UNIVERSIDADE FERADERAL DE VISOÇA - *CAMPUS* FLORESTAL
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
CCF-252

TRABALHO PRÁTICO - MONTADOR RISC-V

ERICH PINHEIRO AMARAL - 5915
PEDRO PAULO PAZ DO NASCIMENTO - 5937

FLORESTAL - MG
2025

ERICH PINHEIRO AMARAL
PEDRO PAULO PAZ DO NASCIMENTO

TRABALHO PRÁTICO - MONTADOR RISC-V

Relatório do trabalho prático da disciplina CCF-252 , sendo o objetivo desenvolver um montador RISC-V. Avaliação feita pelo Prof. Jose Augusto Miranda Nacif, UFV *Campus Florestal*.

FLORESTAL - MG
2025

RESUMO

O trabalho consiste na implementação de uma versão simplificada de um montador para a arquitetura RISC-V, seguindo especificações de uma tabela formada para cada aluno com um conjunto de 7 instruções que deve ser implementada. Por fim, para cada instrução, em RISC-V, deve ser devolvida, em linhas correspondentes, a saída em código binário, sendo escrita no terminal ou em um arquivo de texto.
Palavras-chave: montador, instruções, arquivo, terminal

ABSTRACT

The work consists of implementing a simplified version of an assembler for RISC-V architecture, following the specifications of a table formed for each student with a set of 7 instructions that must be implemented. Finally, for each instruction in RISC-V, the output in binary code must be returned on the corresponding lines, either written to the terminal or to a text file.
Keywords: assembler, instructions, terminal, file

---

1 INTRODUÇÃO

Para que uma máquina consiga realizar instruções e consiga ler códigos em qualquer que seja a linguagem de programação, é preciso de um compilador que traduz da linguagem de programação para a linguagem Assembly, e após a linguagem passa por um montador que a traduz para uma linguagem de máquina, para que os processadores entendam as instruções na forma de números binários, reduzindo cada vez mais o nível de abstração.

Este trabalho teve um foco no montador simples para instruções RISC-V, uma arquitetura de conjunto de instruções (ISA) moderna e muito utilizada em pesquisas e desenvolvimento de processadores.

Compreender como um código de alto nível é convertido para instruções que o hardware entende(baixo nível abstração) é de grande importância para entendermos sobre o funcionamento de uma máquina, e a implementação de um montador é uma boa forma de visualizar isso na prática. Além de nos permitir entender melhor os conceitos fundamentais de organização de computadores e arquitetura de processadores.

2 OBJETIVOS

O objetivo principal deste trabalho prático foi a implementação de um montador para a arquitetura RISC-V, responsável pela tradução de código Assembly para linguagem de máquina. Ao decorrer, vimos que o trabalho demandou a aplicação prática

de conceitos fundamentais aprendidos na matéria. E, ao implementarmos o conjunto de instruções (sendo todas as instruções disponíveis no trabalho), enfrentamos o desafio de compreender e codificar as particularidades de cada instrução e seus diferentes formatos (R, I, S, SB).

A construção do algoritmo do montador exigiu um estudo aprofundado e resultou na aquisição de conhecimentos específicos nas seguintes áreas:

- Aprendizado da linguagem (Python);
- Instruções em RISC-V;
- Tipos de instruções;
- Lógica para algoritmo;
- Leitura de arquivos (.asm).

