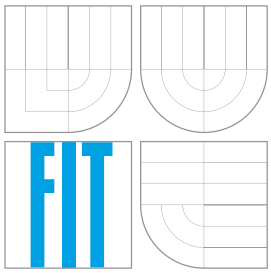
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VYTVOŘENÍ MODELU PROCESORU RISC-V

MODELING OF PROCESSOR RISC-V

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MILAN NOSTERSKÝ

VEDOUCÍ PRÁCE
SUPERVISOR

prof. Ing. TOMÁŠ HRUŠKA, CSc.

BRNO 2016

Zadání bakalářské práce/17675/2015/xnoste00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Nosterský Milan**
Obor: Informační technologie
Téma: **Vytvoření modelu procesoru RISC-V**
RISC-V Model Creation
Kategorie: Počítačová architektura

Pokyny:

1. Seznamte se s popisným jazykem CodAL a s integrovaným vývojovým prostředím Codasip Studio, určeným pro souběžný návrh hardwaru a softwaru. Nastudujte způsob generování programovacích a emulačních nástrojů v tomto prostředí.
2. Seznamte se s architekturou procesoru RISC-V.
3. Navrhněte vybrané části základní verze modelu procesoru RISC-V.
4. Po poradě s vedoucím zvolte některou z rozšiřujících variant procesoru RISC-V a vytvořte a odlaďte její model.
5. Otestujte a zhodnoťte provedené řešení.

Literatura:

- Manuál prostředí Codasip Studio a jazyka CodAL;
- Manuál procesoru Altera NIOS II;
- Dle doporučení vedoucího;

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

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

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Hruška Tomáš, prof. Ing., CSc.**, UIFS FIT VUT
Konzultant: Husár Adam, Ing., CODASIP
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

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

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Bakalářská práce se zabývá implementací modelu procesoru RISC-V v jazyce pro popis architektur CodAL.

Teoretická část práce se zaměřuje na popis jazyka CodAL a klasifikaci procesorů. Praktická část práce se věnuje samotné implementaci procesoru RISC-V na úrovni instrukčního modelu a jeho testování. Dále se práce zabývá implementací MMU, časovačem a analýzou proxy kernelu.

Abstract

This bachelor thesis deals with the implementations of RISC-V processor model in the language for architecture description CodAL.

The theoretical part of thesis is focused on the description of CodAL language and classification of processors. The practical part of thesis deals with the implementation of processor RISC-V on instruction accurate level and the model testing. The thesis also deals with the implementation of MMU, timer and analysis of the proxy kernel.

Klíčová slova

CodAL, Codasip, procesor, RISC-V, ADL, model, MMU, časovač, proxy kernel

Keywords

CodAL, Codasip, processor, RISC-V, ADL, model, MMU, timer, proxy kernel

Citace

Milan Nosterský: Vytvoření modelu procesoru RISC-V, bakalářská práce, Brno, FIT VUT v Brně, 2016

Vytvoření modelu procesoru RISC-V

Prohlášení

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

.....
Milan Nosterský
16. května 2016

Poděkování

Chtěl bych poděkovat vedoucímu své práce panu Prof. Ing. Tomáši Hruškovi, CSc., zaměstnancům firmy Codasip s.r.o a především Ing. Adamu Husárovi, Ph.D. za vstřícnost, odborné rady a věnovaný čas. Dále bych chtěl poděkovat rodině a přátelům za veškerou podporu a pomoc během studia.

Obsah

Kapitola 1

Úvod

S neustále zvyšujícím se počtem chytrých zařízení a jejich zdokonalováním, se stává velmi důležitou součástí vývoje i rychlost a ověření správnosti návrhu procesoru, který tato zařízení řídí. Obě tyto důležité součásti vývoje lze zajistit pomocí návrhu modelu procesoru. Model se vyznačuje několika výhodami například model lze jednoduše ověřit a veškeré úpravy modelu mohou být ihned otestovány.

Z důvodu jednoduchosti úprav modelu se s tímto modelem i snáze experimentuje což vede k jednoduššímu a rychlejšímu hledání optimalizací procesoru. Za další výhodu softwarového modelu lze označit i to, že pokud návrh modelu zabere kratší dobu a model se snadno optimalizuje, tak se šetří čas i peníze, které by bylo nutné investovat do případných prototypů.

Samotná práce se zabývá návrhem modelu procesoru RISC-V na úrovni instrukcí. Procesor RISC-V byl původně vyvinutý na univerzitě v Berkley, díky otevřené instrukční sadě se stává stále více populární a brzy by měl konkurovat procesorovým jádrům jako je ARM a nebo MIPS. [16] Tento model je implementován v jazyce CodAL, kterému se věnuje kapitola 2. V této kapitole se nachází popis jazyka CodAL i jeho sématika. Kapitola 3 se věnuje popisu jendotlivých skupin procesorů například procesory s harvardskou architekturou apod.

Samotnou implementací modelu procesoru RISC-V se zabývá kapitola 4, kde je implementovaný model představen s jeho základní instrukční sadou a jedním rozšířením.

Kapitola 5 popisuje periferie procesoru RISC-V, které jsou nezbytné pro procesor v případě spouštění jádra operačního systému (Linuxu).

Kapitola 2

Jazyk CodAL

2.1 Zařazení jazyka CodAL

Použitím jazyků pro popis architektury se redukuje čas, který je potřebný pro návrh procesoru, a díky tomu se snižují náklady na tvorbu procesoru i výsledná cena procesoru.

Máme dva druhy jazyků, těmi jsou HDL (Hardware description language) a ADL (Architecture description languages). HDL neboli jazyky pro popis hardware slouží pro návrh a simulaci hardware. HDL má řadu nedostatků, například jejich simulace je pomalá a neobsahují některé informace (např. syntaxe assembleru). [8]

Tyto nedostatky překonávají ADL, tedy jazyky pro popis architektury procesorů. ADL se dělí na tři kategorie, kterými jsou:

- jazyky zaměřené na instrukční sadu,
- jazyky zaměřené na strukturu a
- jazyky zaměřené na instrukční sadu a strukturu.

Jazyky zaměřené na instrukční sadu

Tyto jazyky se zabývají pouze popisem instrukční sady procesoru, ale ne jejich hardwarovou implementací. Oddělují model chování od modelu struktury. Slouží především ke generování přenositelných překladačů vyšších programovacích jazyků. Zde spadají jazyky nML, ISDL, Valen-C a CSDL.

Jazyky zaměřené na strukturu

Jazyky zaměřené na strukturu se zaměřují na hardwarovou implementaci instrukční sady na přiměřené úrovni abstrakce. Od HDL se liší tím, že umožňují modelovat s různými stupni abstrakce, kdy při nižším stupni abstrakce získáváme detailnější popis architektury. Nevýhodou těchto jazyků je pomalé generování simulátorů. Patří sem jazyky MIMOLA, AIDL.

Jazyky zaměřené na instrukční sadu a strukturu

Spojují předchozí dva druhy ADL, patří sem jazyk CodAL, jehož popisu bude věnována zbývající část kapitoly. Dalšími zástupci jsou FlexWare, Mdes, PEAS, RADL a LISA.

2.2 CodAL

Jak bylo zmíněno v předchozí části jazyk CodAL patří mezi jazyky, sloužící pro popis architektury i instrukční sady procesoru. Nástroje pro simulaci a ladění samotného procesoru jsou poté generovány ze samotného modelu procesoru popsaného jazykem CodAL. [3, 6].

Základem celého modelu jsou dvě části: platforma a ASIP (Application Specific Instruction-set Processor). Toto rozdělení umožňuje využít ASIP na různých platformách bez nutnosti jeho změny.

Platforma popisuje nejbližší okolí ASIP, například popis paměti a periferních zařízení (časovač, dělička apod.).

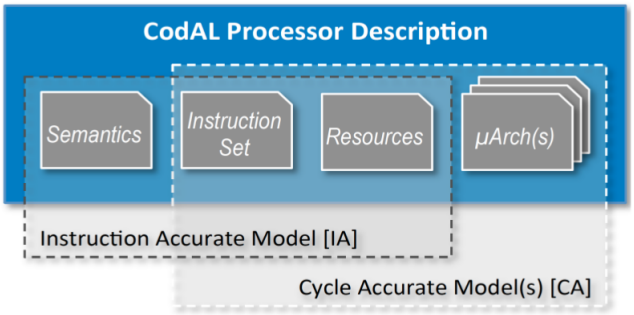
Popis ASIPU má 4 části, kterými jsou:

- zdroje architektury - například registry a programový čítač,
- instrukční sada - jména instrukcí a jejich binární podoba,
- sémantika - popis chování instrukcí a
- implementace - definice pro popis microarchitektury.

Máme dva druhy ASIP modelů:

- IA - Instruction Accurate Model obsahuje zdroje architektury, instrukční sadu a sémantiku a
- CA - Cycle Accurate Model obsahuje zdroje architektury, instrukční sadu a implementaci.

Toto rozdělení je znázorněno na obrázku 2.1.



Obrázek 2.1: IA a CA [5]

Práce se dále zabývá jen instrukčním modelem (IA), kdy syntaxe je obdobná například s jazykem C. V následující části se uvádí popis základních částí IA.

