalizací. Vybral jsem pět testů, pro jejichž běh nejsou potřeba žádné knihovny, tak aby zahrnovali různé kategorie výpočtů. Jejich zdrojové kódy jsou na přiloženém CD.

Jedná se o následující testy:

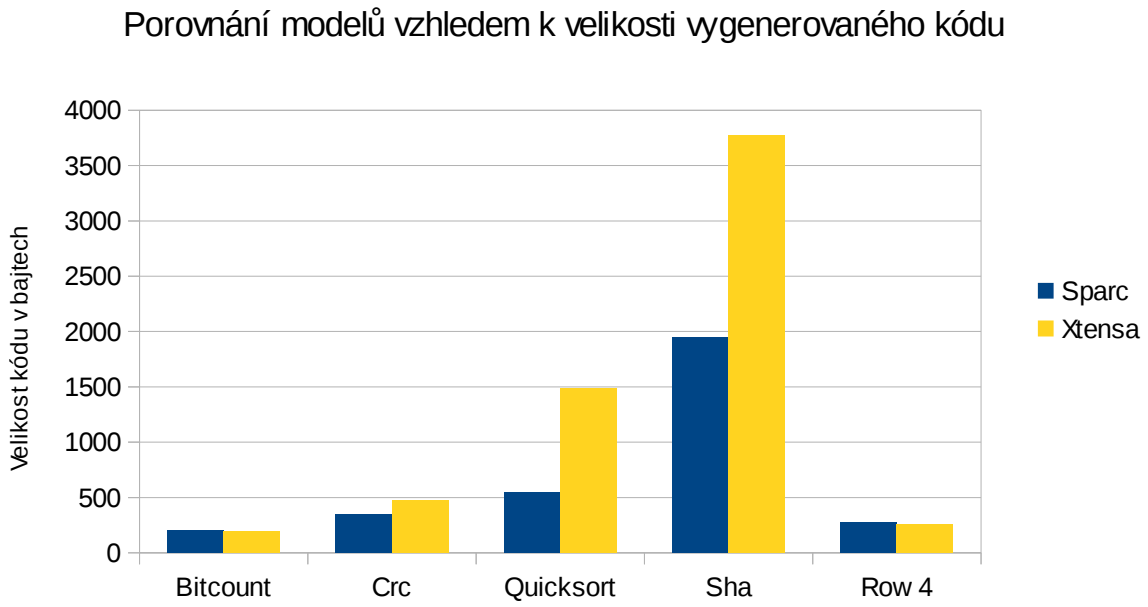
- Bitcount – Test počítá bity v poli dat, byl zařazen do kategorie *automobilový průmysl*
- Crc – Test počítá cyklický redundantní součet, používaný pro detekci chyb při přenosu dat. Patří do kategorie *sítě*

- Quicksort – Jedná se o známý řadící algoritmus, byl opět zařazen v kategorii *automobilový průmysl*
- Isqrt – Velmi jednoduchý program pouze vrací hodnotu
- Sha – Program provádí používaný hashovací algoritmus. Patří do kategorie *bezpečnost*

Procesory jsem porovnával podle dvou kritérií: podle počtu provedených instrukcí a podle velikosti vygenerovaného binárního kódu. Výsledky jsou znázorněny na grefech 8.1 a 8.2. Ve většině testů skončil lépe procesor Sparc. Zvlášť překvapivé je to při porovnání velikosti generovaného kódu když uvážíme, že instrukce procesoru Sprac mají 32 bitů, zatímco instrukce procesoru Xtensa pouze 24 bitů.



Graf 8.1: Výsledky testů podle počtu cyklů



Graf 8.2: Výsledky testů podle velikosti kódu

9 Závěr

V jakyku CodAL jsem úspěšně vytvořil modely dvou procesorů. Modely byly upraveny, aby k nim bylo možné vygenerovat simulátor, assembler a překladač jazyka C. Byly otestovány na sadě programů a zařazeny do portfolia firmy Codasip. Modely byly testovány simulací na vybraných benchmarcích a výsledky této simulace byly analizovány. Modely mohou být použity dalšími uživateli prostředí Codasip Studio nebo pro testování této aplikace.

Model těchto procesorů je možné rozšířit o model s přesností na úrovni cyklů, aby byly výsledky případných dalších simulací přesnější. Dále je možné doplnit do procesoru Xtensa rozšíření nabízená výrobcem a do procesoru Sparc vytvořit model koprocesoru a jednotky pro práci s čísly s pohyblivou řádovou čárkou a rozhraní těchto komponent.

Během práce jsem odhalil několik nepřesností ve fungování vývojového prostředí Codasip Studio, které se týkali například zacyklení generátoru překladače jazyka C při optimalizaci relativních skoků, nebo nenahlášení chyby, když jsem nesprávně nastavil jako ukazatel na zásobník a ukazatel na zásobníkový rámec stejný registr. Chyby byly nahlášeny a budou použity při údržbě a dalším vývoji této aplikace.

Při práci na těchto modelech jsem se detailně seznámil s architekturou dvou procesorů a jejich instrukční sadou. Prohloubil jsem své znalosti ADL jazyků a naučil se modelovat procesory v jazyku

CodAL. Dále jsem se seznámil s problematikou překladačů a s mezikódem používaným ve vývojovém frameworku.

Literatura

1. HENNESSY, John L. 2007. Computer architecture: a quantitative approach. 4th ed. San Francisco: Morgan Kaufmann, 1 sv. (různé stránkování). ISBN 01-237-0490-1.

2. HUSÁR Adam, PŘIKRYL Zdeněk, MASAŘÍK Karel and HRUŠKA Tomáš. ASIP Design using Architecture Description Language ISAC. In: ACACES 2009 - Poster Abstracts. Ghent: High Performance and Embedded Architecture and Compilation, 2009, ISBN 978-90-382-1467-2.

3. CODASIP S.R.O. CODASIP STUDIO MANUAL: User's Guide. 6.5. Brno, 2014.

4. CODASIP S.R.O. CodAL Manual: reference guide. 5.1. Brno, 2014.

5. TENSILICA, INC. Xtensa Instruction Set Architecture (ISA): Reference Manual. 2010.1. USA, 2010.

6. SPARC INTERNATIONAL, INC. The SPARC Architecture Manual. 8. USA, 1992. Dostupné také z: http://www.gaisler.com/index.php/downloads/leongrlib

10 Seznam příloh

- Příloha 1. Seznam rozšíření procesoru Xtensa

11 Seznam rozšíření procesoru Xtensa

Core Architecture

Code Density Option

Floating-Point Coprocessor Option

Miscellaneous Operations Option

Loop Option

MAC16 Option

Interrupt Option

Boolean Option

Extended L32R Option

16-bit Integer Multiply Option

32-bit Integer Multiply Option

Miscellaneous Operations Option

Coprocessor Option

Multiprocessor Synchronization Option

Conditional Store Option

Exception Option

Unaligned Exception Option

High-Priority Interrupt Option