3 RESULTADOS

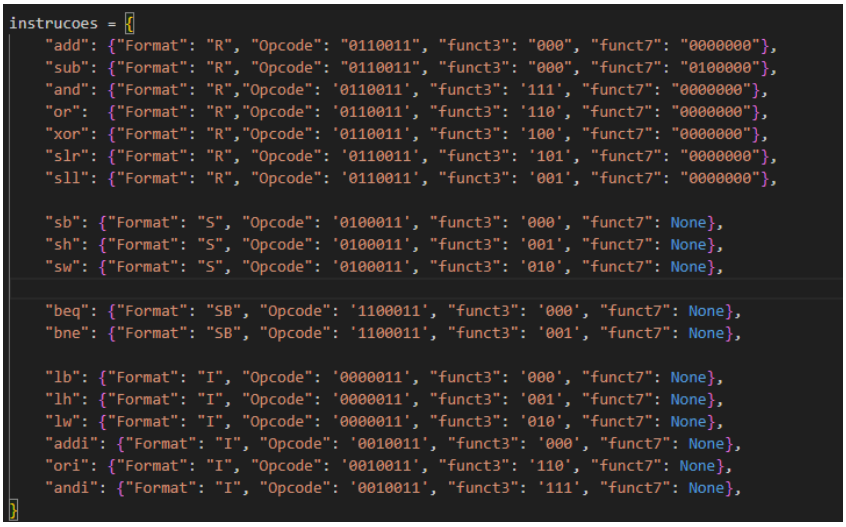
A implementação do montador RISC-V em Python apresentou desafios iniciais, principalmente relacionados ao entendimento da linguagem. Porém, o algoritmo do processo estava clara desde o princípio: receber a linha de instrução assembly, realizar a limpeza de caracteres inúteis para extrair os componentes relevantes (instrução, registradores e imediatos), converter esses componentes para suas cópias binárias e, por fim, montar a instrução final em código de máquina de 32 bits, respeitando o formato específico de cada tipo (R, I, S, SB).

Durante o desenvolvimento, notamos que instruções do tipo R foram implementadas com menos dificuldade, enquanto os tipos S e SB exigiram maior atenção devido a sua alta complexidade no posicionamento dos bits do valor imediato. O tratamento de imediatos, abrangendo diferentes bases (decimal, hexadecimal e binário) e a correta conversão de números negativos (complemento de dois), representou um ponto de dificuldade significativo. Porém, a linguagem Python oferece funções que facilitaram essas conversões e manipulações de bits.

Com pesquisa e o auxílio de ferramentas de IA, conseguimos solucionar problemas específicos que surgiram, como o posicionamento correto de valores, o tratamento de valores nulos (None) e a lógica do algoritmo para conversão de valores nulos. Por fim, o resultado final foi um montador funcional para todo o conjunto de instruções do trabalho prático, e o processo proporcionou um aprendizado prático valioso sobre a arquitetura RISC-V.

3.1 FIGURAS

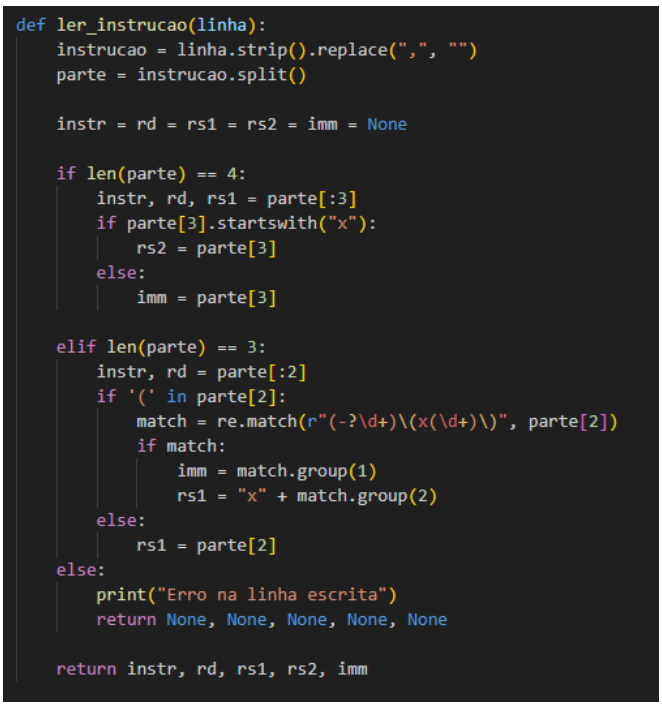
A figura 1 mostra uma dicionário de instruções que e usado no código que armazena dados necessários para montar as instruções RISC-V em binário, para cada instrução ela define o formato, opcode, Funct3 e Funct7.



```
instrucoes = {
    "add": {"Format": "R", "Opcode": "0110011", "funct3": "000", "funct7": "0000000"},
    "sub": {"Format": "R", "Opcode": "0110011", "funct3": "000", "funct7": "0100000"},
    "and": {"Format": "R","Opcode": '0110011', "funct3": '111', "funct7": "0000000"},
    "or":  {"Format": "R","Opcode": '0110011', "funct3": '110', "funct7": "0000000"},
    "xor": {"Format": "R","Opcode": '0110011', "funct3": '100', "funct7": "0000000"},
    "slr": {"Format": "R", "Opcode": '0110011', "funct3": '101', "funct7": "0000000"},
    "sll": {"Format": "R", "Opcode": '0110011', "funct3": '001', "funct7": "0000000"},

    "sb": {"Format": "S", "Opcode": '0100011', "funct3": '000', "funct7": None},
    "sh": {"Format": "S", "Opcode": '0100011', "funct3": '001', "funct7": None},
    "sw": {"Format": "S", "Opcode": '0100011', "funct3": '010', "funct7": None},

    "beq": {"Format": "SB", "Opcode": '1100011', "funct3": '000', "funct7": None},
    "bne": {"Format": "SB", "Opcode": '1100011', "funct3": '001', "funct7": None},

    "lb": {"Format": "I", "Opcode": '0000011', "funct3": '000', "funct7": None},
    "lh": {"Format": "I", "Opcode": '0000011', "funct3": '001', "funct7": None},
    "lw": {"Format": "I", "Opcode": '0000011', "funct3": '010', "funct7": None},
    "addi": {"Format": "I", "Opcode": '0010011', "funct3": '000', "funct7": None},
    "ori": {"Format": "I", "Opcode": '0010011', "funct3": '110', "funct7": None},
    "andi": {"Format": "I", "Opcode": '0010011', "funct3": '111', "funct7": None},
}
```

