

# Primeiro Trabalho Prático de Organização de Computadores 1 (CCF 252)

## Montador RISC-V (Versão Simplificada)

Lucas da C. Moreira (05377)<sup>1</sup>, Ana Luísa M. Rodrigues (05389)<sup>1</sup>

<sup>1</sup>Universidade Federal de Viçosa (Campus Florestal) (UFV-CAF)  
Rodovia LMG 818, km 06, Campus Universitário, Florestal - MG, 35690-000

lucas.costa.moreira@ufv.br, ana.luisa.rodrigues@ufv.br

**Abstract.** *This documentation describes the development and results of the assembly code translation program to machine code, in binary. The project requires the creation of a simplified RISC-V assembler, processing input files with the .asm extension and generating output in binary code. Each group is assigned to implement a specific subset of instructions.*

**Resumo.** *Esta documentação descreve o desenvolvimento e os resultados do programa de tradução de código assembly para código de máquina, em binário. O projeto requer a criação de um montador RISC-V simplificado, processando arquivos de entrada com extensão .asm e gerando saídas em código binário. Cada grupo é designado para implementar um subconjunto específico de instruções.*

### 1. Introdução

Neste primeiro projeto da disciplina CCF 252, ficamos encarregados da implementação de um montador simplificado para a arquitetura RISC-V, seguindo especificações detalhadas. O montador deve traduzir um subconjunto específico de instruções em Assembly para instruções binárias em linguagem de máquina, e as referentes ao nosso grupo(14) foram: "andi", "or", "sub", "lh", "sh", "beq" e "srl".

14	lh	sh	sub	or	andi	srl	beq
----	----	----	-----	----	------	-----	-----

Este projeto permite aos alunos explorar os conceitos de montagem de código, tradução de instruções e implementação de montadores, enquanto desenvolvem habilidades práticas em programação e compreendem melhor o funcionamento interno de uma arquitetura de um processador.

### 2. Desenvolvimento

Para implementar o montador RISC-V, o grupo escolheu a linguagem Python, pela facilidade em lidar com strings, junto do Manual RISC-V [1]. Dividimos o problema em várias partes. São elas:

- Obtenção das entradas a partir de um arquivo e definição da saída com a palavra chave "-o" para arquivo, ou na falta dela, para o terminal

```
#o arquivo de entrada precisa necessariamente ter o nome "teste.asm"
if sys.argv[1][-4:] != ".asm":
    print("Formato de arquivo errado!")
    exit()

#saida no terminal
if len(sys.argv) == 2:
    out = "terminal"
#saida em arquivo
elif len(sys.argv) == 4 and sys.argv[2] == "-o":
    out = "arquivo"
```

- Criação de um dicionário para extrair os dados das funções quando for necessário

```
# Dicionário de instruções RISC-V para suas representações em linguagem de máquina
instrucoes = {
    "lh": {"type": "I", "opcode": "0000011", "funct3": "001"},
    "sh": {"type": "S", "opcode": "0100011", "funct3": "001"},
    "sub": {"type": "R", "opcode": "0110011", "funct3": "000", "funct7": "0100000"},
    "or": {"type": "R", "opcode": "0110011", "funct3": "110", "funct7": "0000000"},
    "andi": {"type": "I", "opcode": "0010011", "funct3": "111"},
    "srl": {"type": "R", "opcode": "0110011", "funct3": "101", "funct7": "0000000"},
    "beq": {"type": "SB", "opcode": "1100011", "funct3": "000"}
}
```

- Tratamento da string de cada linha e definição do opcode e do funct3 para verificar qual instrução está sendo utilizada

```
def Traduz_Instrucao(linha):
    linha = linha.replace("(", " ").replace(")", " ").replace(" x", " ")
    linha = linha.replace(",", " ").split()

    #linha = [sub, 1, 2, 3]
    #linha = lh x1, 1(x2) = [lh, x1, 1, x2]
    #a linha vai ser por exemplo: a instrução "sub x6 x6 x5", se tornando "sub 6 6 5"

    #Define o opcode e o funct3 a partir do dicionário de instruções, usando a instrução da linha como chave
    opcode = instrucoes[linha[0]]["opcode"]
    funct3 = instrucoes[linha[0]]["funct3"]
```

