

Projeto Montador RISC-V

Alvaro G. S. Neto¹, Esdras A. Ferreira¹

¹Universidade Federal de Viçosa - Florestal

alvaro.g.neto@ufv.br (RA: 5095), esdras.ferreira@ufv.br (RA: 5102)

Abstract. *The main objective of this work is to synthesize the knowledge related to the process of code execution on a processor. To achieve this goal, the project involves the creation of a program that resembles an assembler. In other words, the program will be responsible for converting assembly instructions specific to the RISC-V architecture into machine language*

Resumo. *O objetivo principal deste trabalho é sintetizar os conhecimentos relacionados ao processo de execução de códigos em um processador. Para atingir esse objetivo, o projeto envolve a criação de um programa que se assemelhe a um assembler. Em outras palavras, o programa será responsável por converter instruções assembly específicas da arquitetura RISC-V em linguagem de máquina*

1. Execução do Projeto

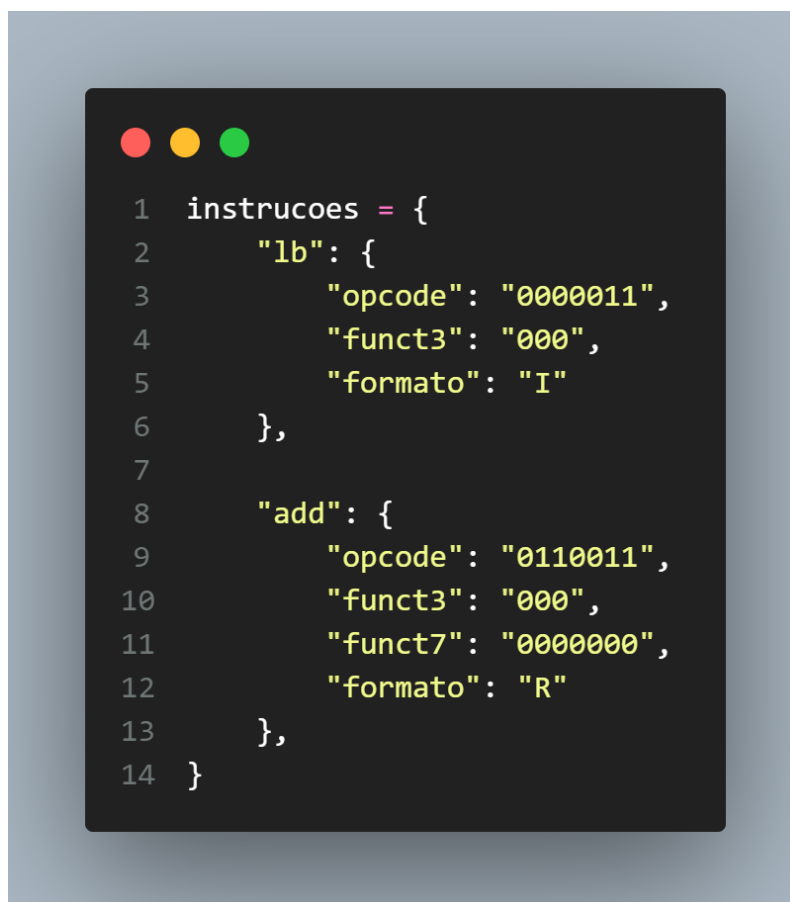
Com o intuito de atender à demanda proposta, optamos por utilizar a linguagem Python, que consideramos a tecnologia mais versátil para esse tipo de projeto. Para executar o programa, basta utilizar o comando “python montador.py”. Uma vez executado, você pode escolher entre fornecer um arquivo .asm como entrada ou inserir as instruções diretamente no terminal. Para finalizar a entrada de instruções, basta digitar 'fim'. Se optar por fornecer um arquivo como entrada, há a possibilidade de obter a saída por meio de um arquivo .bin ou diretamente no terminal.

2. Estrutura do Projeto

2.1. Escalabilidade e Dicionário

Em relação à escalabilidade do código, um dos principais objetivos do projeto foi criar um programa que pudesse ser utilizado em diversas entradas, sem que fosse necessário fazer muitas alterações no código. Para isso, foi utilizado o conceito de encapsulamento, que permite que as funções possam ser reutilizadas em diferentes partes do código, sem que haja interferência entre elas.

