UNIVERSIDADE FEDERAL DE LAVRAS

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Relatório Técnico

Simulador Funcional do Processador UFLA-RISC

Grupo 6

Clarisse Lacerda Pimentel
Daniel Silva Ferraz Neto
Helder Jose Avila
José Victor Miranda de Oliveira
Marco Tulio Franco Silva

Disciplina: Arquitetura de Computadores II (GCC123/PCC507)
Professor: Luiz Henrique A. Correia
Semestre: 2º/2025

Lavras, MG
Novembro de 2025

Sumário

1 Resumo Executivo

Este relatório descreve a implementação de um simulador funcional para o processador RISC de 32 bits **UFLA-RISC**, desenvolvido em Python 3.8+.

1.1 Características implementadas

- Pipeline de 4 estágios (IF, ID, EX/MEM, WB)
- 32 registradores de 32 bits
- Memória de 64K palavras (256KB)
- 30 instruções (22 básicas + 8 adicionais)
- Interpretador Assembly completo
- CPI de 4.0 ciclos por instrução
- 11 programas de teste validados

2 Introdução

2.1 Contexto e Objetivos

O trabalho prático visa consolidar o aprendizado sobre arquitetura de processadores através da implementação de um simulador funcional. Este tipo de simulador divide o processador em blocos funcionais que refletem uma implementação real em hardware.

Objetivos principais:

1. Implementar pipeline de 4 estágios com separação clara de responsabilidades
2. Gerenciar banco de registradores e memória de forma eficiente
3. Codificar e executar conjunto completo de instruções RISC
4. Validar correção através de bateria extensiva de testes
5. Fornecer ferramenta didática para compreensão de arquitetura de processadores

2.2 Metodologia de Desenvolvimento

O desenvolvimento seguiu abordagem modular e incremental:

1. **Fase 1:** Implementação de estruturas básicas (CPU State, Memory, Utils)

2. **Fase 2:** Desenvolvimento da ALU e decodificador de instruções
3. **Fase 3:** Implementação do pipeline e controle de fluxo
4. **Fase 4:** Desenvolvimento do interpretador/assembler
5. **Fase 5:** Testes extensivos e refinamentos
6. **Fase 6:** Documentação e validação final

Ferramentas utilizadas:

- Python 3.8+ (linguagem de implementação)
- Git/GitHub (versionamento e colaboração)
- Visual Studio Code (IDE)
- pytest (testes automatizados)

3 Arquitetura do UFLA-RISC

3.1 Especificações Técnicas

Componente	Especificação
Arquitetura	RISC load-store de 32 bits
Registradores	32 registradores (R0-R31)
Memória	64K palavras × 32 bits = 256KB
Endereçamento	Por palavra (16 bits)
Pipeline	4 estágios
Flags	neg, zero, carry, overflow

Tabela 1: Especificações técnicas do UFLA-RISC

3.2 Pipeline de 4 Estágios

O processador implementa um pipeline clássico de 4 estágios:

```
  IF     ->    ID     ->  EX/MEM   ->    WB
Fetch        Decode       Execute       Write
```

Estágios:

1. **IF (Instruction Fetch):** Busca instrução e incrementa PC
2. **ID (Instruction Decode):** Decodifica e lê registradores
3. **EX/MEM (Execute/Memory):** Executa operação e acessa memória
4. **WB (Write Back):** Escreve resultado no registrador

4 Decisões de Implementação

4.1 Linguagem: Python

Justificativa:

- Desenvolvimento rápido e código legível
- Estruturas de dados de alto nível
- Multiplataforma
- Ideal para simulador didático

4.2 Pipeline de 4 estágios

Por que não 5 estágios?

- Combina EX e MEM em um único estágio
- Simplifica controle de hazards
- Adequado para instruções RISC simples

4.3 Gerenciamento de Flags

- Calculados pela ALU
- Sincronizados com CPU após operações
- Apenas instruções ALU afetam flags
- Operações lógicas zeram carry/overflow

4.4 Registrador R0

- Sempre retorna 0
- Escritas são ignoradas silenciosamente
- Padrão em arquiteturas RISC (MIPS, RISC-V)

5 Conjunto de Instruções

5.1 Instruções Básicas (22)

Categoria	Instruções
Aritméticas	add, sub
Lógicas	xor, or, and, passnota, zeros, passa
Shifts	asl, asr, lsl, lsr
Constantes	lch, lcl
Memória	load, store
Controle	beq, bne, jal, jr, j
Especial	halt

