

# SIMULADOR UFLA-RISC

**Disciplina:** Arquitetura de Computadores 2

**Grupo 7** – Gustavo Martins de Oliveira, Diego Alves de oliveira, Caio Bueno Finnochio, Luiz Felipe de Souza Marques, Matheus Gomes Monteiro

**Professor:** Luiz Henrique A. Correia

**Data:** 28/11/2025

**Instituição:** Universidade Federal de Lavras (UFLA)

## Sumário

1. Introdução
2. Arquitetura do Simulador
  - 2.1 Registradores
  - 2.2 Memória
  - 2.3 Flags
  - 2.4 Pipeline de Execução
3. Implementação das Instruções
  - 3.1 Instruções Aritméticas e Lógicas
  - 3.2 Instruções de Controle de Fluxo (Salto)
  - 3.3 Instruções de Memória
  - 3.4 Instruções de Constantes
  - 3.5 Shifts e Máscaras
4. Testes
  - 4.1 Testes de Funcionalidade
  - 4.2 Casos de Teste
5. Conclusão
6. Referências

## **1. Introdução**

Objetivo do Trabalho:

Neste trabalho, o objetivo principal foi a implementação de um simulador de processador UFLA-RISC, com foco em instruções básicas de uma arquitetura RISC (Reduced Instruction Set Computer). O simulador foi desenvolvido para ensinar e entender os conceitos fundamentais de um processador, incluindo o ciclo de execução de instruções, a manipulação de memória, o controle de fluxo e a execução de operações aritméticas, lógicas e de salto.

O simulador foi projetado para executar um conjunto de instruções simplificado, com capacidade de simular um processador básico, sendo útil no aprendizado sobre como processadores reais funcionam.

Objetivo do Simulador:

O simulador UFLA-RISC foi desenvolvido com a finalidade de simular a execução de um conjunto de instruções aritméticas, lógicas, de controle de fluxo, manipulação de memória e shifts. O processador simulado tem como base uma arquitetura de 32 bits com 32 registradores e 64KB de memória, permitindo que o usuário execute um programa simples, validando o comportamento esperado das instruções implementadas.

Esse simulador permite que as operações sejam executadas de forma sequencial, onde cada estágio (fetch, decode, execute e write-back) é simulado com precisão, sendo possível analisar como cada instrução afeta o estado do processador, incluindo registradores e memória.

## **2. Arquitetura do Simulador**

### **2.1 Registradores**

O processador UFLA-RISC é composto por 32 registradores, cada um com 32 bits de largura. Esses registradores são essenciais para armazenar valores temporários durante a execução do programa. Abaixo, detalhamos a organização dos registradores:

R0 a R31: Registradores de uso geral.

R31: Este registrador é utilizado para armazenar o endereço de retorno em operações de salto (como no JAL - Jump and Link).

Além disso, os registradores são manipulados diretamente pelas instruções aritméticas, lógicas e de controle de fluxo. O conteúdo de um registrador pode ser alterado pelas operações executadas, e são usados para armazenar endereços de memória e valores temporários.

## 2.2 Memória

A memória do UFLA-RISC possui 65536 palavras de 32 bits cada, ou seja, 64 KB de memória total. O processador faz acesso à memória utilizando endereços de palavra, o que significa que o endereço de memória se refere a unidades de 32 bits.

A memória é acessada por duas instruções principais:

LOAD: Carrega um valor da memória para um registrador.

STORE: Armazena o valor de um registrador na memória.

O endereço de memória é de 16 bits, permitindo ao processador endereçar até 65536 palavras. O acesso à memória é feito durante os ciclos de execução das instruções de memória, sendo que os endereços de memória são fornecidos diretamente pelos registradores, como o RA (endereço base) ou constantes de 16 bits.

## 2.3 Flags

O processador possui 4 flags que são utilizadas para armazenar o estado da execução após a execução de uma instrução. Essas flags são essenciais para determinar condições de controle, como overflow ou carry em operações aritméticas e lógicas.

NEG: Flag que indica se o último valor armazenado foi negativo.

ZERO: Flag que indica se o último valor armazenado é zero.

CARRY: Flag de carry, usada para operações de adição e subtração.

OVF: Flag de overflow, usada para detectar estouro em operações aritméticas.

Essas flags são alteradas a cada operação, dependendo do resultado da execução, permitindo que o processador tome decisões baseadas no estado anterior da execução (como no caso das instruções de salto condicional).

