VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

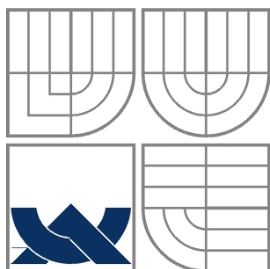
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# GENERÁTOR MANUÁLU INSTRUKČNÍ SADY

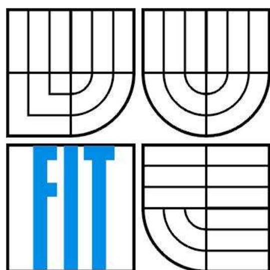
BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE MICHAL KŘEN
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# GENERÁTOR MANUÁLU INSTRUKČNÍ SADY
GENERATOR OF INSTRUCTION SET MANUAL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MICHAL KŘEN
AUTHOR

VEDOUCÍ PRÁCE — PROF. ING. TOMÁŠ HRUŠKA, CSC.
SUPERVISOR

BRNO 2010

## Abstrakt

Tato bakalářská práce se zabývá vytvořením generátoru manuálu instrukční sady, který je součástí projektu Lissom. Model mikroprocesoru je popsán v jazyce ISAC pro popis architektury mikroprocesoru i instrukční sady, kde si vývojář k jednotlivým deklaracím doplní speciálně formátované komentáře. Na základě těchto údajů jsou do manuálu vybrány pouze patřičné informace a nalezeny vztahy mezi nimi. Pro generování instrukcí je využit mezijazyk pro generátor překladače jazyka C. Výsledný dokument manuálu je uložen ve formátu RTF a obsahuje dvě části. V první je uveden přehled všech zdrojů procesoru a ve druhé je seznam všech instrukcí.



## Abstract

This bachelor thesis describes the design and implementation of a generator of instruction set manual, that is a part of the Lissom project. Model of microprocessor is described using architecture and instruction set description language ISAC with added special marked comments to each one declaration. From this source of data useful information and relationships between them are selected for manual. The source of data for generating of instructions is the intermediate-language for generator of C language compiler. The output generated manual document is saved as RTF file and it contains two parts. First part includes summary of all microprocessor’s resources and second part contains the list of all instructions.



## Klíčová slova

Instrukční sada, manuál, generátor, RTF, assembler, Lissom, ISAC, LISA, HW/SW co-design, ADL, mikroprocesor, C++

## Keywords

Instruction set, manual, generator, RTF, assembler, Lissom, ISAC, LISA, HW/SW co-design, ADL, microprocessor, C++

## Citace

Křen Michal: Generátor manuálu instrukční sady, bakalářská práce, Brno, FIT VUT v Brně, 2010

# Generátor manuálu instrukční sady

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením prof. Ing. Tomáše Hrušky, CSc.
Další informace mi poskytli Ing. Adam Husár, Ing. Karel Masařík, Ph.D. a další členové týmu Lissom.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Michal Křen
17. května 2010

## Poděkování

Na tomto místě bych chtěl poděkovat vedoucímu bakalářské práce prof. Ing. Tomáši Hruškovi, CSc. za jeho odborné rady při vytváření této práce.
Dále bych rád poděkoval Ing. Adamu Husárovi a Ing. Karlu Masaříkovi, Ph.D. za jejich cenné rady při konzultacích.

© Michal Křen, 2010
*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

Obsah ..... 1
1 Úvod ..... 3
2 Analýza problematiky ..... 4
2.1 Procesor ..... 4
2.2 Instrukční sada ..... 4
2.2.1 CISC ..... 5
2.2.2 RISC ..... 5
2.2.3 VLIW ..... 5
2.3 Manuál instrukční sady ..... 5
2.4 Jazyky pro popis architektury procesoru ..... 6
2.4.1 Jazyky pro popis architektury ..... 6
2.4.2 Jazyky pro popis instrukční sady ..... 7
2.4.3 Smíšené jazyky ..... 7
2.5 Jazyk ISAC ..... 8
2.5.1 Zdrojová část ..... 8
2.5.2 Operační část ..... 11
2.6 Stav projektu Lissom a cíl práce ..... 11
2.7 Formát dokumentu RTF a další formáty ..... 12
2.7.1 LaTex ..... 12
2.7.2 OpenDocument ..... 12
2.7.3 Office Open XML ..... 12
2.7.4 RTF ..... 13
3 Návrh generování manuálu instrukční sady ..... 14
3.1 Obecný návrh aplikace ..... 14
3.1.1 Rozhraní pro generování dokumentu ..... 15
3.2 Návrh generování zdrojů ..... 15
3.2.1 Přidání speciálně formátovaných komentářů ..... 16
3.2.2 Registr ..... 16
3.2.3 Sběrnice ..... 17
3.2.4 Ideální paměť a RAM ..... 18
3.2.5 CACHE ..... 19
3.2.6 Mapování paměti ..... 20
3.2.7 Struktury se zřetězeným zpracováním ..... 21
3.2.8 Logický obvod ..... 22

3.2.9 Entita ... 22
3.3 Návrh generování instrukcí ... 23
3.3.1 Přidání komentářů k instrukcím a skupinám ... 24
3.3.2 Generování komentářů a binární podoby ... 25
3.3.3 Zobrazení informací v manuálu ... 26
4 Implementace generátoru ... 27
4.1 Generování zdrojů ... 27
4.1.1 Registry a jejich aliasy ... 27
4.1.2 Mapování pamětí ... 28
4.1.3 Ostatní zdrojové prvky ... 28
4.2 Generování instrukcí ... 28
4.2.1 Komentáře a binární podoba ... 28
4.2.2 Komentář typu Doxygen ... 28
4.3 Generování dokumentu ... 29
4.3.1 Vytvoření dokumentu v RTF ... 29
5 Závěr ... 30
Literatura ... 31
Seznam příloh ... 32

# 1 Úvod

Téměř každý den se v našem okolí setkáváme s elektronickými zařízeními, kterými jsou např. řídící jednotky v automobilech, mobilní telefon, bankomat, avionika v letadlech, zdravotnické přístroje, domácí spotřebiče nebo také herní konzole atd. Ve všech těchto případech se jedná o vestavěné systémy. Ty se od klasických stolních počítačů odlišují především tím, že jsou většinou určeny pro vykonávání jedné specifické aplikace a není zde vyžadována přímá interakce s uživatelem. Jádrem takového vestavěného systému je obvykle procesor s aplikačně specifickými instrukcemi (ASIP), který je spolu s pamětí a dalšími periferiemi umístěn na jeden čip (SoC).

Návrhem těchto specifických systémů se zabývá oblast souběžného návrhu hardwaru a softwaru, kde jde především o nalezení ideálního poměru mezi cenou, spotřebou, velikostí, výkonem apod. Použití tradiční metodologie návrhu, při kterém je nejprve vytvořena hardwarová architektura a pro ni poté software, často nevede k nalezení nejlepšího řešení a také neúměrně narůstá doba vývoje. Z těchto důvodů se vyvíjí jazyky pro popis počítačových architektur a instrukčních sad, které právě umožňují souběžný návrh hardwaru a softwaru. Mezi tyto jazyky patří jazyk ISAC, který je vyvíjen na Fakultě informačních technologií Vysokého učení technického v Brně v rámci projektu Lissom. Tento projekt se také zaměřuje na vytvoření softwarových nástrojů dle popisu v jazyce ISAC, kterými jsou např. simulátor, překladač jazyka C, generátor hardwarové realizace procesoru apod. Více informací o této problematice lze dohledat v [1] a na webu projektu Lissom.

Tato bakalářská práce se zabývá vytvořením nástroje pro automatické generování manuálu instrukční sady daného mikroprocesoru na základě jeho popisu v jazyce ISAC. Manuál obsahuje dvě části. V první je uveden přehled zdrojů mikroprocesoru (registry, RAM apod.) a jejich vzájemné propojení. Druhá část obsahuje seznam všech instrukcí, kterým daný mikroprocesor rozumí. Výsledný manuál by se měl podobat definicím instrukční sady v manuálech některých komerčních mikroprocesorů.

Následující kapitola obsahuje teoretické základy celé práce. Zaměřuje se především na podrobnou analýzu jazyka ISAC z pohledu generování manuálu a také vytyčuje cíl práce. Poté kapitola s názvem „Návrh generování manuálu instrukční sady“ obsahuje koncept dílčích řešení jednotlivých problémů, které jsou poté implementovány. Právě o implementaci pojednává další kapitola. V poslední části je uvedeno zhodnocení dosažených výsledků, návaznost na ostatní projekty a také nástin dalšího možného rozšíření práce.

# 2 Analýza problematiky

Tato kapitola se zabývá popsáním teoretických a odborných základů zkoumané problematiky. V první části jsou vysvětleny základní pojmy související s instrukční sadou procesoru. Dále je kapitola zaměřena na jazyky popisující architekturu procesoru, zejména z rodin LISA. Poté je poměrně podrobně popsán jazyk ISAC, který je součástí projektu Lissom. Jsou zde také formulovány cíle této práce. Na závěr je pojednání o formátech dokumentu výsledného manuálu.

## 2.1 Procesor

Algoritmus můžeme implementovat použitím dvou základních metod. První možností je navrhnout aplikačně-specifický číslicový obvod. Výhodou je, že zde dochází k optimálnímu využití technických prostředků, ovšem nevýhodou je vysoká cena a doba návrhu. Druhou možností je implementace pomocí programu běžícího na procesoru, kde dochází k sekvenčnímu vykonávání jednotlivých instrukcí. Výhodou tohoto přístupu je nízká cena, univerzálnost a flexibilita algoritmu. Nevýhodou je nižší výkonnost.

Procesor obsahuje předem definované elementární operace, které dokáží realizovat libovolný algoritmus. Typickou činností procesoru je transformovat vstupní data dle předpisu elementární operace (instrukce) na výstupní data. Posloupnost jednotlivých instrukcí je definována v programu. Samotný procesor je složený z těchto hlavních částí:

- programový čítač (PC), který uchovává adresu nadcházející instrukce,
- instrukční registr (IR), který uchovává aktuálně zpracovávanou instrukci,
- řadič, který řídí činnost procesoru,
- aritmeticko-logická jednotka (ALU), ve které jsou prováděny matematické operace,
- vstupně-výstupní jednotka (I/O), která umožňuje přenos dat z a do procesoru a
- sada registrů.

Vykonání jedné instrukce probíhá následovně. Nejprve je instrukce načtena, tedy hodnota adresy určená obsahem PC je uložena do IR (instrukčního registru). Poté je instrukce dekódována, což znamená nastavení řídících signálů pro dílčí komponenty procesoru. Na závěr je instrukce provedena, výsledek je uložen na patřičné místo a určí se nový obsah PC [5].

## 2.2 Instrukční sada

Instrukční sada je množina elementárních operací, které podporuje daný procesor. Každá instrukce se obvykle skládá z operačního kódu a operandů. Počet operandů může být různý v závislosti na použité architektuře procesoru. Akumulátorové a zásobníkové architektury mají obvykle malý počet operandů. Naopak registrové architektury mají obvykle dva až tři operandy. Každá instrukce má také zpravidla symbolickou reprezentaci (zkratka naznačující co daná instrukce dělá). Obecný algoritmus lze tedy popsat v jazyce symbolických instrukcí (assembly language). Poté může překladač (asembler) přeložit tyto symbolické instrukce do binární podoby. Samotné instrukce můžeme rozdělit dle jejich funkce na např. aritmetické a logické instrukce, instrukce pro přesuny dat, instrukce pro

předávání řízení apod. V následujících podkapitolách budou zmíněny zajímavé kategorie instrukčních sad [6].

### 2.2.1 CISC

Kategorie CISC (Complex Instruction Set Computer) se vyznačuje poměrně vysokým počtem složitějších instrukcí, které se obvykle dají ještě rozložit na elementárnější instrukce. Pro jejich vykonávání se obvykle používá mikroprogramový řadič, který realizuje pro každou instrukci sekvenci mikrooperací.

### 2.2.2 RISC

Kategorie RISC (Reduced Instruction Set Computer) se naopak vyznačuje malým počtem jednodušších instrukcí. Ty jsou charakteristické tím, že mají stejnou dobu vykonávání, pevnou délku a jednotný styl kódování. Obsahují obvykle vetší počet registrů a všechny operace probíhají nad nimi. Paměťové operace jsou realizovány pomocí vyhrazených instrukcí.

### 2.2.3 VLIW

Procesory využívající velmi dlouhé instrukční slova VLIW (Very Long Instruction Word) jsou navrženy za účelem zvýšení výkonnosti využitím paralelismu. Zde je použita technika paralelismu na úrovni instrukcí ILP (Instruction-Level Parallelism), kde se nejprve zjistí, zda jsou instrukce na sobě nezávislé a poté jsou vykonávány současně. O tuto činnost se stará kompilátor. Samotný procesor se skládá z několika samostatně pracujících jednotek.