2.2.1 Zdroje

Mezi zdoje, které můžeme popsat, patří hlavně registry. Jazyk CodAL dovoluje jak popis běžných registrů, tak i registrů speciálních. Mezi speciální registry patří:

- pc - Program counter neboli programový čítač. Tento registr je automaticky rozpoznán jako architekční registr. V každém modelu procesoru se nachází jeden programový čítač, který určuje adresu následující instrukce.
- arch - Říká, že se jedná o registr architektury.
- alias - Poskytuje přístup k registru přes jiné jméno.

Kromě registrů mohou zdroje ASIP obsahovat i signály, pipeline[1], interface[2], porty apod.

2.2.2 Instrukční sada

Každou instrukční sadu lze popsat dvěma základními konstrukcemi element a set.

Element popisuje celé instrukce nebo jen jejich části. Popis elementu se nachází na ukázce 2.1.

```
element nazev_elementu {     // jméno elementu
  use xy_part as reg;        // použití existujícího prvku
  assembler { "INS" reg };   // reprezentace instrukce v jazyce
                                assembler
  binary {0xFFFFFFF reg};    // binární zakódování instrukce
  semantics { ... };         // definice sémantiky elementu
  return { val; };           // definice návratové hodnoty
  timing { ... };            // událost spojená s elementem
};
```

Zdrojový kód 2.1: Struktura elementu jazyka CodAL

Konstrukce set definuje uskupení elementů nebo set. Set může být definován i uvnitř elementu. Pomocí set lze z elementů vytvořit stromovou strukturu, obsahující celou instrukční sadu.

2.2.3 Sémantika

Sekce pro popis sémantiky můžeme najít v sekcích typu element nebo event[3]. Sémantika popisuje chování těchto bloků za pomocí jazyka ANSI C, kdy CodAL podporuje všechny operátory jazyka ANSI C, kromě ukazatelů a funkce sizeof.

V této sekci se nejčastěji pracuje se zdroji, kdy se načte hodnota ze vstupního zdroje, zpracuje se a uloží do zdroje výstupního. Jako zdroj zde může být hodnota přímo zadaná uživatelem, paměť nebo registr.

[1] Zřetězená linka, umožňuje zřetězené zpracování instrukcí. Zpracování instrukce se rozdělí minimálně na tyto části načtení, exekuce a zápis do paměti. Tento přístup umožnuje zpracovávat více instrukcí v jednom okamžiku a zvyšuje instrukční průchod

[2] Slouží jako rozhraní pro komunikaci s pamětí nebo periferními zařízeními.

[3] Událost - například restartování procesoru nebo hlavní smyčka procesoru.

2.3 Zásuvné moduly

Tvorba zásuvných modulů umožňuje vytvořit komponentu s pomocí jazyka C nebo C++. Takovou komponentou může být například časovač, řadič přerušení, dělička apod. Za výhodu pluginů lze označit jejich univerzálnost, kdy mohou být použity na libovolné platformě. [4, 6]

Pro tvorbu zásuvných modelů se využívá knihovna codasipplugin. Tato knihovna poskytuje základní funkce pro komunikaci s rozhraními a porty, dále umožňuje nastavení proměnné context, která slouží k uchování pernamentních dat. Proměnnou context tvoří datová struktura, v níchž jsou obsažena důležitá data. Tyto data se týkají dané komponenty například: jméno instance, typ simulace a další data využívaná po celý běh simulace. Samotnou implementaci pluginu lze rozdělit na dvě části.

První část tvoří implementace samotné komponenty na platformě modelu. Vytvoří se tedy rozhraní, pomocí kterého bude daná komponenta komunikovat s modelem, případně dalšími komponentami nebo pamětí.

Druhá část poté obsahuje samotnou implementaci zdrojového kódu pluginu. Základní strukturu zdojového kódu vygeneruje samotné Codasip Studio. Tato vygenerovaná struktura obsahuje výčet událostí, které mohou nastat v průběhu simulace. Zakladními událostmi jsou inicializace komponenty, destruktor komponenty a událost volaná při každém hodinovém cyklu procesoru. Další události jsou poté specifické pro každé rozhraní podle jeho určení, to samé platí i pro jednotlivé porty.

Kapitola 3

Rozdělení procesorů

Procesor dokáže vykonávat libovolné algoritmy sestavené z předem definovaných elementárních operací (instrukcí). [1] Každý procesor je specifický nejen svou instrukční sadou, ale i svojí realizací. V dnešní době lze procesory dělit do několika základních skupin podle jejich instrukční sady, architektury nebo zaměření.

3.1 Podle instrukční sady

Podle instrukční sady procesory dělíme na procesory s úplnou instrukční sadou (CISC - Complex Instruction Set Computing) a procesory s redukovanou instrukční sadou (RISC - Reduced Instruction Set Computing).

CISC

Procesory typu CISC se jednoduše programují a efektivně využívají paměť, ale s rostoucím počtem instrukcí se pro překladače stává obtížným využít celou škálu speciálních instrukcí procesorů, které jsou poté využívány jen zřídka. [10]

Základní charakteristika architektury CISC:

- velké množství instrukcí (100 - 250 instrukcí),
- obsahuje instrukce vykonávající speciální funkce,
- velké množství adresovacích módů,
- instrukce o proměnné délce,
- instrukce pracující s operandy v paměti a
- mnoho složitých způsobů adresování paměti.

RISC

RISC procesory namísto volání jedné složité instrukce, tuto instrukci rozdělí na více jednoduchých instrukcí, které postupně zpracovávají. Koncepce RISC se netýká pouze redukované instrukční sady, ale i dalších charakteristických znaků. Architektura RISC se snaží dosáhnout většího výpočetního výkonu za menší cenu a s menší instrukční sadou. [10]

Základní charakteristika architektury RISC:

- minimální instrukční soubor (80 - 150 instrukcí),
- jednodušší adresování paměti,
- pro přístup do paměti se používají pouze operace LOAD a STORE,
- většina instrukcí se provádí v jednom cyklu,
- jeden nebo málo formátů instrukcí,
- zřetězená realizace instrukcí,
- datové operace pouze nad registry a
- rychlé paměti na uložení operandů a registrů,

3.2 Podle architektury

Harvardská architektura

Harvardská architektura odděluje adresný prostor pro data a instrukce.Tento druh architektury umožňuje současně načítat data i instrukce, jelikož se jedná o dva odlišné paměťové prostory. V architektuře běžných počítačů téměř vymizela, ale stále se využívá u mikrokontrolérů. [10]

Von Neumannova architektura

Von Neumannova architektura se využívá u klasických počítačů, kdy je pouze jeden adresný prostor pro data i program. Tento přístup představuje značnou výhodu pro programátory, kteří nemusí brát v potaz nespojené paměťové prostory. [10]

3.3 Podle zaměření

Univerzální procesory

Univerzální procesory jsou určeny pro obecné užití. Nejčastěji se nachází v osobních počítačích, kde řeší libovolné úlohy. Za jejich nevýhodu lze označit nižší efektivitu výpočtů, oproti aplikačně specifickým procesorům.

Aplikačně specifické procesory

Aplikačně specifické procesory jsou procesory optimalizované pro konkrétní funkci. Díky této optimalizaci dosahují vyšší efektivity, ale za cenu složitějšího návrhu a výrobě, což vede i k vyšší ceně. Za další možnou nevýhodou lze označit, že mohou být modifikovány jen v omezené míře. [7, 9]

Do této skupiny procesorů patří:

- Aplikačně specificke instrukční procesory - ASIP - Programovatelný mikroprocesor, jehož instrukční sada i hardware jsou navrhnuty pro jednu speciální aplikaci a nebo aplikační doménu, kterou bude vykonávat.
- Digitální signálové procesory - DSP - Jedná se o programovatelné mikroprocesory, které slouží především ke zpracování signálů. Dosahují velkých rychlostí při zpracování matematických operací jak v pevné, tak i v plovoucí desetinné čárce. Jsou schopny zpracovávat velký objem dat, ale vyžadují velkou přístupovou rychlost do paměti. Využívají se například u pevného disku, jako digitálně analogový převodník, zajišťování komprese zvuku/obrazu, pro zpracování zvukových efektů apod.
- Aplikačně specifický integrovaný obvod - ASIC - jedná se o obvody, u kterých se algoritmus kompletně implementuje na úrovni hardwaru.

Kapitola 4

Procesor RISC-V

RISC-V byl původně navržen na Kalifornské univerzitě v Berkeley pro výzkum a vzdělávání. Nyní se postupně stává standartem pro otevřené architektury[1] v průmyslu.[16]

U projektu RISC-V se počítá s instrukcemi o délce 32,64 i 128 bitů, následující implementace a popis se ale bude zabývat pouze variantou o velikosti instrukce 32 bitů. Procesor RISC-V je specifický i tím, že nevyužívá stavový registr a příznaky, jak tomu je u jiných procesorů.[2]

4.1 Registrová sada

Základní registrová sada, kterou má uživatel k dispozici se skládá z 32 registrů, značených x0 až x31, kdy každý z registrů má délku 32 bitů. Dále má uživatel k dispozici speciální registr programový čítač, který se značí jako pc.