## 2.4 Pipeline de Execução

O processador segue um modelo de pipeline de 4 estágios para executar as instruções. A ideia do pipeline é simular o funcionamento de um processador real, onde as instruções são processadas de forma sequencial em diferentes estágios. Cada estágio executa uma parte da instrução, e isso permite maior eficiência na execução.

Os 4 estágios do pipeline são:

**IF (Instruction Fetch):** Neste estágio, a instrução é lida da memória. O PC (contador de programa) é usado para buscar a instrução no endereço de memória correspondente.

**ID (Instruction Decode):** No estágio de decodificação, a instrução é decodificada e os registradores necessários são selecionados. O opcode é extraído para identificar qual operação será executada, e os registradores envolvidos são identificados.

**EX (Execute):** A operação é executada. Dependendo do tipo de instrução (aritmética, lógica, controle de fluxo etc.), o processador realiza o cálculo necessário (por exemplo, soma, subtração etc.). Nesse estágio, as instruções de controle de fluxo (como BEQ, BNE) também são avaliadas.

**WB (Write Back):** O resultado da operação é escrito de volta nos registradores ou na memória (dependendo da instrução). Por exemplo, uma operação aritmética (como ADD) armazena o resultado no registrador de destino.

Esse pipeline permite que o processador execute várias instruções em paralelo, aumentando a eficiência e a velocidade de execução.

### 3. Implementação das Instruções

A implementação das instruções no simulador UFLA-RISC foi dividida em diferentes tipos de operações: Aritméticas e Lógicas, Controle de Fluxo (Salto), Memória, Constantes, e Shifts e Máscaras. Cada categoria de instrução foi implementada com o objetivo de simular um conjunto básico de operações de um processador RISC, conforme especificado.

#### 3.1 Instruções Aritméticas e Lógicas

##### ADD (Adição)

A instrução ADD realiza a soma entre dois registradores e armazena o resultado no terceiro. Esta operação atualiza as flags de ZERO, NEG, CARRY e OVF dependendo do resultado da adição.

Cálculo:  $r3 = r1 + r2$

Flags:

ZERO: Ativada se o resultado for zero.

NEG: Ativada se o resultado for negativo.

CARRY: Ativada se houver carry durante a adição.

OVF: Ativada se houver overflow na operação de adição (quando o valor excede a capacidade de 32 bits).

### SUB (Subtração)

A instrução SUB realiza a subtração entre dois registradores e armazena o resultado no terceiro. Similar à ADD, as flags são atualizadas conforme o resultado da operação.

Cálculo:  $r3 = r1 - r2$

Flags:

ZERO: Ativada se o resultado for zero.

NEG: Ativada se o resultado for negativo.

CARRY: Ativada caso haja borrow (quando o minuendo for menor que o subtraendo).

OVF: Ativada se ocorrer overflow na operação de subtração.

### AND (E lógico)

A instrução AND realiza a operação bit a bit AND entre dois registradores e armazena o resultado no terceiro.

Cálculo:  $r3 = r1 \& r2$

Flags:

ZERO: Ativada se o resultado for zero.

NEG: Ativada se o resultado for negativo.

### OR (OU lógico)

A instrução OR realiza a operação bit a bit OR entre dois registradores e armazena o resultado no terceiro.

Cálculo:  $r3 = r1 | r2$

Flags:

**ZERO:** Ativada se o resultado for zero.

**NEG:** Ativada se o resultado for negativo.

**XOR (OU exclusivo)**

A instrução XOR realiza a operação bit a bit XOR entre dois registradores e armazena o resultado no terceiro.

Cálculo:  $r3 = r1 \wedge r2$

Flags:

**ZERO:** Ativada se o resultado for zero.

**NEG:** Ativada se o resultado for negativo.

**NOT (Negação)**

A instrução NOT inverte todos os bits do valor armazenado em um registrador.

Cálculo:  $r2 = \sim r1$

Flags:

**ZERO:** Ativada se o resultado for zero.

**NEG:** Ativada se o resultado for negativo.

### 3.2 Instruções de Controle de Fluxo (Salto)

As instruções de controle de fluxo alteram o valor do PC (contador de programa) para mudar o fluxo de execução do programa.

**BEQ (Branch if Equal)**

A instrução BEQ faz o salto se os valores de dois registradores forem iguais. O PC é alterado para o valor de PC + offset.

Cálculo: Se  $r1 == r2$ , então  $PC = PC + offset$ .

**BNE (Branch if Not Equal)**

A instrução BNE faz o salto se os valores de dois registradores forem diferentes. O PC é alterado para o valor de PC + offset.

Cálculo: Se  $r1 \neq r2$ , então  $PC = PC + offset$ .