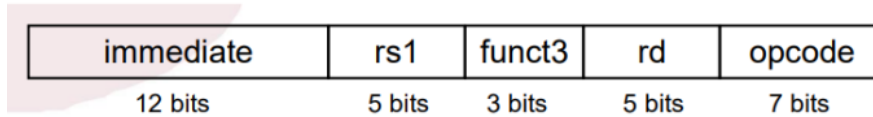
- Tradução e concatenação das partes de cada parte da instrução para binário, em seus respectivos tipos

```
if instrucoes[linha[0]]["type"] == "R":
    if funct3 == "000":
        funct7 = instrucoes[linha[0]]["funct7"]
        rd = linha[1]
        rs1 = linha[2]
        rs2 = linha[3]
        return f"{funct7}{int(rs2):05b}{int(rs1):05b}{funct3}{int(rd):05b}{opcode}"
```

Inicialmente, o algoritmo verifica os argumentos da chamada do programa para obter o arquivo de entrada e o método de saída. Com cada linha da entrada, um tratamento de strings acontece, separando a instrução e os valores dos registradores. Especificamente na instrução BEQ, foi determinado pelo grupo o uso do rótulo como um número, em vez de uma palavra, para uma mais fácil compreensão do código.

Dependendo da instrução, é possível determinar seu tipo (I, R, S, SB), opcode, funct3 e funct7, que serão utilizados para definir seu valor binário. Com esses valores,

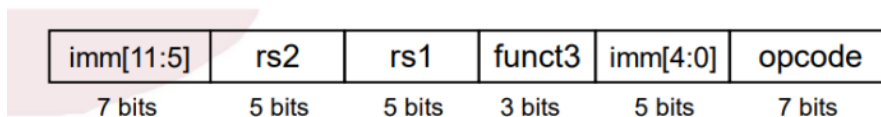
podemos converte-los em e concatena-los em um número binário, que será exibido ao usuário.



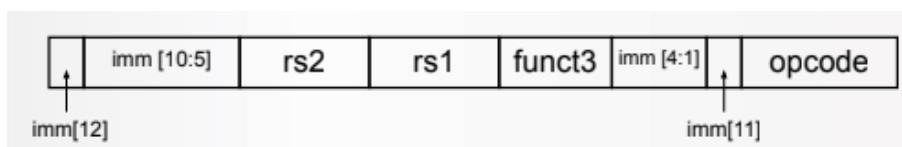
**Figure 1. Formato I**



**Figure 2. Formato R**



**Figure 3. Formato S**



**Figure 4. Formato SB**

No módulo `main()` do programa foi implementado toda a lógica por trás da execução do arquivo no modo terminal ou modo saída.asm, e para isso foi utilizado a verificação da quantidade de argumentos no comando de execução. No modo saída.asm é verificado o argumento "-o" e o nome do arquivo de saída, e no modo terminal é verificado apenas se são apenas dois argumentos, o "main" e a entrada.asm.

