UNIVERSIDADE FEDERAL DE LAVRAS – *CAMPUS* SEDE

Bacharelado em Ciência da Computação

Lana da Silva Miranda

Fábio Damas Valim

Guilherme Lirio Miranda

Marcos Vinícius Pereira

Arthur Soares Marques

DOCUMENTAÇÃO: Simulador de pipeline UFLA_RISC

Lavras

2025

1 RESUMO DA MÁQUINA SIMULADA

A máquina simulada implementa um processador educacional inspirado em arquiteturas RISC. O modelo possui pipeline de 4 estágios (IF, ID, EX, WB), registradores gerais, registrador de flags e memória principal. A execução é organizada em ciclos, permitindo observar o fluxo de instruções passo a passo.

Trata-se de um processador RISC de 32 bits, com:

- 32 registradores gerais (R0–R31), cada um com 32 bits.
- Barramento de dados de 32 bits e barramento de endereços de 16 bits.
- Memória com 65.536 posições (64K palavras), totalizando 256 KB.
- Formato de instruções de 32 bits organizadas em campos (opcode, registradores, constantes).
- Pipeline de 4 estágios: IF, ID, EX/MEM e WB.

Nosso simulador executa instruções ciclo a ciclo, permitindo observar:

- Modificações nos registradores.
- Atualizações de flags (neg, zero, carry, overflow).
- Acesso à memória.
- Alterações do PC (Program Counter).
- Resolução de desvios condicionais e incondicionais.

2 DECISÕES DA IMPLEMENTAÇÃO

As principais decisões do grupo envolveram priorizar legibilidade, modularização e simplicidade estrutural.

- A linguagem escolhida para implementação foi Python, considerando sua facilidade de leitura, alto nível de abstração para simular os componentes, facilidade de lidar com cálculos com double e o fato de ser bem menos verbosa quando comparada com outras linguagens, como o Java, por exemplo.

- O pipeline foi separado nos quatro estágios fundamentais:

  - IF: busca instrução, atualiza o PC
  - ID: decodifica, acessa registradores
  - EX/MEM: executa ALU, resolve branch, acessa memória
  - WB: escreve no registrador destino

Optamos por não adicionar forwarding nem detecção automática de hazards, mantendo o modelo funcional exigido. Cada estágio é acionado sequencialmente no método executar_ciclo().

Trecho de código da estrutura do ciclo:

```
def executar_ciclo(self):
    self.estagio_IF()
    self.estagio_ID()
    self.estagio_EX()
    self.estagio_WB()
```

- Usamos uma classe no processador para inicializar as flags como desligadas e para resetá-las também. As flags são alteradas nas instruções que alteram estado no EX/MEM.
- O simulador também possui um mecanismo de log interno, permitindo que o estado do processador seja impresso no console ou em arquivo.

3 INSTRUÇÕES PROJETADAS

Além das instruções obrigatórias do UFLA-RISC, desenvolvemos instruções adicionais para facilitar, operando sobre registradores e memória, mantendo o padrão RISC de simplicidade. Instruções típicas incluem ADD, SUB, LOAD, STORE, JUMP e variantes com condições baseadas em flags.

3.1 MOVEI: carrega constante inteira direta

Foi criado para facilitar testes simples sem necessidade de usar LCLH e LCLL.

```
# MOVEI rc, valor
self.regs[rc] = valor
```

3.2 INC: incrementa registrador

Foi criado para facilitar o uso em loops e contadores.

```
self.regs[rc] += 1
```

3.3 DEC: decrementa registrador

Complementa INC e auxilia decisões condicionais.

```
self.regs[rc] -= 1
```

3.4 CLEAR: zera registrador

Semelhante à instrução ZERO, mas com sintaxe própria criada. Define o registrador destino como zero.

```
self.regs[rc] = 0
```

3.5 HALT: parada forçada do simulador

Instrução criada para permitir parada manual antes do fim do programa. Quando executada, interrompe imediatamente o loop principal do simulador.

```
self.halted = True
```

3.6 CMP: comparação sem escrita

Compara dois registradores sem alterar registradores, apenas flags. Usado para decisões condicionais (comparações).

```
comp = self.regs[ra] - self.regs[rb]
```

4 TUTORIAL DE USO DO INTERPRETADOR E SIMULADOR

É necessário ter o Python instalado.

1. Clone o repositório:

```
git clone https://github.com/lanamiranda17/ufla-risc-simulador-grupo-2.git
```

2. Entre na pasta do projeto:

```
cd ufla-risc-simulador-grupo-2/
```

3. Escreva seu código:

3.1 Abra o arquivo localizado em `src/interpretador/programa.asm.`

3.2 Apague o conteúdo existente e cole ou escreva o seu código Assembly neste arquivo.

3.3 Na pasta `src/testes/testes_assembly` possui vários arquivos '.asm' que podem ser utilizados como conteúdo para teste, apenas copie e cole dentro de `programa.asm`. O simulador sempre lerá o código deste arquivo específico.

