

LUCAS DOS SANTOS PIMENTEL

**MICROPROCESSADOR MIPS**

CUIABÁ

2024

LUCAS DOS SANTOS PIMENTEL

**MICROPROCESSADOR MIPS**

Trabalho apresentado ao Instituto Federal de Mato Grosso como requisito para a conclusão do curso de Engenharia da Computação.

Professor: Ruy de Oliveira

CUIABÁ

2024

**RESUMO**

sumário

[1 INTRODUÇÃO 5](#_Toc175481219)

[2 Arquitetura e componentes 6](#_Toc175481220)

[2.1 Program Counter 6](#_Toc175481221)

[2.2 Memória de instruções 7](#_Toc175481222)

[2.3 Banco de Registradores 7](#_Toc175481223)

[2.4 ALU 7](#_Toc175481224)

[2.5 Memória de Dados 7](#_Toc175481225)

[2.6 Controlador 7](#_Toc175481226)

[3 Instruções 8](#_Toc175481227)

[3.1 Registradores 8](#_Toc175481228)

[3.2 Barramento 8](#_Toc175481229)

[4 Modos de endereçamento 9](#_Toc175481230)

[5 cojunto de instruções 9](#_Toc175481231)

[6 aplicações e uso do z80 9](#_Toc175481232)

[7 conclusão 9](#_Toc175481233)

[8 referencia bibliografica 9](#_Toc175481234)

# INTRODUÇÃO

**1.1 Contextualização**

A arquitetura MIPS (Microprocessor without Interlocked Pipeline Stages) é uma das arquiteturas de conjunto de instruções mais influentes e amplamente estudadas no campo da engenharia de computadores. Desenvolvida nos anos 1980 como parte do movimento RISC (Reduced Instruction Set Computer), a MIPS foi projetada com o objetivo de simplificar as operações de hardware, melhorando a eficiência e o desempenho dos processadores.

Embora a MIPS tenha evoluído ao longo do tempo para incluir funcionalidades mais avançadas, como o pipeline e a execução fora de ordem, seu núcleo básico continua sendo uma ferramenta educacional essencial para estudantes e profissionais que desejam compreender os princípios fundamentais da arquitetura de computadores. Este trabalho explora a implementação básica da arquitetura MIPS utilizando o Logisim Evolution, uma ferramenta de simulação digital que permite a construção e análise de circuitos digitais.

**1.2 Objetivos do Artigo**

O principal objetivo deste artigo é documentar a implementação da arquitetura MIPS em um ambiente simulado, utilizando o Logisim Evolution. O foco está na criação de um processador MIPS básico, capaz de executar um conjunto de instruções fundamentais sem a complexidade adicional do pipeline. Especificamente, o artigo busca:

1. **Apresentar os conceitos básicos da arquitetura MIPS**, abordando sua estrutura e o conjunto de instruções utilizado.
2. **Detalhar o processo de implementação no Logisim Evolution**, incluindo a configuração da memória, banco de registradores, unidade de controle e ALU (Unidade Lógica e Aritmética).
3. **Demonstrar a execução de instruções básicas** (aritméticas, lógicas, de memória e de controle de fluxo) através de simulações no Logisim.
4. **Discutir as limitações e desafios encontrados durante a implementação**, bem como possíveis melhorias e expansões futuras do projeto.
5. **Contribuir para o aprendizado de estudantes e entusiastas** de arquitetura de computadores, fornecendo um exemplo prático de como construir um processador MIPS funcional em um ambiente de simulação.

# Arquitetura e componentes

O conjunto MIPS é feito com uma arquitetura de 32 bits, porém como estaremos aqui fazendo apenas para fim didáticos com as funções simples, a arquitetura terá apenas 16 bits de largura.

Para entender sobre o MIPS é necessário saber sobre os seus componentes e de como eles estão organizados. Resumidamente ele pode ser separado por 6 grandes blocos que juntos formar a arquitetura como um todo.