## 2.3 Manuál instrukční sady

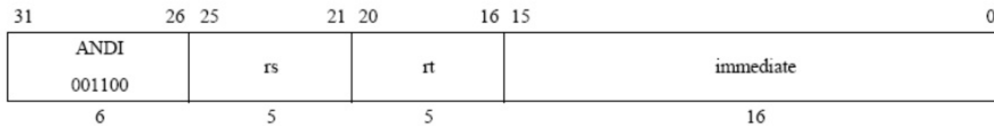
Manuál instrukční sady konkrétního procesoru obsahuje přehled všech instrukcí, které může vývojář v daném mikroprocesoru používat. Z důvodu existence velkého množství různých architektur procesorů a různých druhů použitých instrukčních sad existuje také velké množství různých manuálů. Ty se liší především mírou podrobnosti informací u každé instrukce, způsobem textového formátování a množstvím údajů o samotné fyzické realizaci procesoru. Ukázku manuálů převzatých z procesorů M68HC08 (nahoře) a MIPS32 (dole) lze vidět na obrázku 2.1.

Téměř každý manuál obsahuje tyto základní informace:

- syntaxi instrukce – symbolická reprezentace a operandy určující zápis v jazyce symbolických instrukcí,
- binární kódování – určující jakým způsobem bude instrukce zakódována do binární podoby,
- textový popis – slovní komentář k instrukci,
- sémantiku instrukce – většinou ve formě matematického zápisu dané operace,
- počet cyklů – určující počet cyklů po dobu jedné instrukce.

| Source Form | Operation | Description | Effect on CCR | | | | | | Address Mode | Opcode | Operand | Cycles |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | | | |
| ADC #*opr* | Add with Carry | A ← (A) + (M) + (C) | ↕ | ↕ | – | ↕ | ↕ | ↕ | IMM | A9 | ii | 2 |
| ADC *opr* | | | | | | | | | DIR | B9 | dd | 3 |
| ADC *opr* | | | | | | | | | EXT | C9 | hh ll | 4 |
| ADC *opr*,X | | | | | | | | | IX2 | D9 | ee ff | 4 |
| ADC *opr*,X | | | | | | | | | IX1 | E9 | ff | 3 |
| ADC ,X | | | | | | | | | IX | F9 | | 2 |
| ADC *opr*,SP | | | | | | | | | SP1 | 9EE9 | ff | 4 |
| ADC *opr*,SP | | | | | | | | | SP2 | 9ED9 | ee ff | 5 |

**And Immediate** ANDI




**Format:** `ANDI rt, rs, immediate` MIPS32 (MIPS I)

**Purpose:**

To do a bitwise logical AND with a constant

**Description:** `rt ← rs AND immediate`

*Obrázek 2.1: Ukázka manuálů instrukční sady (M68HC08 a MIPS32)*

# 2.4 Jazyky pro popis architektury procesoru

Jazyky pro popis hardwaru (HDL), mezi které patří např. VHDL a Verilog, slouží především pro modelování elektronických systémů. Procesor popsaný tímto jazykem můžeme simulovat na úrovni logických obvodů, ale informace např. o instrukční sadě zde chybí. Právě tyto nedostatky překonávají jazyky pro popis architektury procesorů (ADL), které umožňují automatické generování softwarových nástrojů sloužící pro vývoj procesoru.

Dle [1] jsou tyto jazyky rozděleny do tří kategorií:

- jazyky pro popis struktury,
- jazyky pro popis instrukční sady a
- smíšené jazyky.

Odlišují se hlavně v míře použité abstrakce. Obecně platí, že při snížení stupně abstrakce nám vzrůstá obecnost popisovaného systému. Následující informace v této kapitole jsou také čerpány z [4].

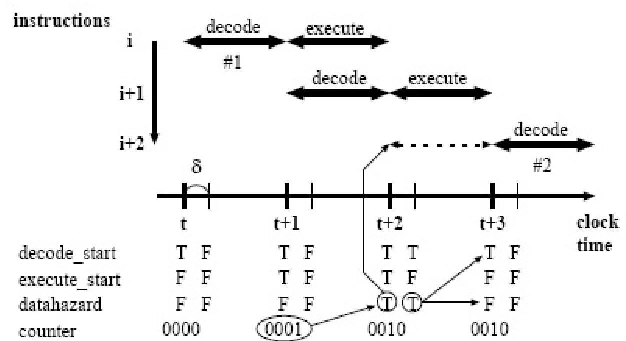
## 2.4.1 Jazyky pro popis architektury

Do kategorie strukturálních jazyků patří např. jazyk MIMOLA (Machine Independent Microprogramming Language). Zde je jako abstrakce použita úroveň meziregistrových přenosů RTL (Register Transfer Level), která se také používá u jazyků pro popis hardwaru. Tato úroveň zachycuje přenosy dat mezi jednotlivými hardwarovými zdroji a logické operace provedené v rámci nich. RTL obecně tedy modeluje chování synchronních číslicových obvodů. Jazyk MIMOLA obsahuje dvě části. První hardwarová část specifikuje mikroarchitekturu na úrovni RTL. Tímto způsobem jsou tedy popsány jednotlivé komponenty architektury a jejich vzájemné propojení. Instrukční sada je generována z tohoto popisu a modulu instrukčního dekodéru. Za účelem vytvoření nástroje

překladače zde musí být dodatečně definovány např. čítač instrukcí, instrukční paměť atd. Druhá softwarová část popisuje aplikační program. Jazyk umožňuje např. nahlížet na fyzické registry jako na proměnné. Výhoda vůči jazykům pro popis hardwaru spočívá v tom, že jediný popis slouží kromě samotné syntézy hardwaru také pro generování softwarových nástrojů (simulátor, generátor kódu apod.). Ovšem jejich nevýhodou je, že z důvodu použité nízké abstrakce je simulování procesoru pomalé.

Dalším jazykem spadajícím do této kategorie je jazyk AIDL (Architecture and Implementation-level Description Language), který dle [3] urychluje 2,4x návrh procesorů oproti použití jazyka pro popis hardwaru VHDL. Je toho docíleno použitím vyšší abstrakce (úroveň architektury a implementace) a možností experimentovat s instrukční sadou již v rané fázi vývoje. Jazyk ovšem postrádá formální popis binárního kódování instrukce a syntaxi jejího asembleru a tudíž neumožňuje automatické generování překladačů. Na obrázku 2.2 lze v levé části vidět ukázku jazyka AIDL popisující linku zřetězení (pipeline). V pravé části je tento kód vysvětlen na časovém diagramu.

```
1 stage decode(decode_start == TRUE
2               && datahazard != TRUE){
3   decode_start <- FALSE;//
4   block(1){
5     execute_start <- TRUE;
6     decode_start <- TRUE;
7     counter<0:3> <- counter<0:3> + 'b0001;
8     if (counter<3> == 'b1) then
9       datahazard <- TRUE;
10    endif;
11 }}
12 stage execute(execute_start == TRUE){
13   execute_start <- FALSE;//
14   block(1){
15     if (datahazard == TRUE) then
16       datahazard <- FALSE;
17       decode_start <- TRUE;
18     endif;
19 }}
```




*Obrázek 2.2: Ukázka popisu pipeline v jazyce AIDL a vysvětlení provádění – převzato z [3]*

## 2.4.2 Jazyky pro popis instrukční sady

Další skupinou jsou jazyky pro popis instrukční sady, které používají vysoký stupeň abstrakce. Detailní popis hardwaru je zde ignorován a instrukce jsou na sémantické úrovni specifikovány opět pomocí meziregistrových přenosů. Výsledný popis architektury se tedy s trochou nadsázky může podobat referenčnímu manuálu instrukční sady. Mezi zástupce patří například formalismus nML, vyvíjený na Technické univerzitě v Berlíně. Výhodou je, že umožňují tvorbu softwarových nástrojů a to hlavně překladače vyšších programovacích jazyků. Ovšem díky chybějícímu detailnímu popisu hardwaru neumožňují použít simulátor založený na cyklech a popsat architektury s linkami zřetězení.

## 2.4.3 Smíšené jazyky

Z důvodu nedostatků dvou výše zmíněných skupin jazyků vznikají smíšené jazyky, které chtějí eliminovat jejich nedostatky, tedy limitovat množství informací, které nejsou potřebné pro rychlou simulaci procesoru, a naopak přidat informace například o linkách zřetězení apod. Opět jsou zde použity meziregistrové přenosy pro specifikaci sémantiky jazyka. Mezi zástupce z této kategorie patří jazyk EXPRESSION, který obsahuje vyvážený pohled jednak na specifikaci chování a také na specifikaci struktury. Do specifikace chování lze zahrnout přesný popis operací, instrukcí a mapování operací a ve specifikaci struktury jsou popsány jednotlivé komponenty architektury, datové cesty a paměťový subsystém. Ve specifikaci chování ovšem chybí informace o binárním kódování, syntaxi

asembleru a také neumožňuje vytváření hierarchie operací . Charakteristika procesoru je založena na cyklech.

Dalším významným jazykem spadajícím do kategorie smíšených ADL je jazyk LISA (Language for Instruction-Set Architecture). Skládá se ze dvou částí. První část tvoří deklarace hardwarových zdrojů a druhá část je tvořena popisem operací. Zde se způsob vyjádření operací přibližuje zápisu nML a podporuje také jejich hierarchii. Hlavní výhodou je, že z jednoho popisu umožňuje automaticky generovat všechny nástroje pro programování a simulaci ASIP. Jazyk LISA se po syntaktické stránce velmi podobá programovacímu jazyku C.

# 2.5 Jazyk ISAC

Dle výše uvedeného dělení spadá jazyk ISAC (Instruction Set Architecture C) do třetí kategorie smíšených jazyků ADL. Jazyk vznikl v rámci projektu Lissom a více informací o něm lze dohledat v [2]. Vychází z jazyka LISA, se kterým má například společná některá klíčová slova, a taktéž se skládá ze dvou základních částí:

- zdrojová část obsahuje deklaraci zdrojů, ze kterých je modelovaný procesor složen a
- operační část obsahuje popis operací a skupin.

## 2.5.1 Zdrojová část

Sekce popisující zdrojovou část v jazyce ISAC začíná vždy klíčovým slovem `RESOURCE`. Zde jsou definovány všechny zdroje modelu, kterými např. jsou registry, paměti, linky zřetězeného zpracování apod. V následujícím textu budou tyto zdroje popsány.

**Registry**

Deklarace registru začíná vždy klíčovým slovem `REGISTER`. Před ním mohou být uvedena klíčová slova `PC` a `CR`, které znamenají o jaký speciální typ registru se jedná. V prvním případě o programový čítač, který smí byt pouze jeden v celém modelu, a ve druhém případě se jedná o kontrolní registr. Tyto registry jsou nezbytné pro simulátor. Pokud neuvedeme dvě výše zmíněná klíčová slova před zápisem registru, jedná se o registr pro běžné použití. Parametr `bit` specifikuje jeho bitovou šířku. Zde je uveden příklad zápisu registru v jazyce ISAC:

```
/**
 * Program Counter
 */
PC REGISTER  bit[16]    pc;
```

Jedná se o 16-bitový registr programového čítače. Za povšimnutí stojí speciálně formátované komentáře, které překladač jazyka ISAC neignoruje. Slouží právě pro slovní popis zdrojových prvků a jejich využití v generátoru manuálu instrukční sady je popsáno v následující kapitole zabývající se návrhem.

**Obecný hardwarový prvek**

Jeho deklarace je pomocí klíčového slova `LC`. Popisuje tedy obecný logický obvod, který má předem nedefinované chování. To je specifikováno až v operační části. Následující čtyři prvky spadají do kategorie paměťových prvků.

### Ideální paměť

V jazyce ISAC má ideální paměť klíčové slovo MEMORY. Mezi její parametry patří velikost bloku a případně podbloku v bitech (BLOCKSIZE). Poté velikost (SIZE) specifikující počet bloků a na závěr příznaky (FLAGS) udávající zda je paměť určena pro čtení, zápis a či obsahuje proveditelný kód. Obecně by programová paměť měla mít příznaky pro čtení a vykonávaní proveditelného kódu a datová paměť by měla být určena pro čtení a zápis.

### RAM

Neideální komponenta RAM (random-access memory) je deklarována pomocí stejného slova RAM. Vůči ideální paměti přidává další parametry a to endianitu (ENDIANESS). Ta udává v jakém pořadí jsou uloženy jednotlivé bajty paměti. Povoluje hodnoty Little-endian, kde je nejméně významný bajt uložen na místo s nejnižší adresou, a Big-endian, kde je naopak nejvíce významný bajt uložen na místo s nejnižší adresou. Dalším parametrem je zpoždění (LATENCY), které specifikuje latenci pro operaci čtení a zápisu.

### Cache

Někdy také překládána jako rychlá vyrovnávací paměť je deklarována užitím klíčového slova CACHE. Ta přidává k parametrům RAM navíc parametry NUM_WAYS specifikující počet cest pro set-asociativní cache. Pokud je hodnota parametru rovna jedné, znamená to, že je cache mapována přímo. Dalším parametrem je LINESIZE udávající počet bloků na jeden řádek a její hodnoty musí být mocniny dvou. Parametry WA_POLICY a WB_POLICY definují strategii write-allocate/back pro cache. Posledním parametrem je CONNECT definující paměťový modul, ke kterému je CACHE připojena. Povolenými paměťovými prvky na připojení jsou RAM, CACHE a sběrnice. Důležité je, aby odpovídala jejich bitová šířka.