Registry ze základního registrové sady jsou přiřazeny pro spefické úkony následovně[3]:

- návratová adresa - registr x1,
- bázový ukazatel - registr x2,
- výsledky funkcí - registry x16, x17, x26 a x27,
- ukazatel na zásobník - registr x14 a
- pomocný registr - registr x30.

Dále jsou ještě registry typu callee[4] (registry x3 až x13) a caller[5] (registry x18 až x25) saved, ale ty nejsou v implementaci vyžadovány. Dalším speciálním registrem je registr x0, u kterého se předpokládá, že za každých okolností se bude jeho hodnota rovnat 0. Pro zajištění nulové hodnoty v tomto registru jsem navrhnul dvojici funkcí pro přístup do registrové sady (zápis a čtení), které zajišťují, že z registru x0 bude vždy přečtena hodnota

[1] Otevřená architektura neboli open-source hardware, je obdoba open-source software, kdy má kdokoliv přístup ke zdrojovým kódům např. jednotlivé linuxové distribuce. Zástupce open source hardware může být jmenován například projekt Arduino

[2] Některé procesory architektury RISC i CISC využívají příznaky pro detekci, například přetečení při provádění operace.

[3] Rozřazení odpovídá implementaci procesoru RISC-V, kdy registry x26 a x27 jsou přiřazeny z důvodu odstranění varovného hlášení překladače.

[4] Registry callee saved slouží k uložení hodnot, které musí být chráněny během volání procedury.

[5] Registry caller saved slouží k uložení dočasných proměných pro jejich ochranu přes volání procedury.

0 a tato hodnota nebude nikdy přepsána. Díky využití funkcí pro přístup do registrové sady se nemůže stát, že hodnota x0 bude přepsána, nebo bude na místě registru přečtena neinicializovaná paměť.

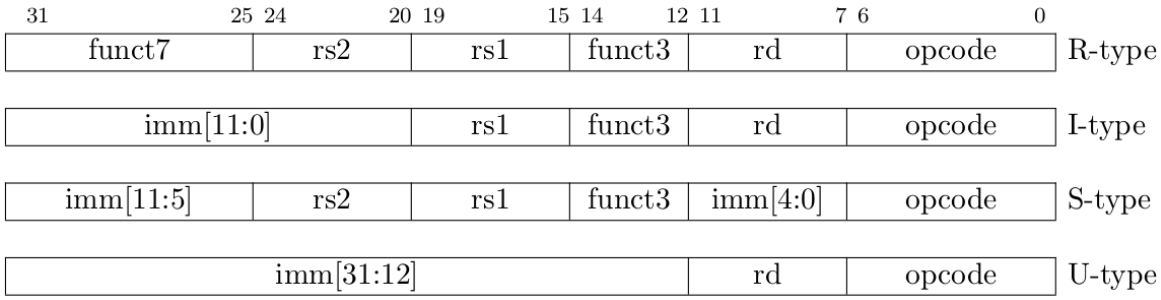
4.2 Kódování instrukcí

Základní instrukční sadu tvoří čtyři typy kodóvání instrukcí, které jsou uloženy na pevné délce 32 bitů a v paměti na 4 bytech.

Jednotlivými typy instrukcí jsou:

- typ R - pro specifikaci 3 registrů v instrukci,
- typ I - umožňuje využití konstanty o délce 12 bitů(Immediate), sloužící jako vstupní operand a dvou registrů,
- typ S - využívá se pouze pro instrukce ukládající do paměti (Store) a
- typ U - tyto instrukce obsahují pouze jeden registr a konstantu délky 20 bitů.

Na obrázku 4.1 lze poté vidět formát jednotlivých instrukcí, kdy rd značí cílový registr, rs1 vstupní registr 1, rs2 vstupní registr 2 a imm označuje konstantu. Jak lze na obrázku vidět, instrukční sada procesoru RISC-V zachovává polohu vstupních registrů i registru cílového a to z důvodu zjednodušení dekódování instrukcí.



Obrázek 4.1: Základní kódování instrukcí[16]

Kódování instrukcí značí, jak je instrukce zapsána v binární podobě. Tato forma odpovídá zápisu v elementu instrukce v sekci binary, na ukázce zdrojového kódu 4.1[6] lze vidět zápis instrukce typu I, kdy v sekci binary je instrukce zapsána ve své binární podobě.

[6]Ukázka znázorňuje pouze sekci binary, jak by mohl vypadat její zápis pokud by byl element tvořen samotnou instrukcí.

```
element i_not
{
    use reg_any as    reg_dst,
                      reg_src1;

    use imm12 as    imm;

    assembler { "ADDI" reg_dst "," reg_src1 "," imm};

    binary { imm reg_src1 0b000:bit[3] reg_dst 0b0010011:bit[7] };

};
```

Zdrojový kód 4.1: Ukázka zápisu sekce binary

4.3 Instrukční sada

Instrukční sada se dělí na jednotlivé sady pro danou bitovou šířku a poté na oficiální, případně neoficiální rozšíření. U 32 bitové instrukční sady jsou dostupné tyto sady:

- I - základní instrukční sada operující s datovým typem integer, realizuje základní aritmetické operace, skoky, přístup do paměti a další speciální operace,
- M - rozšíření základní instrukční sady o násobení a dělení,
- A - rozšíření o atomické paměťové operace, jedná se o instrukce poskytující čtení/-úpravu/zápis,
- F - rozšíření o výpočty v plovoucí řádové čárce s jednoduchou přesností a
- D - rozšíření o výpočty v plovoucí řádové čárce s dvojnásobnou přesností.

Dále jsou poté instrukce sloužící pro přístup do kontrolních a stavových registrů (CSR - control and status registers), které nejsou součástí žádného z výše jmenovaných rozšíření.

Mezi neoficiální rozšíření pro 32 bitovou architekturu patří například "L"decimální aritmetika, "B"bitové operace apod.[11]

4.3.1 I - Základní instrukční sada

Následující část se zabývá implementací základní instrukční sady procesoru RISC-V. Základní instrukční sada obsahuje 47 jednotlivých instrukcí. Z důvodu, že ne všechny instrukce jsou podporovány jazykem CodAL jsou tyto instrukce implementovány tak, aby jim procesor rozuměl, ale choval se jako při exekuci NOP instrukce (NOP - no operation, nic se v cyklu neprovede). Tyto instrukce jsou implementovány i přesto, že nemají žádné chování, aby procesor rozuměl celé instrukční sadě a nemohlo dojít k chybě u překladu instrukcí.

Instrukce budou představeny tak, jak tvoří jednotlivé elementy, které jsou následně přiřazeny jedinému elementu ISA, který obsahuje celou instrukční sadu implementovaného procesoru RISC-V.

Aritmetické instrukce

Jsou zde dva druhy aritemtických operací a to s dvěma vstupními registry nebo s jedním vstupním registrem a konstantou. Každá z těchto variant má jinak zakódovanou instrukci, ale mají 3 společné bity, podle kterých lze detekovat o jakou operaci se jedná.

Při samotném zpracování aritmetické operace se tedy volá funkce alu3op, která na vstupu očekává konstanty. V případě, že je vstupní operand instrukce registr, pak se nejprve načte hodnota tohoto registru a až poté se volá funkce pro zpracování aritmetických operací. Tento způsob volání funkce umožňuje mít pouze jednu implementaci pro dva druhy instrukcí.

Instrukce patřící do této skupiny jsou:

- ADD - aritmetický součet,
- SLL - logický posun doleva,
- SLT - nastaví 1 do cílového registru, pokud je vstup 1 menší než vstup 2, v případě konstanty na vstupu je znaménkově rozšířená,
- SLTU - obdoba SLT, jen konstanta není rozšířena znaménkově,
- XOR - exkluzivní disjunkce nad vstupy,
- SRL - logický posun doprava,
- OR - logická disjunkce,
- AND - logický součin,
- SUB - odčítání vstupů, dostupný jen pokud oba vstupy tvoří registry a
- SRA - aritmetický posun doprava.

Z těchto instrukcí můžeme vytvořit poté další instrukce. Tyto pseudoinstrukce se zapisují v jazyce CodAL jako aliasy.

Implementaci aliasu znázorňuje ukázka zdrojového kódu 4.2, kde se nachazí instrukce NOT. Tato instrukce odpovídá zápisu XORI reg_dest, reg_src, -1, tento zápis poté odpovídá sekci binary v elementu. V samotném elementu se využívá cílový a zdrojový registr. V sekci assembler se nachází v assembleru zapsaná instrukce NOT s operandy, ale v sekci binary už se nachází na prvních 12 bitech číslo -1, zdrojový registr a místo kódu NOT se využije kód XORI. Poté při zpracování instrukce se využije sémantika XORI instrukce.

```
element i_not : assembler_alias(i_reg_imm_inst), compiler_alias(
    i_reg_imm_inst)
{
    use reg_any as   reg_dst,
                     reg_src1;

    assembler { "NOT" reg_dst "," reg_src1};

    binary { 0xFFF:bit[IMM12_W] reg_src1 OPC_XORI:bit[3] reg_dst
        REG_IMM_OPC:bit[ALU_OPC_W] };
};
```

Zdrojový kód 4.2: Implementace instrukce NOT s využitím aliasu

Obdobným způsobem jsou tvořeny i další aliasy. V této sekci to jsou instrukce:

- MV rd, rs - slouží pro přesun rs1 do rd(ADDI rd, rs1, 0[7]),
- SEQZ rd, rs - nastaví rd na 1 pokud rs rovno nule (SLTIU rd, rs1, 1),
- SNEZ rd, rs - nastaví rd na 1 pokud rs není rovno nule (SLTU rd, x0, rs2) a
- NOP - při exekuci si pouze inkrementuje registr pc o 4, což značí posuv na další instrukci (ADDI x0, x0, 0).

