



UNIVERSIDADE FEDERAL DE LAVRAS

ARQUITETURA DE COMPUTADORES II

LUIZ HENRIQUE A. CORREIA

BRUNO CRESPO FERREIRA

DAVI HERMÓGENES SIQUEIRA

GABRIEL ROGERIO APARECIDO PAES SILVA

VINICIUS DE OLIVEIRA FABIANO

**TRABALHO PRÁTICO**

**UFLA-RISC**

LAVRAS - 2023

## 1. Introdução

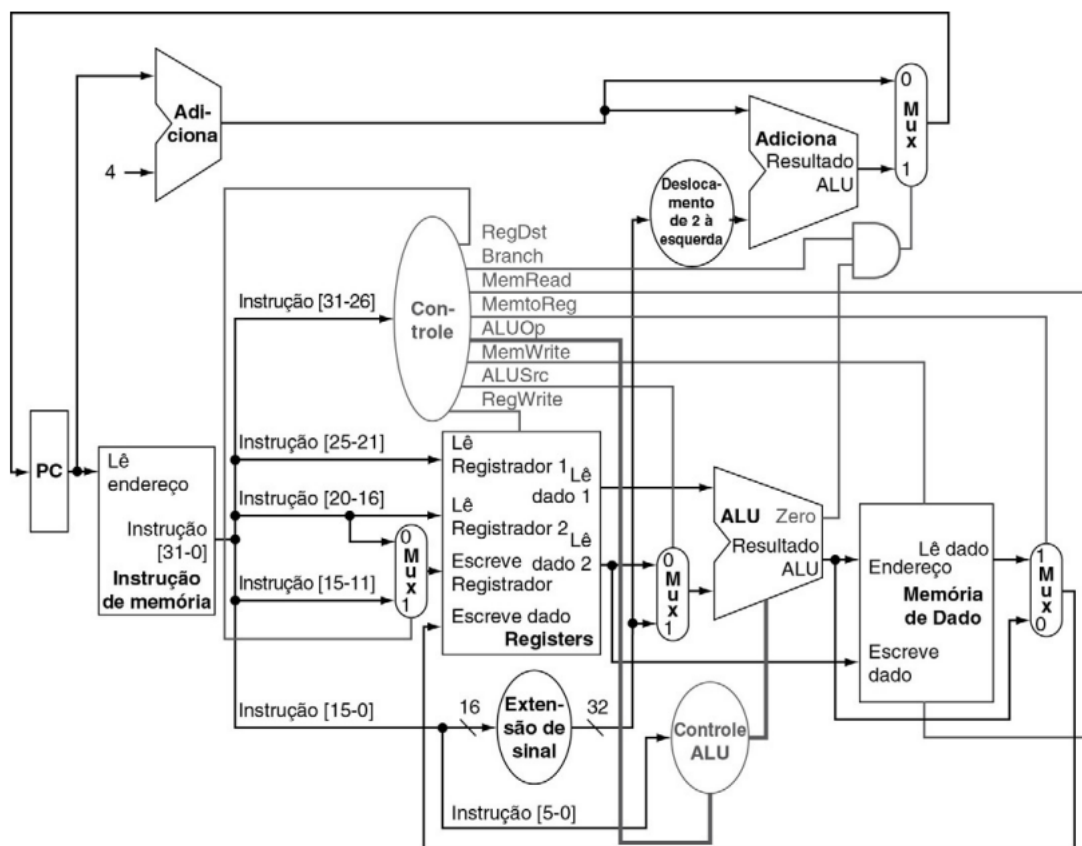
Este trabalho prático tem como objetivo o estudo juntamente com o desenvolvimento e implementação de um simulador funcional para o processador monociclo RISC, utilizando determinados programas escritos em linguagem de programação C++ orientada a objetos, a fim de simular tal processador.

## 2. Implementação

A implementação do processador UFLA-RISC possui muitas estratégias, infraestruturas e semelhanças ao projeto do processador MIPS realizado na disciplina anterior (Arquitetura de Computadores I). Pode-se dizer que as grandes diferenças se encontram no formato das instruções, embora ainda sejam 32 bits, a estrutura de cada uma foi alterada.

A lógica do caminho de dados tende à arquitetura MIPS, representada pela figura 1. Vale ressaltar que o caminho de dados abaixo foi pego como exemplo, não necessariamente valores e técnicas foram usados no UFLA-RISC, como por exemplo não soma-se o  $PC + 4$ , mas agora se faz  $PC + 1$ .

