FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Bakalářská práce

Návrh jádra procesoru architektury RISC-V v FPGA

Studijní program: B0613A140005 Informační technologie
Studijní obor: Inteligentní systémy

Autor práce: **Jaroslav Körner**
Vedoucí práce: Ing. Martin Rozkovec, Ph.D.
Ústav informačních technologií a elektroniky

Liberec 2023

FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Zadání bakalářské práce

Návrh jádra procesoru architektury RISC-V v FPGA

Jméno a příjmení:	**Jaroslav Körner**
Osobní číslo:	M20000041
Studijní program:	B0613A140005 Informační technologie
Specializace:	Inteligentní systémy
Zadávající katedra:	Ústav informačních technologií a elektroniky
Akademický rok:	2022/2023

Zásady pro vypracování:

1. Proveďte rešerši základních druhů architektur instrukčních souborů, seznamte se se specifikacemi ISA RISC-V včetně rozšíření
2. Seznamte se s FPGA Xilinx řady 7 a vývojovým prostředním Vitis
3. V jazyce VHDL navrhněte jádro procesoru splňující základní neprivilegovaný instrukční soubor
4. Funkci jádra ověřte pomocí simulace i demonstračního programu

Rozsah grafických prací:	dle potřeby dokumentace
Rozsah pracovní zprávy:	30-40 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	Čeština

Seznam odborné literatury:

[1] Harris, Sarah L., Harris, David: Digital Design and Computer Architecture, RISC-V Edition; Morgan Kaufmann, 2021, ISBN-13: 978-0128200643

[2] Waterman, A., Asanovic, K, SiFive Inc.: The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, v 20191213. Online, [https://bit.ly/3fNFNOk]

[3] Waterman, A., Asanovic, K, SiFive Inc.: The RISC-V Instruction Set Manual, Volume II: Privileged ISA, v 20211203. Online, [https://bit.ly/3MaacCP]

Vedoucí práce:	Ing. Martin Rozkovec, Ph.D. Ústav informačních technologií a elektroniky
Datum zadání práce:	24. října 2022
Předpokládaný termín odevzdání:	22. května 2023

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

prof. Ing. Ondřej Novák, CSc.
vedoucí ústavu

V Liberci dne 24. října 2022

Prohlášení

Prohlašuji, že svou bakalářskou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Jsem si vědom toho, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má bakalářská práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

22. května 2023 Jaroslav Körner

Návrh jádra procesoru architektury RISC-V v FPGA

Abstrakt

Tato bakalářská práce se zabývá návrhem jedno jádrového procesoru architektury RISC-V32I, tedy procesoru s 32 bitovou adresací paměti pracující nad datovým typem integer. Návrh byl omezen na neprivilegovaný instrukční soubor. Jádro procesoru je navrženo v jazyce VHDL. Tento návrh byl následně otestován pomocí simulace v prostředí Xilinx Vivado. Celková funkčnost je předvedena jednoduchým demonstračním programem spuštěným na desce Avnet ZedBoard.

Klíčová slova: bakalářská práce, 32-bitový mikroprocesor, architektura instrukčního souboru RISC-V32I, návrh hardwaru v jazyce VHDL, programovatelné hradlové pole

RISC-V architecture based processor core in an FPGA

Abstract

This bachelor thesis deals with a design of a single-core processor of the RISC-V32I architecture, i.e. a processor with 32-bit memory addressing and working over the integer data type. The design was limited to unprivileged instruction set. The processor core is designed in the VHDL language. This finished design has been tested using simulation in Xilinx Vivado environment. The overall functionality is demonstrated by a simple demonstration program running on the Avnet ZedBoard.

Keywords: bachelor thesis, 32-bit microprocessor, RISC-V32I instruction set architecture, hardware design in VHDL, field programmable gate array

Poděkování

Rád bych poděkoval mým učitelům ze střední školy, kteří mne zasvětili do zázraků a tajů číslicové techniky. Také pedagogickému kolektivu na ČVUT, kteří mi ukázali základy návrhu architektury procesorů typu MIPS v předmětu APO. Dále chci poděkovat Ing. Martinu Rozkovcovi, Ph.D., jenž se odhodlal být vedoucím mé bakalářské práce, za to, že za celou tu dobu nade mnou nezlomil hůl i když stále zůstávám v srdci programátor a nikoliv architekt.

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

Seznam zkratek

ALU	Aritmeticko-Logická Jednotka
ARM	Advanced RISC Machine / Acorn RISC Machine
AXI	Advanced eXtensible Interface
ce	clock enable
CPU	Central Processing Unit
CISC	Complex Instruction Set Computer
CU	Control Unit
DSP	digital signal procesing
FPGA	Field Programmable Gate Array
GNU	GNU's Not Unix!
HW	Hardware
IDE	integrated development environment
ISA	Instruction Set Architecture
LUT	look up table
MMAP	memory mapped device
NOP	no operation
PC	Program Counter
RISC	Reduced Instruction Set Computer
RISC-V	Reduced Instruction Set Computer - Version 5 (čti: risk pět) [1]
rd	Register Destination
rs1	Register Source 1
rs2	Register Source 2
SoC	system on chip
SW	Software
VHDL	Very High Speed Integrated Circuit Hardware Description Language
WSL	Windows Subsystem for Linux
XLEN	šířka registrů procesoru
XPM	Xilinx Parameterized Macros

1 Předmluva

Jistě se ptáte, co stálo za mým rozhodnutím zvolit si za téma mé bakalářské práce návrh vlastního procesoru?

Již na střední škole ve mne vzbudil velký zájem předmět číslicové technologie a vše s ním spojené. Když jsem poprvé za sebe zapojil čtyři čtyřbitové inkrementální čítače 7493 [2] s příslušnými moduly 24,10,6,10. Čtyři dekodéry na 7-segment 7449 a za ně odpory a displeje. Tuto hromadu švábů a propojovacích vodičů jsem napojil na generátor hodinového signálů (předpokládám značky Tesla), tak se před mým zrakem rozeběhly číslíčka primitivních digitálních hodin. V tu chvíli se mi rozzářily oči a já měl pocit, jako že jsem právě v tom okamžiku ovládl tranzistory a elektrický proud.

Netrvalo to dlouho a naskytla se mi příležitost rozebrat svůj první stolní počítač (těmito slovy myslím kamarádův). Vypojil jsem ho ze zásuvky a s křížovým šroubovákem v jedné ruce jsem hbitě postupoval skříní k jeho niternějším a niternějším útrobám. Nakonec již nezbylo nic, co bych mohl z šasi vyjmout. Rozprostíral se zde přede mnou na kuchyňském linoleu úplně nový svět. Svět plný prapodivných součástek a elektroniky, které bych si dříve nedovedl představit ani v těch nejdivočejších snech. Při kompletaci jsem pokračoval v opačném pořadí, než při demontáži a když má práce ustala, stál zde opět více méně, až na pár šroubků, ten původní počítač. Stačilo ho už jen zapnout. Mé srdce se rozbušilo. Spatřil jsem však ještě jedno tlačítko vzadu na zdroji počítače, které jsem řádně neprozkoumal. Věděl jsem, že musím stůj co stůj přijít na to, co se stane, když ho přepnu. Učinil jsem tak a ozvala se rána, jako když střelí z děla a z útrob počítače se vyvalil začernalý obláček dýmu.

V tu chvíli jsem měl jasno. Ten den na tom místě jsem se zapřisáhl, že neustanu ve svém bádání, dokud neodhalím všechna kouzla, čáry a taje, které v sobě počítačová skříň skrývá.

Tyto a další události vedly k tomu, že dnes stojím právě zde a jsem rozhodnut navrhnout si svůj vlastní procesor.

2 Úvod

Tato bakalářská práce se zabývá vytvořením návrhu procesoru RISC-V32I ve VHDL. Jedním ze základních úkolů této bakalářské práce je seznámení se podrobněji s architekturou procesorů RISC-V.

Cílem bylo popsat návrh procesoru s jedním jádrem, který je vybaven 32 bitovou adresní sběrnicí. Procesor disponující aritmetickologickou jednotkou, která umožňuje sčítání, odčítání a všechny základní operace booleovské algebry nad celými čísly. Procesor bude omezen pouze na neprivilegovaný instrukční soubor. Jednotlivé části návrhu procesoru budou otestovány simulací v prostředí Xilinx Vivado. Poté co bude procesor plně navržen, projde všemi simulacemi a úspěšnou syntézou, tak dojde k jeho nahrání na vývojovou desku Avnet ZedBoard. Po nahrání do FPGA na něm bude ověřena jeho funkčnost pomocí demonstračního programu, jenž otestuje procesor při ovládaní vstupních a výstupních periferií.

Kapitola 3 Důležité body teorie návrhu procesorů na straně 15 seznámí čtenáře se základním dělením architektur počítačů a procesorů.

Kapitola 4 s názvem RISC-V na straně 22 se zabývá obsahem anglické specifikace architektury RISC verze pět z roku 2019 dostupné na webu *riscv.org*, v souladu s kterou je tento procesor navržen. Kapitola je rozdělena do sekcí podle logického dělení instrukčního souboru.

Kapitola 5 Návrh procesoru je rozdělena do sekcí tak, aby jejich pořadí korespondovalo s postupnými kroky návrhu částí procesoru.

Kapitola 6 Programy čtenáře nejprve seznámí s krátkým úvodem pro programování v jazyce symbolických adres. Dále se věnuje překladu zdrojových kódu a jejich nahrání do navrženého procesoru.

Všechny soubory návrhu a programy jsou dostupné v repositáři GitLabu[1] na adrese: *https://gitlab.tul.cz/jaroslav.korner/RISC-V/*

[1]Pro přístup je nutné použít VPN do sítě TUL.

3 Důležité body teorie návrhu procesorů

3.1 Architektura procesoru

Teď již k samotnému návrhu procesoru, ale kde začít? Asi nejstěžejnějším rozhodnutím před tím, než člověk vůbec začne něco navrhovat, je rozhodnout se, jakou použije architekturu a instrukční sadu s ní spojenou. Za základní dělbu architektur lze považovat RISC a CISC.

3.1.1 CISC

Mezi známe zástupce rodiny CISC patří například: Motorola 6800, Intel 8080 (celá větev x86), Zilog Z80.

Hlavní myšlenkou této architektury je poskytnout programátorovi co největší množství instrukcí, ty mohou mít třeba i různou délku. Takový přístup měl nedocenitelný přínos pro programátory v nízko úrovňových jazycích. Základy této architektury byly však položeny v době, kdy takt procesorů neporážel vybavovací dobu operačních pamětí a tak nebyl problém provádět aritmetické operace nad daty uloženými přímo v paměti RAM. Tuto slabinu se dnes snaží dohnat asociativní paměť (cache).

Dnes mezi hlavní reprezentanty bez pochyby patří 32bitová x86 [3] od *Intelu* z něj odvozená architektura AMD64 [4], ke které *AMD* přidalo 64bitovou podporu.

Nevýhoda architektury x86 co se týče vlastního návrhu je v tom, že to Intel jen tak někomu nedovolí [5].

3.1.2 RISC

Z analýzy četnosti výskytu instrukcí provedené na půdě IBM vyšlo, že komplexní instrukce CISC procesorů se v kódu zas tak často nevyskytují. Tak v Berkeley přišli s konceptem architektury *RISC*. Jde o menší, lehčí a pružnější návrh, který se snažil začít s čistým štítem a osekat přebytečný křemík, tak aby vznikla vysoká účinnost na watt i za cenu menšího počtu provedených instrukcí na takt ve srovnání s architekturou CISC [6]. Navrhovaný procesor disponoval značným počtem registrů (běžně 32 u 32bitových procesorů) a pro přístup do paměti používal striktně instrukce **LOAD** a **STORE**. Další z klíčových technologií, na které RISC od začátku stavěl, bylo zřetězené zpracování instrukcí.

Prvním z této rodiny byl MIPS od *MIPS Technologies*, který vznikl na univerzitě ve Stanfordu [7] jako výukový projekt vedený Johnem Hennessey. Ač jsou i tyto procesory vyráběny, tak se na trhu zdaleka neuchytily natolik jako *ARM*.

Architektura procesorů ARM byla vytvořena na Univerzitě v Cambridge [8]. Nevýhoda procesorů ARM je pro nás v tom, že ten kdo je chce vyrábět či navrhovat musí zaplatit tučnou licenci [9].

Jednou z mladších větví je *RISC-V*, který vznikl na Univerzitě v Californii, Berkeley pod vedením Davida Pattersona. Tato architektura je dnes pod open source licencí. Tam, kde ARM licencuje svá jádra pro výrobu či modifikaci, tak RISC-V volně nabízí instrukční sady, které můžete použít k vývoji vašeho procesoru [10]. Této architektuře je věnována stále větší pozornost i na poli velkých hráčů jako jsou: *Intel*, *AMD* či *NASA*.

Spoustu nových projektů vzniká právě nad touto architekturou: nejrychlejší RISC-V procesor *zde*, první notebook s procesorem RISC-V *zde*, Raspberry Pi klon *zde*.

Právě z důvodu otevřené licence a aktuálnosti jsem se rozhodl použít pro svůj návrh procesoru architekturu RISC-V.

3.1.3 Současnost

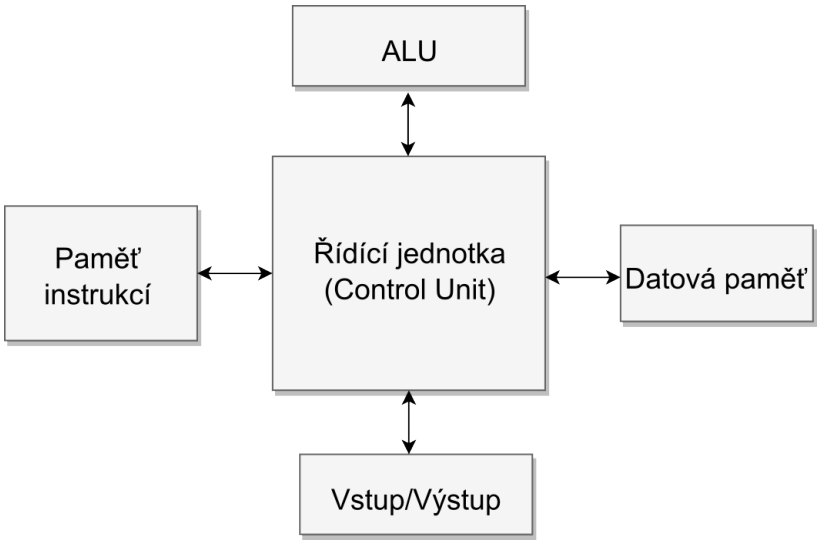
V dnešní době se rozdělení na CISC a RISC rozmazává. Procesory s architekturou CISC jsou vnitřně postavené na rozkladu instrukcí na mikrokód, který se svým vykonáváním již podobá operacím tak typickým pro architektury typu RISC. Naopak instrukční sada procesorů RISC je rozšiřována o instrukce (SIMD) pro práci s vektory a multimediální operace, které svými vlastnostmi splňují charakteristiku přístupu CISC.

