Experimentovanie s vlastnou inštrukčnou sadou

Less Instruction Machine (LIM)

Fakulta elektrotechniky a informatiky
Slovenskej technickej univerzity v Bratislave

Aplikovaná informatika

Ivan Hniedash
xhniedash@stuba.sk

14. decembra 2025

Abstrakt

Táto práca sa zaoberá návrhom a popisom vlastného procesora s architektúrou Limited Instruction Machine — architektúra procesora navrhnutého autorom (LIM) a jeho prvej implementácie LIM M1. Práca stručne a zrozumiteľne vysvetľuje princíp fungovania procesora, pričom väčší dôraz je kladený na jeho programovanie než na samotnú hardvérovú realizáciu.

Text postupne predstavuje základné vlastnosti architektúry LIM, jej inštrukčnú sadu a vlastný assemblerový jazyk navrhnutý špeciálne pre tento procesor. Súčasťou práce je aj vysvetlenie vykonávania inštrukcií na úrovni mikrokódu, čo umožňuje lepšie pochopiť vnútornú logiku procesora a spôsob spracovania dát.

V záverečnej časti sú uvedené praktické príklady algoritmov a programov napísaných v LIM Assembly, ktoré demonštrujú možnosti procesora a princípy jeho efektívneho programovania. Práca tak poskytuje ucelený úvod do návrhu aj programovania jednoduchého, no plne funkčného procesora.

Obsah

Zoznam obrázkov **4**

Zoznam tabuliek **5**

List of Abbreviations **6**

List of Algorithms **7**

List of Listings **8**

1 Aplikacie a simulatory **10**
1.1 Logisim . . . 10
1.2 LIM Assembly Compilator . . . 10
1.3 Odporúčané programy . . . 10

2 LIM architektúra **10**
2.1 Základné komponenty a smerovanie dát . . . 11
2.2 Smerovanie dát a základné princípy práce s údajmi . . . 11
2.2.1 Pseudoparalelizmus . . . 11
2.2.2 Spracovanie pamäte . . . 11
2.3 Vlastnosti aritmetiky a porovnávania . . . 12
2.4 Špecifiká programovania . . . 12
2.5 Kľúčové výhody architektúry . . . 12

3 Inštrukcie a Inštrukčná sada **12**
3.1 Inštrukčná sada . . . 12
3.1.1 Inštrukcie . . . 14
3.2 Príklad . . . 14

4 Vnútorná štruktúra procesora LIM **14**
4.1 Vnútorná štruktúra . . . 15
4.1.1 Prvky procesora . . . 15
4.1.2 Riadiaca jednotka . . . 16
4.2 Príklad mikroprogramu pre inštrukciu `ADD` . . . 17
4.3 Podporované operácie ALU . . . 18

5 Vývoj algoritmu (programu) **19**
5.1 Program na výpočet faktoriálu . . . 19
5.2 Sčítanie viacbajtových čísel . . . 20
5.3 Používanie podmienok a porovnaní . . . 22

Zoznam obrázkov

1 Tok údajov v procesore 11
2 Vnútorná štruktúra na výsokej úrovni 15
3 Vnútorná štruktúra Riadiacej jednotky 17

Zoznam tabuliek

1 Inštrukčná sada 13
2 Bitová štruktúra v inštrukcii 14
3 ADD mikroprogram 17
4 ALU funkcie 18

List of Abbreviations

ALU Aritmetická logická jednotka. 11, 12, 14, 17, 18, 24

CPU Central Processing Unit – centrálna procesorová jednotka. 9

HEX hexadecimálny (šestnástkový) číselný systém. 18

LIM Limited Instruction Machine — architektúra procesora navrhnutého autorom. 2, 3, 9, 10, 12, 14–18, 24

MUX multiplexor. 11

REG skratka pre register. 11, 17, 22

List of Algorithms

1 Ukážkový program na používanie podmienok a porovnaní 22

Listings

1 Lineárny program na výpočet faktoriálu 20
2 Sčítanie viacbajtových čísel 21
3 Ukážkový program na používanie podmienok a porovnaní 23

Úvod

Procesor, ako povedal Al Kzair, Januzi a Blom [1], známy aj ako Central Processing Unit – centrálna procesorová jednotka (CPU), je základná výpočtová jednotka každého počítača alebo elektronického zariadenia. Jeho úlohou je vykonávať inštrukcie programu, spracovávať dáta a riadiť komunikáciu medzi jednotlivými komponentmi systému, ako sú pamäť, vstupné a výstupné zariadenia.

Cieľom tejto práce je ukázať princíp fungovania procesora na jednoduchšom, transparentnom modeli vytvorenom autorom. Navrhnutý procesor s architektúrou LIM (Limited Instruction Machine) a označením LIM M1 je koncipovaný tak, aby bolo možné jednoznačne sledovať tok údajov, vykonávanie inštrukcií a riadenie operácií na úrovni mikrokódu.

Práca sa zameriava najmä na programovanie procesora, jeho inštrukčnú sadu a spôsob vykonávania programov v nízkoúrovňovom assemblerovom jazyku. Vysvetlenie hardvérovej štruktúry slúži predovšetkým ako prostriedok na lepšie pochopenie správania procesora pri vykonávaní algoritmov, nie ako cieľ sám o sebe.

V ďalších kapitolách je postupne predstavená architektúra LIM, vlastný assemblerový jazyk, vnútorná štruktúra procesora a princípy mikroprogramového riadenia. Záverečná časť práce obsahuje praktické príklady algoritmov, ktoré demonštrujú možnosti procesora a ilustrujú efektívne využitie jeho architektúry v praxi.

Aplikacie a simulatory V tejto sekcii sa nachádza základný zoznam potrebných programov na začatie práce s procesorom, ako aj zoznam odporúčaných programov na zrýchlenie a zjednodušenie práce.

LIM architektúra V tejto sekcii nájdete informácie o vlastnostiach architektúry, o jej možnostiach, ako aj sa oboznámite s jej základnými komponentmi.

Inštrukcie a Inštrukčná sada V tejto sekcii je opísaný autorom vyvinutý assemblerový jazyk a zoznam inštrukcií v assemblerovom jazyku aj v jeho strojovej podobe. Taktiež je tu predstavená štruktúra inštrukcie na úrovni strojového kódu.