### Sběrnice

Posledním paměťovým prvkem je sběrnice, která je deklarována pomocí klíčového slova BUS. Jejími parametry jsou BLOCKSIZE a ENDIANESS. Konektivita sběrnic je popsána v části, kde se mapuje paměťový prostor. Během testování generování zdrojů do manuálu instrukční sady jsem zjistil, že se u sběrnice nachází nepotřebný parametr DATAPORT, který udával kolik čtení a zápisů se může provést v jednom taktu. Tento parametr byl následně odstraněn z gramatiky sběrnice.

### Mapování pamětí

Pro simulátor je důležité jasné určení, do kterého adresového prostoru je mapována příslušná paměť. Jazyk ISAC požaduje vždy jedno implicitní mapování použitím identifikátoru defaultmap (nejedná se o klíčové slovo jazyka) a také je umožněno definovat si další mapování. Mapování pamětí vždy začíná klíčovým slovem MEMORY_MAP a následuje seznam parametrů RANGE udávající adresový rozsah mapovaných prvků. Ukázka:

```
/** Default Memory Map */
MEMORY_MAP defaultmap
{
    BUS(testBUS), RANGE(10, 0xC8)->program_mem [(9..8)][(7..0)];
    BUS(testBUS), RANGE(201, 255)->reg_mem [(5..0)];
};
```

Výše uvedený kód je ukázkou mapování dvou paměťových prvků. Oba dva jsou připojeny na sběrnici testBUS. V prvním případě by byla výše deklarována programová paměť typu RAM obsahující 4 banky. Ta je dle uvedeného zápisu mapována na adresu od 0x0A (ve zdrojovém kódu je uvedena adresa v desítkové soustavě) do adresy 0xC8. Dva nejvýznamnější bity adresy - 9. a 8. jsou použity pro určení banku a ostatní bity jsou použity pro adresování buněk. V druhém případě se jedná o registrové pole obsahující celkem 70 8-bitových registrů. Adresa se určuje dle 5. až 0. bitu.

### Struktury se zřetězeným zpracováním

Myšlenkou zřetězeného zpracování (pipeline) je umožnit vykonávat více instrukcí současně. Respektive, že se v procesoru v jednom okamžiku může nacházet více instrukcí v různém stádiu rozpracovanosti. Pokud by nebylo použito pipeline, tak například během fáze načítání instrukce z paměti jsou nevyužity ty části procesoru, které slouží např. pro násobení. Při vykonávání samotného násobení by tomu bylo zase naopak. Současným prováděním více instrukcí je tedy možné zvýšit výkon procesoru. Princip spočívá v rozdělení vykonávané instrukce na fáze (např. načtení instrukce z paměti, dekódování, provedení atd.), která realizuje pouze část z celkového provedení instrukce. Je nutné si ukládat stavy instrukcí a příslušné mezivýsledky do registrů.

V jazyce ISAC je zřetězené zpracování deklarováno užitím klíčového slova `PIPELINE`. Další zápis je vysvětlen na ukázce převzaté z [2]:

```
REGISTER   bit[24]    src1, src2;
REGISTER   bit[24]    letch1, letch2, letch3;

PIPELINE pipe
{
      FE:   [latch1]   (src1);
      DC:   [latch2]   (src2);
      EX:   [latch3];
      WB:   ;
};
```

Jak je z ukázky patrné, za klíčovým slovem `PIPELINE` následuje výčet jednotlivých fázi zřetězeného zpracování. V hranatých závorkách jsou poté uvedeny oddělovací registry pro uložení mezivýsledků a v kulatý jsou stavové registry.

### Alias

Jazyk ISAC umožňuje deklarovat použitím klíčového slova `ALIAS` virtuální zdroj, který nemá exaktní ekvivalent v hardwaru. Alias je dovolen vytvářet pro registry a paměťové prvky. Příklad:

```
REGISTER bit[8] TEST[4];
bit[1] x9 ALIAS { TEST[1] bit[5]  };
bit[4] x7[4] ALIAS { TEST[1]  };
```

V prvním případě je vytvořen jedno-bitový alias x9 do banku s indexem 1 na 5. bit registru TEST. Ve druhém případě je vytvořen 4-bitový alias x7, obsahující 4 banky. Ten je mapován do banku s indexem 1 s bitovým rozsahem 3..0 a 7..4 a dále pokračuje do banku s indexem 2 se stejným bitovým rozsahem – celkem tedy alias obsahuje zmíněné 4 banky. Během testování aliasů jsem zjistil, že překladač jazyka ISAC je poměrně benevolentní v zápisu aliasů, což v určitých případech (hlavně aliasy skrze více banků) vedlo k nepodporovanému chování. Proto byl zaveden striktnější zápis aliasů a také bylo zakázáno vytvářet alias na bankované paměťové prvky.

**Entita**

Nově podporovaným prvkem je entita, která je deklarována užitím klíčového slova `ENTITY`. Po deklaraci následuje výčet jejích prvků, kterými můžou být registr, paměťové prvky, pipeline, logický obvod a alias. Následně může být vytvořena instance této entity s užitím parametrů. Během testování jsem zjistil, že jazyk ISAC chybně umožňoval zápis mapování paměti v entitě. V současnosti je tato chyba již opravena.

### 2.5.2 Operační část

Na nejvyšší úrovni existují dvě možnosti definice:

- operace OPERATION a
- skupiny GROUP

Každá operace i skupina má jedinečné jméno. Pro skupiny je typické, že sdružují operace podobného typu. U skupiny lze také použít klíčové slovo `REPRESENTS`, které udává mezijazyku pro generátor překladače, že nemá brát v úvahu všechny možné varianty této skupiny, ale nahlížet na ni jako na celek. Operace je popsána v následujících sekcích:

- Sekce INSTANCE deklaruje, které operace a skuliny lze používat v dané operaci. Pouze u této sekce se připouští deklarace více než jedenkrát v rámci operace.
- Sekce CODING popisuje binární zakódování instrukce.
- Sekce ASSEMBLER popisuje syntaxi asembleru.
- Sekce BEHAVIOR a EXPRESSION popisují chování operace, které je poté během simulace prováděno.
- Sekce ACTIVATION popisuje časování jiných operací vzhledem k popisované.
- Sekce CODINGROOT provádí dekódování instance operace na základě instrukce uložené ve zdroji.

Tato kapitola o jazyku ISAC vychází z [2] a byla doplněna o vlastní poznatky a aktuální změny.

## 2.6 Stav projektu Lissom a cíl práce

Jak již bylo uvedeno v úvodu, tato bakalářská práce je součásti projektu Lissom, který je realizován na Fakultě informačních technologií VUT v Brně. V čase zadání této práce (říjen 2009) fungovala většina generujících se softwarových nástrojů. Také vývojové prostředí Eclissom (využívající Eclipse) umožňuje základní popis modelu procesoru v jazyce ISAC, následné připojení k Middlewaru (může být použit testovací server na FIT či lokální instalace), vytvoření nástrojů a simulování kódu v asembleru.

Pro generování zdrojů do manuálu jsou všechny potřebné části již implementované. Překladač jazyka C je v době psaní této práce ve vývoji. Mezijazyk pro generátor tohoto překladače se již generuje a pro např. procesor MIPS funguje dobře. Problém je např. s procesorem ARM, kde se generuje příliš mnoho instrukcí. Ovšem syntaktický analyzátor mezijazyka (cgirparserl) nebyl na konci dubna 2010 ještě hotový. Pro generování instrukční sady do manuálu se právě využívá tento mezijazyk.

Cílem této bakalářské práce je seznámit se s projektem Lissom, nastudovat si jazyky pro popis počítačových architektur a to zejména vyvíjený jazyk ISAC. A také si osvojit práci se zdrojovými kódy v projektu a zvládnout formát manuálu RTF. Tato teoretická část práce je popsána převážně v této kapitole. Následně na základě těchto nastudovaných znalostí je navrhnout a implementován generátor manuálu instrukční sady. O tom pojednávají další kapitoly.

## 2.7 Formát dokumentu RTF a další formáty

Úkolem této práce je také analyzovat, zda zvolený formát manuálu instrukční sady RTF je nejlepším možným. Požadavkem je, aby výsledný dokument šlo snadno editovat z důvodu např. doplnění určitých částí do manuálu, které nelze generovat. Tento požadavek v podstatě vylučuje formát PDF a další podobné formáty, jelikož je nelze jednoduše editovat použitím běžných textových editorů. Také webový formát dokumentů HTML není vhodný, protože není primárně určen pro tisk. Pokud dále vezmeme v potaz, že formát by měl umožňovat pokročilejší sazbu textu (různé fonty, tabulky, okraje atd.), tak nám zbývají formáty RTF, LaTex, Office Open XML, OpenDocument apod. O nich pojednávají další podkapitoly.

### 2.7.1 LaTex

LaTex rozšiřuje sázecí systém Tex o řadu maker, což vede k uživatelsky přívětivějšímu vytváření dokumentů. Svým zaměřením je vhodný převážně na sazbu matematických, jazykových, vědeckých a také inženýrských dokumentů. Jedná se o značkovací jazyk a tedy dokument je uložen v textové podobě. Tento jazyk je určen pro ruční psaní jednotlivých sázecích příkazů v textovém editoru, ovšem existují i editory generující tento jazyk. Hlavně z tohoto důvodu, že by vývojář musel ovládat LaTex nebo používat konkrétní editor a dále překladač, který by vytvořil například PDF soubor, je tento formát nevhodný pro manuál instrukční sady.

### 2.7.2 OpenDocument

OpenDocument (ODF) vychází ze staršího souborového formátu používaného aplikacemi OpenOffice.org a ty jej také samozřejmě mají jako svůj nativní formát. Jedná se o otevřený souborový formát založený na XML. V současné době existuje velké množství editorů, které tento formát podporují včetně např. nejnovějších Microsoft Office a nejnovějšího Wordpadu ve Windows 7. Existuje také množství knihoven, převážně ve vyšších programovacích jazycích, umožňující generování dokumentů v tomto formátu.

### 2.7.3 Office Open XML

Office Open XML (OOXML) je formát ukládání dokumentů vyvinutý firmou Microsoft. Poprvé byl použit v Microsoft Office 2007. Jedná se o ZIP soubor, který v sobě obsahuje XML a další potřebné soubory. Dokument pro textový procesor má příponu docx. Existují knihovny převážně pro platformu .NET, které umožňují generovat tento dokument.

### 2.7.4 RTF

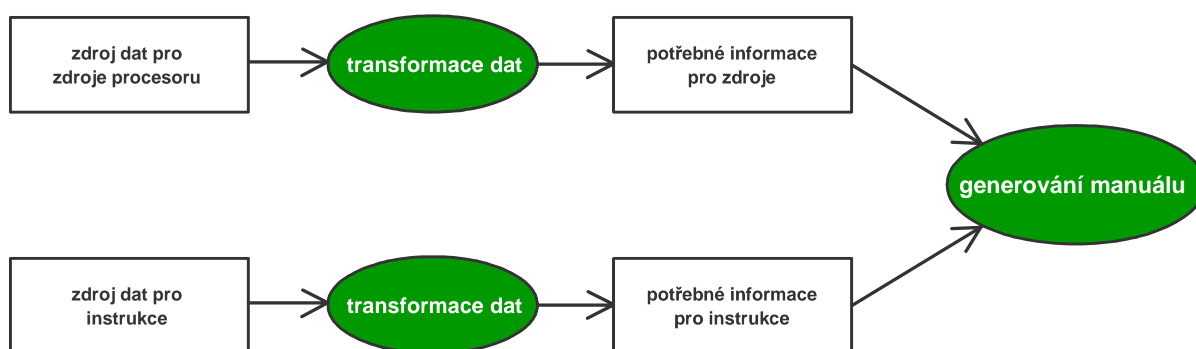
Naopak vzhledem k formátu LaTex formát dokumentu RTF není určen pro ruční psaní příkazů, ale pro využití textových procesorů generující tento formát. RTF je vyvíjený společností Microsoft jako na platformě nezávislý formát souborů. Tento formát má více než dvacetiletou historii, a proto jsou jeho konstrukce ustáleny a základní formátování je vykreslováno ve většině editorů stejně. Právě podpora většiny textových procesorů je jedna z největších výhod tohoto formátu. Dokonce lze RTF dokument číst i na přenosném přehrávači hudby iPod od společnosti Apple a také Google dokumenty jej podporují. To jistě vypovídá o faktu, že se nejedná o mrtvý formát a je zde velká pravděpodobnost, že i za 10 let jej budou editory podporovat v nezměněné verzi. Ve většině základních instalací operačního systému existuje aplikace, která podporuje tento formát. V poslední řadě je velkou výhodou RTF, že jej lze snadno generovat s použitím libovolného programovacího jazyka [7]. Z těchto důvodů se formát RTF jeví jako nejvhodnějším výstupem generátoru manuálu instrukční sady.

# 3 Návrh generování manuálu instrukční sady

Tato kapitola vychází z teoretických základů uvedených v předešlé kapitole. Z obecného pohledu se zabývá návrhem získání potřebných dat pro generování manuálu instrukční sady a konceptem tisku těchto údajů do dokumentu. V první části je navrhnutý princip realizace celé aplikace. Dále je tato kapitola rozdělena na dvě velké části. První řeší problematiku generování zdrojů procesoru a druhá generování operací.

