

# Trabalho Prático - Montador RISC-V

Letícia Cristina A. Silva<sup>1</sup>, Melissa Alanis S. Oliveira<sup>1</sup>

<sup>1</sup>Ciência da Computação – Universidade Federal de Viçosa (UFV)  
Florestal – MG – Brazil

leticia.c.silva@ufv.br, melissa.alanis@ufv.br

## 1. Introdução

A presente documentação aborda questões acerca da implementação de um **montador RISC-V**, sendo designado como uma arquitetura de conjunto de instruções baseadas nos princípios RISC, livre para ser usado em qualquer finalidade. O principal objetivo deste trabalho é simular a função do **Assembler**, que se constitui como responsável pela “tradução” da linguagem de baixo nível para a de máquina. Em termos gerais, o **Assembly** reflete em uma série de comandos os códigos realizados pelo usuário em programas de alto nível, de forma a facilitar o montador no processo de transformação para a linguagem que o processador é capaz de compreender.

Nesse sentido, tem-se como proposta do trabalho prático a criação de uma versão simplificada do montador RISC-V, em que se recebe um arquivo.asm contendo as instruções em Assembly, a fim de gerar a saída binária para cada tipo de comando, seja via terminal ou arquivo de saída.

## 2. Desenvolvimento

Para a implementação do montador, a equipe optou pela utilização da linguagem **Python**, devido a algumas de suas funções prontas que otimizam o tempo de desenvolvimento do projeto. A seguir, estão as instruções fornecidas ao grupo para a tradução, além das extras, que também estão inclusas na tabela criada para facilitar não só a passagem para binário no momento da decodificação, como também a identificação dos diferentes formatos **R, I** e **S**.

### 2.1. Leitura arquivo de entrada

Antes de mais nada, é importante destacar que todas as funções do montador deste trabalho prático giram em torno da leitura do arquivo de entrada - pois ele é o ponto de partida para a chamada dos demais procedimentos. Inicialmente, o algoritmo se utiliza do módulo `sys.argv`, do Python, que transforma os argumentos passados pelo usuário nas linhas de comando em elementos de um vetor e, com base nisso, verifica e direcionar a forma como a saída dos dados será tratada: caso seja informado no prompt apenas o arquivo de entrada, então a saída será no terminal; se tanto o arquivo de entrada, quanto o de saída forem dados, então os resultados estarão no arquivo de saída.

O arquivo de entrada é percorrido da seguinte maneira: a cada linha lida, o programa substitui as vírgulas, os parênteses e os x's por espaços, além de separar todos os elementos em conjuntos de caracteres (strings), a fim de facilitar a passagem de argumentos para as funções de formatação que são chamadas logo em seguida.

```

nome_arquivo = sys.argv[1]
print(sys.argv)
base = 2 #se meu usuario nao informar a base, ela sera 2
# Se o usuário fornecer um arquivo de saída, abra o arquivo para escrita
if len(sys.argv) > 3 and sys.argv[2] == "-o": #para reconhecer um arquivo de saída precisa do -o, e len(sys.argv) > 3 indica que pessoa informou um
    # arquivo de saída ou base para printar no terminal
    nome_arquivo_saida = sys.argv[3]
    if len(sys.argv) == 5: #A pessoa digita arquivo.py entrada.asm -o saída base
        base = int(sys.argv[4])
    saida = open(nome_arquivo_saida+".asm", "w") # "w" cria um arquivo com o nome fornecido ou, caso já exista um com esse nome, subreescreve o conteúdo
elif len(sys.argv) == 3: #A pessoa digita arquivo.py entrada.asm base
    base = int(sys.argv[2])
with open(nome_arquivo, "r") as arq: # Leitura do arquivo
    for linha in arq:

        palavras = linha.replace(',', ' ') # Retirando as Vírgulas
        palavras = palavras.replace('(', ' ') # Retirando os colchetes
        palavras = palavras.replace(')', ' ') # Retirando os colchetes
        palavras = palavras.replace('x', ' ') # Retirando o X
        palavras = palavras.split() # Separando os elementos
        # Comparando a primeira string de cada linha p/ identificar o tipo
        if (palavras[0] == "add" or palavras[0] == "sll" or palavras[0] == "or" or palavras[0] == "sub"):
            resultado = FormatoR(palavras[0], palavras[1], palavras[2], palavras[3]) #instrucao rd rs1 rs2
        elif (palavras[0] == "andi" or palavras[0] == "addi" or palavras[0] == "xori"):
            resultado = FormatoI(palavras[0], palavras[1], palavras[2], palavras[3]) #instrucao rd rs1 imm
        elif (palavras[0] == "lh"):
            resultado = FormatoI(palavras[0], palavras[1], palavras[3], palavras[2]) #instrucao rd imm(rs1)
        elif (palavras[0] == "sh"):
            resultado = FormatoS(palavras[0], palavras[1], palavras[3], palavras[2]) #instrucao rs1 imm(rs2)
        elif (palavras[0] == "bne"):
            resultado = FormatoS(palavras[0], palavras[1], palavras[2], palavras[3]) #instrucao rs1 rs2 imm