## 2.1 Program Counter

Quando se fala de processadores no geral, o program counter sempre aparece sendo essencial e indispensável. Ele tem o papel de apontar o endereço da memória a ser usada que contém uma instrução. No final do ciclo de clock normalmente é somado 1 ao endereço que estava guardando para apontar para a próxima instrução.

## 2.2 Memória de instruções

Aqui é armazenado todas as instruções que o programa irá rodar, podendo ser previamente carregado em um arquivo com as instruções com código de máquina. Cada endereço dessa memória se refere a um instruções que no caso desse projeto são de 16 bits.

## 2.3 Banco de Registradores

No banco de registradores temos um conjunto de registradores que serve para auxiliar em operações e armazenar dados temporários dependendo de cada instrução. Pode-se colocar valores previamente em algum dos registradores, porém diferentemente da memória de instruções aqui durante a execução do programa valores podem ser escritos dentro dos registradores do mesmo jeito que pode ser lidos.

## 2.4 ALU

A unidade lógica aritmética ou em inglês arithmetic logic unit (ALU), é onde é feito todas as operações lógicas e aritméticas das instruções, presente em quase todas as instruções.

## 2.5 Memória de Dados

Na memória de Dados é armazenado os valores em definitivo das operações a qual o usuário deseja guardar, sendo ele volátil, ou seja, caso o programa seja encerrado de maneira indevida, os dados que estavam na memória se perderão.

## 2.6 Controlador

A unidade de controle, é fundamental para qualquer processador. É a parte responsável por controlar entradas e saída de dados de cada bloco dependendo de cada instrução. Mesmo sendo essencial é possível o programa rodar sem ele, porém os dados teriam que ser direcionados manualmente, o que demanda mas tempo porém é melhor para o aprendizado.

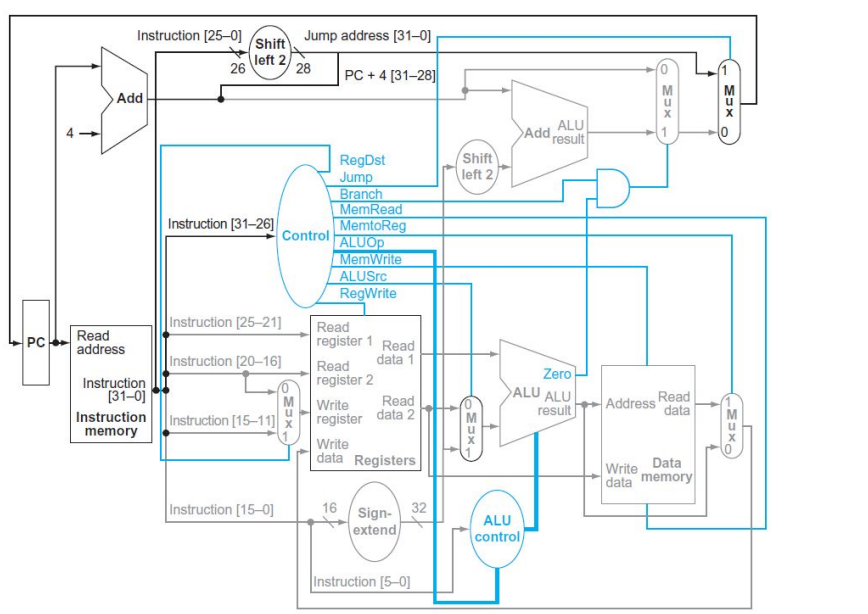


Figura 1: Arquitetura MIPS resumida

# Instruções

As instruções do MIPS se divide mem três tipos (R, I e J) e cada uma tem funcionalidades diferentes e usam números de registradores diferentes.

As instruções do tipo R, são as instruções mais básicas de lógicas e aritméticas, onde pega um valor em um registrador faz a operação com o valor de outro registrador e escreve o resultado em um terceiro registrador, sendo assim utilizando três registradores dentro da sua instrução. Ao todo serão 8 tipos de instruções do tipo R: ADD, SUB, AND, OR, XOR, NAND, NOR e SLT. O que define o que será cada uma é a parte chamada funct que será explicada mais a frente.