## 3.1 Obecný návrh aplikace

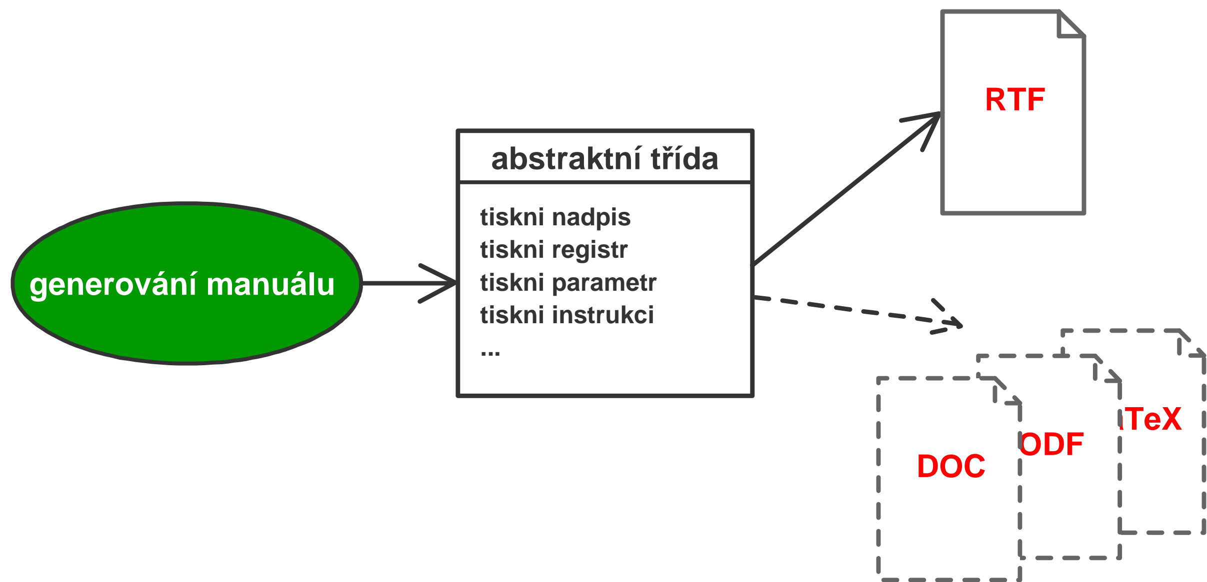
Na celý problém generování manuálu lze na nejvyšší úrovni nahlížet jako na transformaci vstupních dat popsaného mikroprocesoru do výstupních dat ve formátu generovaného dokumentu. V programování většinou jdou proti sobě dva protichůdné parametry – rychlost programu a paměťové nároky. Podobně je tomu i zde při návrhu aplikace. První možností je při tisku jednotlivých prvků do manuálu využít přímý přístup ke zdrojům dat a v případě potřeby dohledávat další vazby. Při tomto přístupu nevznikají žádné větší paměťové nároky, ale z pohledu rychlosti není dohledávaní vazeb až v případě potřeby ideální. Druhou možností znázorněnou na obrázku 3.1 je nejprve projít všechny zdroje dat a uložit si potřebné údaje a nalezené vazby. Poté se pouze tyto získané data projdou a vytisknou. Nevýhodou tohoto přístupu jsou vyšší paměťové nároky (ovšem i v nejnáročnějších modelech by se měli pohybovat v řádech jednotek MB) a mnohdy také duplicita dat. Výhodou je, že zdrojový kód aplikace je přehlednější díky existenci tříd popisujících zaznamenávané informace pro manuál. Také vstupní informace se procházejí pouze jednou. Pro realizaci aplikace jsem zvolil druhou možnost z důvodu výše vyjmenovaných výhod.




*Obrázek 3.1: Navrhnutý obecný model aplikace*

### 3.1.1 Rozhraní pro generování dokumentu

Za cílový formát dokumentu byl zvolen RTF. Ovšem, jak je u softwarových produktů běžné, s odstupem času se může ukázat, že je tento formát nevhodný z dnes neznámých příčin. Z tohoto důvodu by nebylo dobré řešení generovat manuál přímo do formátu RTF při procházení jednotlivých prvků.




*Obrázek 3.2: Navrhnutý tisk do formátu RTF přes abstraktní třídu*

Manuál se bude generovat s využitím rozhraní abstraktní třídy, jak je znázorněno na obrázku 3.2. Tato třída specifikuje všechny metody, které musí mít případný formát manuálu implementované. V rámci této bakalářské práce byl implementovaný pouze formát RTF a navržený princip tisku do něj je následující. Nejprve generátor manuálu zavolá u abstraktní třídy patřičnou metodu pro tisk například registru. Poté dle aktuální instance třídy skutečného formátu je realizována patřičná metoda pro tisk například registru.

## 3.2 Návrh generování zdrojů

Tato podkapitola se zabývá transformací zdrojových dat zdrojů procesoru na potřebné údaje, jak lze vidět na obrázku 3.1. Nejprve je nutné navrhnout, co bude zdrojem dat. Předpokladem je, že existuje model procesoru popsaný v jazyce ISAC. Zde jsou uvedeny všechny potřebné informace pro generování manuálu v poměrně přívětivé formě. Ovšem nebylo by vhodné získávat potřebné údaje přímo z tohoto souboru hlavně z důvodu, že není zkontrolovaný překladačem jazyka ISAC. Tento překladač zkompiluje zdrojový soubor v jazyce ISAC do vnitřního formátu XML. Implementace vlastního parseru by byla zbytečná, protože v projektu Lissom existuje knihovna ParseLib, která právě načte model v XML jako seznam všech operací a seznam všech zdrojů. Celý tento cyklus lze přehledně vidět na obrázku 3.3.

*Obrázek 3.3: Cesta od modelu procesoru k načtenému seznamu zdrojů*

Jako zdroj informací pro generátor zdrojů do manuálu instrukční sady poslouží právě seznam všech zdrojů načtených užitím knihovny ParserLib. V následujících podkapitolách bude uveden návrh transformace těchto dat do manuálu.

Na začátku samotného tisku jednotlivých zdrojových prvků do manuálu bylo třeba vyřešit problém neexistující šablony, která by udávala formát vzhledu a typ informací, které budou o zdrojích uvedeny. Vydat se směrem vytvoření této detailní šablony by nebylo vhodné, protože jsem na začátku práce nevěděl, jaké přesně informace a vazby půjdou zjistit. Také ukládání všech dostupných informací a vazeb by nebylo ideální, protože by jistě mnohé informace v samotném manuálu nebyly uvedeny. Řešením bylo vytvoření náčrtu šablony a detailní formát jednotlivých prvků byl vytvářen spolu s transformací zdrojových dat. Z pohledu softwarového inženýrství se tento proces mohl podobat V-Modelu, kde nejprve u každého prvku byl navrhnut jeho formát a poté následovala implementace. Dále místo testů, verifikace a validace následovala analýza, zda je zvolený formát a druh použitých informací vhodný v kontrastu s ostatními zdroji. Pokud se ukázalo, že by bylo něco vhodnější znázornit jinak, tak se změnil návrh šablony a provedla se kódová úprava.

### 3.2.1 Přidání speciálně formátovaných komentářů

Bylo navrhnuto, aby uživatel měl možnost dopsat si ke každému zdroji krátký textový komentář, který bude poté použit v manuálu. Vydat se cestou přidání parametru ke každému prvku by znamenalo zbytečný zásah do gramatiky jazyka ISAC. Vhodným řešením tedy je určit speciální formát komentářů, které nebude překladač jazyka ISAC ignorovat, ale přiřadí je k prvku uvedenému pod komentářem. Zřejmě nejrozšířenějším formátem komentářů pro dokumentaci je formát, který využívá program Doxygen:

```
/**
 * ... text ...
 */
```

Tento zápis se také používá v jazyce Java a rovněž se jedná o běžný komentář v jazyce C s přidáním další hvězdičky na začátku. Jak již bylo uvedeno, jazyk ISAC se po syntaktické stránce podobá jazyku C a tedy tento styl komentářů je vhodný. Při pohledu na obrázek 3.3 je dobré zmínit, že přidaný textový popis zdroje se stejnou cestou dostane až k příslušnému zdrojovému prvku v seznamu všech načtených zdrojů.

### 3.2.2 Registr

V předešlé kapitole bylo uvedeno, jaké údaje poskytuje jazyk ISAC o registrech. V komerčních manuálech se registry často zobrazují prostým výčtem nebo ve schématu celého CPU. Vykreslovat do manuálu schéma CPU by byl náročný úkol a také řada editorů formátu RTF nepodporuje zobrazení grafických objektů. Často se také v manuálech objevuje význam jednotlivých bitů v registru. Vytvořit například ve stavovém registru příznak přetečení lze v jazyce ISAC napsat vytvořením ALIAS na daný registr. Z tohoto důvodu by nebylo vhodné vytvořit pro ALIAS podkapitolu v manuálu, ale nabízí se uvést příslušný ALIAS přímo u registru. Kvůli přehlednosti bude vhodné uvádět registry spolu s bitovým rozsahem a tudíž bude nutné ošetřit příliš velký bitový rozsah.

Také je nutné vzít v úvahu, že ALIAS může být vytvořen do pole registrů. Zde je tedy nezbytné přesně uvést, do kterého prvku z registrového pole je ALIAS vytvářen. Samotný ALIAS může být také bankovaný (je vytvořeno pole aliasů), a tudíž se tato skutečnost musí také znázornit do manuálu.

V následujícím příkladu lze vidět stavový registr (převzatý z modelu procesoru 8051) popsaný nejprve v jazyce ISAC a následuje ukázka, jak je tento registr zobrazený v manuálu instrukční sady.

**Jazyk ISAC:**

```
/** Program Status Word */
REGISTER bit[8] PSW;
/**  Parity bit  */
bit[1] PARITY ALIAS { PSW bit[0] };
/** Overflow bit  */
bit[1] OV ALIAS { PSW bit[2] };
/** Register bank 0  */
bit[1] RS0 ALIAS { PSW bit[3] };
/** Register bank 1  */
bit[1] RS1 ALIAS { PSW bit[4] };
/** User flag   */
bit[1] F0 ALIAS { PSW bit[5] };
/** Partial carry (between fourth and fiveth bit)  */
bit[1] AC ALIAS { PSW bit[6] };
/** Carry flag  */
bit[1] C ALIAS { PSW bit[7] };
```

**Manuál instrukční sady:**

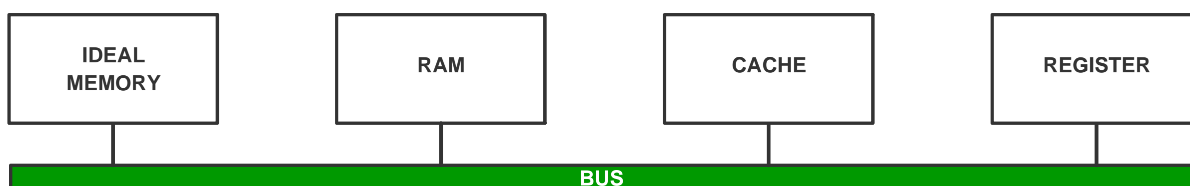
| 8-bit Program Status Word (PSW) | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

*Alias:*

0-bit = Parity bit (**PARITY**)
2-bit = Overflow bit (**OV**)
3-bit = Register bank 0 (**RS0**)
4-bit = Register bank 1 (**RS1**)
5-bit = User flag (**F0**)
6-bit = Partial carry (between fourth and fiveth bit) (**AC**)
7-bit = Carry flag (**C**)

### 3.2.3 Sběrnice

Sběrnice ve své definici obsahuje dva parametry a to informaci o své bitové šířce a údaj o endianitě. Tato data se tedy zobrazí v manuálu u sběrnice jako její parametry. Další informace o sběrnici lze zjistit z mapování prvků do paměti, protože mohou být mapovány právě přes sběrnici, a tudíž jsou na ni připojeny. Všechny podporované zdroje, které lze připojit na sběrnici, lze vidět na obrázku 3.4. Samotné propojení prvku se sběrnicí bude v manuálu uvedeno jak u zdroje, tak i u sběrnice po sekci paramenty v sekci propojení.




*Obrázek 3.4: Seznam zdrojů, které lze připoj na sběrnici.*

V následující ukázce lze vidět, jakým způsobem můžeme v jazyce ISAC připojit paměť RAM ke sběrnici a jak je poté tato skutečnost uvedena v manuálu.

**Jazyk ISAC:**

```
/** Program memory */
RAM bit[8] program_mem[4] {…}
/** System BUS */
BUS bit[8] sBUS
{
    BLOCKSIZE (8, 4);
    ENDIANESS (BIG);
};
MEMORY_MAP defaultmap
{
  BUS(sBUS),RANGE(10,0xC8)->program_mem [(9..8)][(7..0)];
}
```

**Manuál instrukční sady:**

**8-bit System BUS (sBUS):**

*Parameters:*

Block size: 8-bit , SubBlock size: 4-bit

Endianess: big-endian

*Connected to:*

RAM: Program memory (program_mem)

***4x memory - indexed[0..3]***

**Program memory (program_mem):**

*Parameters:*

…

*Connected to:*

BUS: System BUS (sBUS)

### 3.2.4 Ideální paměť a RAM

Situace u těchto zdrojů je poměrně přímočará. Vždy se jedná o přepsání parametrů zdroje do podoby více čitelné uživatelem. Paměť RAM obsahuje stejné parametry jako ideální paměť a k tomu navíc informaci o endianitě a zpoždění při čtení a zápisu. Kromě sekce informující o připojení těchto prvků ke sběrnici a v případě RAM i k rychlé vyrovnávací paměti může být v manuálu uvedena ještě další sekce obsahující případné aliasy. Zde je situace podobná jako u registrů jen s tím rozdílem, že je zakázáno vytvářet alias na bankovanou paměť.