```

Figure 1. Trecho do código de leitura do arquivo

## 2.2. Transformações para binário e complemento de dois

Para converter os valores dos registradores e imediatos, no presente trabalho, foram desenvolvidas duas funções: "Binario" e "Complemento\_Dois", respectivamente. Ambas recebem como parâmetro uma string contendo a representação decimal do valor a ser convertido e retornam o valor binário como uma string, com a quantidade de bits esperada em cada uma. Isso é feito para facilitar a concatenação com os demais valores das instruções.

O bloco de código denominado "Binario" converte o número decimal dos registradores para sua representação binária utilizando uma função nativa da linguagem Python chamada "format". Nessa função, é possível definir tanto a base desejada quanto a quantidade de bits do número, utilizando "05b". A função "Complemento de Dois" lida

```

def Complemento_Dois(valor_decimal):
    if valor_decimal < 0:
        valor_binario = format(valor_decimal*(-1), '012b') # Convertendo para binário, multiplica o valor decimal por -1 pq tem
        #que pegar o numero positivo para as transformações

        numero_invertido = "".join('1' if bit == '0' else '0' for bit in valor_binario) #0 join vai concatenar os valores, e a
        #parte interna do join transforma e inverte os bits

        valor_complemento_dois = bin(int(numero_invertido, 2) + 1)[2:] #Como python não permite somar em binário, eu transformo
        #meu valor em decimal, somo 1 transformo em binário

        return str(valor_complemento_dois) #Retorno meu valor como string para facilitar a concatenação
    else:
        valor_binario = format(valor_decimal, '012b') # Convertendo para binário
        return str(valor_binario)

def Binario(valor_decimal):
    valor_binario = format(valor_decimal, '05b') # Convertendo para binário
    return str(valor_binario)

```

Figure 2. Funções Binário e Complemento de Dois

com números negativos e positivos, com o último é aplicado o método da função supracitada, mudando apenas a quantidade de bits para 12, já com os demais realiza a conversão para complemento de 2, seguindo as etapas abaixo:

1. Primeiramente, o valor positivo do número é convertido para binário, multiplicando-o sua representação na base 10 por "-1" e utilizando a função "format";
2. Em seguida, os bits da representação binária são invertidos e concatenados, por meio de um loop "for" e a função "join";
3. Por fim, é adicionado 1 ao valor invertido, o que requer a conversão do número para decimal, pois Python não permite a soma de números binários, e em seguida é realizada novamente a transformação, desta vez utilizando outro método da linguagem, "bin".

### 2.3. Formatos RISC-V

Os procedimentos para formatação da saída foram desenvolvidos com base na necessidade de "alinhar" os valores binários de acordo com cada tipo de formato estabelecido pelo montador RISC-V. Logo, em cada um dos formatos R, S e I, tem-se uma determinada organização para que as strings de 0's e 1's sejam concatenadas, a depender do opcode, funct3, funct 7, rd, rs1, rs2 e imediato, valores estes que variam dentre as instruções.

Formato S:

SH	imm[11:5]	rs2	rs1	funct3: 001	imm[4:0]	opcode: 0100011
BNE	imm[11:5]	rs2	rs1	funct3: 001	imm[4:0]	opcode: 1100011

Formato I:

LH	imm[11:0]	rs1	funct3: 001	rd	opcode: 0000011
ANDI	imm[11:0]	rs1	funct3: 111	rd	opcode: 0010011

Formato R:

ADD	funct7: 0000000	rs2	rs1	funct3: 000	rd	opcode: 0110011
OR	funct7: 0000000	rs2	rs1	funct3: 110	rd	opcode: 0110011
SLL	funct7: 0000000	rs2	rs1	funct3: 001	rd	opcode: 0110011