Vnútorná štruktúra procesora LIM V tejto sekcii je podrobne rozobraný autorom vyvinutý procesor na hardvérovej úrovni s cieľom presnejšie pochopiť princíp jeho fungovania, čo pomôže programovať procesor efektívnejšie. Je tu opísaná fyzická štruktúra procesora, spôsob jeho fungovania na úrovni logiky, ako aj vysvetlené, čo je mikrokód (mikroinštrukcie), spolu s príkladmi inštrukcií vo forme mikrokódu.

Vývoj algoritmu (programu) Táto sekcia predstavuje praktickú časť práce; sú tu uvedené algoritmy možných úloh pre tento procesor a ich realizácia vo vlastnom assemblerovom jazyku s podrobným vysvetlením toho, čo sa deje.

1 Aplikacie a simulatory

Nižšie sú uvedené programy (aplikacie) nevyhnutné pre túto prácu, ako aj zoznam odporúčaných aplikácií na optimalizáciu práci.

1.1 Logisim

Na vytvorenie tohto procesora bol použitý simulátor logických obvodov **Logisim-Evolution** [9]. Toto je fork – vylepšená verzia pôvodného simulátora **Logisim** [2]. Na testovanie alebo simuláciu úloh na tomto procesore je táto aplikácia nevyhnutná.

1.2 LIM Assembly Compilator

LIM Assembly Compilator [6] je kompilátor strojového kódu napísaný autorom Hniedash [5]. Prijíma assemblerový kód a potom vytvára niekoľko súborov. Súčasťou je aj čistý binárny súbor pripravený na priame načítanie do pamäte, binárny zásobník príkazov a pripravený obraz pamäte pre simulátor Logisim. Algoritmus momentálne vyžaduje nainštalovanú najnovšiu verziu Python, alebo použite verziu programu kompilovanú v jazyku C. Sú dostupné pre Mac a Windows.

Tento kompilátor nie je povinný softvér, ale výrazne urýchľuje následnú prácu so strojovým kódom. Inak by písanie programov pre procesor zahŕňalo zapisovanie binárnych čísel do pamäte pomocou tabuľky.

1.3 Odporúčané programy

- **CST** alebo akýkoľvek program na rýchly prevod čísel medzi rôznymi číselnými sústavami. Je odporúčané použiť konzolovú aplikáciu, ktorá je napísana autorom Hniedash [5] — **CST** [4]. Vyvinutá pre rýchlu prácu s procesorom pred napísaním kompilátora. Program je stále užitočný, ak chcete upraviť inštrukcie alebo pridať vlastné mikroinštrukcie.
- **Visual Studio Code** [8] je praktický editor kódu so vstavaným terminálom. Je ideálny na rýchle a jednoduché písanie programov a ich kompiláciu. Je to vynikajúca voľba pre prácu s vlastným kompilátorom.

2 LIM architektúra

Procesor a architektúrou LIM nevyužíva konvejerové spracovanie. Každá inštrukcia sa vykonáva ako jeden ucelený krok, pričom ďalšia inštrukcia sa začne spracúvať až po dokončení predchádzajúcej. Výsledky výpočtu sú preto dostupné ihneď po ukončení vykonania inštrukcie.

Procesor je postavený na princípe „od začiatku do konca“: každá operácia sa vykoná úplne bez čakania na dáta z externých zdrojov alebo medziľahlých vyrovnávacích pamätí. Na rozdiel od pipeline procesorov to zjednodušuje programovanie a eliminuje typické oneskorenia spojené s čakaním na dáta medzi fázami.

- Nazov architektúry: Limited Instruction Machine (LIM)
- Názov procesora: LIM M1
- Typ: 8-bitový nonpipeline (sekvenčný) procesor s matematickými výpočtami n-bitovými číslami

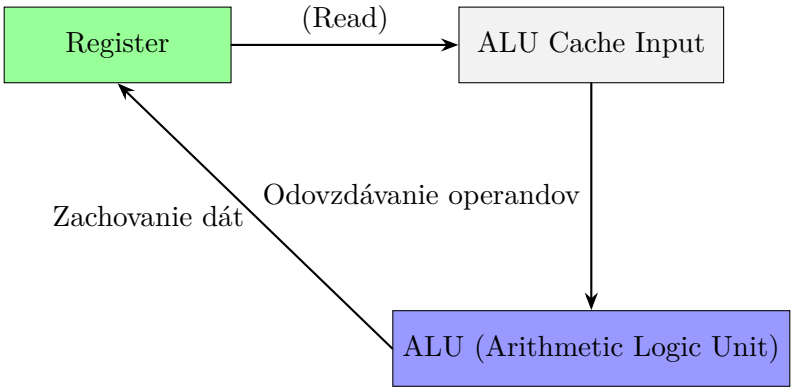
2.1 Základné komponenty a smerovanie dát

- **Registre**: skratka pre register (REG). „Registre sú súčasťou procesora, ktoré uchovávajú hodnoty (napätia). Tieto registre obsahujú malé množstvo rýchlej pamäte, ktorá uchováva hodnoty uložené v nich. Vďaka tomu sú registre jedinečné v tom, že budú súčasťou dôležitých inštrukcií“ – Al Kzair, Januzi a Blom [1]. Štyri registre na všeobecné použitie, používané na ukladanie operandov a výsledkov.
- **ALU**: Aritmetická logická jednotka (ALU), ktorá podporuje sčítanie, odčítanie, násobenie a delenie. Má vstavaný „kreditový systém“ na prenos/požičiavanie si pri práci s číslami presahujúcimi štandardnú bitovú hĺbku. Je navrhnutá tak, aby práca s extrémne veľkými číslami nebola problémom.
- **ALU cache**: Dve interné cache určené na ukladanie operandov pred spracovaním, čo pomáha minimalizovať čas prístupu a organizovať pseudoparalelné výpočty.
- **Riadiaca jednotka**: riadi postupnosť mikroinštrukcií a obsahuje špeciálny blok – „Dátový mostík“, ktorý vybera medzi internými a externými dátami na spracovanie, ALU a vyrovnávacími pamäťami bez použitia spoločnej internej zbernice.
- **Multiplexory**: multiplexor (MUX) Umožňujú výber zdrojov údajov pre ALU a riadiacu jednotku, poskytujú flexibilné smerovanie a možnosť priameho obídenia registrov, čím sa šetrí čas.