Instruções do tipo I são aquelas que usam acesso direto a algum endereço ou usando esse valor como constante chamado também de valor imediato. No geral são 5 instruções do tipo I: LW, SW, ADDI, BEQ e SLTI. Diferente do tipo R o tipo I usam apenas 2 registradores do banco de registradores e nos casos LW e SW também usam a memória.

Por último temos do tipo J que trás apenas uma instrução, JUMP, responsável por pular para o endereço indicado por ela mesmo, ou seja, em vez das instruções serem executadas de maneira sequencial, quando chega no JUMP ele realizará um salto na memória podendo ir para frente ou voltar na memória de instruções.

## Formato das instruções

Para falar sobre cada instrução utiliza-se uma linguagem pra definir o que é cada instrução, chamamos essa linguagem de assembly. Para podermos representarmos cada instrução é usado o mnemônico (add, sub, lw, sw ...) e em seguida os endereços ou constantes a ser utilizados.

Para a arquitetura MIPS existe uma simbologia para o que estamos acessando: quando tem o símbolo ‘$’ junto com um número trata-se de um registrador que em específico nesse trabalho possui apenas 8 ou seja os registradores vão de $0 até $7, como está sendo tratado de registradores foi acrescentado a letra r no meio ficando $r0 até $r7 os registradores.

Quando estamos falando de um valor imediato ou endereço de algum registrador é colocado apenas o esse valor sem simbologia nenhuma antes e nem depois.

Usa-se uma sintase diferente caso a instrução peça um acesso a memória, que no nosso caso refere-se as instruções LW e SW. Para falar o endereço da memória a ser acessado é colocado uma constante junto com um registrador dentro de um parênteses, com isso é falado que o endereço é o resultado da soma da constante com o valor dentro do registrador.

Para mostrar o que foi falado aqui vai um exemplo: add $r0, $r1, $r2, essa é uma instrução do tipo R que retrata a seguinte operação -> ($r0) = ($r1) + ($r2), ou seja, o registrador r0 irá receber a soma dos conteúdos dos registradores r1 e o r2 respectivamente. Outro exemplo podemos ver em ->lw $r1, 5($r3), a instrução lw carrega da memória determinado valor, portanto nessa instrução temos que no registrador r1 receberá o valor que está na memória no endereço 5 +($r3), caso no registrador r3 estivesse o valor 8, por exemplo, então r1 iria receber o conteúdo da memória na posição 13 (5+8).

A tabela a seguir mostra todas as instruções com os seus respectivos formatos

|  |  |  |  |
| --- | --- | --- | --- |
| **Tipo** | **Instrução** | **Assembly** | **descrição** |
| **R** | ADD | add $ra,$rb, $rc | ($ra) = ($rb) + ($rc) |
| SUB | sub $ra,$rb, $rc | ($ra) = ($rb) - ($rc) |
| AND | and $ra,$rb, $rc | ($ra) = ($rb) ^ ($rc) |
| OR | or $ra,$rb, $rc | ($ra) = ($rb) ᵛ ($rc) |
| NAND | nand $ra,$rb, $rc | ($ra) = ($rb) ⊼ ($rc) |
| NOR | nor $ra,$rb, $rc | ($ra) = ($rb) ⊽ ($rc) |
| XOR | xor $ra,$rb, $rc | ($ra) = ($rb) ⊕ ($rc) |
| SLT | slt $ra,$rb, $rc | ($rb) < ($rc), ($ra) = 1; ($ra) = 0 |
| **I** | LW | lw $ra, k ($rb) |  |
| SW |  |  |
| ADDI |  |  |
| BEQ |  |  |
| SLTI |  |  |
| **J** | JUMP |  |  |

## Convertendo para o código de máquina

Se rodássemos um programa contendo apenas as instruções em assembly, ele não rodaria, pois não há nada que traduza o código para o computador entender. Portanto, iremos transformar a instrução em executável para o computador, o que muda para cada função, tendo em comum o tamanho de 16 bits. Para cada instrução formada, também se forma uma palavra, como é chamado.