Poslední instrukce patřící do této kategorie jsou LUI a AUIPC. Instrukce LUI slouží k načtení konstanty do horních 20 bitů registru, kdy se spodních 12 bitů naplní nulami.

LUI se využívá v instrukci pro překladač, která se stará o načtení konstanty délky 32 bitů. Načtení konstanty se poté skládá z pěti kroků, jak je znázorněno na ukázce kódu 4.3

```
instr load_imm32, ok,
{ gpreg_0 = regop(regs), imm_1 = immop() },
%7 = i32 imm_1;
gpreg_0 = %7;
,
"LUI" gpreg_0 "," imm_1 " >> 12\n"     //Načtení horních 20 bitů
"ORI R30 , R0 ," imm_1 " & 0xFFF\n"    //Načtení spodních 12 bitů
"SLLI R30 , R30 , 20\n"           //Nastavení horních 20 bitů na 0
"SRLI R30 , R30 , 20\n"
"OR" gpreg_0 "," gpreg_0 ", R30\n",    //Výsledek v gpreg_0

0b00000000 0b00000000 0b00000000 0b00000000
0b00000000 0b00000000 0b00000000 0b00000000
0b00000000 0b00000000 0b00000000 0b00000000
0b00000000 0b00000000 0b00000000 0b00000000
0b00000000 0b00000000 0b00000000 0b00000000,
"",
"",
"",
{{0}}
```

Zdrojový kód 4.3: Instrukce překladače pro načtení 32 bitové konstanty

Tento přístup se využívá z důvodu, že instrukce ORI načtenou konstantu znaménkově rozšíří z 12 bitů na 32 bitů, proto se musí horních 20 bitů načtené konstanty nulovat.

Instrukce AUIPC načte 20 bitovou konstantu do horních 20 bitů a spodní naplní nulami stejně jako LUI, ale navíc tento výsledek přičte k obsahu registru pc.

Nepodmíněné skoky

Pro nepodmíněný skok slouží instrukce JAL, JALR a pseudoinstrukce J. Instrukce JAL provede skok na adresu, která se získá výpočtem offsetu ze zadané konstanty a přičtením k registru pc, kdy se původní hodnota registru inkrementována o 4[8] uloží do cílového registru. Registr pc se před uložením musí inkrementovat, aby při návratu z funkce nedošlo k opětovnému skoku a následnému zacyklení procesoru na určitém bloku instrukcí.

[7] I v instrukci značí, že využívá konstantu (Immediate)
[8] Tato hodnota odpovídá délce jedné instrukce v bytech.

JALR instrukce dovoluje namísto přičítání pouze k registru pc, přičíst offset ke kterémukoliv registru a výsledek poté vložit do pc, který se stejně jako v případě instrukce JAL ukládá inkrementován o 4 do cílového registru.

V případě instrukcí nepodmíněných skoků bylo nutno implementovat alias pro překladač assembleru a jazyka C, který přesně specifikuje co se děje při volání a co při skoku.

Pseudoinstrukce J pracuje stejně jako instrukce JAL, jen neukládá pc do cílového registru, tedy lze ji zapsat následovně: JAL x0, imm .

Podmíněné skoky

U podmíněných skoků se vždy porovnají dva registry a na základě tohoto porovnání se pokračuje, nebo se provede skok. Skok se provádí na adresu získanou z vynásobení offsetu dvěma a přičtení výsledku k registru pc.

Procesor RISC-V má pouze následující podmíněné skoky:

- BEQ/BNE - skok se provede, pokud si jsou/nejsou registry rovnym,
- BLT[U] - skok se provede, pokud je registr menší, využívá znaménkové/neznaménkové porovnání a
- BGE[U] - skok se provede, pokud je registr větší nebo roven, využívá znaménkové/-neznaménkové porovnání.

Další podmíněné skoky, jako například BGT (větší než) nebo BLE (menší nebo rovno) se získají výměnou vstupních registrů u BLT respektive BGE.

Tyto instrukce se tvoří pomocí specifikací pro překladač, které se nachází v souboru user_rulelib.rl.

Instrukce pro přístup do paměti

Jak bylo zmíněno v části 3.1, procesory typu RISC využívají pro přístup do paměti pouze operace LOAD a STORE. Instrukce RISC-V pro LOAD a STORE se dělí ještě podle velikosti a způsobu ukládání dat.

Instrukce LOAD načítají podle typu instrukce určitý počet bitů do cílového registru z adresy, která se získá sečtením offsetu s hodnotou zdrojového registru. Pro získání základní adresy se provede nad získanou hodnotou operace and s maskou. Adresa se následně zadá na vstup rozhraní pro komunikaci s pamětí v režimu čtení. Další parametry pro čtení z rohraní jsou konečná adresa a počet přečtených bytů. Operace pro čtení z paměti jsou následovné:

- LB - načte 8 bitů z paměti do cílového registru a rozšíří znaménkově na 32 bitů,
- LH - načte 16 bitů z paměti do cílového registru a rozšíří znaménkově na 32 bitů,
- LW - načte 32 bitů z paměti do cílového registru,
- LBU - načte 8 bitů z paměti do cílového registru a rozšíří nulami na 32 bitů a
- LHU - načte 8 bitů z paměti do cílového registru a rozšíří nulami na 32 bitů.

Operace STORE pracují obdobně jako operace LOAD. Jediný rozdíl tvoří rozhraní pro komunikaci s pamětí, kdy se rozhraní nachází v režimu zápisu. Parametry funkce tvoří data k zápisu, adresy stejně jako v předchozím případě a kolik bytů se zapisuje.

Instrukce pro zápis do paměti jsou poté pouze 3 a to:

- SB - uložení spodních 8 bitů zdrojového registru,
- SH - uložení spodních 16 bitů zdrojového registru a
- SL - uložení spodních 32 bitů zdrojového registru.

Operace s pamětí

V případě implementace více jader RISC-V na jednom čipu může docházet ke kolizi při práci s paměťovým prostorem. K řešení tohoto problému slouží instrukce FENCE a FENCE.I. První jmenovaná slouží ke garanci pořádí operací, které přístupují do paměti. Instrukce FENCE.I slouží k vynucení zápisu do paměti.

Tyto instrukce jsou v rámci implementovaného modelu RISC-V navrhnuty jako operace NOP. Implementovaný model se skládá pouze z jednoho jádra a není tedy nutné využívat tyto speciální instrukce.

Systémové instrukce

Slouží k přístupu k systémovým funkcím. V základní instrukční sadě procesoru RISC-V se nachází 8 systémových instrukcí. První dvě jsou SCALL a SBREAK, zbylých šest slouží k obsluze čítačů a časovačů.

Instrukce SCALL slouží k vytvoření požadavku na některou z funkcí operačního systému. K její implementaci se využívá funkce jazyku CodAL jménem codasip_syscall, která se postará o veškerou obsluhu jednotlivých volání. Instrukce SBREAK slouží debugerům[9] k navrácení řízení programu.

Instrukce pro časovače a čítače jsou implementovány stejně jako SCALL s pomocí funkcí jazyka CodAL. Instrukce jsou:

- RDCYCLE - zapíše do cílového registru spodních 32 bitů počtu hodinových cyklů procesoru,
- RDTIME - zapíše do cílového registru spodních 32 bitů reálného času systému procesoru a
- RDINSTRET - zapíše do cílového registru spodních 32 bitů čítače instrukcí.

Další instrukce jsou H varianty výše zmíněných (RDCYCLEH, RDTIMEH, RDINSTRETH), které slouží k zapsání bitů 32 až 64 do cílového registru. Spojením např. instrukcí RDCYCLE a RDCYCLEH lze lehce získat 64 bitové číslo.
Pro načtení jen určitého rozpětí bitů se využívá v jazyce CodAL zápis [x1..x2], kde x1 a x2 značí první respektive poslední bit načítané sekvence. Na ukázce kódu 4.4 lze vidět uložení horních 32 bitů čítače cyklů do proměnné, která se následně zapíše do cílového registru instrukce.

```
res = (uint32) (codasip_get_clock_cycle()[63..32]);
```

Zdrojový kód 4.4: Ukázka sémantiky instrukce RDCYCLEH

[9] Debugger softwarový nástroj využívající se pro hledání chyb a ladění softwaru.

4.3.2 Rozšíření "M"

Jedná se o standardní rozšíření základní instrukční sady procesoru RISC-V o matematické operace násobení, dělení a výpočtu zbytku po dělení.