2.2 Smerovanie dát a základné princípy práce s údajmi

Údaje sa pohybujú v cykle: Dáta sa presúvajú cyklicky. Tabuľka 1 popísuje ten to proces. To zjednodušuje logiku procesora a umožňuje spoľahlivé výpočty. Táto organizácia eliminuje potrebu vyrovnávacích pamätí pre prerušenie reťazca a udržiava vysokú rýchlosť spracovania.

Obr. 1: Tok údajov v procesore



2.2.1 Pseudoparalelizmus

Izoláciou požičiavania–pretečenia operácií násobenia–delenia a súčtu–rozdielu je možné operácie logicky vykonávať „bezprostredne za sebou“ bez vzájomného rušenia, čím sa zrýchľuje práca s veľkými číslami. Na prerušenie „kontextu“ vykonávania musíte použiť inštrukciu `NOP`.

2.2.2 Spracovanie pamäte

Procesor podporuje načítavanie údajov z pamäte a ich spätný zápis do pamäte s možnosťami podmieneného vetvenia a riadenia toku.

2.3 Vlastnosti aritmetiky a porovnávania

- **Veľké čísla**: Procesor je navrhnutý tak, aby pracoval s diskrétnymi prirodzenými číslami s prakticky neobmedzenou bitovou hĺbkou, vrátane sčítania, odčítania, násobenia a delenia.
- **Pokročilé možnosti porovnávania**: Procesor dokáže vykonávať priame porovnávania (a > b, a < b) a nepriame porovnávania (a ? b → <, >, =), čo je obzvlášť užitočné pre matematické dôkazy a prácu s abstraktnými numerickými štruktúrami.
- **Systém kreditov ALU**: Interné spúšťače umožňujú akumuláciu prenosov-požičiavania medzi číslicami počas sčítania-odčítania, ako aj korekciu čiastočných výsledkov násobenia a delenia bez potreby externej synchronizácie.

2.4 Špecifiká programovania

- Jednoduchý a deterministický model vykonávania, kde sa každá operácia vykonáva do konca, ako uvedené (pozr. diagram 1)
- Nie je potrebné vkladať NOP ani čakať na údaje.
- Mikroinštrukcie sú postavené na koncepcii „získanie údajov → vykonanie operácie → potvrdenie výsledku“.

2.5 Kľúčové výhody architektúry

- Podpora prakticky nekonečných čísel je jedinečnou vlastnosťou 8-bitových procesorov.
- Pokročilé porovnávacie možnosti pre zložité matematické a vedecké výpočty.
- Jednoduchosť programovania a predvídateľnosť vykonávania vďaka kompletnému operačnému modelu bez pipeline.
- Absencia spoločnej internej zbernice znižuje latenciu a zjednodušuje logiku smerovania.
- Flexibilné smerovanie dát cez riadiaci mostík a multiplexory.

3 Inštrukcie a Inštrukčná sada

Nasleduje tabuľka príkazov assembleru. Tabuľka bude obsahovať samotné príkazy, ich popisy, ich strojový kód v hexadecimálnom formáte a výslednú cestu. Ako už bolo spomenuté, pre pohodlie odporúčam použiť program na rýchly preklad z rôznych výpočtových systémov [4, CST].

3.1 Inštrukčná sada

Ako uvedel Gašparovič [3] vo svojej praci – Assembly je rodným jazykom procesora. Procesor chápe assembly ako postupnosť núl a jednotiek. Túto formu nazývame strojový kód. Pretože vytvára programy priamo v jednej z číselných sústav (binárnej, desiatkovej alebo šestnástkovej). Syntax jazyka vyvinutého pre architektúru LIM je popísaná v tabuľke 1

Tabuľka 1: Inštrukčná sada

`Operačný Kód`	**Mnemotech.**	**Popis / Operácia**	**Výsledok**
Aritmetické operácie			
`0x0`	NOP	Bez operácie (No Operation) Vymaže pôžičky/prekročenia	—
`0x1`	ADD	Sčítanie: RG1 + RG2	→ R4 (ACC)
`0x2`	SUB	Odčítanie: RG1 - RG2	→ R4 (ACC)
`0x3`	MUL	Násobenie: RG1 * RG2	→ R4 (ACC)
`0x4`	DIV	Delenie: RG1 / RG2	→ R4 (ACC)
Operácie s pamäťou a registrami			
`0x5`	LMR	Načítanie (Load) z PAMÄTE(R0) do R1 (podľa R2 bytov)	→ R4 (ACC)
`0x6`	LRM	Uloženie (Save) dát z R1 do PAMÄTE(R2)	→ R4 (ACC)
`0x7`	LRR	Načítanie (Load) dát z R1 do R2	→ R4 (ACC)
Operácie porovnávania			
`0x8`	CMP	Porovnanie R1 a R2	→ R4 (ACC)
`0x9`	EQL	Sú R1 a R2 rovné? (Are R1 and R2 Equal?)	→ R4 (ACC)
`0xA`	GRT	Je R1 väčšie ako R2? (Is R1 more than R2)	→ R4 (ACC)
Logické operácie			
`0xB`	AND	Logické A (Logic AND)	→ R4 (ACC)
`0xC`	OR	Logické ALEBO (Logic OR)	→ R4 (ACC)
`0xD`	NOT	Logické NIE (Logic NOT)	→ R4 (ACC)
`0xE`	XOR	Exkluzívne ALEBO (Logic XOR)	→ R4 (ACC)
Skok (Riadenie toku)			
`0xF`	JMP	Skok (Jump) na adresu v R1, ak je R2 pravdivé	—

3.1.1 Inštrukcie

Inštrukcia predstavuje základnú jednotku programu, ktorú procesor dokáže priamo vykonať. Každá inštrukcia určuje konkrétnu operáciu, ktorú má procesor vykonať, spolu s informáciami o tom, kde sa nachádzajú vstupné údaje (operandy) a kam sa má uložiť výsledok.

Z hľadiska architektúry procesora možno inštrukciu chápať ako binárny kód, ktorý riadi činnosť vnútorných funkčných blokov — aritmeticko-logickej jednotky (ALUs), registrov, riadiacej logiky a pamäte. Procesor teda interpretuje jednotlivé bity inštrukcie ako súbor signálov, ktoré určujú, aké vnútorné operácie sa majú vykonať a v akom poradí.