3.2 Schéma počítače

Na procesory lze také nahlížet z pohledu jak pracují s pamětí. Existují dva směry přístupu, Von Neumanovo a Harvardské schéma počítače.

3.2.1 Harvardské schéma počítače

Harvardská architektura byla navržena Howardem Aikenem na Harvardské univerzitě ve třicátých letech dvacátého století. Na této architektuře byl postaven například počítač ENIAC. Vyznačuje se tím, že má oddělenou instrukční a datovou paměť [11].



Obrázek 3.1: Harvardské schéma (převzato z [12])

Mezi výhody, které tento přístup přináší, lze zařadit například:

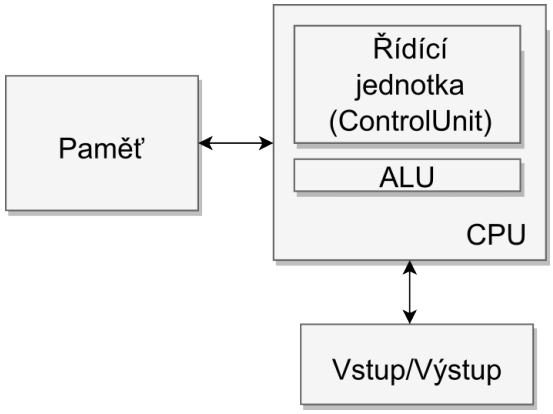
- Větší propustnost při komunikaci (najednou lze načítat z paměti jak data, tak i příští instrukci).
- Paměť může využívat různé technologie (instrukční PROM, datová RAM).
- Instrukční paměť je chráněna proti nechtěnému zápisu.

Mezi nevýhody patří:

- Nemožnost změnit program za běhu.
- Architektura může mít větší požadavky na paměť oproti Von Neumannově.
- Architektura má zvýšenou komplexitu návrhu přidanou sběrnicí. [13]

3.2.2 Von Neumanovo schéma počítače

Svůj název schéma získalo podle autora přednášky „First Draft of a Report on the EDVAC“ Johna von Neumanna [14]. Jde o jednoduché schéma počítače, kdy jsou instrukce procesoru i data ke zpracování uložena v jedné paměti přistupované po společné sběrnici. Nelze tedy najednou načítat jak program tak data a sběrnice do operační paměti se stává úzkým místem. [15]



Obrázek 3.2: Von Neumannovo schéma (převzato z [12])

Návrh popisoval počítač skládající se z těchto částí:

- **Operační paměť** ukládá všechna data,
- **ALU** provádí aritmetické a logické operace,
- **Řídící jednotka** která určuje co se má v danou chvíli vykonávat,
- **Vstupní zařízení** slouží pro komunikaci s vnějším světem,
- **Výstupní zařízení** slouží pro komunikaci s vnějším světem.

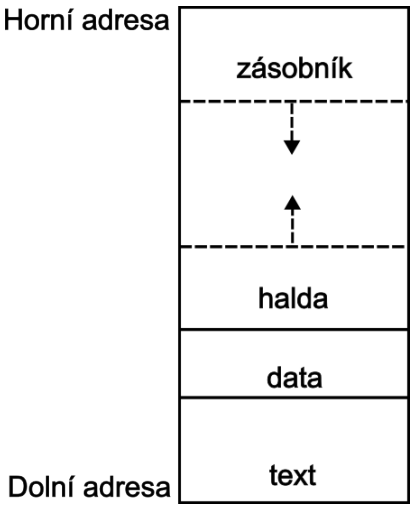
3.2.3 Současnost

V praxi mají počítače mezi procesorem a operační paměti ještě mezistupeň, kterým je asociativní paměť (cache, i ta je dnes víceúrovňová). Přístup do paměti je násobně pomalejší než vykonávání základních aritmetických operací na procesoru a tak do hry vstoupila výše zmíněná asociativní paměť, malá a rychlá paměť, která umožňuje zrychlení opakovaného přístup na stejná data či instrukce (ty jsou vícekrát po sobě načítány zejména v cyklech). Asociativní paměť dnes bývá rozdělena na instrukční a datovou. Tak vzniká hybridní návrh. Procesor může nezávisle přistupovat ke svým instrukcím a datům (dokud jsou načtené v asociativní paměti), zatímco z procesoru ven vede do operační paměti jen jedna datová sběrnice [16].

3.3 Organizace paměti programu

Operační paměť v procesoru lze z pohledu spuštěné aplikace rozdělit na tyto části:

- **Zásobník** obsahuje lokální proměnné funkcí.
- **Halda** dynamicky[1] alokovaná paměť. Když data uložená v alokovaném prostoru již nejsou potřeba, tak tuto paměť musíme před dalším použitím uvolnit (dealokovat).
- **Data** je sekce ve kterém jsou uložené globální konstanty programu.
- **Text** začíná od adresy 0, roste směrem nahoru. Je zde uložený program. Při spuštění počítače se čítač programu nastaví na nulu a program se začne vykonávat z této adresy, tedy v sekci text.



Obrázek 3.3: Segmenty paměti (překresleno podle [17])

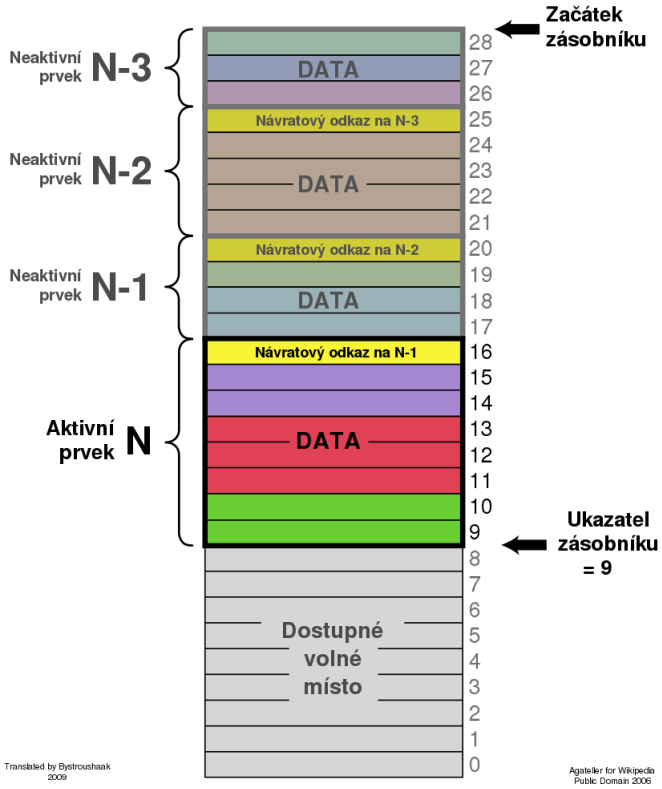
Na obrázku 3.3 je vyobrazeno jak jsou tyto segmenty paměti poskládány. Na nejnižší adrese začíná segmentem text, následují ji data a halda se zásobníkem, které se opticky dělí o jeden společný prostor. [17]

3.3.1 Zásobník

Segment zásobník (stack) začíná na konci adresního prostoru a roste ze shora dolů. Zásobník obsahuje jednotlivé rámce. Ukazatel na vrchol rámce se standardně ukládá do jednoho z registrů procesoru. Při volání funkce se vytvoří nový rámec o velikosti potřebné pro argumenty a lokální proměnné dané funkce (hodnota v registru sp se aktualizuje).

[1] Název je přejat od slova dynamit, u toho taky nikdy nevíte jak velká ta díra bude dokud to nevybouchne.

Tento paměťový prostor bývá standardně pro aplikace omezen a tak se velké datové struktury alokují na haldu (heap). Nejkrajnějším omezením zásobníku může být hranice oblasti **text** ve které je uložen program. Další zápis do zásobníku by vedl k tomu, že by se přepsal spouštěný program a aplikace by již nemohla zaručit své správné vykonání. Tento problém se při detekování běžně označuje za „stack overflow“. [18]



Obrázek 3.4: Zásobník (převzato z [19])

Na obrázku 3.4 je vyobrazená sekce zásobník. Velké „N“ značí lokální kontext běžící funkce, „N-1“ je předchozí rámec, ze kterého byla tato funkce zavolána. Řádky značí jednotlivé proměněné délky výchozího slova systému. První řádek rámce obsahuje návratovou adresu do programu předchozí funkce.

3.4 Jazyk pro popis hardware

Jazyk VHDL (Very High Speed Integrated Circuit Hardware Description Language) poskytuje návrhářům hardwaru vyšší úroveň abstrakce než je tomu v metodologii návrhu přes booleovské rovnice a pravdivostní tabulky, ze kterých vzniká schema obvodu, nebo intuitivní schématický návrh s použitím obvodů SSI. To umožňuje díky tomu že vznikl právě za účelem popisu chování integrovaných obvodů. Za tímto jazykem popisující hardware stojí Ministerstvo obrany USA a stal se standardem IEEE. Jeho syntaxe vychází z jazyka ADA. Jde o jazyk silně typový, který vyžaduje použití konverzních funkcí pokaždé když se mění typ signálu.

Specifikace RISC-V slovně a za pomoci tabulek s kódy operací a vzory instrukcí formalizuje vlastnosti architektury. Díky vlastnostem jazyka VHDL je možné přeložit tuto textovou předlohu do kódu popisujících návrh procesoru.

3.4.1 Strukturní vs behaviorální popis

Čistě behaviorální popis může krok za krokem popisovat co se má vykonávat při jednotlivých instrukcích. Takovýto postup může být pro člověka čitelnější a jednoduší na tvorbu, jelikož přímo koresponduje s formulaci významu instrukcí ve specifikaci a veškerá zákoutí návrhu potřebná pro pochopení funkce daných instrukcí jsou na jednom místě v programu.

Nevýhoda takového návrhu je však v menší modularitě, ze které plyne náročnější testování a simulace (test bench). Do budoucna je v plánu rozšířit návrh procesoru RISC-V o zřetězené zpracování instrukcí. To by však bylo v takovém přístupu velmi náročné.

Zvolil jsem si proto cestu kombinace strukturního a behaviorální popisu. Práce je tak rozdělena do souborů podle jednotlivých logických celků. To umožňuje opakovaného použití HW (entit) na více místech v návrhu. Testování těchto dílčích částí je teoreticky mnohem snazší, protože například ALU nebo dekodér instrukcí je jedním z bloků návrhu. ALU je tedy možné instancovat zvlášť a otestovat ji podle specifikace chování aritmetiky celých čísel (integer) pro sčítání a odčítání, podobně otestujeme i funkce booleovské algebry. Tyto nezávislé (unit) testy poskytují návrhářovi důvěru v jednotlivé kusy návrh zatímco z nich skládá větší funkční celky.

4 RISC-V

Při návrhu procesoru jsem postupoval podle anglické specifikace a tak většina informacím k architektuře procesoru (jako je například význam a formát instrukcí, adresování paměti, skoky v programu a další) je překladem specifikace RISC-V a to verze *riscv-spec-20191213* [20].

Ta určuje několik RISC-V ISA základu, které operují nad datovým typem integer, ale liší se šířkou registrů a datové sběrnice. Přehled je znázorněn v tabulce 4.1.

Tabulka 4.1: Základní ISA RISC-V

Rozšíření	**Bitovost**	**Datový typ**
RV32I	32 bitů	integer
RV64I	64 bitů	integer
RV128I	128 bitů	integer

Jsou zde také definovaný i rozšíření ISA, tabulka 4.2, která lze libovolně kombinovat, podle požadovaných vlastností výsledného procesoru.

Tabulka 4.2: Rozšíření ISA

Rozšíření	**Anglicky**	**Česky**
M	Multiplication and Division	Násobení a dělení
A	Atomic instructions	Atomické instrukce
F	Single-Precision Floating-Point	Floating-Point základní přesnost
D	Double-Precision Floating-Point	Floating-Point dvojitá přesnost
Q	Quad-Precision Floating-Point	Floating-Point čtyřnásobná přesnost
C	Compressed	Komprimovaná (16-bitová)

4.1 RV32I

Tento procesor implementuje instrukční sadu RISC-V32I. Jde o nejzákladnější podmnožinu operací na 32bitovém celočíselném procesoru RISC-V. Má 40 instrukcí (jejichž přehled je uveden v na obrázku 4.1), ale při jednodušší implementaci lze popsat

instrukci ECALL/EBREAK[1] pouze jedinou instrukcí SYSTEM. Instrukci FENCE[2] lze implementovat jako instrukci NOP, za cenu ztráty podpory více vláknových aplikací. Tím se sníží rozsah instrukční sady na pouhých 38 instrukcí. Programově lze na této ISA emulavot všechny ostatní rozšíření kromě A (atomických instrukcí).

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Obrázek 4.1: Instrukce ISA RV32I (převzato z [20])

[1]Instrukce pro exception-call, tedy softwarová přerušení.
[2]Synchronizace zápisu do paměti a propsání z asociativní paměti do RAM.

4.2 Registry

V tabulce 4.3 jsou popsány všechny adresovatelné registry základní neprivilegované ISA. Architektura RV32I má 32 celočíselných registrů, každý 32 bitů široký, tedy XLEN=32[3]. Registr x0 je pevně zapojen všemi svými bity na 0. Obecné registry x1–x31 obsahují hodnoty, které různé instrukce interpretují jako:

- kolekci booleovských hodnot,
- dvojkový doplněk (znaménková binární celá čísla),
- bezznaménková binární celá čísla.

Tabulka 4.3: Registry a jejich symbolických pojmenování (převzato z [21])

Název registru	**Symbolický název**	**Popis**	**Ukládá**
x0	zero	Vždy nula	
x1	ra	Návratová adresa	Volající
x2	sp	Ukazatel na stack	Volaný
x3	gp	Globální ukazatel	
x4	tp	Ukazatel na vlákno	
x5	t0	Dočasný / alternativní návratová adresa	Volající
x6–7	t1–2	Dočasný	Volající
x8	s0/fp	Ukládaný / ukazatel rámce	Volaný
x9	s1	Ukládaný	Volaný
x10–11	a0–1	Argument funkce / návratová hodnota	Volající
x12–17	a2–7	Argument funkce	Volající
x18–27	s2–11	Ukládaný	Volaný
x28–31	t3–6	Dočasný	Volající

ISA nespecifikuje který registr musí být použit jako sp[4] nebo ra[5]. Mezi osvědčené postupy však patří využívat registr x1 jako návratovou adresu (alternativně x5). Stejně tak registr x2 běžně slouží jako ukazatel na zásobník. Tabulka 4.3 obsahuje přehled symbolických názvů registrů. Lze je použít při programování v jazyce symbolických adres (assembler), pro tvorbu přehlednějšího kódu a kvalitní překladač provede jejich přeložení na index registru. Tabulka 4.3 též obsahuje sloupeček „Ukládá“, který značí kdo je zodpovědný za ukládání obsahuje registrů na zásobník a jejich obnovení, před návratem z funkce.

[3] Terním XLEN značí šířku integerového registru.
[4] sp - stack pointer, ukazatel na zásobník
[5] ra - return address, návratová adresa z funkce

ISA definuje ještě jeden registr pc[6], který obsahuje adresu aktuální instrukce, zobrazený v tabulce 4.4.