### Figura 1 - Caminho de dados



fonte: [\[1\] DAVID A. Patterson E JOHN L. Henessy. Organização e projeto de computadores: A interface Hardware Software. Morgan Kaufmann, Tradução da 4ª Edição, 2014.](#)

## 2.1. Interpretador de Instruções

O interpretador de instruções está ligado ao processador, logo ao criar o processador, o programa “Conversor.cpp” é acionado e pede-se ao usuário o nome de um arquivo texto armazenado na pasta *files* para a sua conversão.

A criação de um código e seu armazenamento são explicados essencialmente na seção [3. Como usar o software](#). Portanto, aqui explicaremos como as instruções são transformadas em binário e armazenadas na memória.

O programa funciona da seguinte maneira:

1. Pede-se o nome do arquivo .txt ao usuário.
2. Linha por linha, o programa cria instruções em binário da seguinte maneira:
  - a. Faz a leitura da linha.
    - i. Se for vazia, pula-se para a próxima linha.
    - ii. Se for “EOF”, encerra-se a leitura.

- b. A instrução “address” pode ser a primeira.
  - i. Caso não for, a posição inicial da memória de texto é igual a 0.
  - ii. Caso for, ele não é escrito na memória de instruções, na posição dada pelo próprio é inicializado a instrução logo abaixo dele.
- c. As demais instruções que virão sofrem a seguinte formatação:
  - i. Remove-se qualquer ‘,’ da linha.
  - ii. Remove-se qualquer ‘R’ da linha.
  - iii. Remove-se qualquer ‘#’ e quaisquer outros caracteres após ele.
  - iv. Remove-se espaços adjacentes.
  - v. A instrução é armazenada em um vector de strings.
- d. É chamado um método, que baseado na posição 0 do vector, cria a correspondente instrução em binário.
- e. A instrução é armazenada na memória de texto.

Ex: imagine um código fonte somente com:

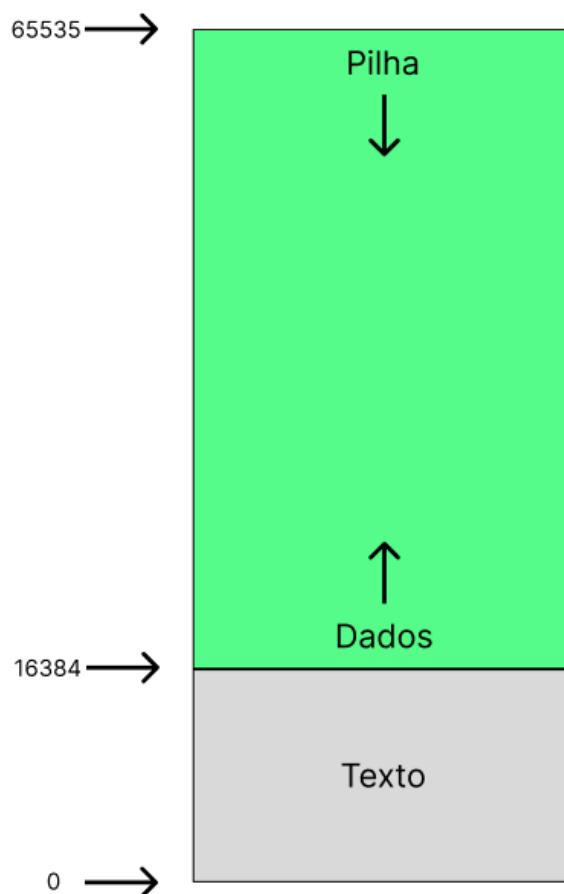
add R8, R16, R17 # é uma instrução de add

A formatação correspondente seria: add 8 16 17

## 2.2. IF

O funcionamento do estágio IF (Instruction Fetch) simplesmente lê as instruções a partir do começo da memória de texto ou do começo estipulado pela instrução *address* (caso ela for a primeira).

A memória é representada por um vetor bitset de tamanho 65536 endereços, sendo que cada posição possui 32 bits. A memória é dividida de acordo com a figura 2.