Násobení

Instrukce pro násobení:

- MUL - násobení dvou 32 bitových čísel a uložení spodních 32 bitů výsledku,
- MULH - násobení dvou znaménkových 32 bitových čísel a uložení horních 32 bitů výsledku,
- MULHU - násobení dvou neznaménkových 32 bitových čísel a uložení horních 32 bitů výsledků a
- MULHSU - násobení znaménkového 32 bitového čísla s neznaménkovým číslem stejné velikosti a uložení horních 32 bitů výsledku.

Instrukce násobení jsou rozděleny tak, že vždy vrací pouze 32 bitový výsledek a to buď spodních, nebo horních 32 bitů. Pokud by byl potřebný výsledek 64 bitový, musela by se využít jedna z kombinací:

- MULH+MUL,
- MULHU+MUL a
- MULHSU+MUL.

V případě využití takové sekvence musí být zachováno pořadí registrů cílových i zdrojových. Navíc registr pro cílových horních 32 bitů nesmí být žádný z registrů zdrojových, z důvodu jeho přepsání v první části výpočtu celkového výsledku.

Dělení

K dělení slouží instrukce:

- DIV - dělení dvou čísel o velikosti 32 bitů,
- DIVU - dělení dvou bezznaménkových čísel o velikosti 32 bitů,
- REM - zbytek po dělení dvou 32 bitových čísel a
- REMU - zbytek po dělení dvou 32 bitových čísel bez znaménka.

Výsledek všech operací se vždy uloží jako 32 bitové číslo.
V případě přetečení při dělení nebo dělení nulou není vyvolána vyjímka, ale jsou přesně definovány výsledky operací. Jednotlivé definice jsou znázorněny v tabulce 4.1.

Podmínka	**Dělenec**	**Dělitel**	**DIV**	**REM**	**DIVU**	**REMU**
Dělení nulou	x	0	$2^X LEN - 1$	x	-1	x
Přetečení (pouze znaménkové)	$-2^{(XLEN-1)}$	-1	-	-	$-2^{(XLEN-1)}$	0

Tabulka 4.1: Výsledky jednotlivých operací při vyjímkách [16, 12]

4.4 Události modelu

O hlavní funkcionalitu modelu se starají dvě základní události (eventy) main a reset. Událost reset inicializuje procesor tak, že nastaví hodnoty všech jeho registrů na hodnotu 0.

Událost main v každém hodinovém cyklu procesoru načte instrukci, inkrementuje registr pc o délku instrukce a zahájí zpracování instrukce pomocí funkce isa, která je set tvořící strom zakódovaných instrukcí.

4.5 Architektura a rozhraní modelu

V sekci architektura jsou specifikovány základní informace o architektuře, které jsou neměnné pro každou platformu. V implementovaném procesoru RISC-V zde specifikuji velikost a počáteční hodnotu registru pc, specifikace registrového prostoru pro registry x0 až x31 a adresový prostor.

V rozhraní modelu jsou implementovány pouze dvě rozhraní pro přístup do paměti, která se nachází v platformě modelu. Jedno rozhraní slouží pro práci s pamětí pro instrukce typu LOAD a STORE. Z toho vyplývá, že slouží pro zápis do paměti i čtení z paměti. Druhé rozhraní, sloužící pro načtení instrukce povoluje jen čtení z paměťového prostoru. Další atributy mají rozhraní společné, jedná se o rozhraní typu MASTER, délka adresy v paměti a délka slova mají shodně 32 bitů, 8 bitů tvoří byte a uspořádání paměti je big endian.

4.6 Platforma

Platforma specifikuje pouze paměťový prostor, který se skládá ze dvou rozhraní pro připojení modelu. Paměť musí mít co se týká atributů stejné uspořádání, délku adresy, délku slova i poslední parametr, udávající kolik bitů tvoří byte. Oproti těmto shodným parametrům se specifikuje velikost paměťového prostoru. Jednotlivá rozhraní paměti poté musí být rozhraní typu SLAVE, ale musí být nastaveny stejně jako v modelu, tedy pro načítaní instrukcí pouze pro čtení a pro instrukce typu LOAD a STORE musí být umožněno čtení i zápis.

Samotné propojení rozhraní pak probíhá pomocí funkce connect, jak je znázorněno na zdojovém kódu 4.5, který znázorňuje paměťový prostor implementovaný na platformě a jeho propojení s modelem procesoru RISC-V.

```
memory mem {
  bits = {ADDR_W, WORD_W, LAU_W};
  size = MEM_SIZE;
  endianness = ENDIAN;
  latencies = {1, 1};
  unaligned = true;

  interface if_fetch
  {
    type = MEMORY:SLAVE
    flag = R;
  };

  interface if_ldst
  {
    type = MEMORY:SLAVE
    flag = RW;
  };

};

connect riscv.if_fetch => mem.if_fetch;
connect riscv.if_ldst => mem.if_ldst;
```

Zdrojový kód 4.5: Paměťový prostor a jeho propojení s modelem procesoru

4.7 Testování modelu

Testování modelu jsem prováděl již v průběhu návrhu, kdy Codasip studio[10] dovoluje vygenerovat překladač jazyka assembler.

S pomocí překladače assembler bylo možné překládat již od začátku návrhu jednoduché algoritmy v jazyce assembler. Tyto jednoduché programy jsem následně pomocí režimu debug v Codasip studiu otestoval a případně provedl potřebné úpravy v samotném návrhu procesoru.

Režim debug dovoluje krokovat algoritmus a sledovat stav registrů, paměti, případně později u jazyka C stav proměnných.

Po dokončení implementace základní instrukční sady procesoru bylo možné vygenerovat překladač jazyka C. Před exekucí programů v jazyce C se musí provést startovací kód v jazyce assembler, ten se nachází v souboru crt0.s. Startovací kód obsahuje inicializaci zásobníku, volání těla hlavního programu v jazyce C a po ukončení hlavního těla se stará o ukončení celého programu.

Poslední testování proběhlo na testovací sadě pro procesory od firmy Codasip. Tato testovací sada obsahuje celkem 702 testovacích souborů pro základní instrukční sadu a rozšíření "M". Výsledkem exekuce testovacího skriptu je jedna z možností:

- OK - test proběhl v pořádku,
- SK - test byl přeskočen,
- CC - problém při kompilaci[11] testu,

[10] Softwarový nástroj pro návrh procesorů v jazyce CodAL od firmy Codasip

[11] Překladu programu do jazyka stroje.

- LNK - problém při linkování[12],
- TO - vypršel čas přidělený pro test,
- EC - test skončil se špatným návratovým kódem,
- SIM - selhání simulace a
- NEC - nebyl vygenerován soubor s návratovým kódem.

Po vyhodnocení všech testů se poté nachází v pracovním adresáři soubory s přesnějším popisem problému.

Výsledek testování s optimalizací o0:

- OK - 688 testů,
- CC - 15 testů a
- LNK - 0 testů.

[12]Sestavování samostně přeložených modulů a knihoven do funkčního celku.

Kapitola 5

RISC-V a Linux

Následující kapitola se zabývá rozšířením modelu procesoru RISC-V[1] o potřebné periferie a instrukce, které jsou nezbytné pro úspěšné spuštění Linuxu. Veškeré poznatky o periferiích jsou získány ze simulátoru procesoru RISC-V jménem SPIKE a zdrojových kódů Linuxu. Periferie se následně k procesoru připojí na úrovni platformy, kde budou vystupovat jako zásuvný modul neboli plugin.

5.1 Linux

Linux je jádro operačního systému, které spolu s dalšími programy tvoří samotný operační systém. Jádro operačního systému lze označit za jeho nejnižší a nejzákladnější část. Jádro navazuje na hardware a zcela ho pro uživatele a uživatelské aplikace zapouzdřuje. Při startu operačního systému se jádro zavádí jako první a beží po celou dobu jeho běhu. Programy mohou jádro žádat o služby pomocí systémových volání.[2, 13]

Linux spadá pod licenci GNU GPL (GNU[2] General Public License), která dovoluje volně šířit a modifikovat obsah zdrojových kódů. GNU GPL také zajišťuje, že jakýkoliv dále upravený obsah bude distribuován pod stejnou licencí.

Díky možnosti prohlédnout si zdrojové kódy Linuxu, lze jednoduše vypátrat instrukce, využívané v inline assembleru[3], které musí model procesoru RISC-V podporovat.

Tyto instrukce jsou:

- Instrukce sloužící pro přístup do CSR (Control Status Registers).
- Operace nad registrem fcsr. Registr fcsr je 32 bitový registr, který slouží jako stavový a řídící registr. Tento registr je zaveden s rozšířením "D".
- AMO - Atomické paměťové operace, jedná se o instrukce poskytující čtení, úpravu a zápis atomických hodnot.
- wfi - Wait for interrupt, dostupná v režimech S,H a M. [4][15]

[1] Z důvodu implementace 64-bitové instrukční sady a řady rozšíření se budou veškeré úpravy aplikovat na procesor, který byl navrhnut ve firmě Codasip.