Tabulka 4.4: Registr čítače programu (převzato z [20])

XLEN-1 0
pc
XLEN

4.3 Kódování instrukcí

ISA RV32I specifikuje čtyři základní instrukční formáty (R/I/S/U), tabulka 4.5. Všechny instrukce mají fixní délku 32 bitů a musí být zarovnány na 4 bajty. Pokud při skocích v programu adresa následující instrukce není zarovnána na 4 bajty, IALIGN=32 (bitů)[7], je generovaná výjimka „instruction-address-misaligned“. Při dekódování instrukce jejíž opcode nedává smysl je vyvolána výjimka „illegal-instruction“.

Tabulka 4.5: Formát kódování instrukcí (převzato z [20])

31 25	24 20	19 15	14 12	11 7	6 0	
func7	rs2	rs1	func3	rd	opcode	R-typ
imm[11:0]		rs1	func3	rd	opcode	I-typ
imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode	S-typ
imm[31:12]				rd	opcode	U-typ

Jednotlivé typy instrukcí v tabulce 4.5 nesou své jméno podle:

- `R-typ` je podle anglického slova **register** (registr),
- `I-typ` je od slova **immediate** (okamžitý),
- `S-typ` je od slova **store** (ukládat),
- `U-typ` je od slova **upper** (horní) a používá se u instrukcí s 20 bitovými konstantami.

[6] pc - program counter, neboli čítač programu

[7] Termín IALIGN (měřeno v bitech) používáme k odkazu na omezení zarovnání instrukce adresy, které implementace vynucuje. IALIGN je 32 bitů v základním ISA.

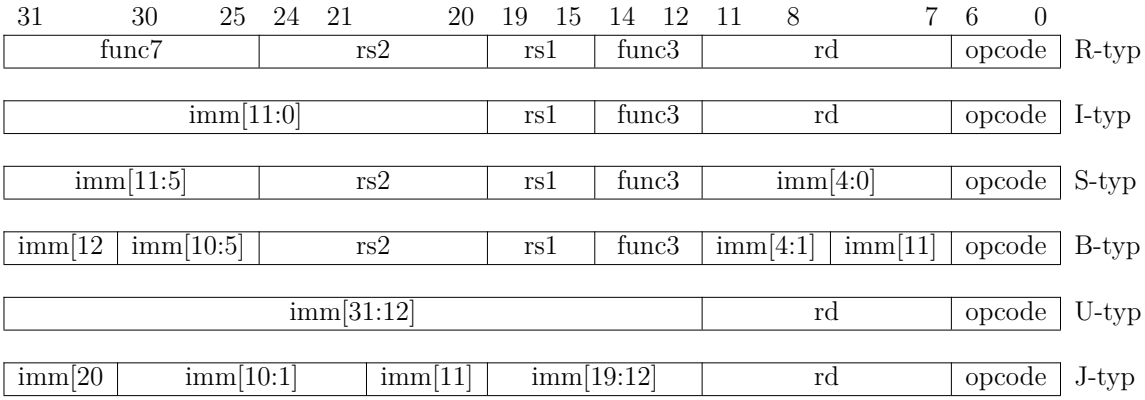
Z tabulky 4.5 je dále patrné, že instrukce ihned (immediate) se vykonávají s jedním registrem (rs) a hodnotou uloženou přímo v kódu programu (konstantou), výsledek se uloží do cílového registru (rd). Naproti tomu registrové instrukce mají dva operandy uložené ve zdrojových registrech rs1 a rs2, výsledek operace se opět ukládá do registru rd. Pozice operandů rs1, rs2 a rd se napříč formáty neliší, což umožňuje jednoduché dekódování instrukcí.

Immediate operandy jsou vždy nejprve znaménkově rozšířeny na délku 32bitů. V tabulce 4.5 je imm argument uveden s rozsahem bitů v hranatých závorkách. Například imm[11:0] znamená, že se jedná o 12 bitovou hodnotu, která je znaménkově rozšířena na 32 bitů. Znaménkové rozšíření je jedna z nejdůležitějších operací na okamžitých instrukcích. V RISC-V je znaménkový bit pro všechny okamžité hodnoty vždy uložen v 31 bitu instrukce, aby znaménkové rozšíření mohlo probíhat paralelně s dekódováním instrukce.

4.3.1 Kódování okamžitých instrukcí

Základní čtyři formáty kódování instrukcí lze rozšířit ještě o formáty (B/J) podle toho jak pracují s okamžitou hodnotou, tabulka 4.6.

Tabulka 4.6: Formát kódování immediate operací (převzato z [20])

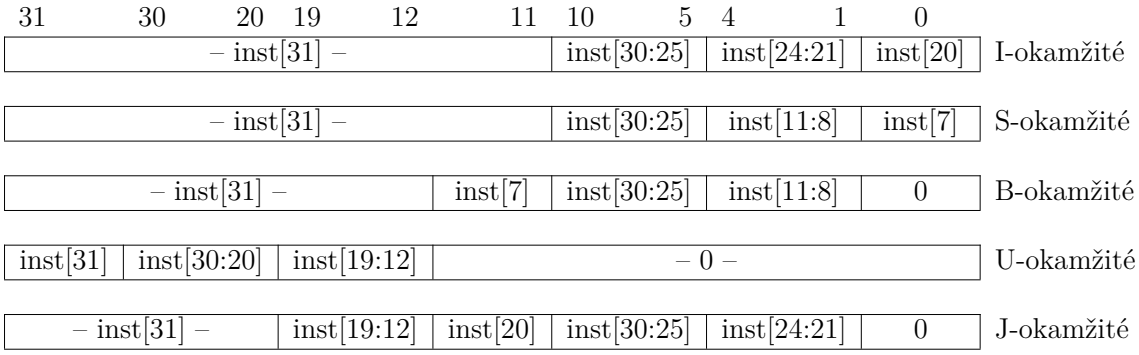


Jediný rozdíl mezi formáty S a B je v tom, že 12bitové pole bitů okamžité hodnoty se používá k zakódování větvení v násobcích 2 v B formátu. Místo toho, aby se všechny bity v okamžitém kódování instrukce posunuly vlevo o jeden bit v hardwaru, jak je obvyklé, zůstávají střední bity (imm[10:1]) a znaménkový bit na pevných pozicích, zatímco nejnižší bit ve formátu S (inst[7]) kóduje bit vyššího řádu ve formátu B.

Podobně jediný rozdíl mezi formáty U a J je v tom, že 20bitové pole bitů okamžité hodnoty se posune vlevo o 12 bitů, aby vytvořilo konstantu u typu U a o 1 bit, aby vytvořilo konstantu typu J. Umístění instrukčních bitů okamžité hodnoty v U a J formátech je vybráno tak, aby se maximalizoval překryv s ostatními formáty a mezi sebou.

Tabulka 4.7 ukazuje okamžité hodnoty vytvořené ze všech základních instrukčních formátů a je označena tak, aby ukázala, který bit instrukce (inst[y]) produkuje jaký bit okamžité hodnoty. Například okamžitá hodnota pro formát I je vytvořena z inst[31:20], zatímco okamžitá hodnota pro formát S je vytvořena z inst[31:25] a inst[11:7].

Tabulka 4.7: Znaménkové rozšíření immediate instrukcí (převzato z [20])



4.4 Instrukce celočíselných operací

Základní aritmeticko-logické operace jsou rozdělené do dvou prakticky totožných skupin: registrových (registr) a bezprostředních/okamžitých (immediate).

Bezprostřední operace (immediate) jsou operace, které provádíme nad jedním registrem a konstantou (register-immediate), a jsou zakódovány pomocí formátu typu I. Patří sem například instrukce: `ADDI`, `SLTI`, `SLTIU`, `XORI`, `ORI`, `ANDI`, `SLLI`, `SRLI`, `SRAI`.

Tabulka 4.8: Registrové operace (převzato z [20])

31 25	24 20	19 15	14 12	11 7	6 0
func7	rs2	rs1	func3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Registrové operace (register) jsou operace, které provádíme nad dvěma registry (register-register), a jsou zakódovány pomocí formátu typu R, tabulka 4.8. Patří sem například instrukce: `ADD`, `SUB`, `SLL`, `SLT`, `SLTU`, `XOR`, `SRL`, `SRA`, `OR`, `AND`.

0perace načítají operandy z registrů rs1 a rs2 a zápis výsledku do registru rd. Položky funct7 a funct3 vybírají typ operace, bližší přehled hodnot které nabývají naleznete na obrázku 4.1.

`ADD[I]`[8] přičte rs2, nebo ve variantě I znaménkově rozšířenou (sign-extended) 12bitovou konstantu k hodnotě v registru rs1. Přetečení je ignorováno a výsledek (dolních XLEN bitů) je uložen do registru rd.

`SUB` provede odečtení hodnoty v registru rs2 od hodnoty v registru rs2 (rs1 - rs2). Přetečení je ignorováno a výsledek (dolních XLEN bitů) je uložen do registru rd.

`SLT[I]` (znaménkové) a `SLT[I]U`[9] [10] (bezznaménkové) porovnání hodnoty (set less than immediate). Pokud hodnota v registru rs1 $<$ rs2 (nebo imm), tak se do registru rd zapíše 1, jinak 0.

`AND[I]`, `OR[I]`, a `XOR[I]` provádí bitové logické operace.

`SLL[I]` je logický posun doleva (nuly jsou posunuty do nižších bitů).

`SRL[I]` je logický posun doprava (nuly jsou posunuty do horních bitů).

`SRA[I]` je aritmetický posun doprava (původní znaménkový bit je zkopírován do vyprázdněných horních bitů).

4.4.1 Přetečení při aritmetických operacích

Součástí základní sady instrukcí není speciální podpora instrukcí pro kontrolu přetečení. Pro obecný znaménkový součet jsou po součtu vyžadovány tři další instrukce, využívající pozorování, že součet by měl být menší než jeden z operandů, pokud je druhý operand záporný, ukázka kódu 1.

```
add t0, t1, t2
slti t3, t2, 0
slt t4, t0, t1
bne t3, t4, overflow
```

Listing 1: Kontrola přetečení u obecného znaménkového součtu

[8] `ADDI rd, rs1, 0` je použito pro přesun hodnoty z registru rs1 do registru rd (assemblerovská pseudoinstrukce `MV rd, rs1`).

[9] `SLTU rd, x0, rs2`, nastaví rd na 1, pokud rs2 není rovno nule, jinak nastaví rd na nulu (assemblerovská pseudoinstrukce `SLTZ rd, rs`).

[10] `SLTIU rd, rs1, 1` nastaví rd na 1, pokud rs1 je rovno nule, jinak nastaví rd na nulu (assemblerovská pseudoinstrukce `SEQZ rd, rs`).

4.4.2 NOP

> Nedělám nic, ale dělám to dobře.
>
> (True — Linux manual page [22])

Instrukce NOP nezmění žádný architektonicky viditelný stav, kromě posunu pc a inkrementace všech příslušných počítadel výkonu. Jde o takzvanou pseudoinstrukci a pro její zakódování použito například `ADDI x0, x0, 0`.

4.5 Instrukce pro práci s pamětí

Procesor disponuje těmito instrukce pro načítání a ukládání hodnot: LUI, LB, LH, LW, LBU, LHU, SB, SH, SW.

RISC-V má pro všechny přístupy do paměti jediný bajtově adresovatelný adresní prostor velikosti 2^{XLEN} bajtů. Paměť lze rozdělit do elementárních celků:

- **halfword** (půl slovo) je 16 bitů (2 bajty),
- **word** (slovo) paměti je definováno jako 32 bitů (4 bajty),
- **doubleword** (dvojslovo) je 64 bitů (8 bajtů),
- **quadword** (čtyř slovo) je 128 bitů (16 bajtů).

Instrukce přístupu do paměti podrobněji:

- LW načte 32-bitovou hodnotu z paměti do registru rd.
- LH načte 16-bitovou hodnotu z paměti, poté ji rozšíří na 32 bitů a uloží do registru rd.
- LHU (Load halfword unsigned) načte 16-bitovou hodnotu z paměti, poté ji rozšíří na 32 bitů nulami a uloží do registru rd.
- LB a LBU (Load byte unsigned) jsou analogicky definovány pro 8-bitové hodnoty.
- SW, SH a SB uloží 32-bitové, 16-bitové a 8-bitové hodnoty z nižších bitů registru rs2 do paměti.

LUI (load upper immediate) nepracuje přímo s pamětí, ale používá se k vytvoření 32-bitových konstant a využívá formát U. LUI umístí hodnotu U-okamžitá do horních 20 bitů cílového registru rd, jehož nejnižších 12 bitů vyplní nulami.

4.6 Čítač programu a instrukce skoku

RV32I poskytuje dva typy instrukcí pro předání řízení:

- nepodmíněné skoky,
- podmíněné skoky.

Instrukce pro práci s registrem čítače instrukcí: `AUIPC`, `JAL`, `JALR`, `BEQ`, `BNE`, `BLT`, `BGE`, `BLTU`, `BGEU`.

4.6.1 Nepodmíněné skoky

Všechny nepodmíněné skoky používají adresování relativní k registru pc.

`AUIPC` (add upper immediate to pc) se používá k vytvoření adresy relativní k registru pc a používá formát U. `AUIPC` vytvoří 32-bitový offset z 20-bitové U-okamžité hodnoty, kde se vyplní nejnižší 12 bitů nulami. Pak se přičte tento offset k adrese instrukce `AUIPC` a výsledek uloží do registru rd.

`JAL`[11] (jump and link) instrukce používá formát J, kde J-okamžitá hodnota zakódovává znaménkový offset v násobcích 2 bajtů. Offset je rozšířen na 32 bitů a přičten k adrese instrukce `JAL`. Výsledná adresa je cílem skoku. `JAL` uloží adresu instrukce následující po skoku (pc+4) do registru rd. Standardní softwarová konvence volání používá x1 jako registr pro návratovou adresu.

`JALR` (jump and link register) je instrukce skoku na adresu z registru, která používá kódování typu I. Cílová adresa se získá přičtením znaménkově rozšířeného 12bitové I-okamžité hodnoty k registru rs1, poté se nejnižší bit výsledku nastaví na nulu. Adresa instrukce následující po skoku (pc+4) se zapíše do registru rd. Registr x0 lze použít jako cíl, pokud výsledek není vyžadován.

Když je instrukce `JALR` použita s bází rs1=x0, může být použita k implementaci jedno instrukčního volání podprogramu z nejnižších 2 KiB nebo nejvyšších 2 KiB adresního prostoru odkudkoli z programu, což lze použít k implementaci rychlého volání malých funkcí.

Instrukce `JALR` byla definována tak, aby umožnila dvou instrukční sekvenci, která skočí kamkoli v 32bitovém absolutním adresním rozsahu. Instrukce `LUI` nejprve může načíst rs1 s horními 20 bity cílové adresy, poté `JALR` může přidat dolní bity. Podobně instrukce `AUIPC` následovaná `JALR` může skočit kamkoli v 32bitovém relativním adresním rozsahu.

`JAL` a `JALR` instrukce vygenerují výjimku (instruction-address-misaligned), pokud cílová adresa není zarovnána na hranici čtyřbajtového bloku.