**Figura 2 - Divisão da memória**

Fonte: autoria própria

A parte de texto e dados depende da instrução *address*, porém por conveniência, foi imposto que a memória de texto não pode ultrapassar um quarto da memória total, logo o tamanho da memória de texto - consequentemente a quantidade de instruções possíveis - foi definido como 16384 endereços/instruções.

A cada instrução lida pelo PC, o mesmo é incrementado em uma unidade na posição (tamanho de uma instrução) e tal instrução é enviada para o caminho de dados nas próximas seções.

### 2.3. ID

Esta seção se propõe única e exclusivamente a esclarecer os métodos empregados na implementação do ID (Instruction Decoder).

O estágio de decodificação de instrução é encarregado de decodificar a instrução de 32 bits obtida do IF (instruction fetch), que por sua vez obtém da memória de instruções.

A implementação do ID (Instruction decode) se deu da seguinte forma:

- Duas classes foram criadas a fim de melhor organizar seu funcionamento.
- A classe Id é responsável apenas por receber a instrução de 32 bits e recolher os campos (bits) do opcode, ra, rb, rc, const16 e const24 (vale ressaltar que nesta Arquitetura não há o campo funct, isto é, a instrução é decodificada única e exclusivamente através do opcode).
- A Classe Controle é a encarregada de alterar os sinais de controle a depender da instrução. Um objeto da Classe Controle é criado na classe Id, passando como parâmetro para o construtor apenas o opcode da instrução. Após isso, os valores dos registradores ra, rb e rc são recolhidos através da Classe Registradores, para serem utilizados em estágios seguintes, como operandos da operação.

De modo geral, o estágio ID obtém informações sobre os registradores envolvidos na operação e determina os sinais de controle necessários para executar tal instrução.

## 2.4. EX/MEM

Por sequência, a etapa ExMem tem como objetivo realizar as operações no estágio de Execução/Memória após a decodificação. Ela é responsável por executar diferentes tipos de instruções, como operações aritméticas, instruções de desvio e operações de leitura e escrita na memória.

A classe ExMem possui os seguintes membros:

- alu: Um objeto da classe Alu que lida com as operações aritméticas, de desvio e de memória.
- regs: Um ponteiro para a classe Registradores, que contém os registradores utilizados nas operações.
- ifStage: Um ponteiro para a classe If, representando o estágio de busca de instruções.
- controle: Um ponteiro para a classe Controle, responsável por controlar os sinais de controle das instruções.

Durante a execução, a classe ExMem realiza as seguintes operações:

1. Verifica os sinais de controle para determinar o tipo de instrução a ser executada.
2. Se os sinais de controle indicarem que a instrução não é um desvio, nem uma operação de leitura ou escrita na memória, são executadas as operações aritméticas, chamando o método `instrucoesAritmeticas()` da Alu.
3. Se os sinais de controle indicarem que a instrução é um desvio, são executadas as operações de desvio chamando o método `instrucoesDeDesvio()` da Alu.
4. Se os sinais de controle indicarem que a instrução é uma operação de leitura ou escrita na memória, são executadas as operações de memória chamando o método `instrucoesDeMemoria()` da Alu.

Essas operações são realizadas utilizando os registradores, o estágio de busca (If), o estágio de decodificação (Id) e os sinais de controle fornecidos pela classe Controle.

Após identificadas as naturezas das operações, a classe Alu (Unidade Lógica Aritmética) fica responsável por tratar cada tipo de instrução de acordo com sua operação específica.

Para as operações aritméticas, a Alu executa cálculos matemáticos, como adição e subtração, utilizando os operandos fornecidos pelos registradores e seguindo as regras de codificação da arquitetura UFLA-RISC. Os resultados dessas operações são armazenados em registradores específicos ou em variáveis temporárias.

No caso das instruções de desvio, a Alu realiza cálculos relacionados aos desvios condicionais ou incondicionais, levando em consideração os sinais de controle e os valores presentes nos registradores. Esses cálculos envolvem a verificação de condições lógicas e a atualização do contador de programa (PC) para determinar o próximo endereço de instrução a ser buscado.

Já para as operações de leitura e escrita na memória, a Alu é responsável por realizar o acesso aos endereços de memória específicos, utilizando os registradores

para obter os dados a serem lidos ou escrever os dados fornecidos nos locais de memória desejados.

## 2.5. WB

O estágio WB é responsável pela escrita no banco de registradores do nosso processador.