Tabela 2: Instruções básicas do UFLA-RISC

5.2 Instruções Adicionais (8 do grupo)

Instrução	Opcode	Operação	Justificativa
slt rc, ra, rb	0x17	rc = (ra ¡ rb) ? 1 : 0	Comparação para loops
mul rc, ra, rb	0x18	rc = ra × rb	Multiplicação eficiente
div rc, ra, rb	0x19	rc = ra ÷ rb	Divisão inteira
mod rc, ra, rb	0x1A	rc = ra % rb	Resto da divisão
neg rc, ra	0x1B	rc = -ra	Negação aritmética
inc rc, ra	0x1C	rc = ra + 1	Incremento simplificado
dec rc, ra	0x1D	rc = ra - 1	Decremento simplificado
nop	0x1E	-	Alinhamento/padding

Tabela 3: Instruções adicionais implementadas pelo grupo

5.3 Exemplo de Uso: SLT para Loops

Listing 1: Loop de 0 a 10

```
# Loop de 0 a 10
lch r1, 0x0000
lcl r1, 0x0000      # i = 0
lch r2, 0x0000
lcl r2, 0x000A      # limite = 10

loop:
    slt r3, r1, r2  # r3 = (i < 10)
    beq r3, r0, fim
    inc r1, r1      # i++
    j loop
fim:
    halt
```

6 Módulos do Simulador

6.1 Estrutura Geral

```
src/
 simulador/
    cpu_state.py          # Registradores, PC, IR, flags
    memory.py             # Memória 64K palavras
    alu.py                # Operações aritméticas/lógicas
    control_unit.py       # Branches e jumps
    instruction_decoder.py # Decodificação de instruções
    simulator.py          # Pipeline principal
    utils.py              # Funções auxiliares

 interpretador/
     parser.py             # Análise léxica/sintática
     encoder.py            # Codificação binária
     opcodes.py            # Tabela de opcodes
     assembler.py          # Orquestrador
```

6.2 Principais Módulos

6.2.1 cpu_state.py

- Gerencia 32 registradores + PC + IR
- Controla flags de condição
- Fornece snapshots para debug

6.2.2 memory.py

- Implementa 64K palavras de memória
- Carrega programas de arquivos texto/binário
- Suporta breakpoints

6.2.3 alu.py

- Executa todas as operações aritméticas/lógicas
- Calcula e atualiza flags
- Suporta shifts, multiplicação, divisão

6.2.4 simulator.py

- Orquestra os 4 estágios do pipeline
- Controla fluxo de execução
- Calcula estatísticas (CPI)

7 Interpretador/Assembler

7.1 Processo de Montagem

```
Arquivo .asm  →  [Parser]  →  [Encoder]  →  Arquivo .bin
                     ↓
                 [Labels]
```

Duas passagens:

1. **Primeira passagem:** Identifica labels e seus endereços
2. **Segunda passagem:** Codifica instruções em binário

7.2 Exemplo Completo

Entrada (programa.asm):

Listing 2: Exemplo de programa assembly

```
address 0
lch r1, 0x0000
lcl r1, 0x000A      # r1 = 10
lch r2, 0x0000
lcl r2, 0x0014      # r2 = 20
add r3, r1, r2      # r3 = 30
store r0, r3        # mem[0] = 30
halt
```

Saída (programa.bin):

Listing 3: Código binário gerado

```
address 0000000000000000
00001110000000000000000000000001
00001110000000000000101000000001
00001110000000000000000000000010
00001110000000000001010000000010
00000001000000010000001000000011
00010001000000000000001100000000
11111111111111111111111111111111
```

8 Testes e Validação

8.1 Programas de Teste

Arquivo	Descrição	Instruções Testadas
01_teste_add.asm	Adição básica	ADD, LCH, LCL
02_teste_sub.asm	Subtração	SUB, flags
03_teste_logicas.asm	Lógicas	XOR, OR, AND, NOT
04_teste_shifts.asm	Deslocamentos	ASL, ASR, LSL, LSR
05_teste_memory.asm	Memória	LOAD, STORE
06_teste_branches.asm	Desvios	BEQ, BNE
07_teste_jumps.asm	Saltos	JAL, JR, J
08_teste_adicionais.asm	Novas instruções	MUL, DIV, MOD, SLT
09_fatorial.asm	Fatorial recursivo	Programa completo
10_fibonacci.asm	Fibonacci	Loops
11_soma_vetor.asm	Soma de vetor	Memória + loops

Tabela 4: Bateria de testes implementada

8.2 Exemplo de Teste: Soma Básica

Listing 4: Teste de adição

```
address 0
lch r1, 0x0000
lcl r1, 0x000A      # r1 = 10
lch r2, 0x0000
lcl r2, 0x0014      # r2 = 20
add r3, r1, r2      # r3 = 30
halt
```