[11] Čistě nepodmíněné skoky (pseudoinstrukce assembleru J) jsou zakódovány jako `JAL` s rd = x0.

4.6.2 Podmíněné skoky

Všechny instrukce větvení používají formát instrukce B. 12-bitová B-okamžitá hodnota obsahuje znaménkový posun v násobcích 2 bajtů. Posun je znaménkově rozšířen a přičten k adrese instrukce větvení, aby se získala cílová adresa. Rozsah podmíněného větvení je ±4 KiB.

Instrukce větvení porovnávají dva registry. Jejich výčet je následující:

- `BEQ` větví kód pokud registry rs1 a rs2 mají stejnou hodnotu.
- `BNE` větví kód pokud registry rs1 a rs2 mají různou hodnotu.
- `BLT` a `BLTU` větví kód, pokud je rs1 menší než rs2, používají se příslušná znaménková porovnání.
- `BGE` a `BGEU` větví kód, pokud je rs1 větší nebo rovno rs2, používají se příslušná znaménková porovnání[12].

Podmíněné skoky vygenerují výjimku (instruction-address-misaligned), pokud cílová adresa není zarovnána na hranici čtyřbajtového bloku a podmínka skoku je splněna. Pokud podmínka skoku není splněna, výjimka se nevyvolá.

[12] `BGT`, `BGTU`, `BLE` a `BLEU` lze syntetizovat prohazováním operandů instrukcí `BLT`, `BLTU`, `BGE` a `BGEU`.

5 Návrh procesoru

Pro procesor jsem založil nový projekt v IDE Vivado. Všechny soubory jsou psány v revizi jazyka VHDL z roku 2008. Procesor se měl skládat z několika menších bloků návrhu a bylo potřeba mezi nimi sdílet definice typů, konstant a funkcí. Za tímto účelem jsem vytvořil balíček `JKRiscV_types`. Ten je do dílčích souborů vkládán příkazem: `use work.JKRiscV_types.all;`

Výhoda tohoto přístupu je v tom, že definice všech konstant popisují architekturu procesoru (například XLEN) jsou na jednom místě.

5.1 Balíček JKRiscV_types

Balíček `JKRiscV_types` definuje konstanty potřebné k návrhu procesoru. Patří mezi ně třeba definice `true` a `false`.

True (pravda) je v mé implementaci RISC-V32i reprezentována jako 1 (tedy 31x'0' & '1'). False (lež) je reprezentována jako 0 (tedy 32x'0').

Balíček `JKRiscV_types` definuje proměnné:

- `JKRiscV_true` jako `to_signed(1, 2)`, tedy: „01“
- `JKRiscV_false` jako `to_signed(0, 2)`, tedy: „00“

Při použití ve VHDL kódu je potřeba tyto konstanty znaménkově rozšířit na požadovanou délku výsledku příkazem:

```
result_signed := resize(JKRiscV_true, result_signed'length)
```

Výhoda toho zápisu je v tom, že popisuje konstanty `true` a `false` pro všechny ISA základy bez ohledu na to jak mají dlouhé registry.

V balíčku `JKRiscV_types` jsou též definovány výčtové datové typy (jejich definice je uvedena v kódu 2 pro:

- stav procesoru, blíže v kapitole 5.7,
- `opcode`, který určuje kód operace, podrobnosti jsou uvedeny v kapitole 4.1 na obrázku 4.1,
- `funct_3`, který slouží pro jemnější rozdělení jednotlivých operací a dále se dělí podle typu na `ALU`, paměť (`memory`) a operace skoku (`branch`),
- `alufunc` slouží pro nastavení funkcí, jež bude vykonávat ALU.

```
type t_ENCODING is (R_TYPE, I_TYPE, S_TYPE, B_TYPE, U_TYPE, J_TYPE,
↪  ERROR);

type t_OPCODE is (LUI_INST, AUIPC_INST, JAL_INST, JALR_INST,
↪  FENCE_INST, ECALL_BREAK_INST, BRANCH, LOAD, STORE, IMMEDIATE,
↪  REGISTR, ERROR);

type t_FUNCT_3_ALU is (ARIT_O, SLT_O, SLTU_O, XOR_O, OR_O, AND_O,
↪  SLL_O, SR_O, ERR_O);

type t_FUNCT_3_MEMORY is (BYTE_O, HALF_O, WORD_O, UBYTE_O, UHALF_O,
↪  ERR_O);

type t_FUNCT_3_BRANCH is (BEQ_O, BNE_O, BLT_O, BGE_O, BLTU_O,
↪  BGEU_O, ERR_O);

type t_ALUFUNC is (ADD_F, SUB_F, SLL_F, SLT_F, SLTU_F, XOR_F, SRL_F,
↪  SRA_F, OR_F, AND_F, ERR_F);
```

Listing 2: Přehled výčtových typů

Balíček dále předepisuje funkce potřebné pro práci procesoru, jako jsou konverze z datových typů `std_logic_vector` na příslušný výčtový typ a naopak. Jednoduchý enkodér pro tvorbu instrukcí vhodný pro testování částečného návrhu procesoru. Balíček implementuje také umístěné funkce generující řídící signály procesoru. Jejich společnou vlastností je že za argument přijímají `opcode` a další signály. Řídící signály určují zda se bude zapisovat do registru, do kterých bajtů paměti se bude ukládat hodnota, nebo zda se jedná o validní adresu při přístupu do paměti.

5.2 Dekodér instrukcí

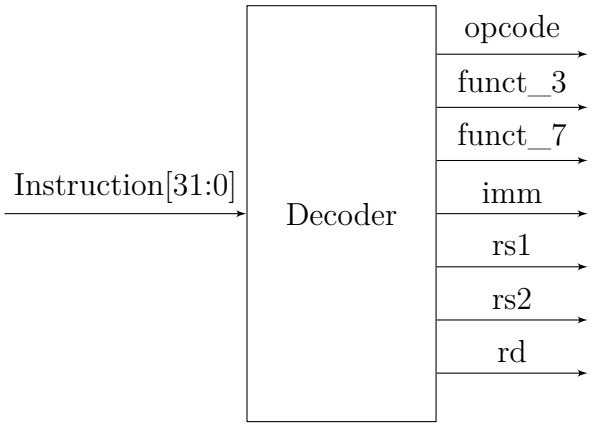
Pro zvýšení čitelnosti kódu dekodér instrukcí nemá na svém výstupu pouze rozkouskovanou instrukci do příslušných signálů, ale překládá je na výčtový datový typ. Díky tomu je dále v kódu možné referovat k jednotlivým instrukcím slovním názvem a nikoli pouze posloupností nul a jedniček. Schéma dekodéru se nachází v příloze A.4, dekodér je popsán v souboru `instruction_parser.vhd`.

V dekodéru se rozšiřuje signál `imm`[1] na délku 32 bitů. To z jakých bitů signálu a zda znaménkově nebo o doplnění nulami určuje typ instrukce (`t_encoding` výňatek z kódu 2), blíže v kapitole 4.3.

Vznikají zde signály `opcode`, `funct_3`[2], `funct_7`, `rd`, `rs1`, `rs2`. Jejich umístění v instrukci a hodnoty odpovídají přehledu instrukcí v obrázku 4.1. Zjednodušená

[1] Zkratka imm z anglického immediate (okamžitý).

[2] Písmeno „O“ na konci zástupců výčtového typu `funct_3` značí zkratku slova operace.



Obrázek 5.1: Blokové schéma dekodéru instrukcí

reprezentace dekodéru je vyobrazena v blokovém schéma 5.1.

Dekodér se skládá z 49 LUTů.

5.3 Registry procesoru

Registry procesoru jsou realizované jako pole délky 32 bázového typu `std_logic_vector(C_DATA_WIDTH - 1 downto 0)`. Registr na indexu 0 je trvale připojen na nulu. Registrové pole má latenci přístupu 1 takt a jeho výstupy nevedou přes další registr. Na výstupu z registrového pole je pro každou adresu (rs1, rs2) jeden velký multiplexer (proto tento návrh také spotřebová značně velké množství F7 a F8 multiplexerů). Entita je popsána v souboru `registers.vhd`.

Tabulka 5.1: Využití prostředku FPGA pro registry

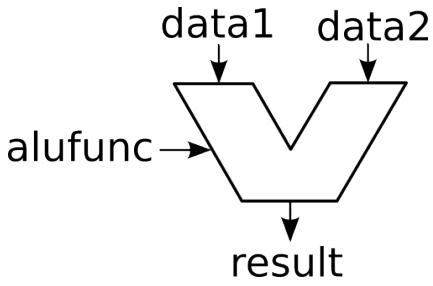
Název	**Slice LUTs**	**Slice Registers**	**F7 Muxes**	**F8 Muxes**
registry	749	992	256	70

Povšimněte si že sloupce v tabulce 5.1 sloupec registry přesně odpovídá 31 (jeden registr je připojený na nuly) registrům po 32 bitech, $32 * 31 = 992$.

Registry bylo možné navrhnout jako dvojici DP BRAM (dvou portová bloková paměť), ale registrové pole je poměrně malé a tak jsem zvolil implementaci ze slice registrů (kterých je na technologii FPGA více než dost).

5.4 Aritmeticko-logická jednotka

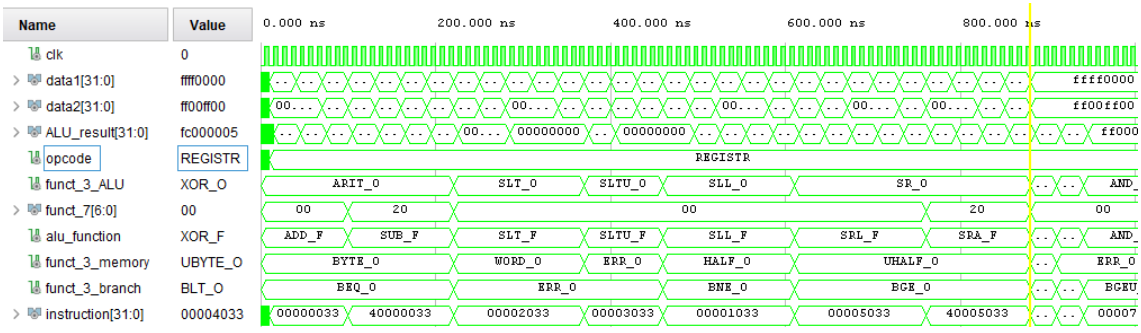
ALU je tvořena sekvenčním příkaz case, který vede na paralelní výpočet jednotlivých funkcí, jejichž výsledek se vybírá na multiplexeru podle `opcode`. Entita ALU v souboru `ALU.vhd` má nepovinný generický parametr `C_OUTPUT_REG`, jenž slouží k výběru zda se má na výstup z jednotky připojit registr. V obrázku 5.2 je znázorněno blokové schéma ALU s připojenými signály.



Obrázek 5.2: Blokové shéma ALU (převzato z [23])

5.4.1 Testování aritmetiko logické jednotky

V souboru `tb_ALU_control.vhd` je obsažná simulace testující funkci ALU a jejího řízení.



Obrázek 5.3: Úspěšná simulace ALU

V simulaci se testují operace sčítání, odčítání, bitový posun, rotace, porovnání a operace booleovské algebry.

V obrázku 5.3 je uveden záznam signálů ze simulace v prostředí Vivado. Simulace využívá příkazu `assert` pro porovnání očekávané hodnoty a té vypočtené v ALU. Při zaznamenaní neshody se do konzole vypíše chybové hlášení.

5.4.2 Využití prostředků na FPGA

V přílohách je uvedeno schéma ALU A.2, tak jak jej vygenerovalo IDE Vivado podle popisu VHDL kódu.

Tabulka 5.2: Využití prostředku FPGA pro ALU

Název	Slice LUTs	F7 Muxes	DSP
ALU	442	26	0

Syntéza nepřidělila návrhu žádné DSP[3] bloky, jelikož pracuje nad kratšími slovy než je 32 bitů. Další z důvodů je že pro sčítání a logické operace její použití není tolik přínosné, jako tomu je například u násobení. Instrukci násobení tento procesor ale neimplementuje.

5.5 Čítač instrukcí

Obvod čítače instrukcí drží hodnotu adresy paměti, kde se nachází aktuální instrukce. Při přechodu na další instrukci se k hodnotě v registru přičte $+4$[4]. Když se má vykonat skok na instrukci, tak se nepřičítá čtyři, ale dojde k přičtení buď hodnoty `imm`, nebo se do registru uloží výsledek z ALU. Entita je popsána v souboru `PC_driver.vhd`.

Obvod je vybaven detektorem nezarovnaných adres, pokud takový stav nastane, nastaví signál `pc_error <= '1'`. Tabulka využití HW prostředků FPGA 5.3.

Tabulka 5.3: Využití prostředku FPGA pro čítač instrukcí

Název	Slice LUTs	Slice Registers
čítač instrukcí	20	32

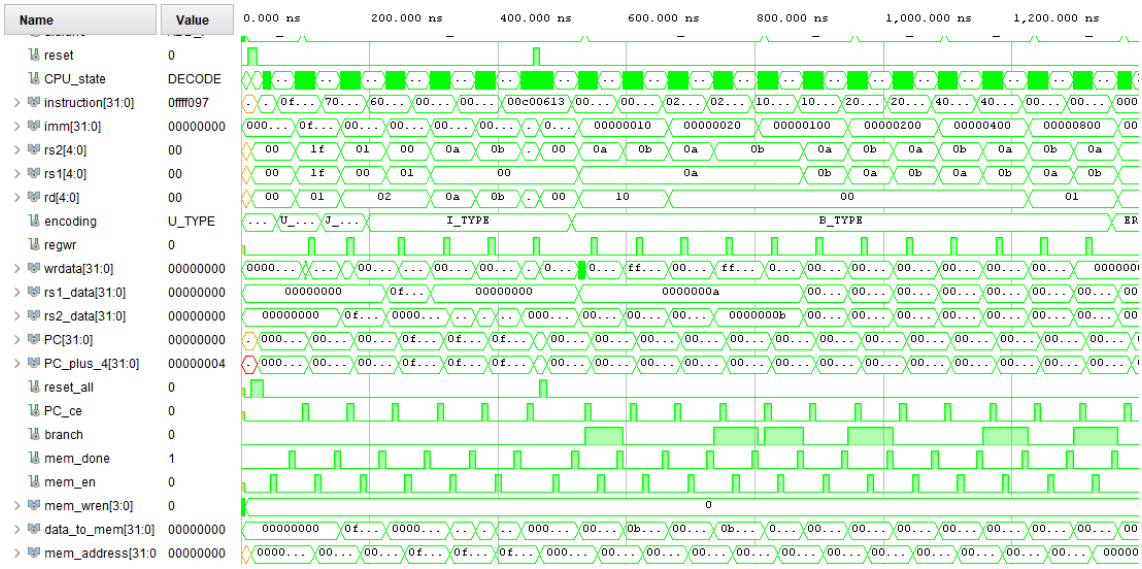
Simulace čítače instrukcí se nachází v souboru `tb_PC_control.vhd`. Povšimněte si v simulaci 5.4 jak je signál `branch` nastavován do jedničky v případech, kdy se má skákat v programu.