Každá inštrukcia v procese má dĺžku 8 bitov a skladá sa z troch častí:

$$\text{inštrukcia} = \underbrace{xxxx}_{\text{OP-kód}} \quad \underbrace{yy}_{\text{operánd 1}} \quad \underbrace{zz}_{\text{operánd 2}} \tag{1}$$

kde:

- **OP-kód** — štyri najvyššie bity (bity 7–4), ktoré určujú, aký typ operácie má byť vykonaný,
- **Operánd 1** — dva stredné bity (bity 3–2), ktoré špecifikujú prvý register alebo vstupný operand,
- **Operánd 2** — dva najnižšie bity (bity 1–0), ktoré definujú druhý register alebo operand.

Tento formát sa označuje ako `xxxxyyzz`, tento formát je uvedený v tabuľke 1, kde každá skupina bitov má presne určený význam. Vďaka pevnej dĺžke inštrukcie (8 bitov) je dekódovanie jednoduché a môže byť realizované priamo hardvérovo — každá časť inštrukcie je priamo priradená k vstupom dekódera, ktorý následne aktivuje príslušnú mikroinštrukciu v riadiacej jednotke. Tabuľka 2 podrobnejšie vysvetľuje význam každej časti inštrukcie.

Tabuľka 2: Bitová štruktúra v inštrukcii

Bity	Pole	Popis
7–4	OP-kód	Typ operácie (napr. ADD, SUB, AND, OR)
3–2	Operánd 1	Číslo prvého registra (R_1)
1–0	Operánd 2	Číslo druhého registra (R_2)

3.2 Príklad

$$0001\,01\,10_2 \Rightarrow \text{OP-kód} = 0001 \text{ (ADD)}, \quad R_1 = 01 \text{ (R1)}, \quad R_2 = 10 \text{ (R2)}$$

Teda inštrukcia 00010110_2 zodpovedá operácii `ADD R1, R2`.

4 Vnútorná štruktúra procesora LIM

Na rozdiel od vysokoúrovňových programovacích jazykov, ako sú **Python**, **C**, **C#**, **Java** alebo **Lua**, **assembler** nie je univerzálny jazyk.

Predstavuje *rodinu nízkoúrovňových jazykov*, z ktorých každý je úzko spätý so štruktúrou konkrétneho procesora. Preto sa syntax a množina dostupných inštrukcií môžu výrazne líšiť nielen medzi rôznymi architektúrami (napríklad medzi **x86** a **ARM**), ale aj medzi procesormi tej istej architektúry.

Napríklad assembler pre **Intel 8086** a pre moderné procesory **Intel Core** formálne patria do architektúry **x86**, no ich množina inštrukcií, spôsoby adresovania, podporované registre a mikropokyny sa výrazne líšia.

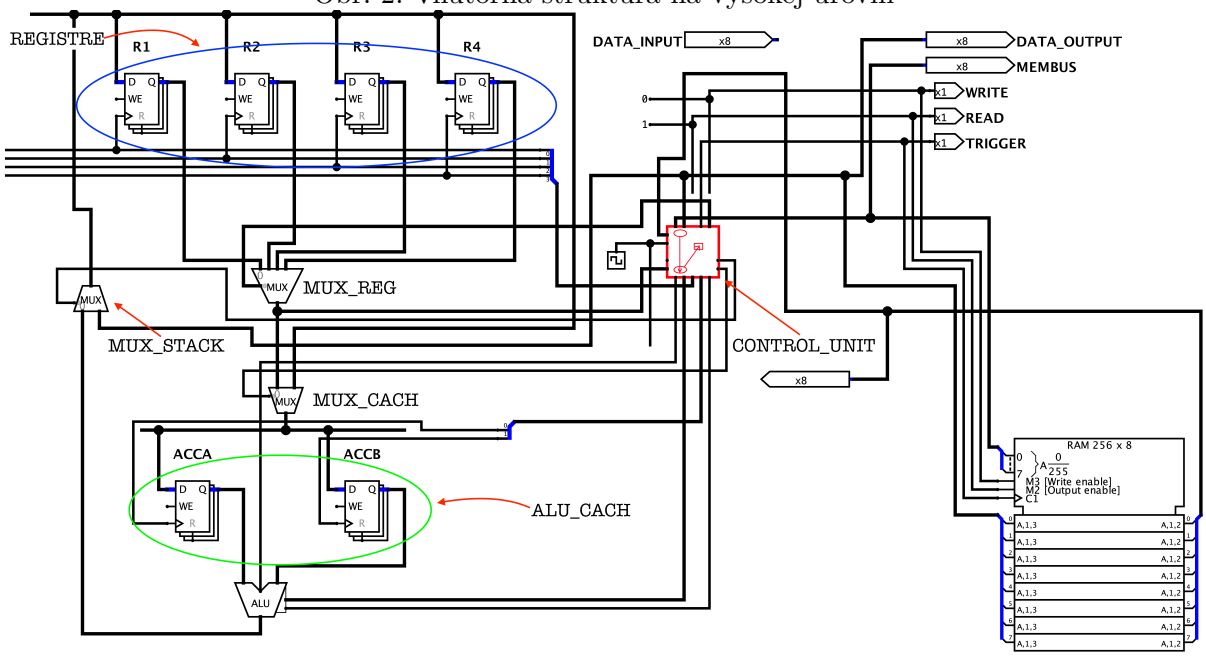
Moderné procesory obsahujú rozšírené sady inštrukcií (ako **MMX**, **SSE**, **AVX** a ďalšie) a odlišnú mikroarchitektúru, čo robí ich assemblerové programovanie omnoho zložitejším, ale zároveň funkčne bohatším v porovnaní s prvými procesormi Intel.

Preto, aby bolo možné písať algoritmy/programy na procesore LIM, je potrebné presne vedieť, ako funguje, aké sú jeho výhody a nevýhody, aby bolo možné správne a efektívne spravovať dáta, pričom je potrebné zohľadniť možné nepríjemnosti spojené s nedokonalou domácou architektúrou procesora.