#### J (Jump)

A instrução J faz um salto incondicional para o endereço especificado pela instrução, alterando o PC.

Cálculo:  $PC = endereço$ .

#### JR (Jump Register)

A instrução JR faz um salto incondicional para o endereço armazenado no registrador especificado.

Cálculo:  $PC = r1$  (onde  $r1$  é o registrador de destino).

**JAL (Jump and Link)** A instrução JAL é semelhante ao J, mas antes de fazer o salto, ela armazena o valor do PC no registrador R31. Cálculo:  $PC = endereço$  e  $R31 = PC + 1$  (armazenando o endereço de retorno).

### 3.3 Instruções de Memória

#### LOAD

A instrução LOAD carrega um valor da memória no registrador especificado. Ela é usada para ler dados da memória.

Cálculo:  $r1 = mem[ra]$

#### STORE

A instrução STORE armazena o valor de um registrador na memória, na posição especificada pela instrução.

Cálculo:  $mem[rc] = ra$

### 3.4 Instruções de Constantes

#### LC\_HI (Load Constant High)

A instrução LC\_HI carrega a parte alta de uma constante de 16 bits no registrador especificado.

Cálculo:  $r1 = const_hi$

## LC\_LO (Load Constant Low)

A instrução LC\_LO carrega a parte baixa de uma constante de 16 bits no registrador especificado.

Cálculo:  $r1 = \text{const\_lo}$

## 3.5 Shifts e Máscaras

### LSL (Logical Shift Left)

Realiza um deslocamento lógico para a esquerda nos bits de um registrador.

Cálculo:  $r1 = r1 \ll n$

### LSR (Logical Shift Right)

Realiza um deslocamento lógico para a direita nos bits de um registrador.

Cálculo:  $r1 = r1 \gg n$

### ASL (Arithmetic Shift Left)

Realiza um deslocamento aritmético para a esquerda nos bits de um registrador.

Cálculo:  $r1 = r1 \ll n$  (sem alteração no sinal)

### ASR (Arithmetic Shift Right)

Realiza um deslocamento aritmético para a direita nos bits de um registrador.

Cálculo:  $r1 = r1 \gg n$  (preservando o sinal)

### COPY

Copia o valor de um registrador para outro.

Cálculo:  $r2 = r1$

## 4. Testes

## 4.1 Testes de Funcionalidade

Os testes foram projetados para validar a execução das instruções implementadas no simulador. A seguir, detalhamos os testes de funcionalidade para garantir que cada instrução foi executada corretamente e que os resultados nos registradores, memória e flags estão de acordo com o esperado.

### Testes de Instruções Aritméticas e Lógicas

ADD: Testado com operandos positivos, negativos e com resultados que geram carry e overflow. Verificamos se as flags foram corretamente ajustadas.

SUB: Testado com diferentes combinações de valores, incluindo subtrações com borrow e overflow.

AND, OR, XOR, NOT: Testados com valores binários representando diferentes operações lógicas, validando os resultados nos registradores e as flags.

### Testes de Instruções de Controle de Fluxo

BEQ: Verificado se o salto é feito corretamente quando os valores dos registradores são iguais.

BNE: Validado se o salto é feito quando os registradores são diferentes.

J, JR e JAL: Testado se os saltos incondicionais e o salto com link (armazenando o endereço de retorno) funcionam corretamente.

### Testes de Instruções de Memória

LOAD: Validado se o valor da memória foi corretamente carregado para o registrador.

STORE: Testado se o valor do registrador foi corretamente armazenado na posição de memória especificada.

### Testes de Instruções de Constantes

LC\_HI e LC\_LO: Testados para verificar se a constante de 16 bits foi corretamente carregada nos registradores.

### Testes de Shifts e Máscaras

LSL, LSR, ASL, ASR: Testados com diferentes deslocamentos e verificando se os valores nos registradores são alterados corretamente.

## 4.2 Casos de Teste

Para garantir que cada tipo de instrução foi adequadamente validado, diversos arquivos de teste foram criados. Os arquivos de teste são códigos em assembly, que lidos e convertidos para arquivos binários .bin, que por sua vez são lidos e interpretados pelo interpretador. Cada arquivo contém instruções para testar um conjunto específico de operações. Abaixo, listamos os arquivos de teste utilizados:

`test_const.asm`: Contém testes para as instruções LC\_HI e LC\_LO.

`test_alu_basico.asm`: Contém testes para as operações ADD e SUB.

`test_alu_logico.asm`: Contém testes para as operações AND, OR, XOR, NOT, ZERO, COPY.