5.6 Paměť

RAM je řešena jako dvouportová paměť, která je jedním portem připojena přes odvod `memory_wpraper` na datovou sběrnici procesoru RISC-V a druhým portem přes sběrnici AXI do procesoru ARM. Paměť je nakonfigurovaná s latencí dva takty

[3] DSP - digital signal procesing.

[4] Je to posun o $+32b = +4 * 8b$, protože paměť je bajtově adresovatelná.



Obrázek 5.4: Úspěšná simulace řízení skoků v programu

a je popsána v souboru `bram_xpm_wrapper.vhd`.

Součástí adresního prostoru paměti RAM jsou také MMAP[5] periferie, ty jsou připojené po společné sběrnici kterou tvoří:

- adresa,
- data do periferie,
- data z periferie,
- povolení zápisu do bajtů,
- signál data vybavena z paměti.

Obvod `memory_wpraper` obaluje každou periferii zvlášť a zajišťuje výběr periferie se kterou procesor právě hovoří. Výběr se provádí na základě rozsahu adres, které se předávají jako generický parametr, pokud je periferie vybrána dává se jí to najevo jedničkou na signálu hodiny povoleny (ce).

Paměť RAM je tvořena právě jednou blokovou pamětí.

5.6.1 Nezarovnaný přístup do paměti

Přístup do paměti je realizován po slovech (32b), ale ISA určuje že paměť musí umožnit přístup na jednotlivé bajty. Podrobnosti najdete v kapitole 4.5 na straně 29.

Tabulka 5.4 reprezentuje všechny možné pozice validního zarovnání dat, které podporuje má implementace přístupu do paměti. Kontrolu provádí funkce

[5]Periferie mapované do paměti.

Tabulka 5.4: Zarovnaná data v paměti

Bajt 3	**Bajt 2**	**Bajt 1**	**Bajt 0**
slovo			
půl slovo			
	půl slovo		
		půl slovo	
bajt			
	bajt		
		bajt	
			bajt

`unaligned_address_check` na základě vybrané `funct_3_memory` (slovo, půlslovo, bajt) a adresy posunutí bajtu.

Zapisovaní dat se dává paměti najevo nastavením signálu povolující zápis na úrovni jednotlivých bajtů, tím se řeší zápisy menší než celé slovo.

Pokud se zapisuje do paměti s adresou bajtu jinou než 0, musí se data před jejich odesláním patřičně posunout do správné pozice.

Při načítání dat z paměti, která jsou posunuta, je třeba je opět zarovnat vpravo. O potřebné zarovnání se stará funkce `data_from_memory_formater`, která umožňuje doplnění nulami, nebo znaménkové rozšíření.

Tabulka 5.5: Nezarovnaná data v paměti

Bajt 3	**Bajt 2**	**Bajt 1**	**Bajt 0**
	slovo[31:8]		
slovo[7:0]			
		slovo[31:16]	
slovo[15:0]			
			slovo[31:24]
slovo[23:0]			
půl slov[15:8]			
			půl slovo[7:0]

Tabulka 5.5 popisuje nezarovnaný způsob uložení dat v paměti. Adresování takto uložených dat nelze se stávající architekturou realizovat pouze jedním přístupem do paměti, proto tento přístup není povolen a dojde k vyvolání výjimky[6].

[6]Problém však lze vyřešit softwarově při jejím zachycení.

5.6.2 Připojení paměti k procesoru

První návrh počítal se schématem Harvardské architektury počítače, měl tedy oddělenou paměť pro data a instrukce. Výhoda tohoto řešení byla v tom, že je na implementaci jednodušší a disponuje vyšší teoretickou propustností paměti, protože je možné najednou adresovat jak instrukční, tak datovou paměť.

S touto architekturou jsem otestoval funkčnost načítání příkazů z paměti na programu Fibonacciho posloupnosti, kód uvedený v příloze 7. Vše se zdálo velmi nadějné, ale do budoucna to nebyl vhodný přístup. Mít paměť RAM umístěnou uvnitř procesoru a rozdělenou na část pro program a pro data není praktické, protože nemá unifikovaný přístup (nevede k ní jedna společná externí sběrnice).

Nabízely se různé řešení:

- Vyvést z procesoru dvě oddělené sběrnice.
- Přidat vyrovnávací paměť připojenou na jednu sběrnici vedoucí z procesoru.
- Přesunout paměti mimo procesor a spojit je v jednu.

Dvě sběrnice přidávají do návrhu zbytečnou komplexitu, tak jsem toto řešení zahrnul.

Obalení obou pamětí za pomoci asociativní pamětí (cache), která by měla oddělený přístup jak pro data, tak instrukce je velmi praktické řešení, ale návrh takové paměti je nad rámec zadání práce.

Proto jsem se v dalším kroku návrh rozhodl přesunout paměť mimo procesor. To mělo umožnit připojení periferii mapovaných do paměti a její snazší přeprogramování přes sběrnici AXI (jejíž připojení implementoval vedoucí mé práce).

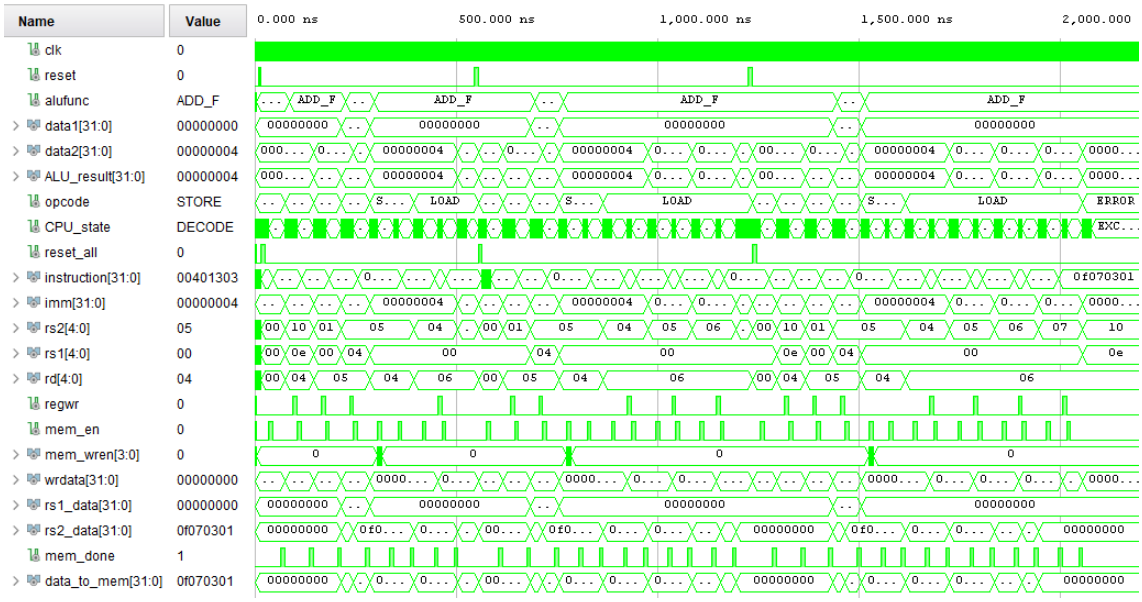
Znamenalo to přejít z Harvardského schématu počítače 3.2.1 na Von Neumanovo 3.2.2, tedy takové, které má jednotnou datovou a adresní sběrnici pro instrukce i data. Byl to nemalý zásah do návrhu a přinesl několik úskalí. Dekodér instrukcí (kombinační logika) předpokládal, že má instrukci stále k dispozici. To se ale s jednotnou sběrnicí změnilo, měl k dispozici buď načtenou instrukci, nebo data.

Řešením bylo přesunout řízení paměti na stranu procesoru, tak aby ven komunikoval pouze s jednou pamětí, ale vnitřně poskytoval registry pro uchovávání jak dat, tak i instrukce. Došlo tedy k přepracování řídící jednotky tak, že do ní byli přesunuty všechny funkce pro formátování dat **z** a **do** paměti. Nyní adresu pro komunikaci při jednotlivých fázích procesoru vybírá řídící jednotka. Při načítání instrukcí se na adresu do paměti připojí hodnota registru `pc` a ve fázi zápisu nebo čtení dat zase výsledek z ALU. Řídíc jednotka pak setrvá v dané fázi než paměť pošle signál data vybavena.

Výsledkem této změny návrhu bylo zpomalení práce fází procesoru, které manipulují s pamětí (načti instrukci a zapiš do paměti), o jeden takt (než se data vybaví a propíší do příslušného registru v řídící jednotce).

5.6.3 Simulace přístupu do paměti

Simulace `tb_memory_control.vhd` ověřuje čtení a zápis do paměti. Kontroluje zda nezarovnaný přístup vyvolává výjimku. Proto si můžete všimnout, že se v simulaci 5.5 několik aktivuje signál `reset_all`, který resetuje celý procesor včetně hlášení o výjimkách.



Obrázek 5.5: Úspěšná simulace přístupu do paměti

5.6.4 Periferie mapované do paměti

RISC-V nemá žádné instrukce pro přímou komunikací s vnějšími periferiemi. Pro jejich ovládání se využívá přístupu zařízeních mapovaných do paměti. Na příslušném rozsahu adres jsou místo paměti RAM připojeny například GPIO I/O registry či frame buffer displeje, nebo jiné periferie.

V mém návrhu procesoru se využívá periferií LED a přepínačů. Při nahrání hodnoty 0 na příslušnou pozici v rozsahu adres **směru** se pin periferie na daném bitu nastaví jako výstupní. Při nahrání hodnoty 1 se nastaví jako vstupní.

V paměti jsou reprezentovány jako 4x32b paměti, kde:

- první 4B (adresa+0) jsou směr (direction: in/out) (nastavení vstupu/výstupu periferie),
- další 4B (adresa+4) slouží pro zápis do periferie,
- následné 4B(adresa+8) slouží pro vyčítání hodnot z periferie,
- poslední 4B (adresa+12) nejsou obsazeny a jsou zde pouze pro zarovnání paměti na násobky dvou.

Na periferie lze přistupovat stejně jako do pole, ukázka kódu 3 který nastaví LEDky jako výstupní, přepínače jako vstupní a pak vezme hodnotu z přepínačů a nastaví je na LEDky.

```
# LED addresses (LED_ADDRESS -> RAM_SIZE)
li t3, 4096
# LED directions, 32 output pins
sw zero, 0(t3)
# LED values OFF
sw zero, 4(t3)

# SWITCH addresses (SWITCH_ADDRESS -> RAM_SIZE + LED_SIZE)
# 4096 + 16
addi t4, t3, 16
# SWITCH directions
lui t5, 0xFFFFF
ori t5, t5, -1
sw t5, 0(t4)

# nacteni hodnoty z prepinace
lw s0, 8(t4)
# zapis do led
sw s0, 4(t3)
```

Listing 3: Demonstrační kód pro zápis z přepínačů do LED

5.7 Řídící jednotka

Zdaleka nejsložitější částí procesoru pro návrh je řídící jednotka. Právě v ní se rozhoduje co bude která část procesoru v daný okamžik vykonávat.

V průběhu navrhování procesoru prošla řídící jednotka několika různými verzemi. Došlo například k přesunu dekodéru instrukcí do jejích útrob, stejně jako obvodů pro řízení paměti 5.6.2 a primitivní správy výjimek.

Řídící jednotka také vyhodnocuje kdy došlo ke skoku v programu.

Obrázek 5.6: Fáze procesoru

Procesor je navržen s ohledem na zřetězené zpracování instrukcí, symbolicky naznačené v diagramu 5.6. Zatím jej však nepodporuje. V bodovém seznamu je uvedeno pět klasických fází zřetězeného vykonávání procesorů architektury RISC-V:

- **Fetch** - načtení instrukce,
- **Decode** - dekódování instrukce,
- **Execute** - vykonání,
- **Memory** - paměť,
- **Writeback** - zápis do registru.

Řídící jednotka je navržena jako stavový automat. Jde o modifikovaný Mealyho automat, jehož všechny výstupy jsou vyvedeny přes registr. Automat je popsán vývojovým diagramem na obrázku 5.6.

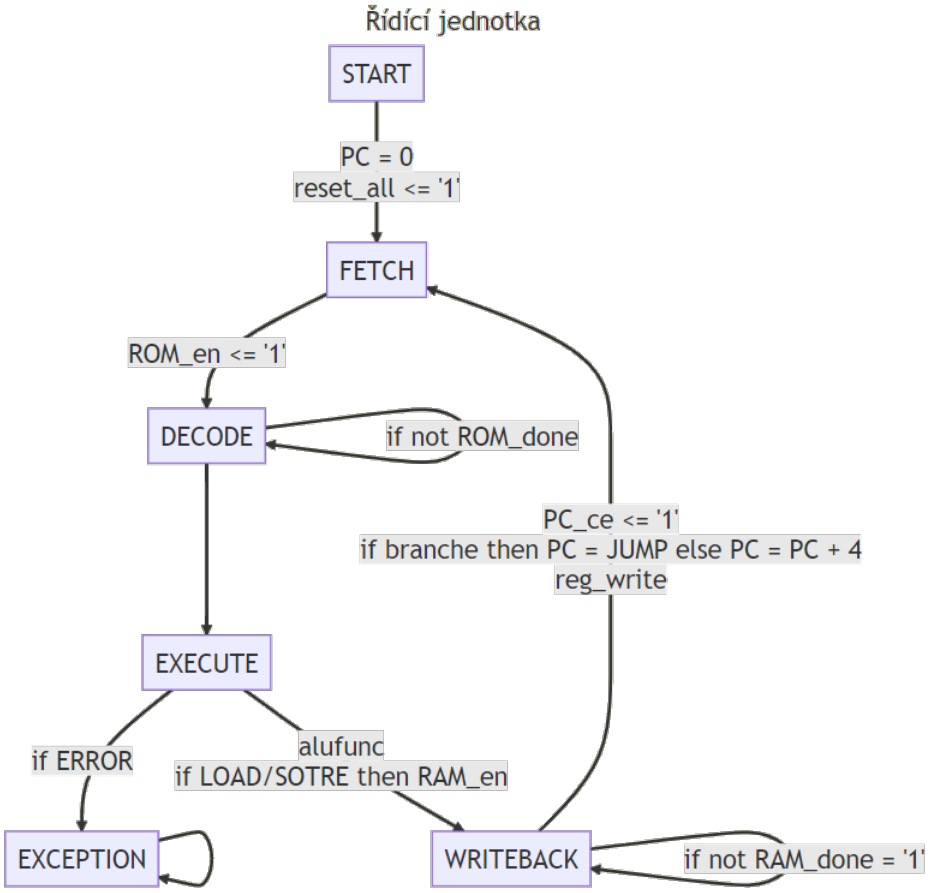
Fáze mého stavového automatu se skládají z: `START`, `FETCH`, `DECODE`, `EXECUTE`, `MEMORY`, `WRITEBACK`, `HALT`, `EXCEPTION`.

Při restartování procesoru se nastaví fáze stavového automatu na `START`. Když nastane vyjímka tak automat skočí na `EXCEPTION` a následně přejde do stavu `HALT`, ve kterém setrvá až do restartování.