Figure 3. Tabela dos Formatos

Para exemplificar melhor a função relatada no parágrafo anterior, tem-se um trecho do código do Formato I, que chama o procedimento de complemento de dois para o seu imediato e o de tornar o número binário para os seus valores rs1 e rd. Nesse sentido, fora utilizado pela equipe os dicionários em Python, que atrelam às chaves (que seriam as instruções) os seus valores correspondentes, sendo os iguais tratados com uma mesma chave. Ex: **funct3** = ("add": "000", "iguais": "001"). Esse conceito trouxe ao

grupo uma vantagem no momento da programação: evitar a digitação repetitiva de 0's e 1's, visto que durante o processo podem ocorrer muitas falhas. Logo, as strings geradas pela chamadas das instruções via dicionários e das funções de transformação para valores binários em cada um dos formatos são concatenadas e, então, retornadas para que sejam “printadas” no terminal ou no arquivo de saída.

```
def FormatoI(instrucao,rd, rs1,imm): # Formato I: imm, rs1, funct3, rd, opcode

    imm = Complemento_Dois(int(imm))
    rs1 = Binario(int(rs1))
    rd = Binario(int(rd))

    if(instrucao == "lh"):
        funct3r = funct3["iguais"]
        opcode = opcode[instrucao]
    else:
        funct3r = funct3[instrucao]
        opcode = opcode["iguaisI"]

    resultadoI = imm + rs1 + funct3r+ rd + opcode
    return resultadoI
```

Figure 4. Função Formato I

### 3. Extras implementados

O grupo optou por expandir o conjunto de instruções do processador, incluindo as operações **addi**, **sub** e **xori**, além das instruções já atribuídas anteriormente. Tais operações seguem os formatos previamente estabelecidos (Tópico: 2.3. Formatos RISC-V), com uma adaptação na nomenclatura dos opcodes, agora designados como "iguaisI" e "IguaisR" para indicar quais são iguais entre os modelos. Além disso, foi necessário introduzir um novo campo "funct7" para a instrução "sub", assim como novos "funct3" para as demais operações. Salienta-se também que, para que o replace("x", " ") não conflitasse com o "x" da palavra "xori", assim que o algoritmo a identifica como "ori" (instrução esta que não foi concedida ao grupo, portanto não há problema) após a remoção deste, ela é concatenada com o "x" novamente para que a passagem de parâmetros funcione normalmente. Para concluir essas mudanças, o sistema de leitura de arquivos foi modificado, com o objetivo de incluir e interpretar as novas instruções.

É importante destacar que as operações adicionais foram escolhidas considerando a implementação de pseudo-instruções, sendo selecionadas as seguintes: **nop**, **mv**, **not** e **neg**. Elas foram implementadas conforme descrito em “The RISC-V Instruction Set Manual”(WATERMAN; ASANOVIĆ, 2017), o que é informado no comentário presente no código py [figura 6]. Na chamada das funções, a pseudo-instrução é identificada e convertida para a instrução RISC-V correspondente, juntamente com seus valores associados. Por exemplo, a pseudo-instrução "neg" é chamada da seguinte forma: "resultado = FormatoR("sub", palavras[1], "0", palavras[2])". Vale ressaltar que os valores decimais são passados como strings, pois são convertidos dentro das funções de formato.

## EXTRAS

### Formato I:

ADDI (EXTRA)	imm[11:0]	rs1	funct3: 000	rd	opcode: 0010011
XORI (EXTRA)	imm[11:0]	rs1	funct3: 100	rd	opcode: 0010011

### Formato R:

SUB (EXTRA)	funct7: 0100000	rs2	rs1	funct3: 000	rd	opcode: 0110011
----------------	--------------------	-----	-----	----------------	----	--------------------

Figure 5. Instruções Extras

```
- Pseudo Instruções:
Pseudo-Instrucao : Instrucao do riscv : Funcionalidade
[nop]             : [addi x0, x0, 0]   : Nao realiza operacao
[mv rd, rs]       : [addi rd, rs, 0]   : Copia oq esta em rs para rd
[not rd, rs]      : [xori rd, rs, -1]  : Inverte os bits do numero
[neg: rd, rs]     : [sub rd, x0, rs]   : Complemento de 2 do rs
```