Resultado esperado:

```
Total de ciclos: 24
Total de instruções: 6
CPI: 4.00
R1: 0x0000000a (u32: 10, s32: 10)
R2: 0x00000014 (u32: 20, s32: 20)
R3: 0x0000001e (u32: 30, s32: 30)
```

8.3 Critérios de Validação

Critérios atendidos:

- CPI sempre igual a 4.0

- Flags corretos após operações ALU
- Registradores com valores esperados
- Memória corretamente atualizada
- R0 sempre zero
- Branches/jumps funcionando corretamente

9 Como Usar

9.1 Instalação

Listing 5: Clonando o repositório

```
git clone https://github.com/MarcoTFranco/ufla-risc-simulador-grupo6-gcc123.git
cd ufla-risc-simulador-grupo6-gcc123
```

9.2 Montar Programa Assembly

Listing 6: Montagem de programa

```
python src/interpretador/main.py programa.asm binarios/programa.bin
```

9.3 Executar no Simulador

Modo normal (apenas resumo):

```
python src/simulador/main.py binarios/programa.bin
```

Modo verbose (ciclo a ciclo):

```
python src/simulador/main.py binarios/programa.bin --verbose
```

10 Resultados Obtidos

10.1 Estatísticas de Execução

Programa de exemplo (11 instruções):

```
Total de ciclos: 44
Total de instruções: 11
CPI: 4.00
```

10.2 Validação do Pipeline

- Todos os 11 programas de teste executados com sucesso
- CPI consistente em 4.0 (ideal)
- Flags corretamente calculados
- Memória e registradores validados
- Sem erros de execução

11 Conclusão

O simulador funcional do UFLA-RISC foi implementado com sucesso, atendendo a todos os requisitos do trabalho prático.

11.1 Objetivos alcançados

- Pipeline de 4 estágios funcionando corretamente
- 30 instruções implementadas e testadas
- Interpretador/assembler completo
- CPI ideal de 4.0 ciclos por instrução
- Bateria extensiva de testes (11 programas)
- Documentação completa

11.2 Principais aprendizados

- Compreensão profunda de pipeline RISC
- Gerenciamento de hazards e controle de fluxo
- Codificação e decodificação de instruções
- Desenvolvimento de software modular

11.3 Trabalhos futuros

- Implementar forwarding para reduzir hazards
- Adicionar predição de branches
- Cache de instruções/dados
- Interface gráfica para visualização

12 Referências

1. PATTERSON, D.; HENNESSY, J. **Computer Organization and Design**. 5th ed. Morgan Kaufmann, 2014.
2. HARRIS, S.; HARRIS, D. **Digital Design and Computer Architecture**. 2nd ed. Morgan Kaufmann, 2012.
3. Material didático GCC123/PCC507 - UFLA

A Formato de Instruções

A.1 Instruções de 3 Registradores

```
[opcode(8)] [ra(8)] [rb(8)] [rc(8)]
```

A.2 Instruções com Imediato

```
[opcode(8)] [const16(16)] [rc(8)]
```

A.3 Instruções de Branch

```
[opcode(8)] [ra(8)] [rb(8)] [offset(8)]
```

A.4 Instruções de Jump

```
[opcode(8)] [address(24)]
```

B Tabela Completa de Opcodes

Opcode	Mnemônico	Tipo
0x01	ADD	3reg
0x02	SUB	3reg
0x03	ZEROS	1reg
0x04	XOR	3reg
0x05	OR	3reg
0x06	NOT	2reg
0x07	AND	3reg
0x08	ASL	3reg
0x09	ASR	3reg
0x0A	LSL	3reg
0x0B	LSR	3reg
0x0C	PASSA	2reg
0x0E	LCH	imm16
0x0F	LCL	imm16
0x10	LOAD	2reg
0x11	STORE	2reg
0x12	JAL	jump
0x13	JR	1reg
0x14	BEQ	branch
0x15	BNE	branch
0x16	J	jump
0x17	SLT	3reg
0x18	MUL	3reg
0x19	DIV	3reg
0x1A	MOD	3reg
0x1B	NEG	2reg
0x1C	INC	2reg
0x1D	DEC	2reg
0x1E	NOP	none
0xFF	HALT	none

Tabela 5: Tabela completa de opcodes

Relatório elaborado em: Novembro de 2025
Repositório: https://github.com/MarcoTFranco/ufla-risc-simulador-grupo6-gcc123