**Jazyk ISAC:**

```
/** Data memory */
RAM bit[8] data_mem
{   ENDIANESS (BIG);
    BLOCKSIZE (8, 8);
    SIZE (255);
    FLAGS (R, W);
};
/** Test Alias */
bit[32] AliasRAM[20..25]  ALIAS { data_mem[2] };
```

**Manuál instrukční sady:**

**Data memory (data_mem):**

*Parameters:*

Block size: 8-bit , SubBlock size: 8-bit
Consists of an array 255 (0xFF) blocks => 255 bytes
Flags: read-write, non-executable
Endianess: big-endian
Read Latency: 1 ,Write Latency: 1

*Alias:*

6x 32-bit Test Alias (**AliasRAM** [20..25]) offset: 0x2 from start of memory

Ve výše uvedené ukázce lze vidět zápis paměti RAM a aliasu a následné navrhnuté zobrazení v manuálu. Za povšimnutí stojí fakt, že ALIAS má 32-bitovou šířku a tedy jeden jeho prvek je přes 4 prvky v paměti RAM. Také tento alias není mapován na začátek paměti, ale s offsetem dva.

### 3.2.5 CACHE

Rychlá vyrovnávací paměť (CACHE) obsahuje všechny parametry jako paměť RAM a navíc zahrnuje údaje týkající se například strategie při výměně dat v CACHE, dále velikost zásobníkuu a jednoho řádku a také počet cest pro set-asociativní CACHE. Také je možné rychlou vyrovnávací paměť připojit k dalšímu prvku, kterým může být ideální paměť, RAM, sběrnice a další CACHE. V následující ukázce lze opět vidět popsanou paměť CACHE v jazyce ISAC a následuje návrh zobrazení v manuálu. Za povšimnutí stojí připojení CACHE k programové paměti typu RAM, která z důvodu úspory místa není ve zdrojovém kódu uvedena.

**Jazyk ISAC:**

```
/** Test CACHE */
CACHE bit[8]  reg_cache  {
    FLAGS(R,W);
    BLOCKSIZE(8,8);
    SIZE(8);
    NUM_WAYS(4);
    LINESIZE(2);
    RPL_POLICY(LRU);
    WA_POLICY(ALWAYS);
    WB_POLICY(ALWAYS);
    CONNECT(program_mem);
};
```

**Manuál instrukční sady:**

**Test CACHE (reg_cache):**

*Parameters:*

Block size: 8-bit , SubBlock size: 8-bit
Consists of an array 8 (0x8) blocks => 8 bytes
Flags: read-write, non-executable
Endianess: little-endian
Read Latency: 1 ,Write Latency: 1

Number of ways for set-associative cache: 4
Number of blocks in the line: 2
Write-Allocate strategy: Always, Write-Back strategy: Always
Replacement strategy: Least recently used
Selective write buffer size: 0 lines of cache

*Connected to:*

RAM: Program memory (program_mem)

Rychlá vyrovnávací paměť je posledním paměťovým prvkem, a proto je v následující tabulce číslo 3.1 uveden přehled paměťových prvků a jejich možností v aktuální verzi jazyka ISAC, a tudíž i v generovaném manuálu.

| **vlastnost** | **Ideal Mem.** | **RAM** | **CACHE** | **BUS** | **Register** |
|---|---|---|---|---|---|
| Lze vytvořit ALIAS | ANO | ANO | NE | NE | ANO |
| Lze připojit ke sběrnici | ANO | ANO | ANO | NE | ANO |
| Lze připojit k CACHE | ANO | ANO | ANO | ANO | NE |

*Tabulka 3.1: Přehled vlastností paměťových prvků*

### 3.2.6 Mapování paměti

V mnohých manuálech procesorů je mapování paměti zobrazeno ve formě tabulky, kde jsou na levé straně uvedeny příslušné adresové rozsahy (většinou v hexadecimálním formátu) a na pravé odpovídající paměťové prvky. Povolenými prvky pro mapování v jazyce ISAC jsou všechny paměťové prvky a registry. V popisu modelu je kromě samotného adresového rozsahu ještě uvedena informace o tom, jak je adresa mapována na indexy příslušného zdroje, a zda je připojena přes sběrnici. Navrhnutý formát zobrazení v manuálu odpovídá výše uvedenému formátu a předešlý údaj o tvorbě adresy, zde není uveden. Propojení se sběrnicí je uvedeno u samotných prvků a v mapování paměti není zobrazeno. Jazyk ISAC vyžaduje vytvořit jedno standardní mapování a umožňuje také definovat další mapování pamětí. Na to je také myšleno při návrhu zobrazení v manuálu.

Na následujícím příkladu, který nemá z pohledu modelu procesoru žádný význam, lze vidět tři typy pamětí a jejich zobrazení v manuálu. Adresy na sebe nenavazují, a proto jsou v manuálu tato místa vyznačena jako neimplementované.

**Jazyk ISAC:**

```
/** test memory */
RAM bit[8] test_mem[4] {…}
/** test array of registers */
REGISTER bit[8] regs[5..76];
/** test CACHE */
CACHE bit[8]  r_cache  {…}
MEMORY_MAP defaultmap
{
      RANGE(10, 0xC8)->test_mem [(9..8)][(7..0)];
      RANGE(201,255)->regs [(25..20)];
      BUS(myBUS2),RANGE(261)->r_cache [(1..0)];
};
```

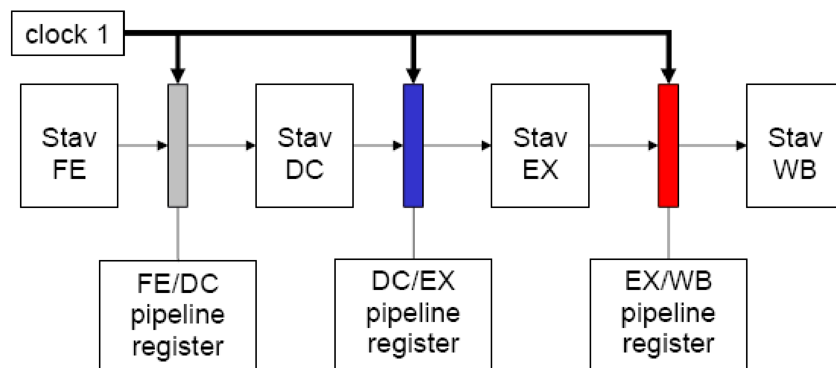
**Manuál instrukční sady:**

DEFAULT MEMORY MAP:

| | |
|---|---|
| 0x000000<br>0x000009 | **UNIMPLEMENTED**<br>10 BYTES |
| 0x00000A<br>0x0000C8 | **test memory (test_mem)**<br>191 BYTES |
| 0x0000C9<br>0x0000FF | **test array of registers (regs)**<br>55 BYTES |
| 0x000100<br>0x000104 | **UNIMPLEMENTED**<br>5 BYTES |
| 0x000105 | **test CACHE (r_cache)** |

## 3.2.7 Struktury se zřetězeným zpracováním

Struktury se zřetězeným zpracováním se většinou zobrazují jako schéma s výčtem jednotlivých stavů, kterým příslušejí oddělovací a stavové registry. To lze vidět na obrázku 3.5. Ve formátu RTF by toto schéma šlo realizovat pomocí tabulek (pokud pomineme podporu kreslení v některých editorech RTF). Po zkušenostech s tabulkami v RTF jsem dospěl k názoru, že výsledek by nepřinesl u všech typů linek zřetězení zpřehlednění problému vůči prostému výčtu stavů s odpovídajícími registry. Proto jsem zvolil tuto možnost a také registry, které se používají jako stavové či oddělovací u pipeline, již nejsou uvedeny v první podkapitole v přehledu registrů.




*Obrázek 3.5: Linka zřetězení – převzato z [4]*

Následující příklad zobrazuje popsaný model zřetězeného zpracování uvedeného v předešlé kapitole, proto jsou vynechány registry a následuje navržené zobrazení v manuálu.

**Jazyk ISAC:**

```
/** test pipeline */
PIPELINE pipe
{      FE:   [latch1]   (src1);
       DC:   [latch2]   (src2);
       EX:   [latch3];
       WB:   ;
};
```

**Manuál instrukční sady:**

**Test pipeline (pipe):**

*States:*

**FE:**

State Registers:

8-bit (src1)

Latch Registers:

8-bit (latch1)

**DE:**

State Registers:

8-bit (src2)

Latch Registers:

8-bit (latch2)

**EX:**

Latch Registers:

8-bit (latch3)

**WB:**

### 3.2.8 Logický obvod

Ze zdrojové sekce lze o tomto obecném hardwarovém prvku zjistit pouze jeho název a dopsaný komentář. Proto jsou tyto prvky uvedeny v manuálu jako prostý výčet. Zobrazení prvku lze vidět v následujícím jednoduchém příkladu.

**Jazyk ISAC:**

```
/** Test ALU */
LC ALU;
/** Test LC  */
LC TLC;
```

**Manuál instrukční sady:**

LOGIC CIRCUITS:

**Test ALU (ALU)**

**Test LC (TLC)**

### 3.2.9 Entita

Na entitu lze nahlížet jako na samostatně funkční jednotku pro zvýšení výkonu procesoru. V jazyce ISAC je nejprve popsána deklarace entity a následně se vytvoří instance, která může obsahovat parametry, kterým může být například registr. Samotné deklarace entity se v manuálu nebudou nijak zobrazovat a všechny zdroje entity a parametry se uvedou až u její instance. Návrh lze vidět na následujícím příkladu:

**Jazyk ISAC:**

```
ENTITY ent_t(a)
{     /** Entity test register */
      REGISTER bit[8] ent_reg [1..3];
};
/** Test Entity Instance */
ent_t instance_ent_t(pc);
```

**Manuál instrukční sady:**

ENTITY:

**Test Entity Instance (instance_ent_t):**

*Connection:*

REGISTER: Program Counter (pc)

*Resources:*

REGISTERS:

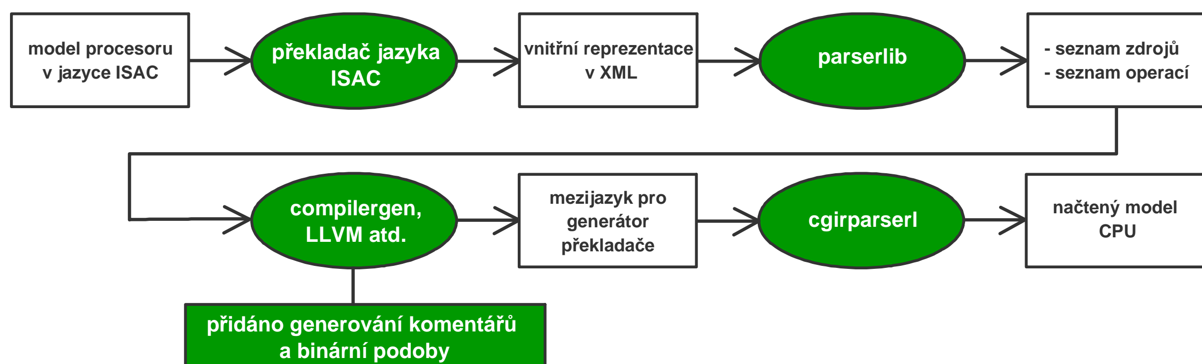
*3x register - indexed[1..3]*

| 8-bit Entity test register (ent_reg) | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Entita byla posledním zdrojovým prvkem v manuálu. Kompletní přehled všech zdrojových prvků, jejich parametrů a metod lze vidět v diagramu tříd, který je kvůli svému rozsahu umístěn do přílohy této práce.

# 3.3 Návrh generování instrukcí

Ve druhé kapitole v části týkající se manuálu instrukční sady byly uvedeny informace, které se obvykle v manuálech o každé informaci zobrazují, a také zde byly předloženy formy zobrazení instrukcí. První a zřejmě i nejdůležitější úlohou při návrhu generování instrukcí je určit, co bude zdrojem dat. Jinak řečeno, najít v celém projektu Lissom zdroj (knihovnu apod.), který by poskytoval informace o syntaxi, sémantice, binární podobě apod. pro každou instrukci. Načtený model procesoru, který slouží jako zdroj dat pro zdrojové prvky, poskytuje všechny tyto potřebné informace. Ovšem problém je, že operace a skupiny jsou zde v hierarchickém uspořádání a musel bych navrhnout převod do seznamu instrukcí, ale tím se zabývají jiné části projektu. Proto bylo navrhnuto, aby zdrojem dat byl načtený model CPU, který jako zdroj dat bere mezijazyk pro generátor překladače. Tento mezijazyk je vygenerován převážně použitím knihovny compilergen a LLVM optimalizátoru. Celý tento proces lze vidět na obrázku 3.6.