[2] GNU je rekurzivní zkratka GNU's Not Unix

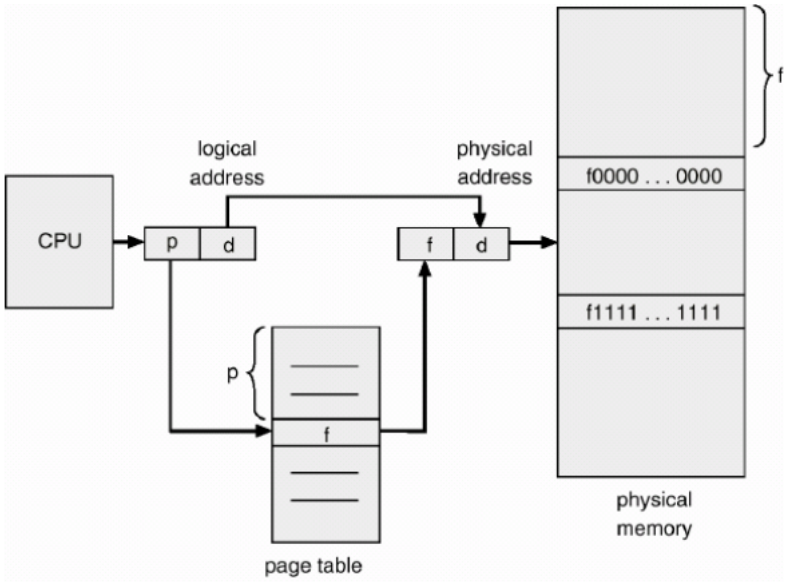
[3] Kód v jazyce assembler vložený mezi příkazy vyššího jazyka

[4] RISC-V má 4 režimy, kterými jsou User/Application (U),Supervisor(S),Hypervisor(H),Machine(M)

5.2 MMU

Memory Managment Unit (MMU) je hardwarová jednotka, sloužící pro překlad logických adres na fyzické. MMU je v dnešní době součástí čipu procesoru. MMU může využívat speciálních registrů, případně i hlavní paměť systému. Pro urychlení může využívat Translation Look-aside Buffer (TLB).

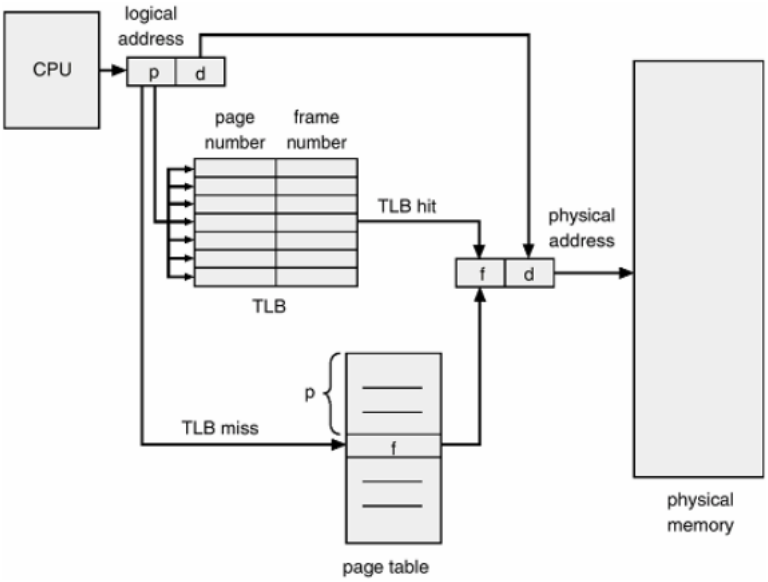
Stránkování je jeden z možných přístupů přidělování paměti, dalšími jsou například Contiguous Memory Allocation nebo segmentace paměti. U stránkování se logický adresový prostor dělí na jednotky pevné velikosti, které se nazývají stránky a fyzický adresový prostor se dělí na jednotky stejné velikosti, které se nazývají rámce. Paměť se přiděluje po rámcích. V nejjednoduším případě tj. jednoúrovňových tabulek stránek, operační systém (OS) udržuje informaci o volných rámcích a pro každý proces tabulku stránek. [14]



Obrázek 5.1: Stránkování[14]

Tabulky stránek jsou udržovány v hlavní paměti. Tabulka stránek obsahuje popis mapování logických stránek do fyzického adresového prostoru a příznaky modifikace, přístupu apod. Každý odkaz na data nebo instrukci v paměti vyžaduje dva přístupy do paměti, kdy se nejdříve podíváme do tabulky stránek a poté na příslušnou adresu ve fyzické paměti. Urychlení je možné s pomocí TLB.

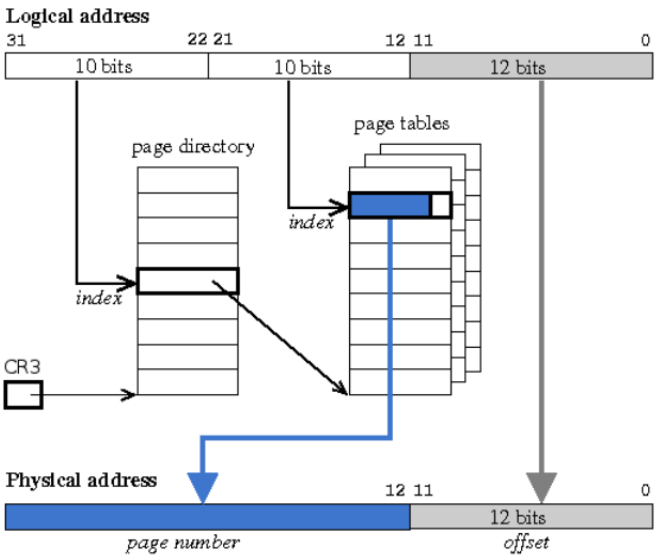
TLB obsahuje dvojice (číslo stránky a číslo rámce) a některý z příznaků, spojený s mapováním. TLB neobsahuje celé stránky nebo rámce. TLB se prohledává paralelně podle čísla stránky a to buď plně nebo jen částečně. U TLB může dojít k TLB miss. V případě TLB miss se u hardwerově řízených TLB automaticky hledá v tabulce stránek. U softwarově řízených TLB se musí v tabulce stránek hledat jádro a poté patřičně upravit TLB.



Obrázek 5.2: Tabulka stránek s TLB[14]

Hierarchie tabulek stránek

Tabulka stránek může být sama stránkována a tím pádem vznikají tabulky tabulek stránek. Obrázek níže znázorňuje dvouúrovňovou tabulku stránek.



Obrázek 5.3: Dvouúrovňová tabulka stránek[14]

5.2.1 Implementace MMU

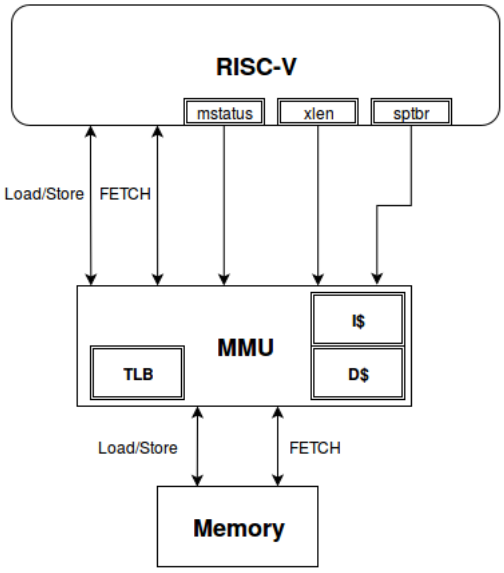
Implementace MMU pro model procesoru RISC-V vychází z implementace MMU ve SPIKE simulátoru tak, aby bylo možné na modelu v budoucnu spustit Linux podporující architekturu RISC-V.

MMU bylo implementováno jako plugin pro platformu procesoru. První proběhl návrh rozhraní MMU a zapojení do modelu na samotné platformě, kdy MMU figuruje mezi procesorem a pamětí. Rozhraní procesoru pro zápis a čtení jsou připojena na MMU, odsud vedou další rozhraní pro Load/Store instrukce procesoru a fetch přímo do paměti. MMU využívá tři stavové registry procesoru, kterými jsou:

- mstatus - informuje v jakém stavu se momentálně procesor nachází ,
- xlen - udává délku instrukce, v našem případě jestli pracujeme s 32 nebo 64 bitovou architekturou procesoru RISC-V a
- sptbr - základní registr tabulky stránek.

Pro přístup k těmto registrům se MMU propojí s procesorem pomocí 3 portů, kdy každý stavový registr má vlastní port. Na tyto porty jsou v každém taktu procesoru přivedeny hodnoty stavových registrů. V implementaci se tato část provádí v eventu main v sekci sémantiky před načtením instrukce.

Na obrázku 5.4 se nachází nákres zapojení procesoru, MMU a paměti. Na nákresu si lze všimnout, že MMU obsahuje TLB i instrukční a datovou cache pamět. Všechny tyto části budou implementovány v rámci jednoho pluginu tak, aby mohly být použity na jakémkoliv modelu RISC-V.