4. Execute o simulador:

4.1 Na raiz do projeto, execute o arquivo principal:

```
python src/main.py
```

4.2 Verifique a Saída: O terminal exibirá:

- Confirmação da compilação: `asm -> .bin`
- Logs ciclo a ciclo da execução (Fetch/Decode).
- Estado final dos registradores e memória.

5 ESTRUTURAS DE REPRESENTAÇÃO DO HARDWARE (DATAPATH)

O datapath é representado por classes internas no módulo simulador. Os principais componentes implementados são:

- Banco de registradores: `self.regs = [0] * 32`
- Registrador PC: `self.pc = 0`
- Unidade de Controle: lê o opcode, ativa ou desativa partes do datapath, escolhe de onde vem e para onde vai cada dado.
- ALU (implementada dentro do EX/MEM): `self.result_ex = self.regs[ra] + self.regs[rb]`
- Memória principal: `self.mem = [0] * 65536`
- Registrador de Flags: classe `Flags()`

6 CÓDIGO FONTE DO SIMULADOR

O código-fonte completo está disponível no repositório https://github.com/lanamiranda17/ufla-risc-simulador-grupo-2/ enviado junto ao trabalho.

6.1 Estrutura de pastas

A estrutura de diretórios está organizada da seguinte forma:

```
ufla-risc-simulador/
│
├──── docs/              → Documentação e todo list
├──── src/               → Código-fonte principal
│     ├──── interpretador/
│     ├──── simulador/
│     └──── testes/
└──── README.md          → Descrição geral e instruções de uso
```

- `docs/`: Destinado à documentação final do projeto, incluindo este relatório e um to do list usado pelos integrantes o grupo para organização interna e para divisão das issues.
- `src/`: Núcleo do simulador, dividido em três módulos fundamentais: interpretador, simulador e testes. Cada subpasta contém componentes que representam partes do datapath e da lógica do processador.

6.2 Interpretador (src/interpretador/)

Responsável por:

- Ler arquivos binários contendo instruções codificadas em 32 bits.
- Interpretar a diretiva `address`, conforme definido no enunciado.
- Mapear cada instrução para seu endereço correto na memória simulada.
- Gerar a representação interna do programa a ser carregado pela classe `Memoria`.

O interpretador atua como a ponte entre o assembly/binário e o simulador, garantindo que o formato da entrada esteja de acordo com as especificações oficiais do UFLA-RISC.

6.3 Simulador (`src/simulador/`)

Aqui estão implementados os componentes fundamentais do datapath e da unidade de controle, estruturados em módulos independentes.

- cache.py: estrutura básica de uma cache (linhas, tags, validade); lógica inicial de leitura e escrita simulada, separada em cache de dados e instruções.
- flags.py: padroniza a manipulação dos flags e permite seu reset e atualização limpa, centralizada.
- Ir.py: define o Instruction Register (IR), responsável por armazenar a instrução trazida do estágio IF
- loader.py: realiza o carregamento dos arquivos binários no simulador. Este módulo é o elo entre o interpretador e a memoria.py.
- memoria.py: responsável por simular a memória principal de 64K palavras (256 KB).
- pc.py: define o Program Counter, componente essencial para o fluxo de execução. O PC é usado diretamente no estágio IF.
- registradores.py: implementa os 32 registradores de 32 bits do UFLA-RISC.
- processador_main.py: núcleo do simulador, onde todos os módulos anteriores são integrados para formar o processador completo. Este arquivo contém o ciclo do pipeline, unidade de Controle embutida, integração dos blocos e execução do programa

7 TESTES REALIZADOS

Para garantir que o simulador UFLA-RISC está funcionando corretamente, foram conduzidos dois tipos principais de testes: testes massivos e testes unitários.

Os testes massivos consistem em executar grandes quantidades de instruções em diferentes combinações, ordens e cenários, como objetivo avaliar o simulador como um todo e principalmente confirmar a correta integração entre os estágios do pipeline (IF → ID → EX/MEM → WB).

Os testes unitários focam na validação de componentes isolados do simulador. Enquanto os testes massivos verificam o sistema inteiro, os testes unitários garantem que cada parte

funcione individualmente, conforme especificado. Um exemplo são os testes das instruções aritméticas: ADD, SUB, AND, OR, XOR, shifts.

A pasta `src/testes` contém:

- programas de verificação do pipeline
- testes de instruções aritméticas
- testes de LOAD/STORE
- testes de flags
- arquivos utilitários usados para simular cenários reais.