Uma das formas de alcançar esse encapsulamento foi a criação de um dicionário com os valores de opcode, formato, funct3 e funct7, que são necessários para a montagem das instruções. Com isso, foi possível resgatar esses valores de forma simples e encaminhá-los para a montagem, sem que fosse necessário escrever diversas linhas de código para cada instrução e facilitando a inserção de novas instruções.



```
1  instrucoes = {
2      "lb": {
3          "opcode": "0000011",
4          "funct3": "000",
5          "formato": "I"
6      },
7
8      "add": {
9          "opcode": "0110011",
10         "funct3": "000",
11         "funct7": "0000000",
12         "formato": "R"
13     },
14 }
```

Figure 1. Dicionário contendo informações sobre as instruções

2.2. Formato das Funções

Os quatro formatos suportados pelo programa são: R, I, S e SB. O formato R é utilizado para instruções que operam apenas com registradores e não possuem imediato. Então haverá bits para o rs1, rs2, funct3, rd e opcode, além do funct7.

Já o formato I é utilizado para instruções que possuem um imediato de 12 bits. Nesse formato, os 12 primeiros bits são usados para identificar o imediato, os próximos contém o rs1, funct3, rd e opcode.

O formato S é utilizado para instruções que armazenam dados na memória, possuindo um imediato de 12 bits e dois registradores. Ele utiliza os sete primeiros bits para conter os bits mais significativos do imediato, seguido dos cinco bits do registrador de origem e mais cinco bits do registrador de destino. Além disso, haverá também funct3, o restante do valor do imediato e o opcode.

Por fim, o formato SB é utilizado para instruções de desvio condicional, que possuem um label de 12 bits e dois registradores. Nesse formato, existe divisão no valor do imediato. O bit mais significativo vem primeiro, depois vem os bits de 2 a 7. Em seguida, vem o rs2, rs1 e o funct3. Depois vem os bits de 8 a 11 e o bit 1 na frente do opcode. Com certeza, esse é o formato mais complexo implementado.

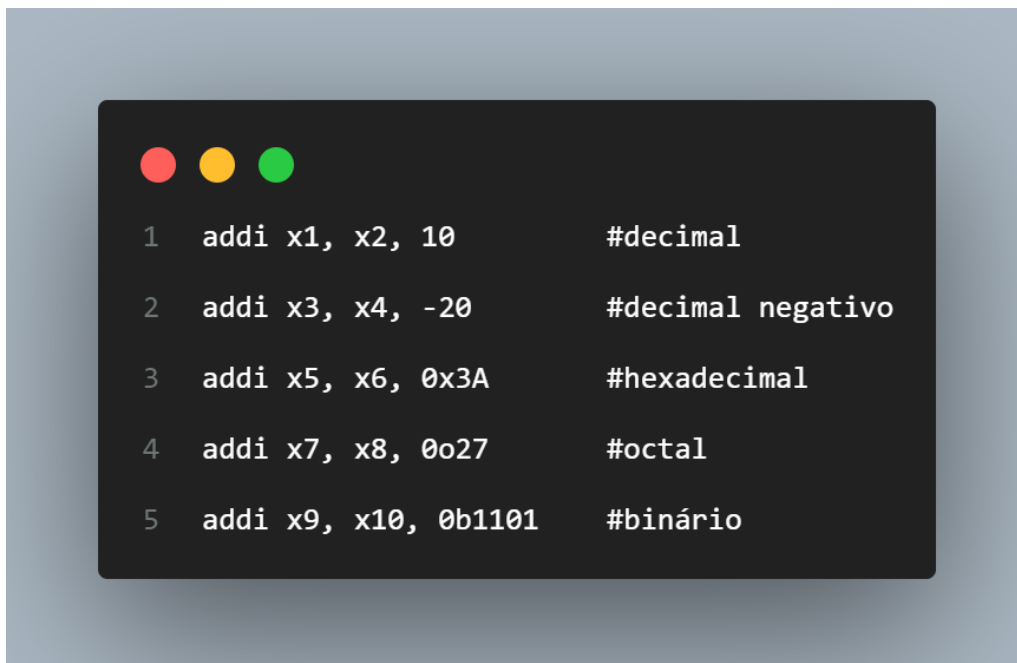
```
1 palavra = funct7 + rs2 + rs1 + funct3 + rd + opcode #formato R
2 palavra = immediate + rs1 + funct3 + rd + opcode #formato I
3 palavra = immediate[0:7] + rs2 + rs1 + funct3 + immediate[7:12] + opcode #formato S
4 palavra = immediate[0] + immediate[2:8] + rs2 + rs1 + funct3 + immediate[8:12] + immediate[1] + opcode #formato SB
```