Figure 6. Pseudo-Instruções

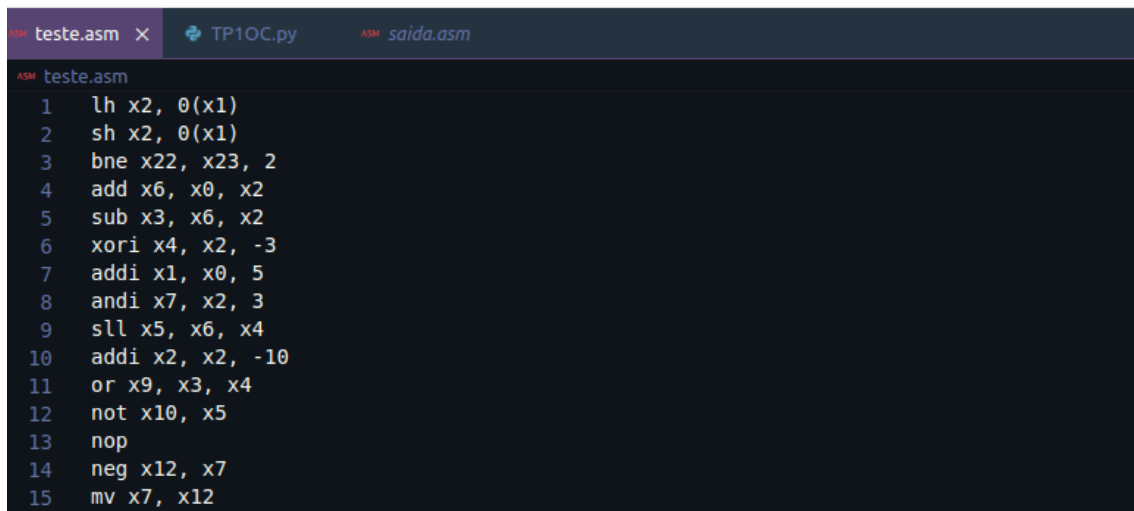
Quanto ao suporte do montador para outras bases numéricas no arquivo .asm, é proporcionado ao usuário a opção de decidir qual base deseja para a impressão dos resultados - binária ou hexadecimal. Essa escolha é feita através dos parâmetros passados na linha de comando, onde a pessoa informa a base desejada, caso não haja uma especificação, o padrão usado será o binário. Durante a impressão do resultado, verificamos a base selecionada e, se o número 16 for indicado, o valor será convertido para essa base, com um padrão de oito bits, utilizando a função supracitada, “format”.

## 4. Execução

A execução do sistema é realizada pelo próprio terminal. Nele, o usuário deve navegar até o diretório onde o arquivo .py está localizado, e executar o seguinte comando “**python3 TP1OC.py entrada.asm**”, se desejar exibir a saída no terminal, ou “**python3 TP1OC.py entrada.asm -o saida**”, para escrevê-la em um arquivo “.asm”. Ademais, se o usuário desejar exibir os resultados em hexadecimal deve colocar o número 16 no final dos comandos (“**python3 TP1OC.py entrada.asm 16**” ou “**python3 TP1OC.py entrada.asm -o saida 16**”). Vale ressaltar também que se o cliente optar por colocar o caminho de cada um dos arquivos passados de maneira individual, sem a necessidade de navegar até o diretório.