4.1 Vnútorná štruktúra

Na obrázku 2 vidno, že štruktúra je trochu preplnená kvôli rozhodnutiu opustiť zdieľanú dátovú zbernicu. To uľahčuje konštrukciu riadiacej jednotky. V niektorých prípadoch je užitočné mať izolovaný tok údajov, čo, ako už bolo spomenuté, znižuje počet operácií procesora. V tejto časti

Obr. 2: Vnútorná štruktúra na výsokej úrovni



sa pozrieme na vnútornú štruktúru procesora na vysokej úrovni a tiež, aby sme pochopili, ako fungujú inštrukcie, sa pozrieme na to, ako funguje ovládač.

4.1.1 Prvky procesora

Na schéme sa stretávajú nasledujúce komponenty:

- **Registre** – Sú ich iba 4 (R1, R2, R3, R4), ale ide o pomerne optimálne riešenie medzi výrazným skomplikovaním procesora a optimálnou prevádzkovou rýchlosťou a jednoduchosťou. Ich účelom je ukladať informácie potrebné pre aktuálne výpočty, či už ide o operandy alebo výsledky predchádzajúcich výpočtov, alebo to môžu byť aj pamäťové adresy.
- **Multiplexory** – Veľmi dôležitá časť procesora, ktorá nahrádza spoločnú zbernicu. V procesore založenom na architektúre LIM sú tri, ale hlavné sú dve: **MUX_REG** a **MUX_STACK**. **MUX_ACC** je záložný multiplexor určený pre ďalšie vylepšenia, jeho

hlavnou myšlienkou je používať dáta priamo z pamäte, v budúcnosti by to malo zrýchliť viacúrovňové výpočty, aby sa nestrácal čas prípravou dát

- **ALU Cache** – Ide o špeciálne registre, ktoré ukladajú dáta pre bezprostrednú matematickú alebo logickú operáciu. Nedajú sa ovládať manuálne; sú prístupné iba na úrovni mikroprogramu, zapísanej pomocou mikroinštrukcií.
- **ALU** – Aritmetická logická jednotka. Jeho úlohou je vykonávať matematické operácie ako sčítanie, odčítanie, násobenie, delenie vrátane porovnávania, ako aj logické operácie ako AND, OR, NOT, XOR. Viac podrobností o možnostiach bude prediskutovaných neskôr: (Tabuľka 4). Keďže ALU má vlastný mechanizmus pre požičiavanie a detekciu pretečenia, po dokončení matematických operácií v jednom kontexte je potrebné použiť inštrukciu NOP, ktorá vymaže príznaky pôžičky a pretečenia.
- **Riadiaca jednotka** – Toto je „mozog“ procesora. Často prijíma inštrukcie a potom na základe mikroprogramov vložených do týchto inštrukcií riadi komponenty procesora, tak aby sa dosiahol požadovaný výsledok. Nasledujúci diagram 3 je zjednodušený diagram nášho ovládača.

4.1.2 Riadiaca jednotka

Schéma zapojenia ovládača zobrazuje počítadlo pamäte a veľmi podobné počítadlo mikroinštrukcií. Jediný rozdiel je v konkrétnej pamäti, ktorú ovládajú.

Za zváženie stojí tri hlavné komponenty riadiacej jednotky, pretože zvyšok je iba výkonná časť, ktorá si nestojí za veľkú pozornosť, pretože je špecifická pre každý iný čip, a preto nemá zmysel sa na ne zaoberať, najmä preto, že priamo neovplyvňujú princíp programovania.

- **Ovládač programovej pamäte** (Na obrázku 3 zvýraznený červenou farbou). Jeho úlohou je dávať inštrukcie procesoru. Najčastejšie je jeho úlohou iterovať pamäťou, vykonávať inštrukcie jednu po druhej, pričom s každou inštrukciou jednoducho zvyšuje adresu pamäťovej bunky o 1.
- **Riadiaca jednotka registra mikroinštrukcií** (Na obrázku 3 zvýraznené modrou farbou) Jeho úloha je takmer rovnaká ako v Ovládači programovej pamäte, len prechádza mikroinštrukciami namiesto bežných inštrukcií.
- **Dekodér mikroinštrukcií** (Na obrázku 3 zvýraznené fialovou farbou) Vysiela elektrický signál na určitom vedení, čím dáva signál požadovanej časti ovládača na vykonanie nejakej jednoduchej úlohy. Napríklad: prepnutie čítacieho registra alebo uloženie dát do vyrovnávacej ALU Cache.
- **Register mikroinštrukcií** Uchováva (Na obrázku 3 zvýraznené zelenou farbou) inštrukčné mikroprogramy a počas prevádzky procesora po prijatí inštrukcie radič mikroinštrukcií presmeruje vykonávanie mikroprogramu na ten, ktorý zodpovedá požadovanej inštrukcii, až po špeciálny koncový príznak `0xF0`

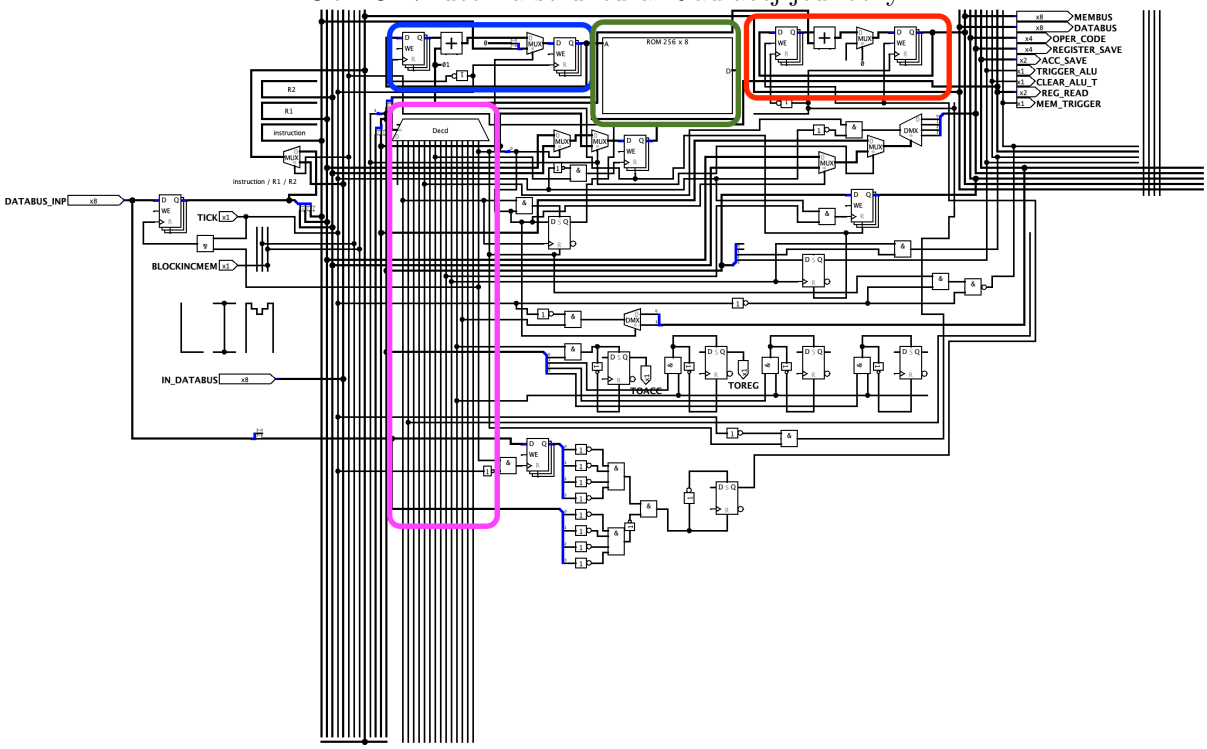
Každá bežná inštrukcia procesora sa dá považovať za *zloženú funkciu*, ktorá sa vo vnútri procesora rozkladá na viacero menších krokov nazývaných **mikroinštrukcie**.