Para isso, são utilizados os sinais de RegWrite, Jump, MemtoReg, RegDst e Overflow para a decisão de onde e com que valores serão feitas as escritas, retornando essas informações para uma função responsável pela escrita nos registradores.

## 3. Como usar o software

Grosseiramente, basta compilar e executar o programa “App.cpp”.

Inicialmente, deve-se criar um arquivo .txt e armazená-lo na pasta files no diretório raiz do projeto, feito isso, ao criar o código, deve-se respeitar as seguintes regras:

1. Uma instrução por linha.
2. Comentários usando # desconsidera até o final da linha.
3. Pode-se deixar linhas em branco.
4. Todo número deve ser representado com inteiros.
5. Os registradores seguem o padrão RN, sendo N o número do registrador (0 a 31).
6. Os operandos devem ser separados por um espaço pelo menos, o uso de vírgula é opcional.
7. O arquivo deve terminar a leitura com a palavra “EOF”.

Abaixo estão as instruções suportadas pelo processador e suas respectivas padronizações.

Obs: *const16* representa uma constante de 16 bits, *endereço* uma constante de 24 bits e *imm* pode ser tanto uma constante de 8 bits, quanto de 16 bits, dependendo da instrução. O campo de opcode e registradores sempre terá 8 bits.



**Tabela 1** - Instruções do projeto

Identificador instrução	Descrição	Opcode	Formato
address	Representa a posição inicial da memória de texto	00000000	address <i>endereço</i>
add	Soma o conteúdo de ra e rb e armazena o resultado em rc	00000001	add rc, ra, rb
sub	Subtrai o conteúdo de rb do conteúdo de ra e armazena o resultado em rc	00000010	sub rc, ra, rb
zeros	Faz o conteúdo de rc zerar	00000011	zeros rc
xor	Efetua xor lógico bit a bit de ra e rb e coloca resultado em rc	00000100	xor rc, ra, rb
or	Efetua or lógico bit a bit de ra e rb e coloca resultado em rc	00000101	or rc, ra, rb
passnota	Faz conteúdo de rc valer o complemento do conteúdo de ra	00000110	passnota rc, ra
and	Efetua and lógico bit a bit de ra e rb e coloca resultado em rc	00000111	and rc, ra, rb
asl	Coloca cada bit rai em rci+1 e preenche com 0 a posição rci. O registrador rb indica quantos bits serão deslocados	00001000	asl rc, ra, rb
asr	Coloca cada bit rai em rci-1 e preenche com o valor do bit ra31 as posições deslocadas. O registrador rb indica quantos bits devem ser deslocados	00001001	asr rc, ra, rb
lsl	Coloca cada bit rai em rci+1 e preenche com 0 as posições deslocadas. O registrador rb indica quantos bits devem ser deslocados	00001010	lsl rc, ra, rb
lsr	Coloca cada bit rai em rci-1 e preenche com 0 as posições deslocadas. O registrador rb indica quantos bits devem ser deslocados	00001011	lsr rc, ra, rb

passa	Faz conteúdo de rc igual ao conteúdo de ra	00001100	passa rc, ra
lch	Carrega nos 2 bytes mais significativo de rc o conteúdo de Const16	00001101	lch rc, <i>const16</i>
lcl	Carrega nos 2 bytes menos significativos de rc o conteúdo de Const16	00001110	lcl rc, <i>const16</i>
load	Carrega no registrador rc o conteúdo da memória endereçada pelo registrador ra	00001111	load rc, ra
store	Carrega na posição da memória endereçada pelo registrador rc o conteúdo do registrador ra	00010000	store rc, ra
jal	Realiza chamadas a procedimentos guardando o endereço do PC atual no registrador R31 (para o retorno após o procedimento) e colocando em PC o valor do endereço do desvio (primeira instrução do procedimento)	00010001	jal <i>endereço</i>
jr	Armazena o conteúdo do registrador rc em PC	00010010	jr rc
beq	Realiza desvio de fluxo condicional. Compara o conteúdo dos registradores ra e rb e se forem iguais o registrador PC recebe o endereço de memória para o qual o fluxo de execução deve ser transferido	00010011	beq ra, rb, <i>endereço</i>
bne	Realiza desvio de fluxo condicional. Compara o conteúdo dos registradores ra e rb e se forem diferentes o registrador PC recebe o endereço de memória para o qual o fluxo de execução deve ser transferido	00010100	bne ra, rb, <i>endereço</i>
j	Realiza desvio de fluxo incondicional. O endereço da próxima instrução a ser executada será o endereço de memória	00010101	j <i>endereço</i>