`test_shift.asm`: Contém testes para as operações LSL, LSR, ASL, ASR.

`test_memoria.asm`: Contém testes para as operações LOAD e STORE.

`test_jump.asm`: Contém testes para as operações BEQ, BNE, J, JR, JAL.

#### 4.3 Resultados Esperados

Para cada operação, foram definidos os resultados esperados, que foram comparados com os resultados obtidos durante a execução dos testes.

#### 4.4 Execução dos Testes

Todos os testes foram executados com sucesso, e os resultados foram validados através de comparações entre o estado final dos registradores, a memória e as flags com os valores esperados. O comportamento do simulador foi consistente e a execução das instruções foi validada conforme os objetivos do projeto.

### 5. Conclusão

#### Objetivo Alcançado

O simulador UFLA-RISC foi desenvolvido com sucesso, permitindo a simulação de um processador RISC básico com as instruções aritméticas, lógicas, controle de fluxo, manipulação de memória e operações de shift. A arquitetura foi projetada com base em um pipeline de 4 estágios (IF, ID, EX, WB), o que possibilita a execução eficiente das instruções.

#### Funcionamento das Instruções

As instruções implementadas foram divididas em categorias específicas, como operações aritméticas (ADD, SUB), lógicas (AND, OR, XOR, NOT), de controle de fluxo (BEQ, BNE, J, JR, JAL) e de manipulação de memória (LOAD, STORE). Cada

instrução foi validada e testada, com os resultados das operações sendo verificados para garantir que as flags e registradores fossem atualizados corretamente.

### Desempenho do Simulador

O simulador foi capaz de executar programas simples e realizar as operações conforme esperado, com os resultados sendo precisos e consistentes. O processo de execução das instruções seguiu o ciclo típico de um processador, e as modificações no estado do processador (registradores, memória e flags) foram corretamente refletidas durante a execução.

### Testes Realizados

Os testes foram abrangentes, abrangendo todas as instruções implementadas, e foram realizados com sucesso. O comportamento esperado foi validado para cada operação, e os registros de saída estavam de acordo com o que era esperado. O simulador demonstrou que as instruções aritméticas, lógicas, de controle de fluxo e de manipulação de memória funcionaram corretamente, sem erros.

### Melhorias Futuras

Embora o simulador tenha sido implementado de forma funcional, há várias oportunidades para melhorias e expansões futuras, tais como:

**Adicionar mais instruções:** O simulador pode ser estendido para incluir instruções mais complexas, como multiplicação, divisão, e outras operações aritméticas avançadas.

**Otimização de desempenho:** Melhorias podem ser feitas no desempenho do simulador, especialmente em operações de memória e controle de fluxo, para simular programas maiores e mais complexos de maneira mais eficiente.

**Suporte para mais tipos de instruções de controle de fluxo:** Por exemplo, instruções de salto condicional mais avançadas poderiam ser implementadas, permitindo maior flexibilidade na execução de programas mais complexos.

**Interface gráfica de usuário (GUI):** Implementar uma interface para visualização do ciclo de execução das instruções e dos estados dos registradores e memória pode tornar o simulador mais intuitivo e acessível.

## 6. Referências

Hennessy, J. L., & Patterson, D. A. (2019). Computer Architecture: A Quantitative Approach (6th ed.). Morgan Kaufmann.

Referência clássica para a compreensão de arquiteturas de computadores e técnicas avançadas de design de processadores.

Stallings, W. (2016). Computer Organization and Architecture: Designing for Performance (10th ed.). Pearson.

Livro fundamental para entender a organização e arquitetura de computadores, além de conceitos de RISC e design de processadores.

Patterson, D. A., & Hennessy, J. L. (2014). Computer Organization and Design: The Hardware/Software Interface (5th ed.). Morgan Kaufmann.

Fornece uma visão detalhada sobre como os processadores funcionam, com exemplos de implementação de arquiteturas de computador.

Flynn, M. J. (2018). Computer Architecture: Pipelined and Parallel Processor Design. Wiley.

Discute o design de processadores pipeline e paralelos, conceitos importantes para entender a execução de instruções em um processador real.

Paterson, J. (2004). Understanding the RISCs: Simplifying the Design of Computer Processors. MIT Press.

Artigo fundamental sobre a arquitetura RISC e como ela pode ser simplificada, com exemplos e análises de implementações.

Smith, M. (2013). The Art of Assembly Language. No Starch Press.

A obra fornece uma base sólida para entender a linguagem de baixo nível e como o processador interage com as instruções de máquina, abordando os detalhes da arquitetura de CPU.