*Obrázek 3.6: Cesta od modelu procesoru k načtenému CPU*

### 3.3.1 Přidání komentářů k instrukcím a skupinám

Ve většině manuálu je u každé instrukce uveden i její slovní popis. Tato informace není v modelu procesoru popsaném v jazyce ISAC nikde uvedena. Proto je nutné se vydat podobnou cestou jako u zdrojových prvků a to přidáním speciálně formátovaných komentářů k instrukcím. Zde je ovšem situace složitější v tom, že jedna instrukce procesoru může být popsána více operacemi a skupinami. Řešením by tedy mohlo být uvedení komentáře vždy u nejníže postavené operace a ten by reprezentoval danou instrukci. Tímto řešením ovšem vznikne velká duplicita dat a například u aritmetických instrukcí by to jistě vedlo vývojáře ke kopírování komentářů a následné změně pouze jednoho slova, např. zda se jedná o sčítání, odčítání apod. Proto bylo navrhnuto zavést do komentářů nadřazených operací proměnné a ty se poté nahradí příslušnou variantou operace. V níže uvedené ukázce lze vidět navržený systém komentářů na modelu procesoru MIPS (vychází z referenčního manuálu). Proměnné začínají symbolem procenta. Komentáře u skupiny mají význam pouze v případě, že skupina má vlastnost REPRESENTS.

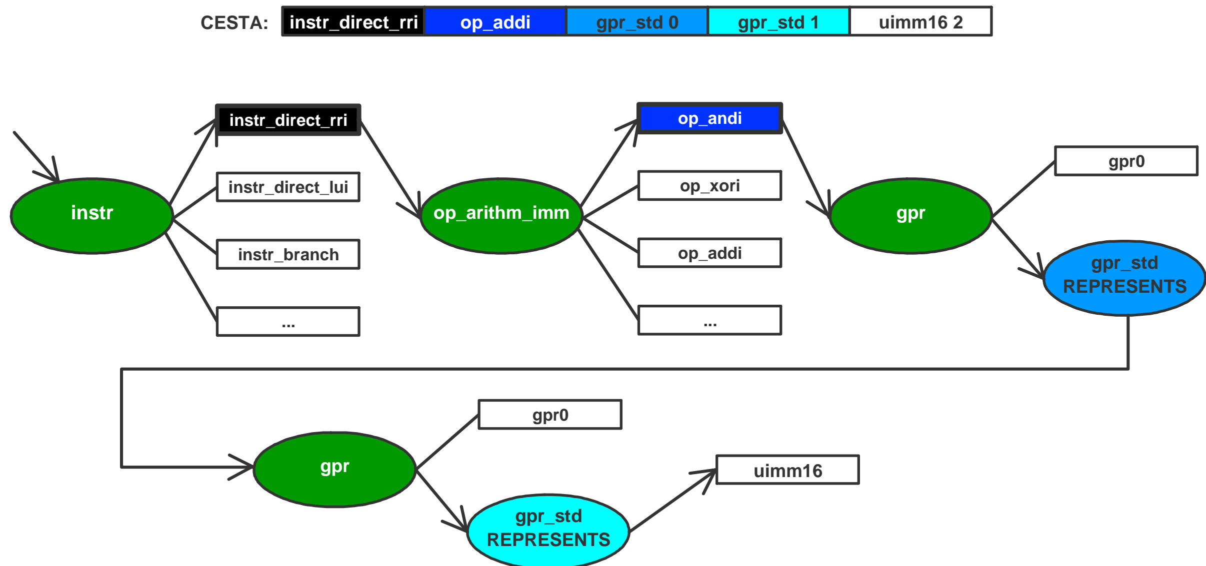
**Jazyk ISAC:**

```
/** GPR */
GROUP gpr_std REPRESENTS gpregs = gpr1 ALIAS { gprat }, gpr2 …

GROUP gpr = gpr_std , gpr0 ALIAS { gprzero };

/** 16-bit immediate */
OPERATION uimm16
{
      ASSEMBLER { immval=#U };
      CODING { immval=0bx[16] };
      EXPRESSION { immval; };
}

/** is added to */
OPERATION op_addi
{
      ASSEMBLER { "ADDI" };
      CODING { 0b001000 };
      EXPRESSION {OP_ADDI; };
}

GROUP op_arithm_imm = op_addi, op_addiu, op_slti, op_sltiu, …

/**
 * The %uimm16 %op_arithm_imm the 32-bit value in %rs R1. The result
 * is placed into %rt R0.
 */
OPERATION instr_direct_rri
{
      INSTANCE gpr ALIAS {rs, rt};
      INSTANCE op_arithm_imm ALIAS { op_arithm_imm };
      INSTANCE uimm16;
      ASSEMBLER{ op_arithm_imm rt "," rs "," uimm16 };
      CODING    { op_arithm_imm rs rt uimm16 };
      BEHAVIOR {…}
```

### 3.3.2 Generování komentářů a binární podoby

V rámci této bakalářské práce bylo také mým úkolem navrhnout a implementovat funkce pro generování komentářů a binární podoby každé instrukce, jak lze také vidět na obrázku 3.6. Rozhraní funkcí již bylo předpřipraveno. Vstupními daty do funkce jsou načtený model procesoru a pole obsahující operandy, názvy operací a skupin, které jsou použity v dané instrukci. Na základě těchto informací lze poté sestavit komentář a binární podobu.

Nejlépe je celý problém generování vysvětlit na příkladu, který vychází ze zdrojového kódu uvedeného v předchozí podkapitole:




*Obrázek 3.7: Ukázka dohledaní binární podoby a komentářů u instrukce*

Nejprve je v modelu nalezena hlavní kódová sekce a tedy i příslušná hlavní operace nebo skupina. To je znázorněno na obrázku šipkou do skupiny instr. Nyní již přijde na řadu pole obsahující názvy elementů, které obsahují potřebné informace pro zrekonstruování komentářů či binární podoby instrukce. První prvek v poli (název operace instr_direct_rri) určuje, která operace ze skupiny bude vybrána. Poté je v této operaci procházena sekce syntaxe a pro každou instanci se dohledá patřičný údaj. Jelikož instancí je skupina op_arithm_imm a ta obsahuje mnoho operací, přijde opět na řadu informace z pole, udávající název operace op_andi. Tak se pokračuje dále a stále se dohledávají údaje z operací. Výjimkou je skupina s vlastností REPRESENTS. V případě sestavování komentářů se použije právě komentář této skupiny a v případě sestavování binární podoby se použije upravený údaj binární podoby první operace z této skupiny. Po skončení algoritmu je tedy vytvořeno pole, které obsahuje pro každou instanci údaj o binární podobě či komentáři.

V případě rekonstrukce komentářů se následně hlavní komentář u operace projde a nahradí se proměnné (instance) nalezenými údaji. Podobným způsobem se také projde binární podoba hlavní operace a jednotlivé instance se nahradí. Konkrétní výsledek pro výše uvedený příklad je uveden níže. Za povšimnutí stojí zaměněné pořadí operandů binární podoby instrukce vůči její syntaxi.

**Komentář:**

```
The 16-bit immediate is added to the 32-bit value in GPR R1. The
result is placed into GPR R0.
```

**Binární podoba:**

```
0b001000  1=0bx[5]  0=0bx[5]  2=0bx[16]
```

### 3.3.3 Zobrazení informací v manuálu

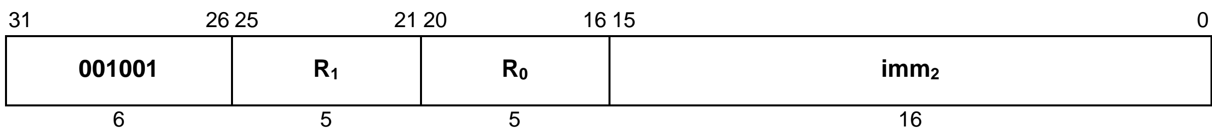
V předchozí podkapitole byl popsán způsob, kterým se pro každou instrukci získá její slovní popis a binární podoba. Další informací, která se často zobrazuje v manuálech, je doba trvání instrukce. Tato informace je obsažena v mezijazyce, a proto je pouze odtud převzata. Syntaktická podoba je také uvedena v načteném modelu CPU, ovšem její podoba není úplně ideální:
**Syntaxe:**
ADDIU 0 , 1 , 2

Proto je dále v načteném modelu dohledán význam jednotlivých operandů. Například pokud je operand registrem, tak je vypsána zkratka „R“ a číslo operandu:
**Syntaxe:**
ADDIU $R_0$ , $R_1$ , $imm_2$

Informace o významu jednotlivých operandů jsou také doplněny k binární podobě instrukce. Pro výpis sémantiky pro vývojáře byla v načteném objektu CPU implementována metoda `pretty_print`. Tímto jsou získány všechny potřebné informace pro zobrazení instrukce v manuálu. Jelikož v manuálu budou zobrazeny i rozsáhlejší informace typu komentáře a sémantiky, tak zobrazení v tabulce, jak lze vidět na obrázku 2.1 nahoře, není vhodné. V následující ukázce lze vidět navrhnutý způsob zobrazení jedné instrukce.

**Manuál instrukční sady:**

ADDIU




*Description:*

The 16-bit immediate is added to the 32-bit value in GPR R1. The result is placed into GPR R0.

*Syntax:*

ADDIU $R_0$ , $R_1$ , $imm_2$

*Latency:*

1

*Semantics:*

[0] cl0#op0 (add (i32 cl0#op1, i32 sext (i16 op2)))

# 4 Implementace generátoru

Jak již bylo uvedeno, tato práce je vytvářena v rámci projektu Lissom a budu v ní využívat již hotové knihovny z projektu. Proto bylo na začátku práce nutné seznámit se se systémem pro správu verzí projektu CVS a také makefile systémem. Jsou také dodržována pravidla pro psaní zdrojového kódu. Jako vývojové prostředí bylo použito Eclipse v operačním systému Kubuntu a implementačním jazykem je C++, přičemž informace byly čerpány převážně z [8] a [9]. V následujících podkapitolách je popsána implementace aplikace dle navrženého řešení z předchozí kapitoly.

## 4.1 Generování zdrojů

Pro načtení modelu procesoru z vnitřní reprezentace v XML je použita knihovna parserlib2. Následně je procházena zdrojová sekce a jednotlivé zdroje jsou ukládány do nově vytvářených objektů reprezentující navržené vlastnosti z předchozí kapitoly. Kompletní diagram tříd zdrojových prvků je umístěn v příloze. Ukazatele na tyto objekty jsou poté ukládány do asociativních polí, kde je indexem jejich jedinečný název. V případě potřeby se také dohledávají patřičné vazby a ukládají se k příslušným objektům. Tímto způsobem se naplní asociativní pole pro každou kategorii. Poté patřičné metody z objektu ManualDoc procházejí tyto pole a realizují jejich tisk pomocí metod abstraktního formátu. Do následujících podkapitol jsou vybrány z implementačního hlediska zajímavé zdroje a je vysvětleno jejich zobrazení.

### 4.1.1 Registry a jejich aliasy

Registry se budou v manuálu zobrazovat dle navrhnutého řešení. Tedy formou dvouřádkové tabulky, kde na prvním řádku bude informace o bitové šířce a názvu registru a druhý řádek bude rozdělen na počet sloupců odpovídající bitové šířce registru. Při realizaci algoritmu na vykreslení je nutné ošetřit dvě krajní podmínky. První je, když bitová šířka registru je příliš velká (17 a víc bitů). V tomto případě by se již rozkreslené bity nevlezly na řádek, proto je zde vždy uveden pouze první a poslední bit. Naopak když je bitová šířka malá (7 a méně bitů), tak je registr vykreslen s minimální šířkou a šířka jednotlivých sloupců je dopočítána. Pokud nenastane žádná z krajních podmínek, tak je pevně stanovena šířka jedné buňky odpovídající jednomu bitu, a tudíž celková šířka registru závisí na jeho bitové šířce. To dává vývojáři grafický přehled jednotlivých šířek registrů.

Dále pokud má daný registr alias, tak je tato skutečnost uvedena přímo pod registrem. V případě, že je registr bankovaný, tak je nejprve vypsána informace, do kterých banků je vytvářen daný alias. Následně mohou nastat čtyři situace. První a zřejmě i nejčastější situací je, když je alias jednobitový a není bankovaný. V tomto případě je vypsána pouze informace o příslušném bitu a názvu aliasu. Pokud není alias jednobitový, tak je vypsán bitový rozsah. Tyto dva případy mohou nastat i za situace, že je samotný alias bankovaný. Zde je tedy nutné uvést všechny bity či bitové rozsahy, kde jejich počet odpovídá velikosti pole aliasů. V následující ukázce lze vidět bankovaný alias do bankovaného registru, přičemž bitová šířka aliasu je větší než registru.

**Manuál instrukční sady:**

***4x register - indexed[0..3]***

| **8-bit (TEST)** | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

*Alias:*

{alias to bank[0..3]} 2x 15..0-bit = Test alias (**x11**[0..1])

### 4.1.2 Mapování pamětí

Mapování paměti popsané v jazyce ISAC nemusí začínat od adresy 0 a také na sebe nemusí adresy jednotlivých mapovaných prvků navazovat. Z tohoto důvodu je nejprve nutné vyplnit adresové prostory, do kterých není mapován žádný zdroj, údajem o této skutečnosti (UNIMPLEMENTED). V samotném zobrazení mapovaného prvku mohou nastat dvě možnosti. První a zřejmě i častější je, že je zobrazen adresový rozsah, kde jsou čísla dle zvyklosti uváděna v hexadecimálním tvaru. Také je u prvku uveden odpovídající počet bajtů, které jsou mapovány do dané adresy. Druhou možností je, že je mapován pouze jeden bajt a tomu odpovídá pouze jedna adresa. Tímto způsobem je vykresleno celé mapování pamětí.

