

Documentação Montador RISC-V (versão simplificada)

Trabalho Prático 1

Pamela Lúcia Lara Diniz - 05898, Júlio César de Souza Oliveira - 05903

¹Instituto de Exatas – Universidade Federal de Viçosa Campus Florestal (UFV-CAF)

1. Introdução

A arquitetura RISC-V é um conjunto de instruções (ISA) de código aberto que permite a criação de processadores para uma variedade de aplicações. Para o seu funcionamento, um código em Assembly deve ser convertido para linguagem de máquina, uma sequência em binário que o processador consegue interpretar e executar. Esse procedimento de conversão é geralmente realizado por um montador (ou assembler), que faz a tradução dos comandos de acordo com uma especificação previamente estabelecida.

Nesse contexto, foi solicitada a elaboração de um programa que simula o comportamento de um montador, sendo capaz de receber um determinado conjunto com diferentes tipos de instruções em Assembly, do Risc-V 32 bits, e realizar a montagem do binário correspondente, de acordo com as regras estabelecidas pela arquitetura.

Com isso, o objetivo principal desse projeto é a aplicação dos conceitos relacionados ao funcionamento interno de um montador da arquitetura RISC V por meio de um viés prático e uma implementação eficiente.

2. Organização

Na imagem a seguir (Figure 1), é possível visualizar a organização do projeto. Na pasta `codigo/` você encontra o arquivo principal do programa `"Risc-V.py"` que contém toda a implementação realizada, além disso, no mesmo diretório devem estar os arquivos de entrada e de saída no formato `.asm`.

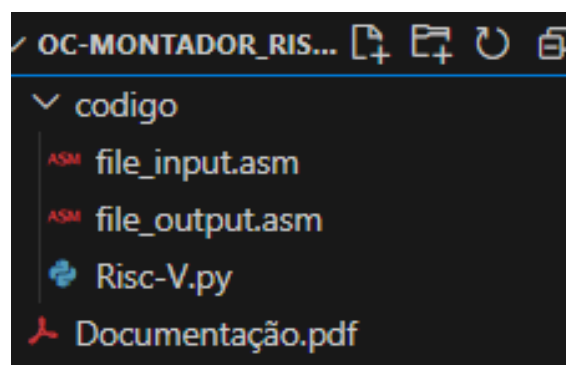


Figure 1. Organização do projeto

3. Desenvolvimento

3.1. main

A construção do main foi pensada para que o programa fosse capaz de receber como entrada o arquivo e escrever (no terminal ou arquivo) a saída; Além disso, o programa

deveria ser capaz de tratar as linhas lidas no arquivo, de forma a remover caracteres especiais, identificar hexadecimal e enviar para a sua respectiva função que trata o tipo da instrução lida.

Portanto, o programa inicia verificando se lhe é passado os argumentos corretos, com um arquivo.asm, e verifica se a saída deve ser exibida pelo terminal ou por arquivo, de acordo com os argumentos passados. Ele tenta abrir o arquivo de entrada e, caso a saída indicada seja por arquivo, abre também o arquivo de saída (Figure 2).

```
def main():
    #Verifica se está sendo passado um arquivo .asm
    if sys.argv[1][-4:] != ".asm":
        print("Entrada inválida")
        exit()
    forma = ""
    # verifica se a saída será por terminal ou arquivo

    if len(sys.argv) == 2:
        forma = "terminal"
    elif len(sys.argv) == 4 and sys.argv[2] == "-o":
        forma = "arquivo"

    #abre o arquivo de leitura
    try:
        arq = open(sys.argv[1], "r")
    except:
        print("Erro ao abri o arquivo de entrada!")
        exit()
    # Se a saída for por arquivo, abre também o arquivo de saída
    if forma == "arquivo":
        try:
            saida = open(sys.argv[3], "w")
        except:
            print("Erro ao abri o arquivo de saída!")

    # Armazena as linhas do arquivo
    instrucoes = arq.readlines()
```

Figure 2. Trecho do main que trata os argumentos passados

A partir disso, ele pode desviar para o trecho do código que imprime a saída no terminal ou para o trecho que armazena a saída em um arquivo baseado nos argumentos, porém, apesar do desvio, as funcionalidades são bem semelhantes.