Figura 1 – Dicionário em Python contendo dados das instruções RISC-V

A figura 2 mostra o código responsável por interpretar a instrução RISC-V, ler e quebrar ela em partes que compõe a instrução, no caso separa em instr, rd, rs1, rs2, e se houver imm, deixando assim essas partes prontas para serem convertidas em binário.



```
def ler_instrucao(linha):
    instrucao = linha.strip().replace(",", "")
    parte = instrucao.split()

    instr = rd = rs1 = rs2 = imm = None

    if len(parte) == 4:
        instr, rd, rs1 = parte[:3]
        if parte[3].startswith("x"):
            rs2 = parte[3]
        else:
            imm = parte[3]

    elif len(parte) == 3:
        instr, rd = parte[:2]
        if '(' in parte[2]:
            match = re.match(r"(-?\d+)\(x(\d+)\)", parte[2])
            if match:
                imm = match.group(1)
                rs1 = "x" + match.group(2)
        else:
            rs1 = parte[2]
    else:
        print("Erro na linha escrita")
        return None, None, None, None, None

    return instr, rd, rs1, rs2, imm
```

Figura 2 – Leitura de Instruções

A figura 3 mostra uma função do código que recebe um registrador ou imediato e o converte para uma representação binária de 5 bits, ela aceita diferentes formatos, como registradores,valores binários, hexadecimais, e números inteiros.

```python
def conversao_binaria(valor):
    if valor is None:
        return None

    if valor.startswith("x"):
        numero = int(valor[1:])
        return format(numero, '05b')

    elif valor.startswith("0b"):
        v_binario = valor[2:]
        return v_binario.zfill(5) if len(v_binario) < 5 else v_binario

    elif valor.startswith("0x") or valor.startswith("-0x"):
        numero = int(valor, 16)
        return format(numero & 0x1F, '05b')

    elif (valor.isdigit()) or (valor.startswith('-') and valor[1:].isdigit()):
        numero = int(valor)
        return format(numero, '05b')

    return None
```

Figura 3 – Função para conversão binária

A figura 4 mostra uma função responsável por converter o valor imediato recebido para sua forma binária, com tamanho correto, que depende do tipo de instrução.

```python
def extrair_imediato(imm, Format):
    if imm is None:
        return None

    if imm.startswith("0x"):
        imm = int(imm, 16)
    elif imm.startswith("-0x"):
        imm = -int(imm[3:], 16)
    else:
        imm = int(imm)

    if Format == 'I' or Format == 'S':
        return format(imm & 0xFFF, '012b')
    elif Format == 'SB':
        return format(imm & 0x1FFF, '013b')
    return None
```