## 2.1. Pontos Extras

Os pontos extras realizados pelo grupo foram: implementação de pseudo instruções e de outras instruções fora do conjunto. As pseudos instruções escolhidas foram o "li" e o "mv", que são instruções para atribuição, sendo a primeira atribuindo a um registrador um valor de um imediato e a segunda atribuição do valor de um registrador para outro registrador.

```
# PONTO EXTRA: pseudo instrução "li"

elif instrucoes[linha[0]] == "li":
    #a instrução li é uma pseudo instrução que carrega um valor imediato para um registrador, de forma similar a addi
    #li x1, 1 = addi x1 x0 1
    #li x1, 1 = [li, x1, 1]
    #addi x1 x0 1 = [addi, x1, x0, 1]
    #li x1, 1 = [addi, x1, x0, 1]
    funct7 = instrucoes["add"]["funct7"]
    rd = linha[1]
    rs1 = "0"
    imm = linha[2]
    return f"{int(imm):012b}{int(rs1):05b}{funct3}{int(rd):05b}{opcode}"
```

Figure 5. Implementação do "li"

```
elif instrucoes[linha[0]] == "mv":
    #a instrução mv é uma pseudo instrução semelhante a li, tendo o seu processo invertido
    #ao invés de adicionar um valor imediato a um registrador, ela copia o valor de um registrador para outro
    #mv x1, x2 = addi x1 x2 0
    #mv x1, x2 = [mv, x1, x2]
    #addi x1 x2 0 = [addi, x1, x2, 0]
    #mv x1, x2 = [addi, x1, x2, 0]
    funct7 = instrucoes["add"]["funct7"]
    rd = linha[1]
    rs1 = linha[2]
    imm = "0"
    return f"{int(imm):012b}{int(rs1):05b}{funct3}{int(rd):05b}{opcode}"
```

Figure 6. Implementação do "mv"

As outras instruções feitas fora do conjunto do grupo foram: lw, sw, add, xor, and, sll. Para lógica do código, foi necessário analisar o tipo de cada instrução, escrever os respectivos opcodes, funct7 e funct3 e concatenar da mesma forma que foi realizado com as instruções designadas ao nosso grupo.

```
# Instruções adicionais do ponto extra: lw, sw, add, xor, and, sll
"sw": {"type": "S", "opcode": "0100011", "funct3": "010"},
"lw": {"type": "I", "opcode": "0000011", "funct3": "010"},
"add": {"type": "R", "opcode": "0110011", "funct3": "000", "funct7": "0000000"},
"xor": {"type": "R", "opcode": "0110011", "funct3": "100", "funct7": "0000000"},
"and": {"type": "R", "opcode": "0110011", "funct3": "111", "funct7": "0000000"},
"sll": {"type": "R", "opcode": "0110011", "funct3": "001", "funct7": "0000000"},
"li": {"type": "I", "opcode": "0010011", "funct3": "111"},
```

Figure 7. Instruções adicionais que estão fora do conjunto.

## 2.2. Comandos para execução

Existem dois modos de execução do programa, o modo saída por arquivo e por terminal. No modo de saída pelo terminal, basta digitar "python3 main.py arquivo\_de\_entrada.asm".

```
python3 main.py teste.asm
```

Figure 8. Comando teste para modo exibição no terminal.

Já o modo de saída pelo arquivo é preciso digitar "python3 main.py nome\_arquivo\_de\_entrada.asm -o nome\_arquivo\_de\_saida".

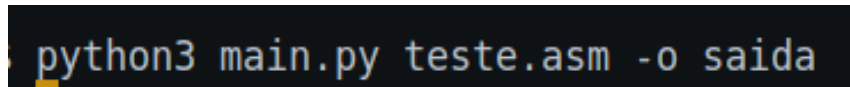
A screenshot of a terminal window with a dark background. The text 'python3 main.py teste.asm -o saida' is displayed in a light blue/cyan monospaced font. A yellow cursor is positioned at the end of the command.

Figure 9. Comando teste para modo exibição em um arquivo de saída.