### 4.1.3 Ostatní zdrojové prvky

U ostatních zdrojových prvků je uvedena pouze formátovaná textová informace, proto je situace jednodušší. V některých případech je pouze k údaji doplněn jeho význam a u jiných je upraven i samotný údaj. Příkladem mohou být příznaky u pamětí pro čtení, zápis atd., kde je na základě těchto informací vývojáři předložena informace vypovídající o tom, že se jedná např. o paměť určenou pouze ke čtení.

## 4.2 Generování instrukcí

Jak již bylo uvedeno, zdrojem dat pro instrukce je mezijazyk pro generátor překladače jazyka C, který byl ovšem v době psaní této práce ještě stále ve vývoji. Hlavním testovacím modelem byl procesor MIPS, pro který se dal vytvořit mezijazyk již v použitelné formě. Problémem ovšem zůstávala knihovna cgirparserl, která právě načítá mezijazyk do virtuálního modelu procesoru. Tato knihovna byla implementována do fáze použitelné pro generátor manuálu až koncem dubna 2010.

Samotný tisk instrukcí z načteného virtuálního CPU je již poměrně jednoduchý. Zde je procházen seznam instrukcí a jsou dohledány potřebné údaje a ty jsou předány abstraktnímu formátu k tisku.

### 4.2.1 Komentáře a binární podoba

Na knihovně cgirparserl ovšem nezávisí vytvoření binární podoby a komentářů pro instrukce, proto tyto části byly implementovány nejdříve přesně dle navrženého řešení z předešlé kapitoly. Nalezení hlavní operace je společné pro obě dvě části a je tedy implementováno funkcí. Stejně tak je vytvořena i rekurzivní funkce pro nalezení operace nebo skupiny s vlastností REPRESENTS v zadané skupině. Pomocí rekurzivní funkce je také implementována hlavní myšlenka algoritmu, která pro jednotlivé instance u operace zjišťuje jejich binární podoby či komentáře a ukládá je do asociativního pole indexovaného názvem instance. Na závěr jsou tyto informace seskládány dohromady.

### 4.2.2 Komentář typu Doxygen

Pro překladač jazyka ISAC jsem také implementoval funkci GetDoxygenCommentText, která odstraní z komentáře jeho uvozovací značky (/** a */), přebytečné bílé znaky a případně hvězdičku na začátku řádku. Tímto vznikne jednořádkový text.

## 4.3 Generování dokumentu

Samotné vykreslování informací do dokumentu je realizováno pomocí čistě virtuálních metod abstraktní třídy AbsGen. Od této bázové třídy musí dědit všechny případné formáty (nyní pouze RTFgen) a implementovat všechny metody. Jako parametry těchto metod jsou použity pouze základní datové typy. Toto řešení vede k polymorfnímu chování a tedy i k pozdní vazbě.

### 4.3.1 Vytvoření dokumentu v RTF

Výstupem aplikace je dokument ve formátu RTF. Objekt ManualDoc ve svém konstruktoru otevírá výstupní souborový stream a právě do něj zapisují metody objektu RTFgen jednotlivé údaje. Následující ukázka reprezentuje titulní stranu manuálu a část prvního registru. Výstup zdrojového kódu jednotlivých metod je v RTF barevně odlišen a parametry jsou uvedeny tučně. První řádek v příkladu uvádí použití ANSI znakové sady a nastavení výchozího fontu nula. Následuje tabulka fontů, ve které je právě uveden font nula, a poté je deklarována tabulka barev (popsaná šedá barva se využívá například u neimplementovaného místa u mapování pamětí). Okraje dokumentu jsou nastaveny na 1000 twips (jednotka použitá ve formátu RTF, kde 1 cm odpovídá cca 567 twips). V zápatí je nastaveno číslování stránek. Následují odstavce udávající obsah titulní strany jsou centrované na střed a odsazeny od začátku strany. Velikost fontu je 40 (RTF kvůli celočíselnému formátu používá dvojnásobek běžné velikosti písma). V závěru ukázky je popsána tabulka a její ohraničení. Za povšimnutí stojí příkaz `\keepn`, který usiluje o udržení daného odstavce na jedné straně. Tímto je docíleno, že například rozepsané bity u registru nebudou uvedeny na další straně, než je název registru.

**Zdrojový kód v RTF:**

```
{\rtf1\ansi\deff0
{\fonttbl
{\f0\fswiss Arial;}
}
{\colortbl ;
\red100\green100\blue100;
\red0\green0\blue255;
\red180\green180\blue180; }
\margl1000 \margr1000 \margt1000 \margb1000
{\footer \pard \qc\plain\f0\chpgn \par}
{\pard\b\fs40 \sb4200 \qc mips\par}
{\pard\b\ul\fs40 \sb200 \qc Instruction Set Manual\par}
{\pard\fs20 \qc 06. May 2010 \par}
\page
{\pard\b\fs30\sa250\sb50 \ql 1) Resources: \par}
{\pard\fs28\sa100\sb150\keepn\ql {\ul CPU REGISTERS:} \par}
{\fs18\b\trowd\trgaph180
\clbrdrt\brdrw15\brdrs
\clbrdrl\brdrw15\brdrs
\clbrdrb\brdrw15\brdrs
\clbrdrr\brdrw15\brdrs\cellx4000
\pard\intbl\qc\keepn  8-bit  (ACC) \cell
\row}
```

# 5 Závěr

Projekt Lissom si klade především za cíl urychlit vývoj procesorů pomocí automaticky generovaných softwarových nástrojů a samotné hardwarové realizace procesoru. Tato bakalářská práce přispívá do projektu dalším softwarovým nástrojem. Tím je generátor, jehož výstupem by byl přehledný a pro vývojáře použitelný manuál instrukční sady. Tento hlavní cíl práce se podařilo realizovat. Aktuální verze aplikace umožňuje generovat přehled všech zdrojů procesoru a jejich propojení. Tato část byla testována a funguje pro libovolný model procesoru popsaný v jazyce ISAC. Dále aplikace generuje informace o všech instrukcích, které poskytuje popsaný model procesoru. Toto bylo testováno pouze na modelu MIPS, protože zdrojem informací je mezijazyk pro generátor překladače jazyka C, který je ve vývoji a poskytuje použitelné výsledky pouze pro tento procesor.

Zdrojová část manuálu je inspirována zažitými formáty zobrazení jednotlivých prvků z komerčních manuálu a poskytuje přehledné informace. Tato část lze dále rozšířit o kreslení schémat například pro linky zřetězeného zpracování, připojení prvků na sběrnici apod. Druhá část obsahuje pro vývojáře všechny potřebné informace o každé instrukci. Dále je možné například syntaxi instrukce rozšířit o přesné názvy operandů a také případně vylepšit funkci pro tisk sémantiky. Rovněž u některých manuálu je na začátku zobrazen přehled všech instrukcí ve formě tabulky. Od zadání práce se zvýšila podpora nových otevřených formátů dokumentů. To může být také předmětem dalšího rozšíření této aplikace.

V rámci testování zdrojů byla zjištěna řada drobných chyb převážně v překladači jazyka ISAC. Některé testovací konstrukce byly použity v nástroji pro automatické testování projektu. Zdrojová část byla také vygenerována pro souběžně vznikající bakalářskou práci zabývající se popsáním mikroprocesoru 8051, kde byly dopsány jejím autorem navrhnuté komentáře.

Z implementačního hlediska byla práce pro mě velkým přínosem, protože bylo nutné psát zdrojový kód co nejvíce obecně z důvodu neustále se vyvíjejícího jazyka ISAC. Také se mi potvrdilo, že je výhodné provést kvalitní návrh aplikace a to převážně tříd a jejich dědičností. Naopak podrobný návrh implementačních detailů se mi příliš neosvědčil.

# Literatura


[1] Masařík, K. *Systém pro souběžný návrh technického a programového vybavení počítačů / Vyd.* 1. Brno : Vysoké učení technické v Brně, 2008. 156 s. ISBN 978-80-214-3863-7

[2] Hruška, T. *Jazyk ISAC – Příručka*. Interní dokumentace, 2004

[3] Morimoto, T., Saito K., Nakanuta H., *Advanced Processor Design Using Hardware Description Language AIDL [online]*. [cit. 2010-04-19]. Dostupné na URL: <http://www.cs.york.ac.uk/rts/docs/SIGDA-Compendium-1994-2004/papers/1997/aspdac97/pdffiles/06a_4.pdf>

[4] Masařík, K. *Jazyky pro popis architektury počítačových systémů [online]*. [cit. 2010-04-19]. Dostupné na URL: <https://www.fit.vutbr.cz/study/courses/IPP/private/Prednesy/IPP_ADL_cz.KM_2010-03-28.pdf>

[5] Drábek, V. *Návrh počítačových systémů – přednášky*. Brno: FIT VUT v Brně, 2006

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

[7] Burke S. *RTF Pocket Guide*, Sebastopol : O'Reilly Media, 2003. 160 s. ISBN 0-596-00475-3

[8] Virius, M. *Jazyky C a C++ :kompletní kapesní průvodce programátora / 1. vyd.* Praha : Grada, 2006. 518 s. ISBN 80-247-1494-9

[9] Eckel, B. *Myslíme v jazyku C++ :knihovna zkušeného programátora.2. díl / 1. vyd.* Praha : Grada Publishing, 2006. 608 s. ISBN 80-247-1015-3

# Seznam příloh

| | |
|---|---|
| **Příloha 1** | Manuál programu |
| **Příloha 2** | Vygenerovaný manuál instrukční sady |
| **Příloha 3** | Diagram tříd zdrojů |
| **Příloha 4** | Obsah CD |

# Příloha 1

**Manuál programu:**
Po přeložení vznikne konzolová aplikace, která má následujícím parametry:

- -i –input vstupní XML soubor obsahující přeložený ISAC model
- -t –input2 vstupní soubor mezijazyka pro generátor překladače jazyka C
- -v –version vypíše verzi generátoru manuálu instrukční sady
- -h –help vypíše nápovědu

Po zadání korektních vstupů je vygenerován soubor manuálu, jehož název je složen z jména procesoru a přípony „-manual.rtf“ (např. mips-manual.rtf).

# Příloha 2

**Vygenerovaný manuál testovacích zdrojů:**
Pozn.: Uvedená kombinace zdrojů nenese žádný konkrétní význam.

## 1) Resources:

### CPU REGISTERS:

| 8-bit Accumulator (ACC) | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

| 32-bit (BIG_REG) | |
|---|---|
| 31 | 0 |

| 8-bit Program Status Word (PSW) | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

*Alias:*

0-bit = Parity bit (**PARITY**)
2-bit = Overflow bit (**OV**)
3-bit = Register bank 1 (**RS0**)
4-bit = Register bank 1 (**RS1**)
5-bit = User flag (**F0**)
6-bit = Partial carry (between fourth and fiveth bit) (**AC**)
7-bit = Carry (**C**)

| 3-bit (SMALL_REG) | | |
|---|---|---|
| 2 | 1 | 0 |

| 1-bit (SMALL_REG2) |
|---|
| 0 |

***72x register - indexed[5..76]***

| 8-bit Test array of registers (nee) | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

*Connected to:*

BUS: System BUS (myBUS2)

| 16-bit Program counter (pc) | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

## BUSES:

**32-bit System BUS (myBUS2):**

*Parameters:*

Block size: 32-bit , SubBlock size: 32-bit
Endianess: big-endian

*Connected to:*

IDEAL MEMORY: (pmem)
CACHE: test cache (register_cache)
REGISTER: Test array of registers (nee)

## IDEAL MEMORIES:

**test comment (TESTMEM):**

*Parameters:*

Block size: 8-bit , SubBlock size: 8-bit
Consists of an array 1000 (0x3E8) blocks => 1000 bytes
Flags: read-only, executable

*Alias:*

16x 32-bit memory alias (**xm0**[0..15]) offset: 0x0 from start of memory
6x 32-bit Alias (**RAMalias**[20..25]) offset: 0x2 from start of memory

***4x memory - indexed[0..3]***

**(pmem):**

*Parameters:*

Block size: 32-bit , SubBlock size: 8-bit
Consists of an array 65536 (0x10000) blocks => 262144 bytes
Flags: read-write, executable

*Connected to:*

BUS: System BUS (myBUS2)

## RAMS:

**Data memory (TESTdata_mem):**

*Parameters:*

Block size: 8-bit , SubBlock size: 8-bit
Consists of an array 255 (0xFF) blocks => 255 bytes
Flags: read-write, non-executable
Endianess: big-endian
Read Latency: 1 ,Write Latency: 1

*Alias:*

6x 32-bit Test Alias (**AliasRAM**[20..25]) offset: 0x2 from start of memory

**Data memory (data_mem):**

*Parameters:*

Block size: 8-bit , SubBlock size: 8-bit
Consists of an array 255 (0xFF) blocks => 255 bytes
Flags: read-write, non-executable
Endianess: big-endian
Read Latency: 1 ,Write Latency: 1