Tieto mikroinštrukcie predstavujú najjednoduchšie, ďalej nedeliteľné operácie, ktoré procesor vykonáva priamo na úrovni svojich vnútorných registrov, zberníc a riadiacich signálov.

Zjednodušene povedané, každá makroinštrukcia (napr. `ADD R1, R2`) je v mikrokóde reprezentovaná ako sekvencia mikroinštrukcií, ktoré postupne zabezpečujú načítanie operandov, vykonanie aritmetickej operácie a zápis výsledku späť do registra.

Týmto spôsobom je možné vytvoriť flexibilný a univerzálny riadiaci mechanizmus, v ktorom sa správanie procesora dá meniť alebo rozširovať úpravou jeho mikroprogramu bez potreby meniť fyzickú logiku obvodov.

Obr. 3: Vnútorná štruktúra Riadiacej jednotky



4.2 Príklad mikroprogramu pre inštrukciu ADD

Nasledujúca tabuľka 3 znázorňuje mikroprogram pre inštrukciu `ADD R1, R2`, ktorá zabezpečuje sčítanie dvoch registrov a uloženie výsledku do akumulátora (`REG3`). Každý krok je reprezentovaný mikroinštrukciou, zapísanou v hexadecimálnej forme, a stručne opísaný jej účel. Každý

Tabuľka 3: ADD mikroprogram

Kód (HEX)	Binárny tvar	Označenie	Popis činnosti
`0x70`	`0111 00 00`	Čítanie #1	Načíta hodnotu registra určeného prvým parametrom inštrukcie (`REG1`)
`0xC0`	`1100 00 00`	Cache #1	Uloží načítanú hodnotu do prvého dočasného registra (cache #1)
`0x71`	`0111 00 01`	Čítanie #2	Načíta hodnotu registra určeného druhým parametrom (`REG2`)
`0xC1`	`1100 00 01`	Cache #2	Uloží načítanú hodnotu do druhého dočasného registra (cache #2)
`0x90`	`1001 00 00`	Režim ALU	Nastaví ALU do režimu sčítania
`0xA8`	`1010 1 000`	Trigger ALU	Spustí výpočet v ALU na základe údajov v cache
`0x53`	`0101 00 11`	Zápis do `REG3`	Zapíše výsledok operácie do akumulátora
`0xA0`	`1010 00 00`	Uzavretie ALU	Ukončí činnosť ALU a obnoví neutrálne riadiace signály
`0xF0`	`1111 0000`	Koniec	Označuje ukončenie sekvencie mikroinštrukcií

binárny kód určuje konkrétnu operáciu ALU a slúži na jej aktiváciu prostredníctvom mikroinštrukcie. hexadecimálny (šestnástkový) číselný systéms (HEXs) kód je uvedený pre rýchlu referenciu a praktické využitie pri mikroprogramovaní.

Tento mikroprogram reprezentuje kompletný vnútorný priebeh vykonania inštrukcie `ADD`, od načítania operandov až po uloženie výsledku do akumulátora. Každá mikroinštrukcia zodpovedá presne definovanej sade riadiacich signálov v mikroinštrukčnom registri procesora.

4.3 Podporované operácie ALU

Nasledujúca tabuľka 4 uvádza všetky operácie, ktoré ALU procesora dokáže vykonať, spolu s ich binárnym a hexadecimálnym kódom.

Tabuľka 4: ALU funkcie

Binárny kód	**HEX**	**Operácia**
0000	0x0	Sčítanie (+)
0001	0x1	Odčítanie (-)
0010	0x2	Násobenie (*)
0011	0x3	Delenie (/)
0100	0x4	Porovnanie: =, <, > (príklad: 0x1, 0x2, 0x3)
0101	0x5	Rovnosť (==)
0110	0x6	Väčšie (>)
1000	0x8	Logické AND
1001	0x9	Logické OR
1010	0xA	Logické XOR
1011	0xB	Logické NOT

5 Vývoj algoritmu (programu)

V tejto sekcii sú predstavené viaceré príklady programov napísané v nízkoúrovňovom jazyku navrhnutom pre vlastný procesor. Každý z nich demonštruje praktické využitie definovaných inštrukcií a spôsob, akým možno pomocou jednoduchých registrových operácií realizovať rôzne výpočtové úlohy.

Programy sú zostavené bez použitia riadiacich štruktúr, ako sú cykly alebo podmienené vetvenie, a teda reprezentujú čisto lineárny prístup k vykonávaniu inštrukcií. Cieľom tejto časti je ukázať princíp práce procesora na úrovni jednotlivých inštrukcií a mikrokrokov, ako aj spôsob, akým možno pomocou týchto základných operácií vytvoriť kompletné výpočtové postupy.