Tabulka 5.6: Využití prostředku FPGA pro řídící jednotku

Název	Slice LUTs	Slice Registers	F7 Muxes
řídící jednotka	857	113	7

V tabulce 5.6 jsou uvedeny HW požadavky na FPGA.



Obrázek 5.7: Vývojový diagram řídící jednotky

Ač můj návrh architektury staví na instrukční sadě RISC kapitola 3.1.2, tak jeho řídící jednotka vykazuje známky přístupu které jsou běžné pro architekturu CISC kapitola 3.1.1. Umožňuje vykonání instrukcí proměněného počtu taktů a to jmenovitě při práci s pamětí. Řídící jednotka při načítání a ukládání dat čeká na signál data vybavena. Tímto způsobem je ošetřena vícetaktová vybavovací doba blokové paměti (2 takty) ze které je nakonfigurovaná RAM pomocí makra XPM. Výhoda tohoto přístupu se naplno projeví až v budoucnu při implementování asociativní paměti (cache) procesoru. Při nalezení (hit) se budou instrukce načítat rychleji (jeden takt) a při případném nenalezení (miss) se vykonávání automaticky pozastaví, protože řídící jednotka bude čekat na signál vybavení dat z paměti RAM.

5.8 Jádro procesoru RISC-V a RAM

V jádru procesoru se propojují jednotlivé části návrhu. Jsou v něm také definovány multiplexery pro výběr zdroje pro zápis do registru a argumentů ALU. K jádru procesoru je ještě připojená paměť a MMAP periferie.

Tabulka 5.7: Využití prostředku FPGA pro jádro procesoru RISC-V

Název	**Slice LUTs**	**Slice Registers**	**F7 Muxes**	**F8 Muxes**
jádro procesoru	1624	1137	263	70

HW požadavky toho návrhu jsou uvedeny v tabulce 5.7.

5.9 Vlastní IP jádro

Aby bylo možné nahlížet do operační paměti procesoru je k ní souboru `computer_wrapper.vhd` ještě připojena AXI sběrnice, která ARM procesoru umožňuje přístup do paměti. Jsou zde také napojené vstupní a výstupní periferie (LED, přepínače a GPIO) na piny FPGA.

Tabulka 5.8: Využití prostředku FPGA pro IP jádro

Název	**Slice LUTs**	**Slice Registers**	**F7 Muxes**	**F8 Muxes**	**Block RAM**	**I/0**
IP jádro	1936	1783	263	70	1	178
IP jádro	3,6%	1,7%	1%	0,5%	0.7%	90%
Zynq7020	53200	106400	26600	13300	140	200

Návrh procesoru využívá přibližně 3,7% LUTů, 1,7% registrů. Informace o využitých I/O je zavádějící protože naprostá většina z nich slouží pro implementaci AXI sběrnice, pokud by design RISC-V uCPU tvořil výsledný návrh, většina z I/O by nebyla připojená na piny FPGA. Takže ve srovnání s prostředky FPGA je můj návrh velmi malý.

6 Programy

Tato práce se kromě návrhu architektury jádra procesoru RISC-V v jazyce VHDL zabývala i vývojem testovacích programů a nástrojů potřebných pro jejich nahrání do paměti.

Pro otestování funkčnosti jsem napsal několik programů v jazyce symbolických adres. Lze je rozdělit do dvou skupin dle určení:

- pro použití v simulaci,
- pro syntetizovaný procesor na desce FPGA.

6.1 Programování procesorů RISC-V

V následujících kapitolách se nejprve lehce seznámíme s psaním programů v jazyce assembly pro procesory RISC-V.

6.1.1 Pseudoinstrukce

Pseudoinstrukce slouží pro větší abstrakci při programování v jazyce symbolických adres (assembly). Umožňují psát program v instrukcích, které procesor sice přímo nepodporuje, ale jejich vykonání je možné syntetizovat pomocí již existujících instrukcí (obrázek 4.1) a to buď zaměněním pořadí argumentů, či případně vynulováním jednoho z nich, viz tabulka 6.1.

Například instrukce MV (přesun obsah registru) jde syntetizovat jako `ADDI rd, ra1, 0`. V tabulce 6.1 je přehled dalších pseudoinstrukcí.

Tabulka 6.1: RISC-V pseudoinstrukce (převzato z [24])

Pseudoinstrukce				**RISC-V instrukce**				**Popis**
nop				addi	x0,	x0,	0	nedělej nic
li	rd,	$imm_{11:0}$		addi	rd,	x0,	$imm_{11:0}$	načti 12-bitovou konstantu
li	rd,	$imm_{31:0}$		lui addi	rd, rd,	$imm_{31:12}$ rd,	 $imm_{11:0}$	načti 32-bitovou konstantu
mv	rd,	rs1		addi	rd,	rs1,	0	přesun
not	rd,	rs1		xori	rd,	rs1,	-1	jedničkový doplněk
neg	rd,	rs1		sub	rd,	x0,	rs1	dvojkový doplněk
seqz	rd,	rs1		sltiu	rd,	rs1,	1	nastav když $= 0$
snez	rd,	rs1		sltu	rd,	x0,	rs1	nastav když $\neq 0$
sltz	rd,	rs1		slt	rd,	rs1,	x0	nastav když < 0
sgtz	rd,	rs1		slt	rd,	x0,	rs1	nastav když > 0
beqz	rs1,	label		beq	rs1,	x0,	label	skoč když $= 0$
bnez	rs1,	label		bne	rs1,	x0,	label	skoč když $\neq 0$
blez	rs1,	label		bge	x0,	rs1,	label	skoč když ≤ 0
bgez	rs1,	label		bge	rs1,	x0,	label	skoč když ≥ 0
bltz	rs1,	label		blt	rs1,	x0,	label	skoč když < 0
bgtz	rs1,	label		blt	x0,	rs1,	label	skoč když > 0
ble	rs1,	rs2,	label	bge	rs2,	rs1,	label	skoč když $\leq$
bgt	rs1,	rs2,	label	blt	rs2,	rs1,	label	skoč když $>$
bleu	rs1,	rs2,	label	bgeu	rs2,	rs1,	label	skoč když $\leq$ (ne znaménkově)
bgtu	rs1,	rs2,	label	bltu	rs2,	rs1,	offset	skoč když $>$ (ne znaménkově)
j	label			jal	x0,	label		skoč
jal	label			jal	ra,	label		skoč a ulož adresu
jr	rs1			jalr	x0,	rs1,	0	skoč na registr
jalr	rs1			jalr	ra,	rs1,	0	skoč na registr a ulož adresu
ret				jalr	x0,	ra,	0	návrat z funkce
call	label			jal	ra,	label		zavolání funkce

6.1.2 Ukázka jak by mohl vypadat program v Assembly

Takto by mohl vypadat demonstrační program 4, který běží v nekonečné smyčce. Pokud program využívá zásobník, tak je potřeba provést inicializaci registru `x2` (`sp` - stack pointer) na horní hranici rozsahu paměti RAM. Operační paměť má nastavenou syntetizovanou velikost 4kB (4096 bajtů). Funkce `return_arg` vrací svůj argument. Program main, volá funkci `return_arg` a ta svůj argument `a0` ukládá jako návratovou hodnotu do registru `a1`. Pokud je návratová hodnota různá od nuly, tak se program vrátí na začátek smyčky `loop`. Pokud je návratová hodnota nula, tak se program ukončí vyvoláním výjimky instrukcí ECALL.

```
1  .text
2  .globl main
3
4  # inicializace ukazatele na zasobnik
5  li sp, 4096          # MEM(0:4092)
6  addi sp, sp, -4      # WORD_SIZE = 4
7
8  j main               # skok na smycku hlavni funkce
9
10 # Funkce: return argumet
11 # Argumenty:
12 #  a0 - argument
13 #  a1 - navratova hodnota
14 return_arg:
15     mv a1, a0        # a1 = a0
16     li a0, 0         # a0 = 0
17     ret
18
19 main:
20     addi a0, x0, 1   # a0 = 1
21
22     loop:
23         jal ra, return_arg  # a1 = return_arg(a0)
24         addi a0, a1, 0      # a0 = a1
25         bne a1, x0, loop    # if a1 is True goto loop
26     ecall
```

Listing 4: Ukázka programu v jazyce symbolických adres

Direktiva `.text` říká, že následující příkazy jsou instrukce programu a budou uložena v sekci paměti **text**, blíže v kapitole 3.3 na straně 19. Direktiva `.globl` nastavuje následující symbol jako globální, ten pak může být použit i v jiných souborech. V tomto případě je to symbol `main`.

6.2 Kompilace zdrojových kódů

Program v jazyce C není zas tak jednoduché přeložit do binárního kódu spustitelného na architektuře RISC-V32I. Při překladu na počítači s procesorem architektury Intel/AMDx64 je potřeba využít takzvaného křížového překladu[1]. Problém je však v tom, že existují běžné překladače pro RISC-V64G[2], tento kód na mém procesoru nelze spustit.

Při překladu z jazyka symbolických adres lze využít běžného RISC-V překladače, pokud však jsou instrukce programu pečlivě voleny tak, aby obsahovali pouze instrukce z ISA RISC-V32I.

Nebo je možné si přeložit překladač GNU od RISC-V Collaboration *riscv-gnu-toolchain* na vlastním počítači [25].

Překladač GNU jsem úspěšně přeložil na stroji s operačním systémem Ubuntu Linux, instalace IDE Vivado byla však na počítači s operačním systémem Windows.

> Ubuntu je pradávné africké slovo nesoucí význam: „Neumím nakonfigurovat Debian“.
>
> (Ubuntu — Urban dictionary [26])

Nabízelo se překladač používat pod WSL. Celkové použití bylo však pro mojí aplikaci příliš složité, ale pokud bych měl v budoucnu překládat delší zdrojové kódy jazyka symbolických adres, případně C, jistě bych se vydal cestou přes WSL.

Z důvodu snazšího ovládáni jsem se rozhodl použít online překladač dostupný na *riscvasm.lucasteske.dev*.

[1] Program je kompilovaný pro jinou rodinu procesoru, než do které patří procesor, který překlad provádí.

[2] G je zkratka pro rozšíření IMAFDZicsr_Zifencei.

6.3 Generování konfiguračních souborů pro nahrání programu do paměti

Přeložený program jde ukládat do několika různých formátů:

- **CEO** - slouží pro konfigurátor IP jader (ten se v poslední návrhu procesoru již nevyužívá, nahradil jej popis paměti přímo v XPM),
- **MEM** - je formát pro blokové paměti. Programy z tohoto formátu načítá simulace.
- **RAW** - slouží k ukládání na SD kartu pro desku ZedBoard.

Pro usnadnění práce s těmito soubory byl vytvořen python script `RISC-V/programs/vivado_datafile_generator.py` ten umožňuje přepis hex--dump souboru do formátu `.coe`, `.mem`, `.raw`. Ukázka výpisu přepínače `--help` pro zmíněný program 5.

```
python.exe .\vivado_datafile_generator.py --help
usage: vivado_datafile_generator.py [-h] -i INPUT [-o OUTPUT]
                                    [-r RADIX] [-f FORMAT]

Aplication for generating COE, RAW and MEM files. For VHDL ROM
initialization. Specific types:
- COE file is used for Xilinx Vivado block ram IP core.
- MEM file is used for block ram.
- RAW file is used for loading program into FPGA memory from SD card.

options:
  -h, --help            show this help message and exit
  -i INPUT, --input INPUT
                        input file
  -o OUTPUT, --output OUTPUT
                        output file
  -r RADIX, --radix RADIX
                        radix
  -f FORMAT, --format FORMAT
                        file format
```

Listing 5: Python skript pro generování Xilinx souborů s programem

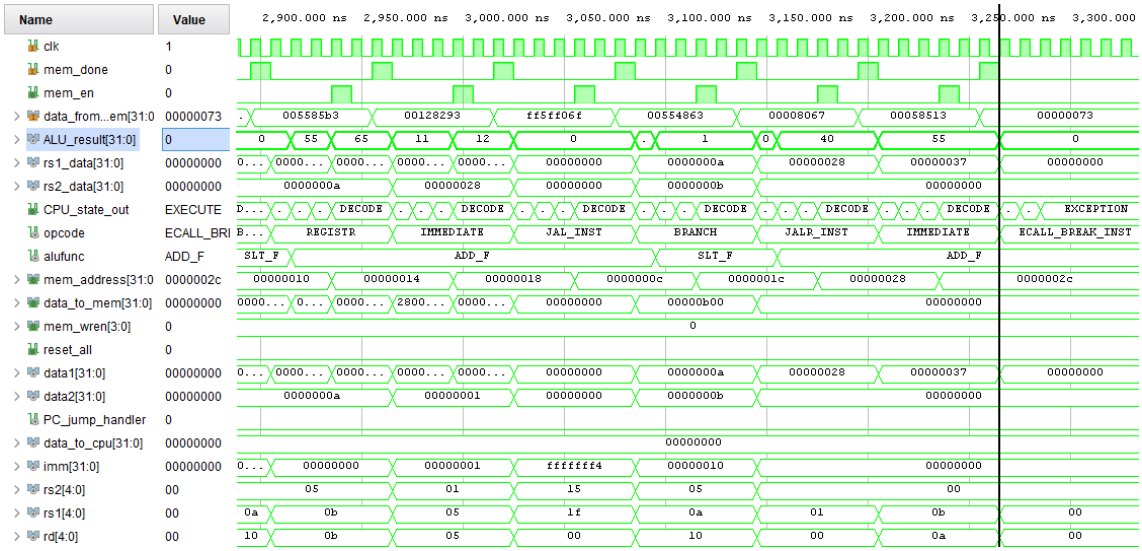
Příklad použití, pro vygenerování nového souboru `.mem` s obsahem paměti ze souboru `sum.txt` zadejte příkaz:

```
python3 vivado_datafile_generator.py -i sum.txt -o program.mem
```

6.4 Demo programy pro otestování RISC-V procesoru v simulaci

6.4.1 Test aritmetiky

Pro základní test aritmetiky byl zvolen program výpočtu sumy. Vzniklo několik jeho variant. První varianta testuje pouze základní aritmetické operace a cyklus for. Druhá varianta testuje i volání funkce.



Obrázek 6.1: Úspěšná simulace se spuštěním programu sum(10)

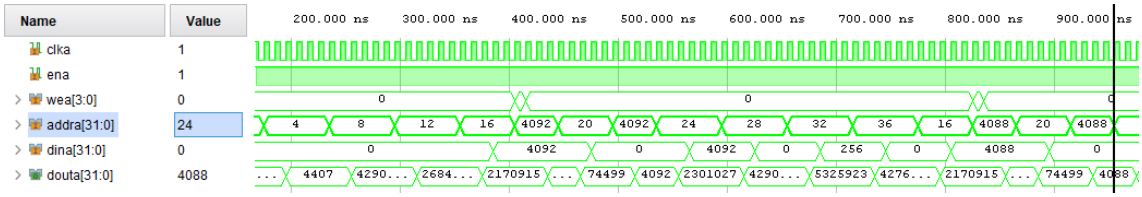
V simulaci 6.1 je spuštěn program sečti posloupnost čísel od 1 do 10. Všimněte si že výsledek je v signálu `ALU_result` má hodnotu 55.