Figura 4 – Função para extrair imediado

A figura 5 mostra a função responsável for reconhecer se existe instrução no dicionário, e se existir, identifica o formato e converte cada parte separada para a forma binária de acordo com seu formato, montando a instrução em sua forma binária.

```python
def montar_instrucao(instr, rd, rs1, rs2, imm):
    if instr not in instrucoes:
        print(f"Instrução inválida: {instr}")
        return None

    info = instrucoes[instr]
    fmt = info["Format"]
    opcode = info["Opcode"]
    funct3 = info["funct3"]
    funct7 = info.get("funct7", "")

    rd_bin = conversao_binaria(rd)
    rs1_bin = conversao_binaria(rs1) if fmt != "U" else extrair_imediato(rs1, fmt)
    rs2_bin = conversao_binaria(rs2)
    imm_bin = extrair_imediato(imm, fmt)

    if fmt == "R":
        rs2_bin = imm_bin if rs2_bin is None else rs2_bin
        return f"{funct7}{rs2_bin}{rs1_bin}{funct3}{rd_bin}{opcode}"

    elif fmt == "I":
        return f"{imm_bin}{rs1_bin}{funct3}{rd_bin}{opcode}"

    elif fmt == "S":
        rs2_bin = rd_bin
        imm_inter = imm_bin[:7] if imm_bin else "0000000"
        imm_ext = imm_bin[7:] if imm_bin else "00000"
        return f"{imm_inter}{rs2_bin}{rs1_bin}{funct3}{imm_ext}{opcode}"

    elif fmt == "SB":
        rs2_bin = rd_bin
        if imm_bin:
            imm12 = imm_bin[0]
            imm10_5 = imm_bin[1:7]
            imm4_1 = imm_bin[7:11]
            imm11 = imm_bin[11]
            return f"{imm12}{imm10_5}{rs2_bin}{rs1_bin}{funct3}{imm4_1}{imm11}{opcode}"

    return None
```

Figura 5 – Recebendo informações sobre a instrução

A figura 6 mostra o ponto de entrada do programa que faz a leitura de um arquivo .asm de entrada o interpreta e retorna um arquivo de texto como saída com as instruções em sua forma binária, respectivamente para cada linha de código em RISC-V.

```python
def main():
    args = sys.argv
    if len(args) == 4 and args[2] == "-o":
        arq_entrada = args[1]
        arq_saida = args[3]
        saidas_binarias = []
        try:
            with open(arq_entrada, 'r') as arquivo:
                linhas = arquivo.readlines()
                for linha in linhas:
                    instr, rd, rs1, rs2, imediato = ler_instrucao(linha)
                    if instr is not None:
                        binario = montar_instrucao(instr, rd, rs1, rs2, imediato)
                        if binario:
                            saidas_binarias.append(binario)
            with open(arq_saida, "w") as saida:
                for linha in saidas_binarias:
                    saida.write(linha + "\n")
            print(f"Instruções convertidas salvas em '{arq_saida}'")
        except FileNotFoundError:
            print("Arquivo de entrada não encontrado.")
```

Figura 6 – Função para leitura de arquivos .asm

A figura 7 mostra uma continuação da função da figura 6, porém nela é tratada as instruções quando são passadas linha por linha pelo terminal, retornando as instruções em sua forma binária no próprio terminal.

```
elif len(args) < 2:
    linhas = []
    print("Digite instruções linha por linha. Para encerrar digite 'fim'.\n")
    while True:
        linha = input("> ")
        if linha.lower() == 'fim':
            break
        linhas.append(linha)

    for linha in linhas:
        instr, rd, rs1, rs2, imediato = ler_instrucao(linha)
        if instr is not None:
            binario = montar_instrucao(instr, rd, rs1, rs2, imediato)
            if binario:
                print(binario)
```

Figura 7 – Função para leitura de linha pelo terminal

4 CONCLUSÕES

Após a implementação de um montador RISC-V na linguagem Python, obtivemos um maior conhecimento prático da arquitetura RISC-V, como por exemplo as instruções e suas diferentes arquiteturas a depender de cada formato(R, I, S, SB), como cada campo da instrução é representado em binário, aprendemos também sobre como funciona a execução da máquina em baixo nível de abstração. Além de aprofundar nosso conhecimento na linguagem de programação Python.