5.1 Program na výpočet faktoriálu

Nasledujúci program 1 predstavuje jednoduchý príklad lineárnej implementácie výpočtu faktoriálu čísla uloženého v registri `R0`. Program slúži na ilustráciu základných princípov práce s registrami a aritmetickými operáciami, bez použitia cyklov alebo príkazov podmieneného skoku. Predpokladá sa, že registre `R1` (počitadlo) a `R3` (akumulátor – výsledok) sú na začiatku inicializované hodnotou `1`.

Princíp výpočtu spočíva v postupnom násobení akumulátora `R3` aktuálnym číslom v registri `R1` a v inkrementácii registra `R1` o jednotku po každom kroku. Program je zapísaný ako sekvencia jednotlivých inštrukcií, aby bol prehľadný a vhodný na demonštráciu fungovania procesora.

Výsledná hodnota faktoriálu je uložená v registri `R3` po ukončení programu.

Listing 1: Lineárny program na výpočet faktoriálu

```
; Step 1: R3 = R3 * R1
MUL R3, R1        ; ACC = 1
ADD R1, R0        ; increment counter

; Step 2: R3 = R3 * R1
MUL R3, R1        ; ACC = 2
ADD R1, R0        ; increment counter

; Step 3: R3 = R3 * R1
MUL R3, R1        ; ACC = 6
ADD R1, R0        ; increment counter

; Step 4: R3 = R3 * R1
MUL R3, R1        ; ACC = 24

; End of program
NOP               ; factorial result is now in R3
```

5.2 Sčítanie viacbajtových čísel

Nasledujúci program 2 demonštruje princíp postupného sčítania dvoch viacbajtových čísel pomocou registrového procesora. Keďže architektúra umožňuje pracovať iba s obmedzeným počtom registrov a každá inštrukcia spracúva len časť dát, sčítanie sa vykonáva po jednotlivých častiach (napríklad po bajtoch) a výsledok sa postupne ukladá do pamäte.

Program ilustruje:

- Postupné načítavanie častí operandov z pamäti do registrov,
- Aritmetické sčítanie,
- Postupné ukladaní výsledku späť do pamäti,
- Lineárnu štruktúru programu bez cyklov a podmienených skokov.

Takýto lineárny prístup je vhodný na demonstráciu práce procesora s väčšími číslami a ukazuje základné princípy manipulácie s dátami na mikroúrovni.

Program načítava **najmladšie bity** (0–7) operandov, vykoná ich sčítanie a výsledok uloží na konkrétnu adresu v pamäti. V kontexte predloženej implementácie sa tento postup opakuje ešte dvakrát pre ďalšie časti čísel:

- Najmladšie bity (0–7) sa sčítajú a výsledok sa uloží na adresu `0xFA`.
- Nasledujúce bity (8–15) sa sčítajú a uloží sa výsledok na adresu `0xFB`.
- Najvyššie bity (16–23) sa sčítajú a uloží sa výsledok na adresu `0xFC`.

Konkrétny príklad:

- Číslo `A` = 00000001 00000001 00000001
- Číslo `B` = `A`

Postup sčítania:

Listing 2: Sčítanie viacbajtových čísel

```
1  LMR 0 1 ;Load next number to register
2  0x01
3  LMR 1 1 ;Load next number to register
4  0x01
5  ADD 0 1 ;Sum of REG1 and REG2
6  LMR 2 1
7  0xFA
8  LRM 3 2 ;Save result to MEM by REG2 pointer
9
10 ;Sum of less significant bits
11
12 LMR 0 1 ;Load next number to register
13 0x01
14 LMR 1 1 ;Load next number to register
15 0x01
16 ADD 0 1 ;Summ of REG1 and REG2
17 LMR 2 1
18 0xFB
19 LRM 3 2 ;Save result to MEM by REG2 pointer
20
21 ;Sum of more significant bits
22
23 LMR 0 1 ;Load next number to register
24 0x01
25 LMR 1 1 ;Load next number to register
26 0x01
27 ADD 0 1 ;Summ of REG1 and REG2
28 LMR 2 1
29 0xFC
30 LRM 3 2 ;Save result to MEM by REG2 pointer
31
32 ;Sum of more significant bits
33
34 NOP      ;End of program. Clear borrowings
```

1. `A(0-7) + B(0-7)` → uložené do `0xFA`
2. `A(8-15) + B(8-15)` → uložené do `0xFB`
3. `A(16-23) + B(16-23)` → uložené do `0xFC`

Výsledok sčítania:

`A + B` = 00000010 00000010 00000010

Tento postup ilustruje princíp postupného sčítania viacbajtových čísel, kde každá časť sa spracúva samostatne a výsledok sa ukladá do pamäťových buniek po častiach.

Po tomto kroku je dôležité použiť inštrukciu `NOP`, ktorá vymaže príznaky pôžičky a pretečenia (borrow a overflow). Tento príklad platí rovnako aj pre ostatné matematické operácie.

Existuje však dôležitý moment pri spracovaní veľkých čísel: ak výsledok súčtu trojbajtového čísla môže prekročiť 3 bajty (teda stať sa 4-bajtovým), stačí v prípade pretečenia jednoducho sčítať dve nuly (`0 + 0`) — tým sa automaticky zohľadní pretečenie.

Podobne to funguje pri násobení: ak sa vynásobí (`1 * 1`) v najvyšších bajtoch, pretečenie sa takisto správne zohľadní.

Pri odčítaní alebo delení takéto riešenie nie je možné ani logicky, ani matematicky, pretože vznik pretečenia alebo pôžičky sa musí riešiť priamo v rámci algoritmu operácie.

5.3 Používanie podmienok a porovnaní

Nasledujúci program 1 bude príkladom toho, ako používať inštrukcie pre rôzne testy, či už ide o `<, >, =, !=` a boolovské funkcie `AND, OR, NOT` a ďalšie.

V nasledujúcom programe 3, ak je číslo v registri REG 3 menšie ako zadané číslo v registri REG 2, toto číslo sa zvýši o 1. Napríklad, toto číslo by bolo 10. Ak je registr REG 3 10 alebo vyššie, program bude pridávať 10, kým nedosiahne 100. Keď sa dosiahne 100, program sa začne odznova.