6.4.2 Test paměti

Program testující práci s pamětí, které bude později využita při ukládání proměnných na zásobník a jejich čtení, nebo při komunikaci se zařízeními mapovanými do paměti. Cyklus for postupně ukládá do paměti data, následně je čte a ověřuje zda jsou shodná. Začne na čísle 0 a postupně inkrementuje až do velikosti paměti. S hodnotou proměnné se posouvá i adresa, na kterou se má data uložit. Pokud se někde vyskytne chyba, program se ukončí. Simulace je uvedena n obrázku 6.2.

6.4.3 Fibonacci

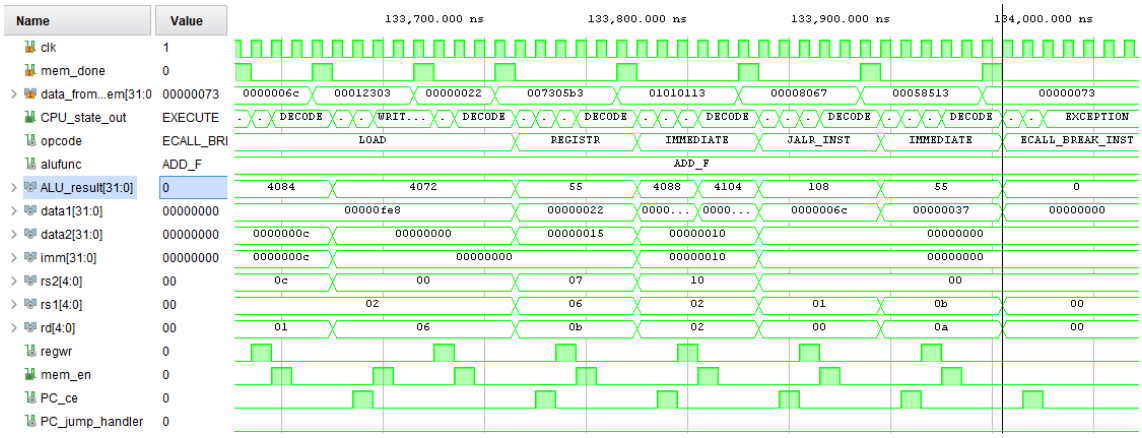
Algoritmus Fibonacciho posloupnosti byl zvolen protože pro výpočet nepotřebuje operace násobení a dělení. Nejdříve byla využita jeho sekvenční verze, která využívá cyklus for, pro otestování základní funkčnosti. Následně byla využita rekurzivní verze, která využívá zásobník. Program v jazyce C je uveden v příloze 6, stejně tak



Obrázek 6.2: Úspěšná simulace se spuštěním programu testu paměti

i ten v jazyce symbolických adres příloha 7.

$$F(n) = \begin{cases} 0, & \text{pro } n = 0; \\ 1, & \text{pro } n = 1; \\ F(n{-}1) + F(n{-}2) & \text{jinak.} \end{cases} \tag{6.1}$$



Obrázek 6.3: Úspěšná simulace se spuštěním programu fib(10)

Při syntéze RAM o velikosti 4kB se na stack se vejde 256 rámců ($256 * 16B = 4kB$). V každém rámci jsou 4B pro uložení adresy návratu a 12B pro uložení argumentů. Tedy maximálně 256 volání rekurze. Při volání výpočtu $fib(10)$ je potřeba zavolat funkci 177 a to se s jistotou do paměti RAM vejde.

Ze simulace 6.3 je patrné, že po uplynutí času $134us$ se procesor dopočítá k hodnotě $fib(10) = 55$, což je očekávaný výsledek.

6.4.4 Nahraní nového programu do simulace

Simulace (test bench) `tb_run_program.vhd` spouští program uložený do souboru: `program.mem`. Pro spuštěné nového programu je potřeba vygenerovaný soubor ve formátu `.mem` přesunout do adresáře `./RISC-V.srcs/sources_1/new/` a restartovat simulaci.

6.5 Demo programy pro RISC-V procesor na desce ZedBoard

Procesor je také napojen na vnější svět pomocí zařízení mapovaných do paměti. Je připojený na přepínače a ledky. Ledky jsou mapované na adresu 4096 a mají rozsah 128b. Přepínače na adresu 4112. Přepínače jsou nastaveny jako vstupní a ledky jako výstupní piny.

Pro otestování vývojové desky byl napsán program `switch_to_LED`, který běží v nekonečné smyčce a zrcadlí stav přepínačů na ledkách.

Funkčnost syntetizované architektury se tímto jednoduchým programem podařila úspěšně ověřit.

6.5.1 Nahrání programu do paměti

Vývojová deska je postavené kolem SoC Xilinx Zynq-7000, který má v sobě jak FPGA, tak i ARM procesor. Pro něj vedoucí mé práce vytvořil program v jazyce C. Program hledá na SD kartě soubor `program.raw`. Když soubor s příponou `.raw` úspěšně nalezne, tak jeho obsah nahraje do paměti RAM procesoru RISC-V který je syntetizovaný v části čipu s FPGA.

Teto program se následně postará i o řízení mého procesoru. Povoluje mu signál hodin (jeden takt, nebo více) a pokud je k ARMu připojen váš počítač po sériové lince, umožní i výpis do konzole v jaké fázi se procesor nachází.

Jak vytvořit soubor s příponou `.raw` pro nahrání programu je podrobněji popsáno v kapitole 6.3.

Vygenerovaný soubor nakopírujte na SD kartu a zasuňte jí do vývojové desky ZedBoard. Desku restartujte. Když spustíte aplikaci pro nahrávání programu na procesoru ARM přes IDE Vitis, tak se provede jeho zavedení do paměti procesoru RISC-V a ten bude uveden v činnost.

7 Závěr

V bakalářské práci jsem se seznámil se specifikací ISA RISC-V. Podle ní jsem navrhl procesorové jádro v základní 32bitové verzi, které pracuje v neprivilegovaném režimu. Můj procesor umožňuje výpočty s celými čísly a je bez jakéhokoliv rozšíření. Navržené jádro jsem propojil s operační paměti a vstupními a výstupními periferiemi.

V simulacích jsem postupně úspěšně otestoval základní vlastnosti tohoto procesoru, jako jsou aritmetickologické operace, přístup do paměti nebo podmíněné skoky v programu.

Podařilo se mi syntetizovaný procesor nahrát do FPGA řady Xilinx Zynq-7000. Nakonec jsem napsal program v jazyce symbolických adres pro zobrazení stavu přepínačů na LED diodách. Tento program se na mém procesoru úspěšně spustil.

Nejpřímočařejším rozšířením mého procesoru by bylo přidání některého z rozšíření které popisuje specifikace ISA RISC-V, jako je například podpora násobní a dělení, nebo výpočtů nad čísly s pohyblivou řádovou čárkou.

Do budoucna se nabízí procesor také rozšířit o zřetězené zpracovávání instrukcí. Při návrhu jsem se snažil postupovat tak, aby implementace tohoto rozšíření byla co nejednodušší.

Dalším způsobem navýšení výkonu mého procesoru by mohlo být jeho doplnění o asociativní paměť.

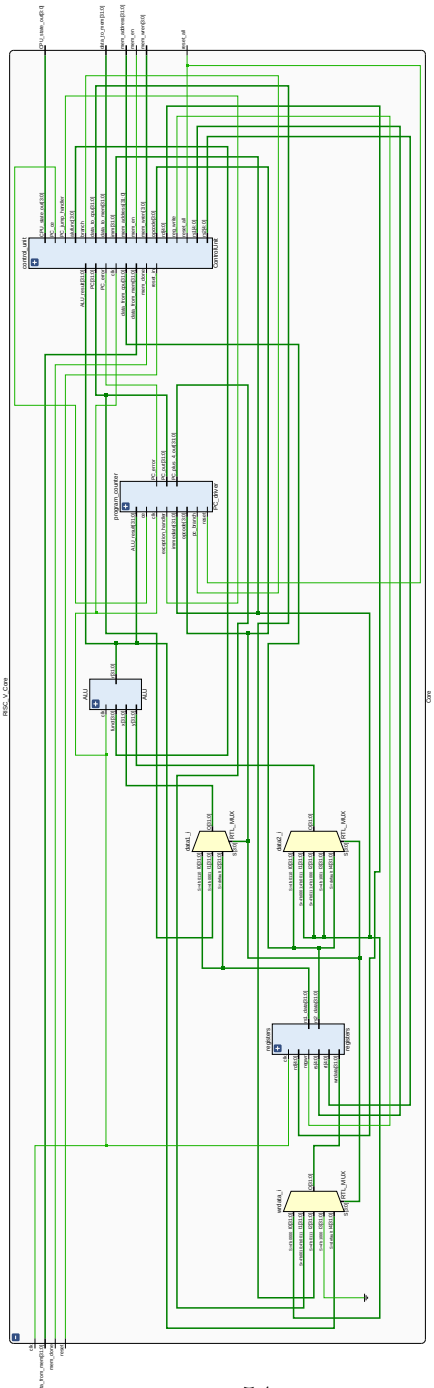
Pokud nastane výjimka při vykonávání programu, tak se můj procesor zastaví. Vhodným rozšířením by tak z tohoto pohledu bylo navrhnout obsluhu výjimek, třeba i s programovou částí řešící limitace základní ISA RISC-V32I jako je například chybějící instrukce násobení nebo podpora práce s nezarovnanou pamětí.

Můj procesor sice nedisponuje nikterak vysokým výkonem, ale jeho výhodou jsou malé rozměry (zabere méně než 4% LUTů) na FPGA řady Xilinx Zynq-7000. Při rozšíření návrhu o sérii čítačů, a podporu přerušení by mohl sloužit jako mikroprocesor pro ovládání jiných návrhů na FPGA, které potřebují procesorové řízení.

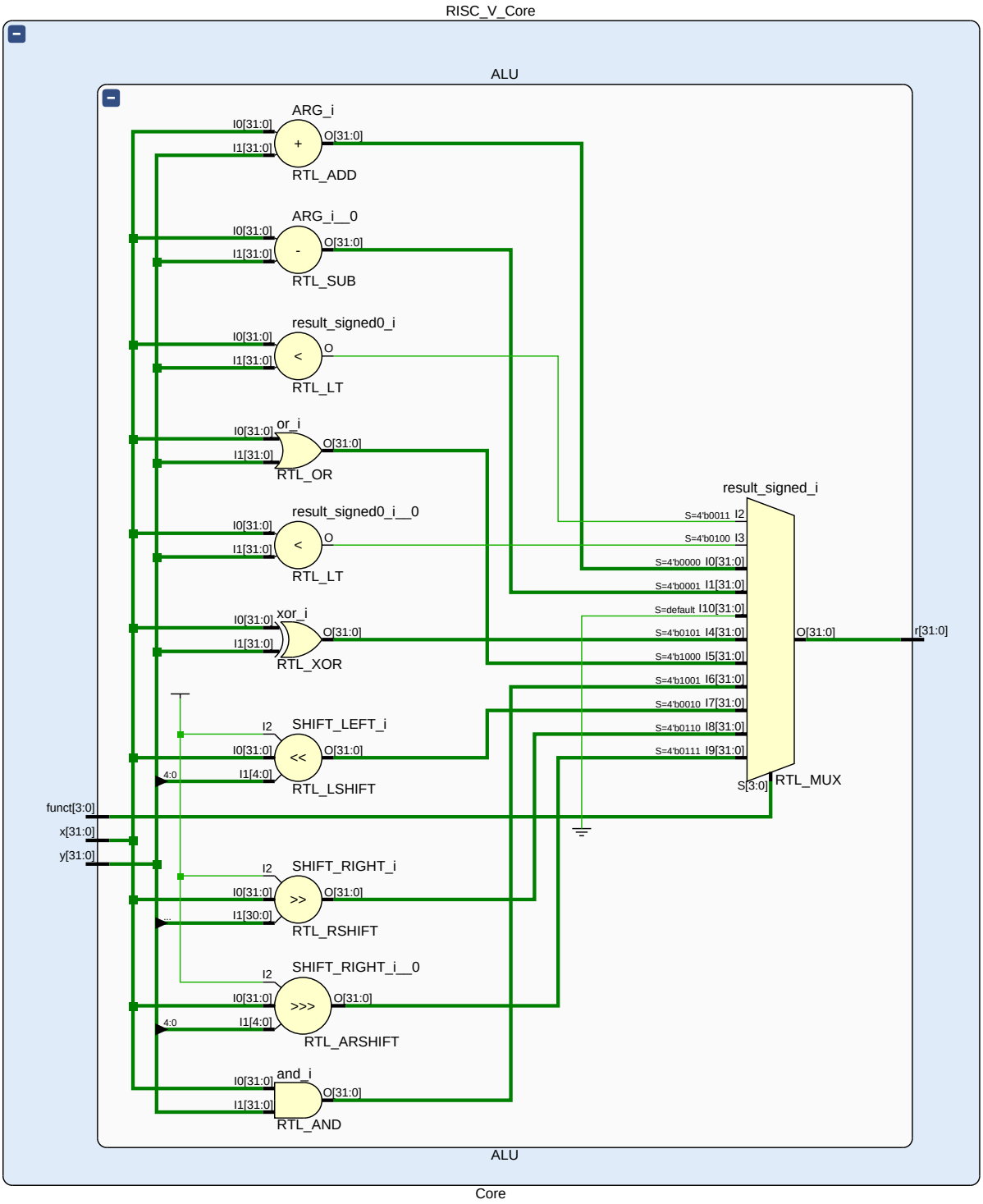
Zajímavé by bylo také prozkoumat tu možnost kdy by se z mého návrhu nechal pouze balíček implementující specifikaci RISC-V a návrh procesoru by se přepsal do čistě behaviorální podoby, která by stavěla na funkcích z balíčku, jejichž funkčnost je již otestovaná.