Figure 2. Organização dos bits de cada formato

2.3. Imediato e Bases

No projeto, para lidar com valores do imediato, foi criada a função “base”. Essa função é responsável por verificar em que base o valor do imediato está e fazer as transformações necessárias. Para realizar essa tarefa, a função verifica se o valor está na base hexadecimal, octal, binária ou decimal. Se for um valor negativo em decimal, a função converte o valor para complemento de dois.

O projeto aceita a entrada de immediatos nas bases hexadecimal, octal, binária, decimal e decimal negativo. Para identificar as bases, é necessário colocar o identificador correto antes do valor. Para hexadecimal é preciso usar o identificador ‘0x’, para octal se usa o ‘0o’, para binário se usa o ‘0b’ e nenhum identificador para decimal. Para valores negativos, é possível utilizar o sinal ‘-’ antes do valor, quando esta na base decimal. Isso garante que o montador consiga lidar com diferentes tipos de valores e que sejam montadas de forma correta.

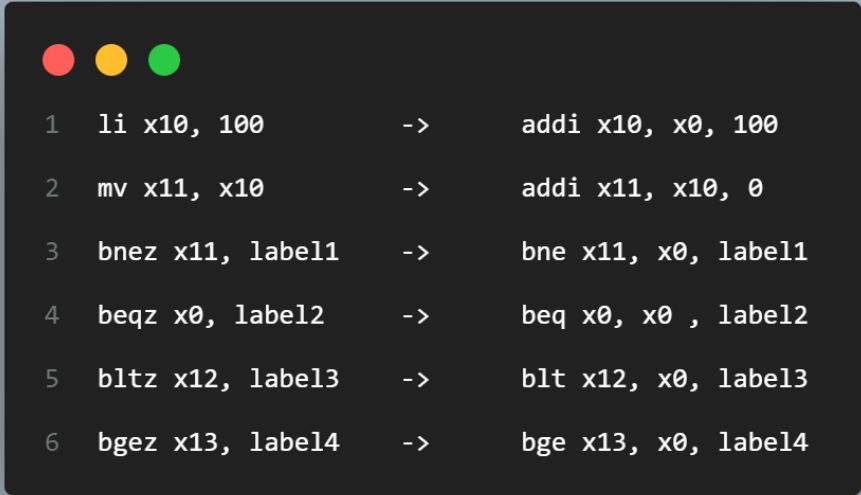
A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It displays five lines of assembly code, each with a line number, an instruction, and a comment indicating the base used for the immediate value.

```
1  addi x1, x2, 10      #decimal
2  addi x3, x4, -20     #decimal negativo
3  addi x5, x6, 0x3A    #hexadecimal
4  addi x7, x8, 0o27    #octal
5  addi x9, x10, 0b1101 #binário
```

Figure 3. Exemplo de entrada

2.4. Pseudo Instruções

Nosso montador atualmente suporta seis pseudo instruções: **li**, **mv**, **bnez**, **beqz**, **bltz** e **bgez**. Para fornecer suporte a essas pseudo instruções, adicionamos uma função específica para cada uma delas que é chamada durante a montagem. Essas funções convertem as pseudo instruções em instruções equivalentes da arquitetura RISC-V adicionando 'x0' e '0' de acordo com a necessidade. Por exemplo, a pseudo instrução 'mv' será convertida em sua instrução equivalente, que no caso é a **addi** somando com o imediato de valor zero. Com isso é possível mover o valor de um registrador para outro de maneira mais legível e simplificada.