	presente na instrução		
halt	Encerra o processador	11111111	halt


**Tabela 2** - Instruções do grupo

Identificador instrução	Descrição	Opcode	Formato
slt	Se o conteúdo de ra é menor que o de rc, é atribuído 1 ao conteúdo de rc, caso contrário 0	00010110	slt rc, ra, rb
slti	Se o conteúdo de ra é menor que o valor de <i>imm</i> , é atribuído 1 ao conteúdo de rc, caso contrário 0	00010111	slti ra, rb, <i>imm</i>
smt	Se o conteúdo de ra é maior que o de rc, é atribuído 1 ao conteúdo de rc, caso contrário 0	00011000	smt rc, ra, rb
inc	Incrementa em uma unidade o conteúdo de rc	00011001	inc rc
dec	Decrementa em uma unidade o conteúdo de rc	00011010	dec rc
addi	Adiciona um valor de até 24 bits no conteúdo de rc	00011011	addi ra, rb, <i>imm</i>
subi	Subtrai um valor de até 24 bits no conteúdo de rc	00011100	subi ra, rb, <i>imm</i>
nand	Efetua nand lógico bit a bit de ra e rb e coloca resultado em rc	00011101	nand rc, ra, rb
nor	Efetua nor lógico	00011110	nor rc, ra, rb

	bit a bit de ra e rb e coloca resultado em rc		
--	---	--	--

Um exemplo da representação de todas as instruções é mostrado abaixo:

**Figura 3** - Exemplo de todas as instruções

 exemploTodasInstrucoes.txt - Bloco de Notas

Arquivo Editar Formatar Exibir Ajuda

# a finalidade desse arquivo é somente mostrar a padronização das instruções

# instruções do processador

address 12182 # o começo da memória de texto começa na posição 12182

add R8, R16, R17

sub R8, R9, R10

zeros R16

xor R16, R17, R18

or R2, R16, R17

passnota R2, R3

and R16, R17, R18

asl R2, R16, R6

asr R3, R16, R5

lsl R2, R16, R6

lsr R3, R16, R5

passa R2, R16

lch R8 30000

lcl R8 14032

load R20, R7

store R20, R6

jal 1234

jr R31

beq R8, R13, 12

bne R8, R13, 76

j 312

halt

# instruções do grupo

slt R8, R2, R8

slti R8, R2, 122

smt R8, R2, R0

inc R8

dec R8

addi R8, R0, 123

subi R29, R29, 4

nand R16, R17, R18

nor R2, R16, R17

EOF

Fonte: autoria própria

Para simular o processador, basta compilar e executar o programa *App.cpp* e ao inserir o nome do arquivo texto, o processador fará todo o trabalho.

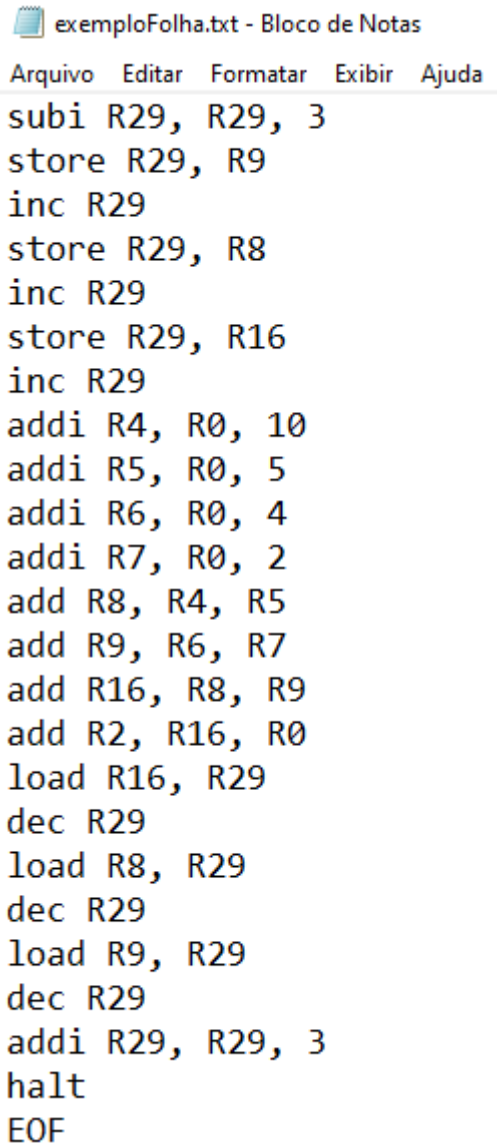
A instrução *address* define a partir de que posição na memória de instruções são guardadas as instruções após ela, ou seja, não se comporta como uma instrução de desvio.