A Přílohy

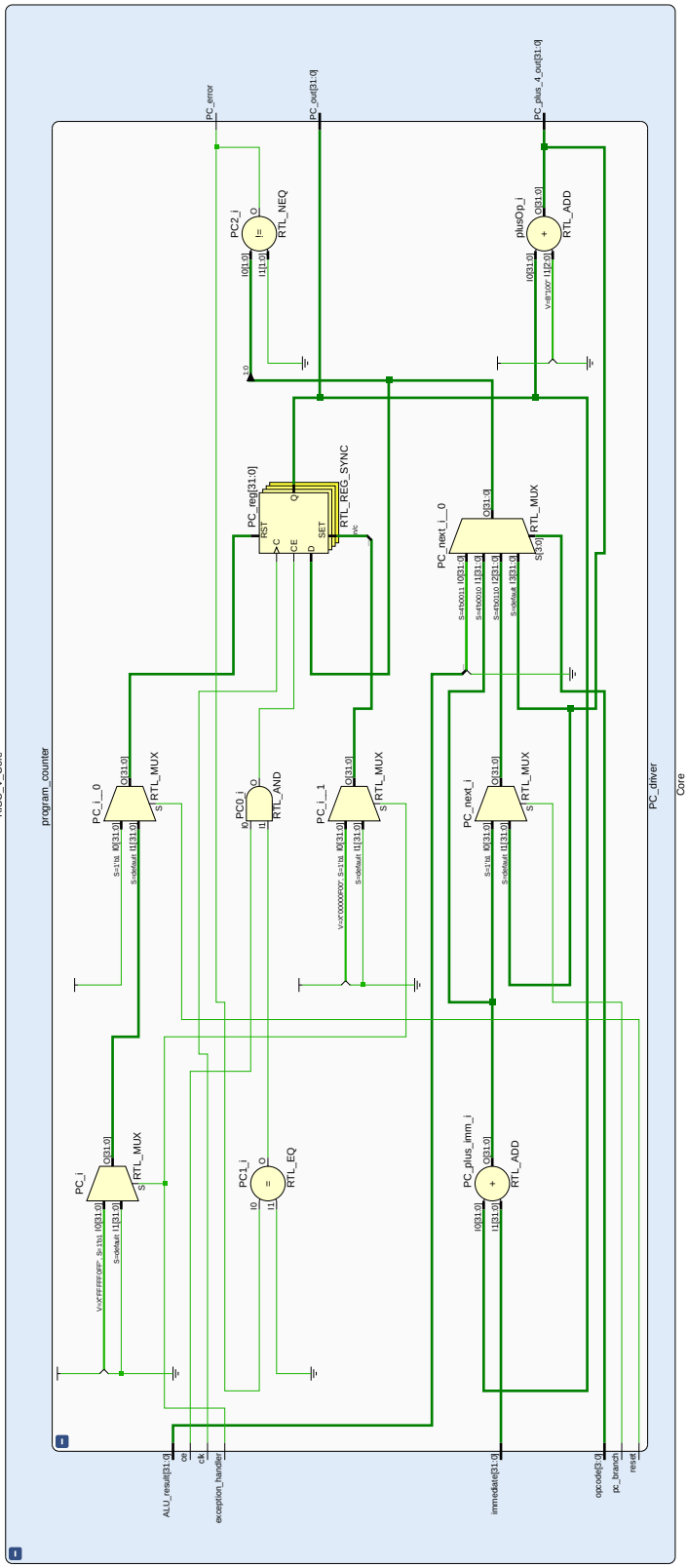
A.1 Shéma jádra procesoru RISC-V



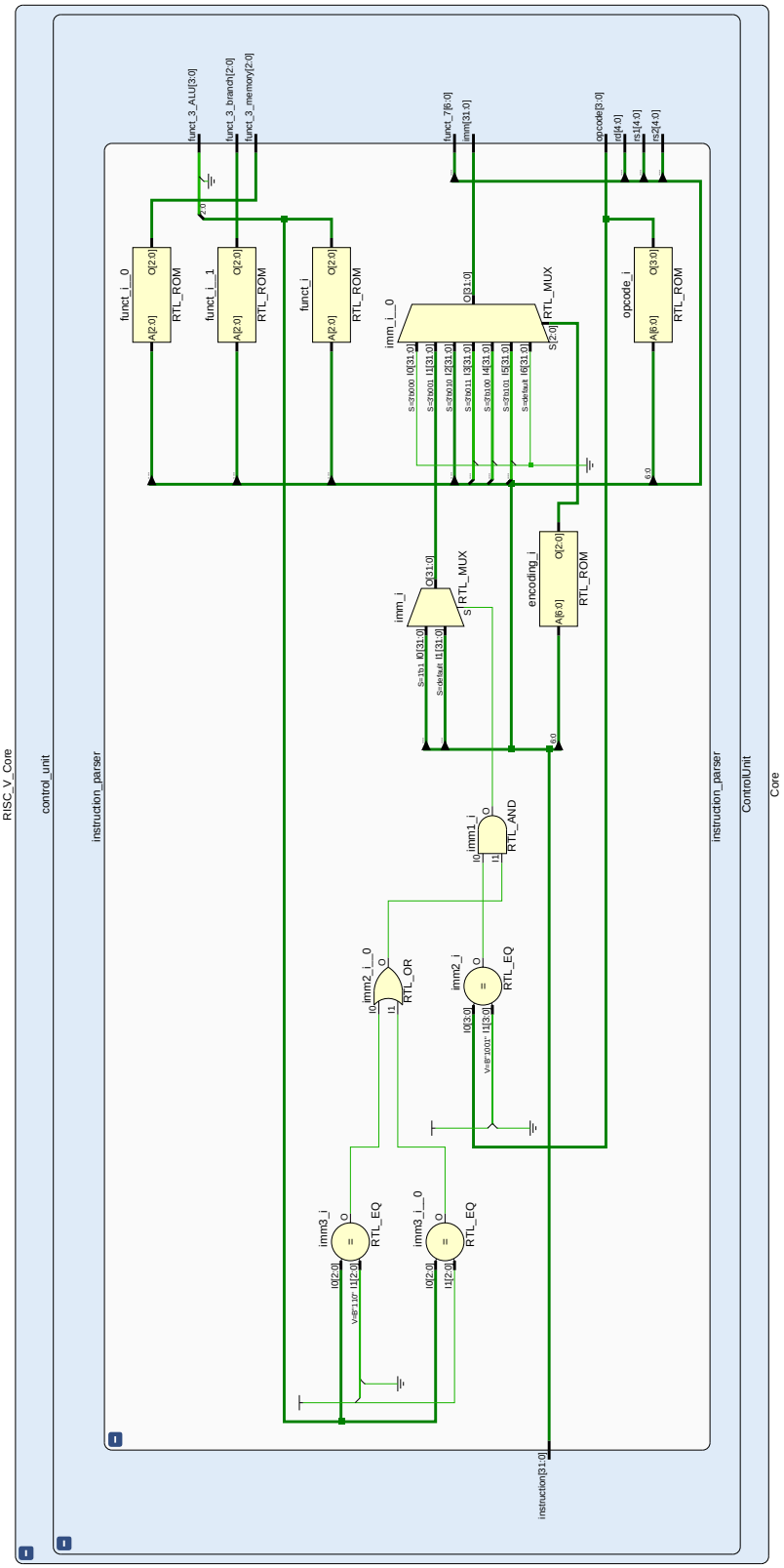
A.2 Shéma ALU


RISC_V_Core
ALU
ARG_i
I0[31:0]
I1[31:0]
+
O[31:0]
RTL_ADD
ARG_i__0
-
RTL_SUB
result_signed0_i
<
O
RTL_LT
or_i
RTL_OR
result_signed0_i__0
xor_i
RTL_XOR
SHIFT_LEFT_i
I2
<<
I1[4:0]
4:0
RTL_LSHIFT
SHIFT_RIGHT_i
>>
I1[30:0]
RTL_RSHIFT
SHIFT_RIGHT_i__0
>>>
RTL_ARSHIFT
and_i
RTL_AND
result_signed_i
S=4'b0011 I2
S=4'b0100 I3
S=4'b0000 I0[31:0]
S=4'b0001 I1[31:0]
S=default I10[31:0]
S=4'b0101 I4[31:0]
S=4'b1000 I5[31:0]
S=4'b1001 I6[31:0]
S=4'b0010 I7[31:0]
S=4'b0110 I8[31:0]
S=4'b0111 I9[31:0]
S[3:0]
RTL_MUX
r[31:0]
funct[3:0]
x[31:0]
y[31:0]
ALU
Core

A.3 Shéma čítače instrukcí



A.4 Shéma dekodéru instrukcí



A.5 Program: Fibonaciho posloupnost v C

```c
int fibonaci(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return fibonaci(n - 1) + fibonaci(n - 2);
}

void main() {
    int number = fibonaci(10);
}
```

Listing 6: Fibonaciho posloupnost v jazyce C

A.6 Program: Fibonaciho posloupnost v assembly

```
.text
.globl main

# inicializace SP
li sp, 4096
addi sp, sp, -8

# Skok na hlavní funkci
j main

# Funkce fibonaci - vypočítá n-té číslo Fibonaciho posloupnosti
# Argumenty:
#   a0 - n (číslo, pro které chceme spočítat číslo Fibonaciho posloupnosti)
# Návratová hodnota:
#   a1 - n-té číslo Fibonaciho posloupnosti
fibonaci:
    # Test na n = 0
    beqz a0, fibonaci_return_0

    # Test na n = 1
    li t0, 1
    beq a0, t0, fibonaci_return_1

    # Výpočet fibonaci(n - 1)
    addi sp, sp, -16
    addi a0, a0, -1
    sw ra, 12(sp)
    sw a0, 8(sp)
    jal ra, fibonaci
    sw a1, 0(sp)

    # Výpočet fibonaci(n - 2)
    lw a0, 8(sp)
    addi a0, a0, -1
    jal ra, fibonaci
    mv t2, a1

    # Návratová hodnota = fibonaci(n - 1) + fibonaci(n - 2)
    lw ra, 12(sp)
    lw t1, 0(sp)
    add a1, t1, t2
    addi sp, sp, 16
    jr ra
```

```
44
45 fibonaci_return_0:
46     # Návratová hodnota pro n = 0 je 0
47     li a1, 0
48     jr ra
49
50 fibonaci_return_1:
51     # Návratová hodnota pro n = 1 je 1
52     li a1, 1
53     jr ra
54
55 # Hlavní funkce
56 main:
57     # Nastavení argumentu pro volání funkce fibonaci
58     addi a0, zero, 10
59
60     # Volání funkce fibonaci
61     jal ra, fibonaci
62
63     # Předání výsledku do registru a0
64     mv  a0, a1
65     # Konec programu
66     ecall
```

Listing 7: Fibonaciho posloupnost v jazyce assembly

Použitá literatura

[1] *What does risc v stand for* [online]. [cit. 2023-05-08]. Dostupné z: https://codasip.com/2021/03/17/what-does-risc-v-stand-for/.

[2] *Seznam logických integrovaných obvodů řady 7400* [online]. San Francisco (CA): Wikimedia Foundation, 2001 [cit. 2023-05-07]. Dostupné z: https://cs.wikipedia.org/wiki/Seznam_logick%C3%BDch_integrovan%C3%BDch_obvod%C5%AF_%C5%99ady_7400.

[3] *8086: 16-BIT HMOS MICROPROCESSOR* [online]. 1990. [cit. 2023-05-08]. Dostupné z: https://datasheetspdf.com/pdf-file/544568/Intel/8086/1.

[4] *AMD64 Technology: AMD64 Architecture Programmer's Manual Volume 1: Application Programming* [online]. 3.23. vyd. 2020. [cit. 2023-05-08]. Dostupné z: https://www.amd.com/system/files/TechDocs/24592.pdf.

[5] *X86* [online]. San Francisco (CA): Wikimedia Foundation, 2001 [cit. 2023-05-08]. Dostupné z: https://en.wikipedia.org/wiki/X86.

[6] *Reduced instruction set computer* [online]. San Francisco (CA): Wikimedia Foundation, 2001 [cit. 2023-05-08]. Dostupné z: https://en.wikipedia.org/wiki/Reduced_instruction_set_computer.

[7] *Mips* [online]. [cit. 2023-05-08]. Dostupné z: https://www.computerhope.com/jargon/m/mips.htm.

[8] KUMARSAHOO, Amit. *Advanced risc machine arm processor* [online]. [cit. 2023-05-08]. Dostupné z: https://www.geeksforgeeks.org/advanced-risc-machine-arm-processor/.

[9] *ARM: Arm Flexible Access* [online]. [cit. 2023-05-08]. Dostupné z: https://www.arm.com/products/flexible-access.

[10] *RISC-V: history* [online]. 2021. [cit. 2023-05-08]. Dostupné z: https://riscv.org/about/history/.

[11] KUTÝ, Michael. *Otázky na státnice: Architektura počítače* [online]. 2014. [cit. 2023-05-08]. Dostupné z: http://michaelkuty.github.io/ssz-ai-hk-3/tech/2.html.

[12] MICHÁLEK, Ondřej. *Principy počítačů: Architekte, kam jsem si ten výpočet uložil...?* [online]. [cit. 2023-05-13]. Dostupné z: https://www.itnetwork.cz/hardware-pc/principy-pocitacu/architekte-kam-jsem-si-ten-vypocet-ulozil.

[13] PANKAJ. *Harvard Architecture* [online]. [cit. 2023-05-13]. Dostupné z: https://www.geeksforgeeks.org/harvard-architecture/.

[14] NEUMANN, John von. First Draft of a Report on the EDVAC [online]. [B.r.], s. 101 [cit. 2023-05-08]. Dostupné z: http://web.mit.edu/STS.035/www/PDFs/edvac.pdf.

[15] PELIKÁN, Jaroslav. *Von Neumannovo schéma* [online]. 1999. [cit. 2023-05-08]. Dostupné z: https://www.fi.muni.cz/usr/pelikan/ARCHIT/TEXTY/VNEUM.HTML.

[16] *Von Neumannova architektura* [online]. San Francisco (CA): Wikimedia Foundation, 2001 [cit. 2023-05-08]. Dostupné z: https://cs.wikipedia.org/wiki/Von_Neumannova_architektura.

[17] CHEN, Jenny a Ruohao GUO. *Stack and Heap Memory* [online]. [cit. 2023-05-13]. Dostupné z: https://courses.engr.illinois.edu/cs225/fa2022/resources/stack-heap/.

[18] *Stack Vs Heap Java* [online]. [cit. 2023-05-13]. Dostupné z: https://www.javatpoint.com/stack-vs-heap-java.

[19] *Zásobník: Základní architektura zásobníku* [online]. San Francisco (CA): Wikimedia Foundation, 2001 [cit. 2023-05-13]. Dostupné z: https://cs.wikipedia.org/wiki/Z%C3%A1sobn%C3%ADk_(datov%C3%A1_struktura)%5C#Z%C3%A1kladn%C3%AD_architektura_z%C3%A1sobn%C3%ADku.

[20] WATERMAN, Andrew a Krste ASANOVI. *The RISC-V Instruction Set Manual: Volume I: Unprivileged ISA* [online]. University of California, Berkeley, 2019 [cit. 2023-05-09]. Dostupné z: https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf.

[21] *RISC-V: Calling Convention* [online]. 2015. [cit. 2023-05-09]. Dostupné z: https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf.

[22] *Linux manual page: True* [online]. [cit. 2023-05-20]. Dostupné z: https://man7.org/linux/man-pages/man1/true.1.html.

[23] LAMBTRON. *ALU block.gif* [online]. San Francisco (CA): Wikimedia Foundation, 2014 [cit. 2023-05-19]. Dostupné z: https://commons.wikimedia.org/wiki/File:ALU_block.gif.

[24] HARRIS, Sarah L. a David Money HARRIS. *Digital design and computer architecture: RISC-V Edition*. Waltham, MA: Morgan Kaufman, 2021. ISBN 978-0-12-820064-3.

[25] *RISC-V GNU Compiler Toolchain* [online]. [cit. 2023-05-15]. Dostupné z: https://github.com/riscv-collab/riscv-gnu-toolchain.

[26] OSUPERDAVEO. *Ubuntu* [online]. [cit. 2023-05-20]. Dostupné z: https://www.urbandictionary.com/define.php?term=ubuntu.

Rejstřík instrukcí a pojmů