### 2.3. Instruções para instalação

Para instalar o compilador ou interpretador Python em uma distribuição **Ubuntu/Debian**, siga as seguintes instruções:

Abra um terminal no seu sistema Linux. Você pode fazer isso pressionando Ctrl + Alt + T ou procurando por "Terminal" no menu de aplicativos. Verifique se o gerenciador de pacotes do seu sistema está atualizado executando o seguinte comando:

**sudo apt update**

Para instalar o interpretador Python padrão (Python 3), execute o seguinte comando: `sudo apt install python3` Este comando instalará o Python 3 e todas as dependências necessárias. Para verificar se o Python foi instalado corretamente, você pode executar o seguinte comando para exibir a versão do Python instalada:

**python3 --version**

Você deverá ver a versão do Python 3 instalada no seu sistema. Opcionalmente, se você deseja instalar o compilador Python (para compilar código Python para código de máquina), você pode instalar o pacote `python3-dev`:

**sudo apt install python3-dev**

Este pacote contém arquivos de cabeçalho e bibliotecas necessárias para compilar extensões Python. Com essas etapas, você terá instalado com sucesso o compilador ou interpretador no seu sistema Linux.

### 2.4. Acesso ao código no Github

Link para acessar o repositório: <https://github.com/ruktk/TP01OC>

## 3. Resultados

Nessa seção será apresentado três dos casos de testes utilizados para verificar os resultados.

Caso de teste 1:

```

TP1- tradutor > ASM teste.asm
1      andi x6, x6, 7
2      sub x6, x6, x5
3      or x0, x5, x6
4      srl x0, x1, x6
5      beq x0, x0, 8
6      8:
7      lh x1, 4(x2)
8      sh x1, 8(x2)
9

```

Figure 10. Instruções em assembly para teste 1

```

anamint@linuxMintAna:~/UFV/3º periodo/OC/TP1- tradutor$ make run_terminal arquivo_de_entrada=teste.asm
python3 main.py teste.asm
Código de máquina RISC-V:
00000000011100110111001100010011
010000000101001100000001100110011
00000000011000101110000000110011
00000000011000001101000000110011
000000000000000000000100001100011
00000000010000010001000010000011
0001000000100010001000000100011
anamint@linuxMintAna:~/UFV/3º periodo/OC/TP1- tradutor$

```

Figure 11. Resultado no terminal para teste 1

```

TP1- tradutor > ≡ saída.txt
1      00000000011100110111001100010011
2      010000000101001100000001100110011
3      00000000011000101110000000110011
4      00000000011000001101000000110011
5      000000000000000000000100001100011
6      00000000010000010001000010000011
7      00010000000100010001000000100011
8

```

Figure 12. Resultado no arquivo de saída para teste 1

Caso de teste 2:

```

TP1- tradutor > asm teste.asm
1      andi x6, x5, 15
2      sub x6, x6, x5
3      or x0, x5, x5
4      srl x0, x1, x5
5      beq x0, x0, 6
6      6:
7      lh x1, 1(x2)
8      sh x1, 1(x2)
9

```

Figure 13. Instruções em assembly para teste 2

```

anamint@linuxMintAna:~/UFV/3º periodo/OC/TP1- tradutor$ make run_terminal arquivo_de_entrada=teste.asm
python3 main.py teste.asm
Código de máquina RISC-V:
00000000111100101111001100010011
010000000101001100000001100110011
00000000010100101110000000110011
00000000010100001101000000110011
0000000000000000000000011001100011
00000000001000100010000100000011
10000000000100010001000000100011
anamint@linuxMintAna:~/UFV/3º periodo/OC/TP1- tradutor$

```

Figure 14. Resultado no terminal para teste 2

```

TP1- tradutor > ≡ saída.txt
1      00000000111100101111001100010011
2      010000000101001100000001100110011
3      00000000010100101110000000110011
4      00000000010100001101000000110011
5      0000000000000000000000011001100011
6      00000000000100010001000010000011
7      10000000000100010001000000100011
8

```

Figure 15. Resultado no arquivo de saída para teste 2

Caso de teste 3:





## References

- [1] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual Volume I: User-Level ISA*. Document Version 2.2. SiFive Inc. & CS Division, EECS Department, University of California, Berkeley. May 2017.