Obrázek 5.4: Zapojení modelu RISC-V, MMU a paměti

Po vytvoření návrhu lze vygenerovat základní strukturu pluginu, kdy se vytvoří funkce pro přístup do MMU z procesoru. Tato vygenerovaná struktura obsahuje funkce MMU pro práci s porty a rozhraními. Implementací těl takto vygenerovaných funkcí se budou zabývat následující části textu.

Inicializace MMU

Nejdříve se při spuštění simulace se zapojeným MMU volá funkce init.V této funkci se inicializují potřebné proměnné, které se uloží do proměnné context, která udržuje data.

Tuto proměnnou definuje struktura context_mmu_t, jež obsahuje nasledující informace:

- název instance,
- typ simulace,
- začátek adresového prostoru paměti,
- velikost paměti,
- registry TLB a
- cache paměti.

Rozhraní pro zápis a čtení dat

Zápis do paměti se provádí na straně procesoru pomocí instrukcí typu STORE. Tyto instrukce přistupují k paměti přes metodu rozhraní write, toto rozhraní je poté připojeno k MMU.

Při provedení zápisu na rozhraní pro data na straně procesoru, se na straně MMU volá funkce write datového rozhraní, komunikujícího s procesorem. Vstupními parametry funkce jsou:

- context,
- data pro zápis,
- adresa pro zápis dat,
- index bloku, od kterého se bude zapisovat a
- počet bytů k zapsání.

Nejdříve se provede výpočet proměnné vpn. Ta se využije pro získání store_tagu z tlb, který se nachází na pozici $vpn\%TLB\_ENTRIES$, kde TLB_ENTRIES uvádí maximální počet záznamů. V případě, že se na takto získané pozici nachází hodnota odpovídající vpn, provede se zápis do paměti na adresu, která se získá součtem offsetu z tlb_data, opět na pozici $vpn\%TLB\_ENTRIES$, a adresy.

Pokud hodnota store_tag neodpovídá hodnotě vpn, tak se pokračuje překladem cílové adresy. Překlad probíhá na základě módu a typu prováděné operace, tedy rozlišuje zda se jedná o operaci LOAD, STORE nebo FETCH. Pokud se procesor při operaci STORE nachází v režimu Machine, pak výsledná adresa odpovídá hodnotě operace AND nad maskou a zadanou adresou.

Pokud se procesor nachází v jiném režimu než Machine, pak se výsledná adresa získá pomocí funkce walk, která zajišťuje průchod tabulkami stránek.

K této získané adrese se poté přičte posun v rámci paměti a následně se provede zápis dat do paměti. Po zápisu se výsledná adresa uloží buď do cache paměti nebo do TLB.

V případě volání některé z funkcí typu LOAD procesorem se na straně MMU volá funkce pro čtení dat. Tato funkce má obdobnou implementaci jako funkce sloužící pro zápis dat, jen s tím rozdílem, že po získání výsledné adresy se na rozhraní LOAD/STORE, které propojuje MMU s pamětí volá funkce čtení.

Rozhraní pro čtení instrukcí

V případě rozhraní pro čtení instrukcí bylo nutné řešit i načtení instrukcí do paměti, které se provádí před samotnou exekucí programu. V případě implementovaného modelu se ukládání instrukcí do paměti provádí přes rozhraní fetch a funkci load.

Vstupní parametry funkce load jsou následující:

- context,
- data pro zápis,
- adresa pro zápis dat,
- index bloku, od kterého se bude zapisovat a
- počet bytů k zapsání.

Na základě těchto údajů poté zapíše do paměti přes rozhraní, které slouží ke čtení instrukcí. Jedná se také o jediný zápis dat přes toto rozhraní, které jinak po celou dobu běhu procesoru slouží pouze pro čtení dat z paměti.

Po zahájení běhu procesoru se poté v každém cyklu provádí načtení instrukce čtením z rozhraní FETCH, kdy hodnota programového čítače odpovídá adrese instrukce. Na straně MMU se poté instrukce načítá v několika krocích. Nejdříve se provede výpočet indexu a následně se načte instrukční záznam. Instrukční záznam tvoří struktura, která obsahuje tag a samotnou instrukci.

Při načítání instrukčního záznamu se nejdříve provede načtení záznamu z instrukční cache paměti. Pokud tag tohoto záznamu neodpovídá adrese instrukce, pak se musí instrukce načíst z paměti a opravit instrukční cache paměť.

Při načítání samotné instrukce se provede kontrola pomocí vypočítané hodnoty vpn a zjistí se, zda tlb neobsahuje adresu dané instrukce. V případě, že se tam adresa nachází, dojde k načtení instrukce z paměti, kdy adresu v paměti udává součet hodnoty z tlb_data a adresy instrukce. Pokud se instrukce v tlb nenachází, dojde k překladu její adresy a následnému naplnění tlb stejně, jako v případě načítání dat z paměti.

Po načtení instrukce se opraví instrukční cache paměť, kdy se vytvoří nový instrukční záznam. Do procesoru se poté vrací instrukce z instrukčního záznamu a to buď vytvořeného nebo načteného z cache paměti.

Implementace pamětí cache

Při implementaci cache paměti se využívá třídy mem_tracer, která se stará o přístup do simulátoru cache paměti. Tato třída obsahuje vektor ukazatelů na připojené paměti cache. Za pomocí tohoto vektoru se určuje, zda se bude přistupovat do cache paměti a případně do které.

Potomek této třídy se nazývá cache_memtracer_t. Ten obsahuje ukazatel na konkretní cache paměť a stará se o tvorbu instancí konkrétní třídy cache paměti. Z cache mem_memtracer_t vychází ještě potomci, kteří zastupují datovou a instrukční cache. Rozdíl mezi těmito potomky tvoří metody interest_in_range a trace, kdy instrukční cache se využívá pouze ve fázi fetch a datová cache paměť se využívá pouze ve fázích Load a Store.

Jak bylo zmíněno dříve o simulaci samotné cache paměti se stará třída cache_sim_t. Tato třída se stará o přístup do cache paměti pomocí metody acces.

Samotný simulátor má poté ještě pro své účely instrukční cache, která je implementována jako pole instrukčních záznamů. Tato instrukční cache se doplňuje s cache pamětí popsanou dříve.

5.3 Časovač

Další komponenta nacházející se ve SPIKE simulátoru je časovač. Časovač slouží k přesnému měření času běhu procesoru. V implementaci SPIKE simulátoru časovač využívá pouze jednu proměnnou rtc. Tato proměnná se jednou za 5000 cyklů procesoru inkrementuje o hodnotu 50. Jedná se o zjednodušenou formu časovače procesoru RISC-V, bez využití porovnávacího registru mtimecomp a registru s aktuální hodnotou času mtime, které by měly sloužit časovači.

Také se v této implementaci nerozlišuje mezi registry určenými pro režim supervisor a registry pro režim machine. V původní implementaci registry machine slouží pouze ke čtení a do registrů typu supervisor se povoluje i zapisovat.

V implementovaném procesoru RISC-V má komponenta pouze jedno rozhraní určené ke čtení hodnoty rtc, která určuje aktuální čas procesoru. Samotná implementace pluginu se skládá pouze z funkcí pro inicializaci, hodinový cyklus a čtení z rozhraní.

Při inicializaci se vytvoří struktura context a inicializují se její hodnoty. Mezi těmito hodnotami se nachází číslo aktuálního cyklu a poté hodnota rtc, která udává čas procesoru. Obě tyto hodnoty jsou inicializovýny na hodnotu 0.

V každém hodinovém cyklu poté dochází k načtení contextu a inkrementace počtu hodinových cyklů. V případě, že hodnota počtu hodinových cyklů odpovídá hodnotě 5000, vypočte se přírustek rtc a aktuální počet cyklů se vynuluje. U požadavku na aktuální čas při instrukci RDTIME se pouze zapíše na rozhraní aktuální hodnota proměnné rtc.

5.4 Proxy kernel

RISC-V proxy kernel slouží k simulaci prostředí, které může hostit binární elf[5] soubory procesoru RISC-V. Toto prostředí slouží k propojení procesoru RISC-V s hostitelským počítačem přes podpůrné funkce proxy kernelu. Mezi podpůrné funkce proxy kernelu patří například systémová volání.

Samotný proxy kernel se volá po inicializaci procesoru RISC-V, kdy s jeho pomocí dojde k načtení parametrů elf souboru a následné načtení samotného souboru do modelu.

Spuštění elf programu

Vstupní bod proxy kernelu se nachází v souboru mentry.S[6]. Po skoku na vstupní bod se provede kontrola stavových registrů, inicializace registrů x0-x31 na hodnotu rovnou 0 a nastavení hodnoty ukazatele na vrchol zásobníku (stack pointer).

Po inicializaci zásobníku se volá funkce init_first_hart. Tato funkce provádí inicializaci stavového registru stroje, v případě implementace rozšíření “D” dojde k inicializaci fcsr a následuje volání funkce boot_loader.

[5] ELF - Executable and Linkable Format formát pro uložení spustielných souborů, linkovatelných objektů, sdílených knihoven a ladících výpisů.

[6] Přípona .S značí soubor v jazyce assembler, který musí být zpracován preprocesorem.