A tabela a seguir mostra como cada palavra é separada de acordo com o seu tipo:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Tipo** | **Opcode** | **Rs** | **Rt** | **Rd** | **funct** |
| **R** | 3 bits | 3 bits | 3 bits | 3 bits | 4 bits |
|  | | | | | |
|  | **Opcode** | **Rs** | **Rt** | **Immediate/address** | |
| **I** | 3 bits | 3 bits | 3 bits | 7 bits | |
|  | | | | | |
|  | **Opcode** | **addresss** | | | |
| **J** | 3 bits | 13 bits | | | |

As instruções do tipo R são utilizadas para operações aritméticas e lógicas que envolvem apenas registradores. Este tipo de instrução é composto pelos seguintes campos:

* **Opcode (3 bits)**: Identifica o tipo geral de operação a ser realizada. No caso das instruções do tipo R, o opcode especifica que a instrução é do tipo R.
* **rs (3 bits)**: Registrador fonte. Contém o primeiro operando para a operação.
* **rt (3 bits)**: Registrador fonte. Contém o segundo operando para a operação.
* **rd (3 bits)**: Registrador de destino. Onde o resultado da operação será armazenado.
* **funct (4 bits)**: Campo de função. Especifica a operação exata (como adição, subtração, etc.) a ser realizada pela ALU (Unidade Lógica e Aritmética).

As instruções do tipo I são utilizadas para operações que envolvem um registrador e um valor imediato, ou para operações de acesso à memória. Este formato é composto por:

* **Opcode (3 bits)**: Identifica o tipo de operação a ser realizada.
* **rs (3 bits)**: Registrador fonte. Contém o operando que será usado na operação ou o endereço base para operações de carga/armazenamento.
* **rt (3 bits)**: Registrador de destino. Onde o resultado da operação será armazenado ou o registrador que contém o dado a ser armazenado na memória.
* **Immediate/address (7 bits)**: Um valor imediato usado diretamente na operação ou um endereço utilizado para operações de acesso à memória.

As instruções do tipo J são utilizadas para saltos incondicionais, onde o fluxo de execução é desviado para um novo endereço. Este formato é composto por:

* **Opcode (3 bits)**: Identifica que a instrução é um salto (jump).
* **Address (13 bits)**: Especifica o endereço para onde o controle deve saltar.

Para exemplificar a próxima tabela trará um exemplo de cada instrução e de como ele ficaria em código de máquina

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **TIPO** | **Assembly** | | **Código de máquina** | | | | |
| **Operação** | **Código** | **Opcode** | **Rs** | **Rt** | **Rd** | **funct** |
| R | ADD | $r1, $r2, $r3 | 000 | 010 | 011 | 001 | 0000 |
| SUB | $r1, $r2, $r3 | 000 | 010 | 011 | 001 | 0001 |
| AND | $r1, $r2, $r3 | 000 | 010 | 011 | 001 | 0010 |
| OR | $r1, $r2, $r3 | 000 | 010 | 011 | 001 | 0011 |
| NAND | $r1, $r2, $r3 | 000 | 010 | 011 | 001 | 0100 |
| NOR | $r1, $r2, $r3 | 000 | 010 | 011 | 001 | 0101 |
| XOR | $r1, $r2, $r3 | 000 | 010 | 011 | 001 | 0110 |
| SLT | $r1, $r2, $r3 | 000 | 010 | 011 | 001 | 0111 |
| I | LW | $r1, 10($r2) | 100 | 010 | 001 | 0001010 (address) | |
| SW | $r1, 10($r2) | 101 | 010 | 001 | 0001010 (address) | |
| ADDI | $r1, $r2, 10 | 111 | 010 | 001 | 0001010 (immediate) | |
| BEQ | $r1, $r2, 10 | 110 | 010 | 001 | 0001010 (immediate) | |
| SLTI | $r1, $r2, 10 | 001 | 010 | 001 | 0001010 (immediate) | |
| J | JUMP | J 10 | 010 | 0000000001010 (address) | | | |

# Modos de endereçamento

# cojunto de instruções

# aplicações e uso do z80

# conclusão

# referencia bibliografica