***4x memory - indexed[0..3]***

**Program memory (program_mem):**

*Parameters:*

Block size: 8-bit , SubBlock size: 8-bit
Consists of an array 255 (0xFF) blocks => 255 bytes
Flags: read-only, executable
Endianess: big-endian
Read Latency: 3 ,Write Latency: 4

## CACHES:

**test cache (register_cache):**

*Parameters:*

Block size: 8-bit , SubBlock size: 8-bit
Consists of an array 8 (0x8) blocks => 8 bytes
Flags: read-write, non-executable
Endianess: little-endian
Read Latency: 1 ,Write Latency: 1
Number of ways for set-associative cache: 4
Number of blocks in the line: 2
Write-Allocate strategy: Always, Write-Back strategy: Always
Replacement strategy: Least recently used
Selective write buffer size: 0 lines of cache

*Connected to:*

BUS: System BUS (myBUS2)
RAM: Program memory (program_mem)

## DEFAULT MEMORY MAP:

| Address | Region |
|---|---|
| 0x000000 | **UNIMPLEMENTED** |
| 0x000009 | 10 BYTES |
| 0x00000A | **Program memory (program_mem)** |
| 0x0000C8 | 191 BYTES |
| 0x0000C9 | **Test array of registers (nee)** |
| 0x0000FF | 55 BYTES |
| 0x000100 | **UNIMPLEMENTED** |
| 0x000103 | 4 BYTES |
| 0x000104 | **(pmem)** |
| 0x000105 | **test cache (register_cache)** |

## ANOTHER MEMORY MAPS:

### Another mem map (anothermap):

| Address | Region |
|---|---|
| 0x000000 – 0x000031 | **UNIMPLEMENTED**<br>50 BYTES |
| 0x000032 – 0x0000FE | **Data memory (data_mem)**<br>205 BYTES |

## PIPELINES:

### test pipeline (pipe):

*States:*

**FE:**

State Registers:

8-bit (fetch)

Latch Registers:

3x - indexed[0..2] 16-bit (fetch_pc)

**DE:**

State Registers:

8-bit test pipeline reg comment (dec_op)
8-bit test pipeline reg comment (dec_dst)
8-bit test pipeline reg comment (dec_src)

**EX:**

State Registers:

8-bit (ex_op)
8-bit (ex_res)
8-bit (ex_dst)

**WB:**

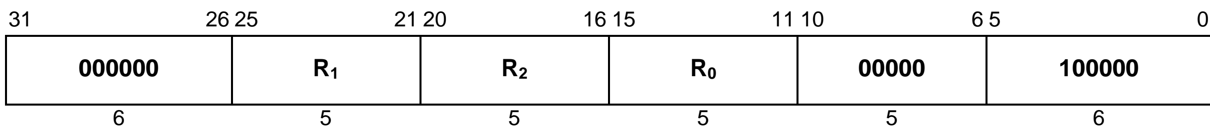
## LOGIC CIRCUITS:

### Logic Circuit (NormalLC)

**Vygenerovaná ukázka instrukcí procesoru MIPS:**

# 2) Instruction Set:

## ADD




*Description:*

The 32-bit word value in GPR R2 is added to the 32-bit value in GPR R1 to produce a 32-bit result. (If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and Integer Overflow exception occurs.) The result is placed into GPR R0.

*Syntax:*

ADD $R_0$ , $R_1$ , $R_2$

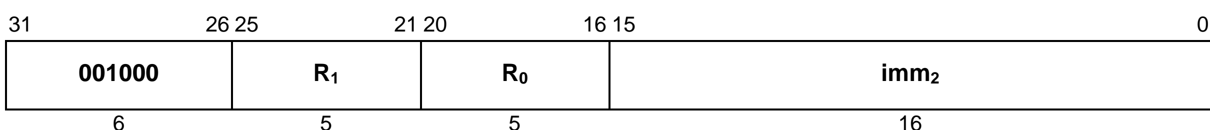
*Latency:*

1

*Semantics:*

[0] cl0#op0 (add (i32 cl0#op2, i32 cl0#op1))

## ADDI




*Description:*

The 16-bit immediate is added (If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and Integer Overflow exception occurs.) to the 32-bit value in GPR R1. The result is placed into GPR R0.

*Syntax:*

ADDI $R_0$ , $R_1$ , $imm_2$

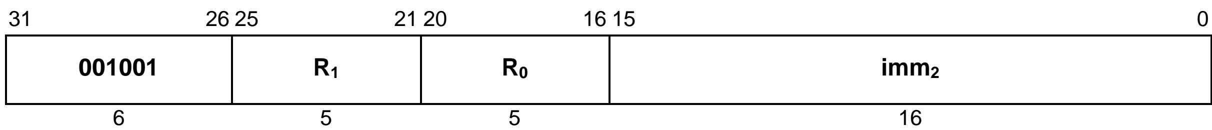
*Latency:*

1

*Semantics:*

[0] cl0#op0 (add (i32 cl0#op1, i32 sext (i16 op2)))

## ADDIU




*Description:*

The 16-bit immediate is added to the 32-bit value in GPR R1. The result is placed into GPR R0.

*Syntax:*

ADDIU $R_0$ , $R_1$ , $imm_2$

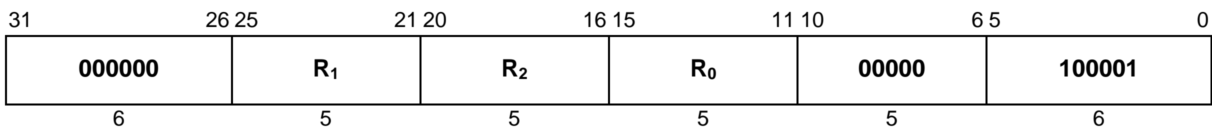
*Latency:*

1

*Semantics:*

[0] cl0#op0 (add (i32 cl0#op1, i32 sext (i16 op2)))

## ADDU




*Description:*

The 32-bit word value in GPR R2 is added to the 32-bit value in GPR R1 to produce a 32-bit result. The result is placed into GPR R0.

*Syntax:*

ADDU $R_0$ , $R_1$ , $R_2$

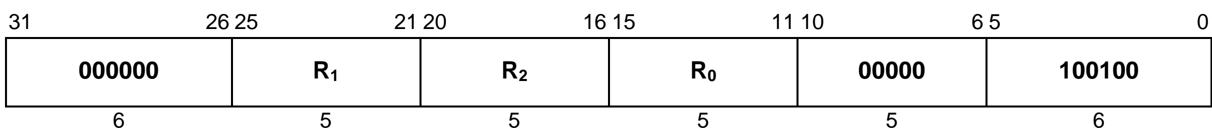
*Latency:*

1

*Semantics:*

[0] cl0#op0 (add (i32 cl0#op2, i32 cl0#op1))

## AND




*Description:*

The 32-bit word value in GPR R2 is combined with the 32-bit value in GPR R1 in a bitwise logical AND operation. The result is placed into GPR R0.

*Syntax:*

AND $R_0$ , $R_1$ , $R_2$

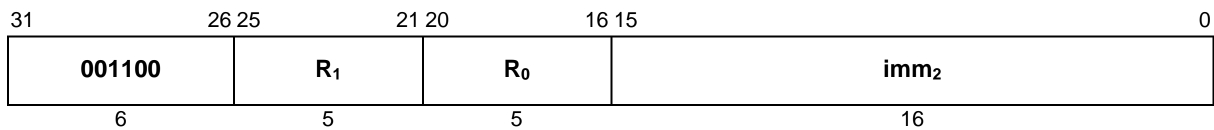
*Latency:*

1

*Semantics:*

[0] cl0#op0 (and (i32 cl0#op2, i32 cl0#op1))

## ANDI




*Description:*

The 16-bit immediate is zero-extended to the left and combined in bitwise logical AND operation with the 32-bit value in GPR R1. The result is placed into GPR R0.

*Syntax:*

ANDI $R_0$ , $R_1$ , $imm_2$

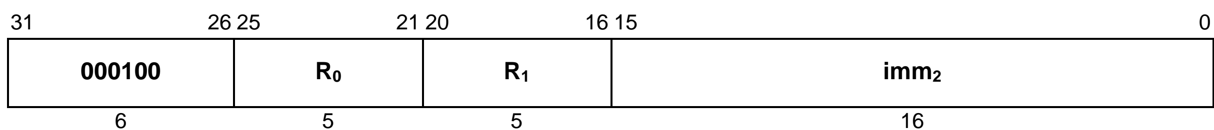
*Latency:*

1

*Semantics:*

[0] cl0#op0 (and (i32 cl0#op1, i32 zext (i16 op2)))

## BEQ




*Description:*

If the contents of GPR R0 and GPR R1 are equal, branch to the effective target address after the instruction in the delay slot executed. An 18-bit signed offset (the 16-bit imm2 field left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

*Syntax:*

BEQ $R_0$ , $R_1$ , $imm_2$

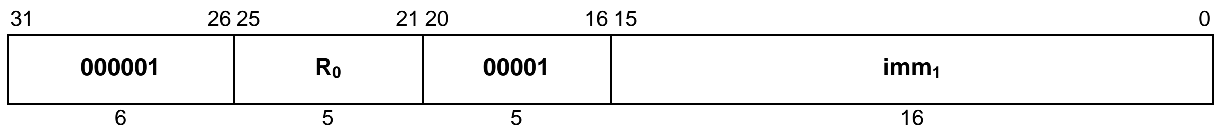
*Latency:*

1

*Semantics:*

[0] %_b = i16 op2
[0] if (seteq (i32 cl0#op0, i32 cl0#op1))
   [1] %_i = shl (i32 sext (%_b), i32 2)
   [1] br (shl (i32 sext (%_b), i32 2))

## BGEZ



| 31 ... 26 | 25 ... 21 | 20 ... 16 | 15 ... 0 |
|---|---|---|---|
| **000001** | **$R_0$** | **00001** | **$imm_1$** |
| 6 | 5 | 5 | 16 |

*Description:*

If the contents of GPR R0 are greater than or equal zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot executed. An 18-bit signed offset (the 16-bit imm1 field left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

*Syntax:*

BGEZ $R_0$ , $imm_1$

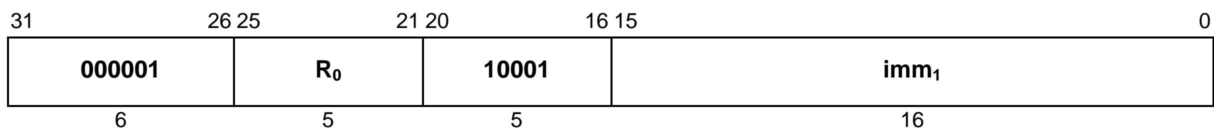
*Latency:*

1

*Semantics:*

```
[0] %_b = i16 op1
[0] if (setge (i32 cl0#op0, i32 0))
   [1] %_i = shl (i32 sext (%_b), i32 2)
   [1] br (shl (i32 sext (%_b), i32 2))
```

## BGEZAL



| 31 ... 26 | 25 ... 21 | 20 ... 16 | 15 ... 0 |
|---|---|---|---|
| **000001** | **$R_0$** | **10001** | **$imm_1$** |
| 6 | 5 | 5 | 16 |

*Description:*

If the contents of GPR R0 are greater than or equal zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot executed. Place the result address link in GPR 31. The return link is the address of the second instruction following the branch, where execution continues after a procedure call. An 18-bit signed offset (the 16-bit imm1 field left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

*Syntax:*

BGEZAL $R_0$ , $imm_1$

*Latency:*

1

*Semantics:*

```
[0] %_b = i16 op1
[0] if (setge (i32 cl0#op0, i32 0))
   [1] cl0#31 (i32 getcurrpc())
   [2] %_j = shl (i32 sext (%_b), i32 2)
   [2] br (shl (i32 sext (%_b), i32 2))
```

# Příloha 3

**Diagram tříd zdrojových prvků:**

# Příloha 4

**Obsah CD:**

- **bin**
  - geninstrman – přeložená aplikace v OS Kubuntu 32-bit
  - mips.xml – přeložený ISAC model procesoru MIPS – model převzat z CVS projektu
  - res_instrsyntsem.txt – mezijazyk pro generátor překladače jazyka C procesoru MIPS
  - mips-manual.rtf – vygenerovaný manuál na základě dvou výše uvedených souborů
- **dokumentace**
  - html – programová dokumentace ve formátu HTML (index.html – hlavní soubor)
  - refmanual.pdf – programová dokumentace ve formátu PDF
- **zdrojovy_kod**
  - geninstrman – zdrojove soubory generátoru manuálu

    .make.mki
    absgen.h
    doxygencomment.h
    doxygencomment.cpp
    geninstr.h
    geninstr.cpp
    geninstrman.cpp
    genresource.h
    genresource.cpp
    manualdoc.cpp
    manualdoc.h
    rtfgen.cpp
    rtfgen.h
    transformresources.cpp
    transformresources.h
  - compilergenl – mnou implementované zdrojové kódy do generátoru překladače

    convbincomm.cpp
    convbincomm.h
- **zprava**
  - xkrenm00.doc – dokument technické zprávy ve formátu DOC
  - xkrenm00.pdf – dokument technické zprávy ve formátu PDF