Funkce boot_loader se stará o:

- vytvoření standartního vstupu,
- vytvoření výstupu,
- vytvoření chybového výstupu,
- vytvoření dvou deskriptorů,
- inicializaci virtuální paměti,
- načtení elf souboru a
- volání načteného elf souboru k exekuci.

Inicializace virtuální paměti spočívá v určení počtu stránek paměti, počtu volných stránek a první volné stránky. S pomocí těchto údajů dojde k tvorbě tabulky stránek. Kromě samotné inicializace paměti se nastaví vrchol zásobníku kernelu.

Načtení souboru programu určeného k exekuci spočívá ve zpracování hlavičky elf souboru, nastavení vstupního bodu a namapování elf souboru do paměti simulátoru.

Poslední krok poté tvoří spuštění samotného elf programu, který byl načten. Zde se nejdříve hlavičky elf programu načtou na uživatelský zásobník, následně jsou na zasobník uloženy parametry programu. Dále se spustí časovač, jeho spuštění spočívá v uložení času t0 s pomocí instrukce rdcycle. Stejným způsobem dojde ke spuštění ostatních čítačů architektury. Nakonec se po vymazání cache paměti přejde k exekuci elf programu.

Systémová volání

Jak bylo zmíněno na začátku kapitoly proxy kernel poskytuje simulátoru podporu systémových volání. Celkem podporuje 41 systémových volání mezi kterými se nachází například volání pro ukončení programu, čtení ze souboru, zápisu do souboru apod. Při systémovém volání dojde na straně procesoru k vyvolání přerušení. Toto přerušení zachytí proxy kernel a klasifikuje jej jako systémové volání. Dále se určí o jaké systémové volání se jedná a přejde se na jeho zpracování. V případě, že proxy kernel potřebuje využít některé ze systémových volání, zavolá funkci frontend_syscall. Tato funkce poté uloží vstupní parametry a provede systémové volání.

Úprava proxy kernelu

Pro možnost využití proxy kernelu na implementovaném procesoru RISC-V je nutné udělat několik úprav. Tyto úpravy se týkají modelu procesoru i zdrojových kódů proxy kernelu.

První fáze úprav spočívá v úpravě modelu procesoru RISC-V a zdrojových souborů proxy kernelu. Do modelu byly přidány následující instrukce:

- eret - slouží k návratu do režimu ve kterém vzniklo přerušení,
- csrw - pseudoinstrukce sloužící k zápisu do CSR,
- csrr - pseudoinstrukce pro čtení z CSR a
- sfence.wm - instrukce fence pro režim supervisor.

Kromě výše zmíněných byly do modelu přidány pseudoinstrukce pro úpravu syntaxe instrukcí lw, sfence.vm, fence a instrukcí pro práci s CSR.

První úprava zdrojových kódů proxy kernelu se týká konfiguračního souboru. Zde se opraví parametry překladače s prefixem nástrojů pro překlad a simulaci proxy kernelu. Ty musí být opraveny tak, aby odpovídaly nástrojům pro simulaci ASIP vygenerovaných s pomocí Codasip studia z implementovaného modelu. Další úpravy spočívají v odstranění nevyužívaných částí proxy kernelu implementovaným modelem a opravě nepřeložitelných částí například nahrazení volání makra, které není podporováno samotným kódem.

Po úspěšné kompilaci zdrojových souborů proxy kernelu bylo nutné ve fázi linkování opět opravit zdrojové kódy, tak aby mohl být proxy kernel složen.

Kapitola 6

Závěr

V první části práce byla implementována základní instrukční sada procesoru RISC-V a její rozšíření "M". Tato implementace proběhla na úrovni instrukčního modelu procesoru. Během implementace jsem se detailněji seznámil s jazykem CodAL a řešením problémů, které se vyskytují během návrhu modelu procesoru. Největším problémem během implementace byla realizace instrukcí skoku, které musely být upřesněny aliasy tak, aby bylo možné vygenerovat překladač jazyka C. Model byl poté ověřen na testovací sadě firmy Codasip s.r.o. Díky široké škále testů bylo možné model důkladně otestovat. V případě nálezu chyby v testování jsem využil Codasip Studio a problémový test debugoval a snáze tak našel chybu v návrhu procesoru. Většina chybných testů byla opravena, zbývající chyby z 4.7 byly po domluvě s vedoucím práce ponechány. Po analýze chybových hlášení a debugování příslušných testů jsem zjistil, že chyba se nachází v knihovně newlib. Až na problémy s knihovnou newlib pracuje model bez problémů.

Ve druhé části práce byly implementovány zásuvné moduly pro model procesoru RISC-V. Tyto komponety jsou MMU a časovač, oba zásuvné modely byly otestovány a pracují správně. Implementací zásuvných modulů jsem se detailněji seznámil s funkcí MMU a cache paměti, která je součástí jeho implementace. V podkapitole 5.4 se nachází popis proxy kernelu, který po domluvě s vedoucím práce byl z důvodu vysoké náročnosti pouze analyzován z pohledu spouštění elf programu a obsluhou systémových volání. Následně byl tento proxy kernel po modifikaci zdrojových kódů prolinkován s modelem procesoru RISC-V implementovaným ve firmě Codasip. Tento model procesoru bylo nutné rozšířit o potřebné instrukce, aliasy a stavové registry tak, aby bylo možné proxy kernel přeložit. Výsledkem této části poté je přeložený proxy kernel na implementovaném procesoru.

Literatura

[1] Princip činnosti procesoru. Materiály k předmětu Návrh počítačových systémů., 2015-10-12 [cit. 2016-05-04].

[2] What is Linux. [ONLINE]., [cit. 2016-05-04].
URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[3] Codasip Ltd.: CODASIP INSTRUCTION ACCURATE MODEL TUTORIAL. Dokumentace firmy Codasip s.r.o., 2015-09-03 [cit. 2016-05-04].

[4] Codasip Ltd.: CODASIP INTERRUPTS AND PERIPHERALS TUTORIAL. Dokumentace firmy Codasip s.r.o., 2015-09-03 [cit. 2016-05-04].

[5] Codasip Ltd.: CODAL LANGUAGE REFERENCE MANUAL. Dokumentace firmy Codasip s.r.o., 2015-09-30 [cit. 2016-05-04].

[6] Codasip Ltd.: CODASIP STUDIO TECHNICAL REFERENCE MANUAL. Dokumentace firmy Codasip s.r.o., 2015-09-30 [cit. 2016-05-04].

[7] Kästner, D.: Embedded Systems. [ONLINE]., 2002 [cit. 2016-05-04].
URL http://www.rw.cdl.uni-saarland.de/~kaestner/es0203/lectdk09.pdf

[8] Masařík, K.; Křivka, Z.: Jazyky pro popis architektury. Materiály k předmětu Principy programovacích jazyků a OOP., 2011-05-04 [cit. 2016-05-04].

[9] QASIM, Y.; JANGA, P.; KUMAR, S.; aj.: APPLICATION SPECIFIC PROCESSORS. [ONLINE].
URL http://web.engr.oregonstate.edu/~qassimy/index_files/Final_ECE570_ASP_2012_Project_Report.pdf

[10] Schwarz, J.; Růžička, R.; Strnadel, J.: Mikroprocesorové a vestavěné systémy. Studijní opora k předmětu Mikroprocesorové a vestavěné systémy., 2006 [cit. 2016-05-04].

[11] Tišnovský, P.: Instrukční sada procesorových jader s otevřenou architekturou RISC-V. [ONLINE]., 2015-11-05 [cit. 2016-05-04].
URL http://www.root.cz/clanky/instrukcni-sada-procesorovych-jader-s-otevrenou-architekturou-risc-v/#k04

[12] Tišnovský, P.: Rozšíření instrukční sady procesorových jader s otevřenou architekturou RISC-V. [ONLINE]., 2015-11-12 [cit. 2016-05-04].
URL http://www.root.cz/clanky/

rozsireni-instrukcni-sady-procesorovych-jader-s-otevrenou-architekturou-risc-v/#ic=serial-box&icc=text-title

[13] Vojnar, T.: Úvod. Studijní materiály k předmětu Operační systémy., 2015-02-10 [cit. 2016-05-04].

[14] Vojnar, T.: Správa paměti. Studijní materiály k předmětu Operační systémy., 2015-04-21 [cit. 2016-05-04].

[15] Waterman, A.; Lee, Y.; Avizienis, R.; aj.: The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.7. Technická Zpráva UCB/EECS-2015-49, EECS Department, University of California, Berkeley, May 2015.
URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-49.html

[16] Waterman, A.; Lee, Y.; Patterson, D. A.; aj.: The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0. Technická Zpráva UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.
URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

Příloha A

Obsah CD

Přiložené CD obsahuje následující adresáře:

- /doc - Obsahuje technickou zprávu ve formátu pdf.
- /model - Obsahuje zdrojové soubory implementace procesoru z první části práce.
- /plugins - Obsahuje implementované zásuvné moduly.
- /pk - Obsahuje zdrojové kódy upraveného proxy kernelu.
- /tex - Obsahuje zdrojové kódy technické zprávy.