O simulador para quando o mesmo lê uma instrução somente formada somente com 1s.

## 4. Exemplos de testes

Os programas implementados pelo grupo são mostrados a seguir:

**Figura 4 - exemploFolha**




```
exemploFolha.txt - Bloco de Notas
Arquivo  Editar  Formatar  Exibir  Ajuda
subi R29, R29, 3
store R29, R9
inc R29
store R29, R8
inc R29
store R29, R16
inc R29
addi R4, R0, 10
addi R5, R0, 5
addi R6, R0, 4
addi R7, R0, 2
add R8, R4, R5
add R9, R6, R7
add R16, R8, R9
add R2, R16, R0
load R16, R29
dec R29
load R8, R29
dec R29
load R9, R29
dec R29
addi R29, R29, 3
halt
EOF
```

Soma dois valores e escreve em um novo registrador.

Fonte: autoria própria

**Figura 5 - swap**

 swap.txt - Bloco de Notas

Arquivo Editar Formatar Exibir Ajuda

```


addi R4, R0, 10
addi R5, R0, 31
subi R29, R29, 1
store R29, R4
inc R29
store R29, R5
load R4, R29
dec R29
load R5, R29
addi R29, R29, 2
halt
EOF

```

Troca os valores entre dois registradores.

Fonte: autoria própria

**Figura 6 - somaElementos**

 somaElementos.txt - Bloco de Notas

Arquivo Editar Formatar Exibir Ajuda

```


lcl R8, 20      # basta alterar o valor inteiro nessa linha para mudar até qual número somar
addi R6, R0, R0
add R6, R6, R8
dec R8
bne R8, R0, 2
halt
EOF

```

Soma os elementos até certo valor e guarda o resultado em um registrador.

Fonte: autoria própria

**Figura 7 - numQuadrado**


 numQuadrado.txt - Bloco de Notas

```
Arquivo  Editar  Formatar  Exibir  Ajuda
addi R6, R0, 5      # numero pra elevar ao quadrado
add R7, R0, R6      # r7 recebe r6
dec R6              # decrementa o 5 em 1
add R8, R8, R7      # acumulando os +5
addi R9, R0, 0      # reg auxiliar
bne R6, R9, 2
halt
EOF
```

Faz o quadrado de um número.

Fonte: autoria própria

**Figura 8 - testeMassivoIncremento**

 testeMassivoIncremento.txt - Bloco de Notas

```
Arquivo  Editar  Formatar  Exibir  Ajuda
add R8, R0, R0      # t0 sofrerá o teste massivo
lcl R4, 30000       # a0 será o ponto de parada
j 384

# teste massivo de incremento
address 384         # armazena as instruções abaixo na posição 384
inc R8              # t0++
smt R9, R8, R4      # t1 recebe 1 se t0 é maior que a0
bne R9, R0, 10      # se t0 é maior que a0, encerramos o programa
j 384

address 10
halt
EOF
```

O valor de um registrador vai de 0 até um certo número.

Fonte: autoria própria

## 5. Conclusões

Com o projeto revisamos conceitos aprendidos na disciplina passada e tentamos melhorá-los neste trabalho.



## 6. Referências bibliográficas

[1] DAVID A. Patterson E JOHN L. Henessy. Organização e projeto de computadores: A interface Hardware Software. Morgan Kaufmann, Tradução da 4ª Edição, 2014.

[2] std::bitset. cplusplus. Disponível em:  
<https://cplusplus.com/reference/bitset/bitset/>. Acesso em: 20 de fev. de 2023.

[3] MIPS 101. Nanyang Technological University of Singapore. Disponível em:  
[MIPS 101 \(ntu.edu.sg\)](https://mips101.ntu.edu.sg/). Acesso em: 20 de fev. de 2023.