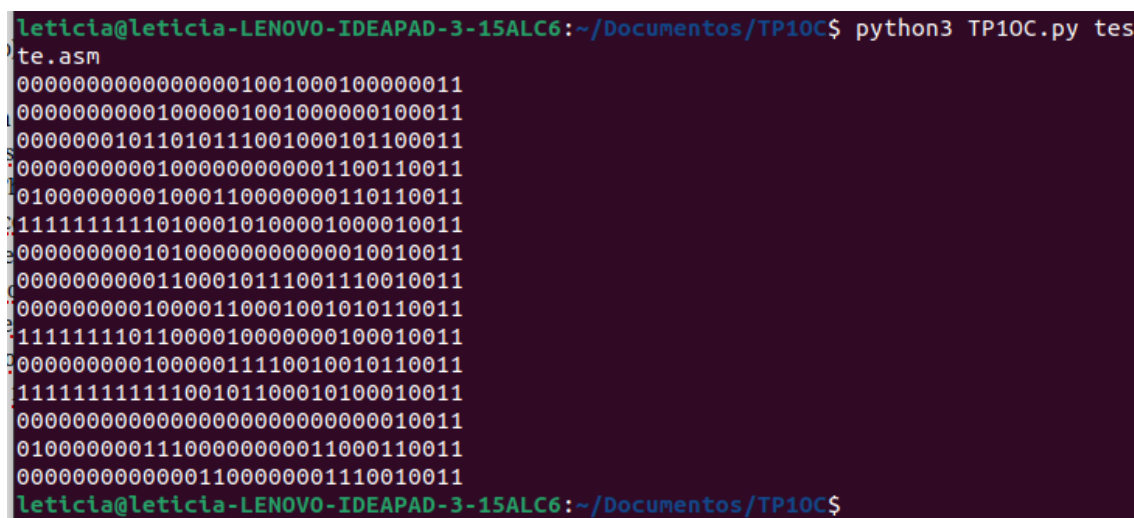
## 5. Resultados e Conclusão

O montador está funcionando exatamente como esperado, tanto para as instruções atribuídas ao grupo (17), quanto para os extras, gerando as saídas esperadas de maneira correta em ambas opções (terminal e arquivo) e bases (decimal e hexadecimal). Abaixo realizamos um teste com todas as instruções e pseudo-instruções implementadas, em ambas as bases, com resultado exibido no terminal:



```
teste.asm x TP10C.py saída.asm
ASM teste.asm
1  lh x2, 0(x1)
2  sh x2, 0(x1)
3  bne x22, x23, 2
4  add x6, x0, x2
5  sub x3, x6, x2
6  xori x4, x2, -3
7  addi x1, x0, 5
8  andi x7, x2, 3
9  sll x5, x6, x4
10 addi x2, x2, -10
11 or x9, x3, x4
12 not x10, x5
13 nop
14 neg x12, x7
15 mv x7, x12
```

Figure 7. Entrada teste



```
leticia@leticia-LENOVO-IDEAPAD-3-15ALC6:~/Documentos/TP10C$ python3 TP10C.py teste.asm
00000000000000001001000100000011
0000000000001000001001000000100011
000000001011010111001000101100011
00000000001000000000001100110011
010000000010001100000000110110011
11111111110100010100001000010011
00000000001010000000000010010011
00000000001100010111001110010011
000000000010000110001001010110011
111111110110000100000000100010011
000000000010000011110010010110011
11111111111100101100010100010011
000000000000000000000000000010011
010000000011100000000011000110011
0000000000000011000000001110010011
leticia@leticia-LENOVO-IDEAPAD-3-15ALC6:~/Documentos/TP10C$
```

Figure 8. Resultado Base Binária

```
leticia@leticia-LENOVO-IDEAPAD-3-15ALC6:~/Documentos/TP10C$ python3 TP10C.py tes
te.asm 16
00009103
00209023
016b9163
00200333
402301b3
ffd14213
00500093
00317393
004312b3
ff610113
0041e4b3
fff2c513
00000013
40700633
00060393
leticia@leticia-LENOVO-IDEAPAD-3-15ALC6:~/Documentos/TP10C$ s
```

Figure 9. Resultado Base Hexadecimal

Por fim, conclui-se que a implementação do montador RISC-V, tanto das instruções designadas quanto das partes extras, foi um sucesso e os objetivos estabelecidos na especificação do trabalho prático foram compreendidos e atendidos corretamente pelo grupo.

## 6. References

1. melissaaalanis/TP1-OC: Simulador de Montador arquitetura RISC-V em Python. Disponível em: <https://github.com/melissaaalanis/TP1-OC>. Acesso em: 19 abr. 2024.
2. RV32I, RV64I Instructions — riscv-isa-pages documentation. Disponível em: <https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html>. Acesso em: 19 abr. 2024.
3. sys — System-specific parameters and functions — documentação Python 3.8.3rc1. Disponível em: <https://docs.python.org/pt-br/3/library/sys.html>.
4. Entrada e Saída. Disponível em: <https://docs.python.org/pt-br/3/tutorial/inputoutput.html>. Acesso em: 19 abr. 2024.
5. AZEVEDO, R. Assembly do RISC-V - Instruções - MC404 - Organização Básica de Computadores e Linguagem de Montagem. Disponível em: <https://www.ic.unicamp.br/~rodolfo/mc404/slides/assembly04/>. Acesso em: 19 abr. 2024.
6. CATUNDA, H. Arquivos de Texto com Python - Como Ler, Editar e Criar? Disponível em: [https://www.hashtagtreinamentos.com/arquivos-de-texto-com-python?gad\\_source=1&gclid=CjwKCAjwrIxBhBbEiwACEqDJXa0fNmErmyaLzFJwBS2J-ovjQX9JjC-fK6BVC24qv37L\\_IjfBpM3BoCKb4QAvD\\_BwE](https://www.hashtagtreinamentos.com/arquivos-de-texto-com-python?gad_source=1&gclid=CjwKCAjwrIxBhBbEiwACEqDJXa0fNmErmyaLzFJwBS2J-ovjQX9JjC-fK6BVC24qv37L_IjfBpM3BoCKb4QAvD_BwE). Acesso em: 19 abr. 2024.
7. Estruturas de dados — documentação Python 3.10.7. Disponível em: <https://docs.python.org/pt-br/3/tutorial/datastructures.html>.
8. O que vimos na aula passada?. Disponível em: <https://www.ic.unicamp.br/>. Acesso em: 19 abr. 2024.

9. WATERMAN, A.; ASANOVIĆ, K. **The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2.** [s.l.: s.n.]. Disponível em: <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>.