Na implementáciu tohto programu je potrebné načítať hlavné hodnoty do registrov: REG 3 = 0, REG 2 = 10, REG 1 = 0, REG 0 = 1. Pre tento algoritmus je REG 3 počítadlo. REG 2 je prvý limit. REG 1 jednoducho vymažeme, pretože ho budeme potrebovať počas vykonávania algoritmu. REG 0 uchováva iterátor. Nasledujúci algoritmus 1 je pohodlnou reprezentáciou toho, čo je napísané:

Algorithm 1 Ukážkový program na používanie podmienok a porovnaní

```
if i < 10 then
    i ← i + 1
else
    while i < 100 do
        i ← i + 10
    end while
    i ← 0
end if
```

Ďalej bude tento algoritmus prepísaný v assembleri pre našu architektúru.

Listing 3: Ukážkový program na používanie podmienok a porovnaní

```
LMR 1 1      ;R1 = 0x0
0x0          ;0x0 (HEX) == 0 (DEC)
LMR 2 1      ;R2 = 0x0
0xA          ;0xA (HEX) == 10 (DEC)
LMR 3 1      ;R3 = 0x0
0x0          ;0x0 (HEX) == 0 (DEC)

;---Ked cislo je mensie za 10---
LMR 0 1      ;riadok programu je 9 (0x9), R1 = 0x1
0x1          ;0x1 (HEX) == 1 (DEC)
ADD 0 1      ;R0 + R1 => R3
LRR 3 1      ;R1 = R3
GRT 2 1      ;R2 > R1 => R3
LMR 0 1      ;R0 = 0x9
0x9          ;0x9 (HEX) == 9 (DEC)
JMP 0 3      ;Program prejde na 9 riadok ak R3 = true (1)
             ;Ak nieje R3 = true processor ide dalej po pamati

;---Ked 10 < cislo < 100
LMR 2 1      ;R1 = 0x64
0x64         ;0x64 HEX = 100 DEC
LMR 0 1      ;riadok programu je 22 (0x16), R0 = 0xA
0xA          ;0xA (HEX) == 10 (DEC)
ADD 0 1      ;R0 + R1 => R3
LRR 3 1      ;R1 = R3
GRT 2 1      ;R2 > R1 => R3
LMR 0 1      ;R0 = 0x16
0x16         ;0x16 (HEX) == 22 (DEC)
JMP 0 3      ;Skokne na 22 riadok programu

;--- Vrati sa na zaciatok ---
LMR 0 1      ;R0 = 0x1
0x1          ;0x1 (HEX) == 1 (DEC)
JMP 0 0      ;Skokne na 1 riadok programu (do zaciatku)
```

Záver

V tejto práci bola podrobne predstavená architektúra vlastného procesora **LIM**, ktorú vyvinul autor, jeho vnútorná štruktúra, súbor inštrukcií a princípy práce s mikrokódom. Ukázalo sa, že každá inštrukcia procesora kombináciu OP-kódu a operandov, ktoré sú hardvérovo interpretované prostredníctvom registrov, ALUs a riadiacich signálov. Na konkrétnych príkladoch — výpočet faktoriálu, sčítanie viacbajtových čísel a podmieňove ulohy. — boli demonštrované princípy lineárneho a cyklického vykonávania programov, interakcia registrov a cache dát, ako aj spracovanie aritmetických operácií na úrovni mikrokódu.

Vytvorenie vlastnej architektúry **LIM** umožnilo autorovi detailne pochopiť vnútorné fungovanie procesora, mechanizmus vykonávania inštrukcií a princíp mikroinštrukcií. Táto práca sa zameriava výhradne na programovú časť systému, čo umožňuje porozumieť fungovaniu procesora bez potreby hlbokých znalostí každého fyzického prvku alebo celej architektúry. Od začiatku bola najprv vyvíjaná softvérová časť procesora a až neskôr sa môže pristúpiť k jej implementácii v reálnej fyzickej podobe. Takýto prístup poskytuje prehľadný a praktický základ pre pochopenie princípov nízkoúrovňového programovania a flexibilného mikroprogramového riadenia, pričom efektivita algoritmov závisí od správneho využitia registrov, ALU a riadiacich signálov. Možná budúca práca by sa potom mohla sústrediť na elektronické a logické schémy procesora až na úrovni tranzistorov.

Pre tých, ktorí by chceli experimentovať s architektúrou **LIM** aj prakticky, je dostupná najnovšia verzia návrhu procesora vo forme súboru pre simulátor Logisim. Tento súbor je zverejnený na GitHube [7] a je možné si ho stiahnuť, otvoriť v Logisim a voľne experimentovať so zapojením, testovať mikroinštrukcie či vytvárať vlastné modifikácie procesora.

Referencie

[1] Christian Al Kzair, Altin Januzi a Andreas Blom. „Understanding the fundamentals of CPU architecture: Bachelor project in Electrical engineering“. Diz. pr. 2018. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-353427.

[2] Carl Burch. *Logisim*. Logisim Official webpage. 2014. URL: https://cburch.com/logisim/ (cit. 10. 11. 2025).

[3] Peter Gašparovič. „Assembler“. In: (2018). Peter Gašparovič - Assembler. URL: https://uim.fei.stuba.sk/wp-content/uploads/2018/02/Uvod_Asembler-1.pdf (cit. 10. 11. 2025).

[4] Ivan Hniedash. *Count System Transformer (CST)*. CSM Source code and compiled applications. 2025. URL: https://github.com/ivan365/CST (cit. 10. 11. 2025).

[5] Ivan Hniedash. *GitHub (ivan365)*. GitHub. 2025. URL: https://github.com/ivan365 (cit. 10. 11. 2025).

[6] Ivan Hniedash. *LIM ASM Compiler*. LIM ASM Compiler — Source code. 2025. URL: https://github.com/ivan365/LIMASM (cit. 10. 11. 2025).

[7] Ivan Hniedash. *LIM CPU M1*. Link to CPU circuit. 2025. URL: https://github.com/ivan365/LIM_CPU_M1 (cit. 10. 11. 2025).

[8] Microsoft Inc. *Visual Studio Code*. Link to Official webpage. 2025. URL: https://code.visualstudio.com (cit. 10. 11. 2025).

[9] Dr. Theo Kluter. *Logisim-Evolution*. BFH-ktt1 GitHub. 2025. URL: https://github.com/logisim-evolution/logisim-evolution (cit. 10. 11. 2025).

AI

Každá časť tejto práce bola gramaticky skontrolovaná pomocou ChatGPT.