1	<code>li x10, 100</code>	->	<code>addi x10, x0, 100</code>
2	<code>mv x11, x10</code>	->	<code>addi x11, x10, 0</code>
3	<code>bnez x11, label1</code>	->	<code>bne x11, x0, label1</code>
4	<code>beqz x0, label2</code>	->	<code>beq x0, x0, label2</code>
5	<code>bltz x12, label3</code>	->	<code>blt x12, x0, label3</code>
6	<code>bgez x13, label4</code>	->	<code>bge x13, x0, label4</code>

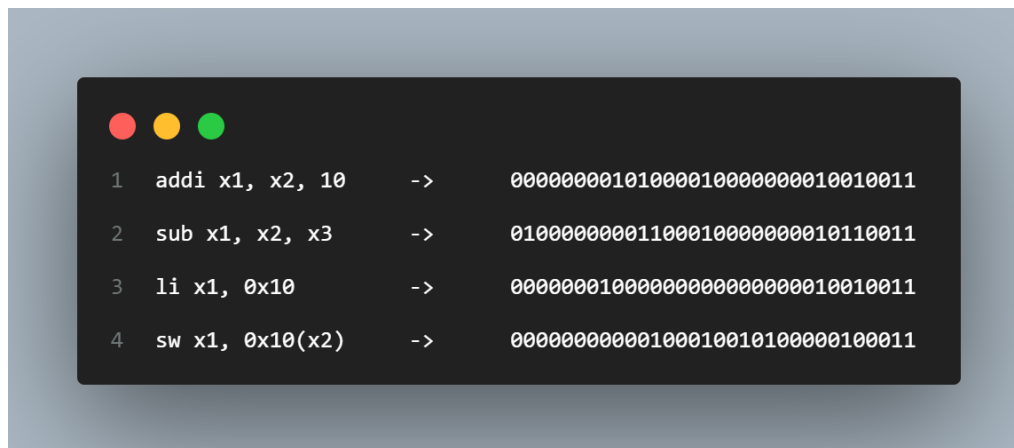
Figure 4. Exemplo de como ocorre as transformações das pseudo instruções

3. Resultados

```
PS C:\Users\alvar\Documents\Programação\montadorAssembly> python montador.py
Montador de código assembly para RISC-V

1 - Para fornecer um arquivo de entrada e um arquivo de saída
2 - Para fornecer um arquivo de entrada e imprimir o resultado no terminal
3 - Para digitar o código no terminal e imprimir o resultado no terminal
```

Figure 5. Menu inicial



A screenshot of a terminal window with a dark background. At the top, there are three colored circles: red, yellow, and green. Below them, four lines of assembly code are listed, each followed by a right-pointing arrow and its corresponding 32-bit binary representation.

Instruction	Binary Representation
1 addi x1, x2, 10	00000000101000010000000010010011
2 sub x1, x2, x3	01000000001100010000000010110011
3 li x1, 0x10	00000001000000000000000010010011
4 sw x1, 0x10(x2)	0000000000100010010100000100011

Figure 6. Exemplo de saída

```
1 - Para fornecer um arquivo de entrada e um arquivo de saída
2 - Para fornecer um arquivo de entrada e imprimir o resultado no terminal
3 - Para digitar o código no terminal e imprimir o resultado no terminal
3
Digite a instrução:addi x1, x2, 10
Digite a instrução:sub x1, x2, x3
Digite a instrução:li x1, 0x10
Digite a instrução:sw x1, 0x10(x2)
Digite a instrução:fim
00000000101000010000000010010011
01000000001100010000000010110011
00000001000000000000000010010011
0000000000100010010100000100011
```

Figure 7. Exemplo de saída pelo terminal

4. Conclusão

Em conclusão, o desenvolvimento do montador em Python para a arquitetura RISC-V permitiu o aprimoramento dos conhecimentos em relação ao processo de compilação de programas em linguagem assembly. A implementação de funções encapsuladas, uso de dicionários e tratamento de pseudo instruções contribuíram para a escalabilidade do código e permitiram que o programa pudesse ser utilizado em diferentes entradas sem a necessidade de muitas alterações. Os testes realizados evidenciaram que o montador foi capaz de realizar com sucesso a conversão das instruções assembly em linguagem de máquina, gerando arquivos binários de saída compatíveis com a arquitetura RISC-V. Embora tenham sido encontradas algumas limitações durante o desenvolvimento, como a falta de suporte a algumas instruções, o projeto serviu como uma base sólida para o aprimoramento do conhecimento em relação ao funcionamento do processo de montagem de programas em assembly.

5. Referências

PATTERSON, D.; HENNESSY, J. *Computer Organization and Design: The Hardware/Software Interface*. 4ª edição, Morgan Kaufmann, 2009.

TANENBAUM, A. S.; WOODHULL, A. S. *Operating Systems: Design and Implementation*. Prentice-Hall, 1997.