Dessa forma, é criado um loop que percorre todas as linhas do arquivo de entrada e para cada linha ele trata a instrução removendo caracteres indesejados e dividindo cada campo da instrução em uma posição do vetor. Uma vez que a instrução já está tratada e seccionada em campos, basta checar qual o tipo dela (o campo que indica o nome da instrução, exemplo "add", "subi" que está relacionado a um tipo), assim chamando a função responsável pelo respectivo formato.

No entanto, na última etapa tem-se uma diferença entre os desvios mencionado anteriormente. Quando o programa é instruído a imprimir por terminal, a instrução é enviada para sua função que retorna seu valor em binário e logo é impressa. Já por arquivo, a string de bits retornada é escrita no arquivo de saída. Dessa forma, esse processo é repetido até que todas as linhas do arquivo sejam lidas.

3.2. Funções

Desde o primeiro momento, o projeto já tinha o objetivo de cobrir o máximo de instruções possíveis. Assim, a primeira etapa foi construir funções com os tipos de instruções es-

pecíficas, onde elas recebiam como parâmetro os campos da instrução, e tratava esses dados de acordo com seu formato (R,I...) (Figure 3). Porém, parte desse processo envolveu converter decimal para binário ou até mesmo converter imediatos da base hexadecimal para decimal e depois para binário. Assim duas funções auxiliares foram criadas para tratar essas conversões (Figure 4) e uma outra que removia caracteres indesejados das instruções. (Figure 5)

```
#Esta função converte um número decimal, positivo ou negativo, para binário com uma quantidade determinada de bits
> def conversao_binario(numero,bits):
    return str(binario).rfill(bits)

#Esta função converte um número hexadecimal para decimal
> def conversao_hexa(numero):
    return str(soma)

#Função que trata instruções do tipo R
> def tipo_r(instru, rd, rs1, rs2):
    return saida

#Função que trata instruções do tipo S
> def tipo_s(instru, rs2, immc, rs1):
    return saida

#Função que trata instruções do tipo SB
> def tipo_sb(instru, rs1, rs2, immc):
    return saida

#Função que trata instruções do tipo U
> def tipo_u(instrucao,rd,imediato):
    return instrucao_perada

#Função que trata instruções do tipo UJ
> def tipo_UJ(instrucao,rd,imediato):
    return instrucao_perada

#Função que trata instruções do tipo I
> def tipo_I(instrucao,rd,rs1,imediato):
    return instrucao_perada

#Função que trata pseudo instruções
> def tipo_pseudo(instrucao,rd,rs_or_imm):
    print("Pseudo instrução inválida")
```

Figure 3. Funções para tratar a instrução de acordo com seu tipo

```
#Esta função converte um número decimal, positivo ou negativo, para binário com uma quantidade determinada de bits
> def conversao_binario(numero,bits):
    return str(binario).rfill(bits)

#Esta função converte um número hexadecimal para decimal
> def conversao_hexa(numero):
    return str(soma)
```

Figure 4. Funções que convertem as instruções entre os sistemas de numeração

```
#Função que trata a instrução removendo caracteres indesejados
> def chr_remove(old, to_remove):
    new_string = old
    for x in to_remove:
        new_string = new_string.replace(x, ' ')
    return new_string
```

Figure 5. Função de remoção de caracteres indesejados

4. Compilação e execução

Para executar o programa em uma máquina Linux ou Windows, primeiramente é necessário ter o interpretador python3 instalado no sistema. Caso não o possua, a instalação pode ser feita através do site oficial python.org. Após verificada a instalação, é necessário abrir o terminal, ou uma IDE compatível, no diretório onde se encontra o script 'Risc-V.py', o comando 'cd' pode ser utilizado para navegar até o local do arquivo. Além disso, é muito importante que no mesmo diretório esteja o arquivo de entrada no formato '.asm'.

Quanto à execução do programa, existem duas possibilidades. Caso queira que a impressão dos resultados seja feita em um arquivo, é necessário que no mesmo diretório

já exista um arquivo para armazenar esses dados, também no formato '.asm', e seja executado o seguinte comando:

```
python3 Risc-V.py nome-entrada.asm -o nome-saida.asm
```

E se preferir que a impressão dos resultados seja mostrada apenas no terminal, basta executar o comando:

```
python3 Risc-V.py nome-entrada.asm
```

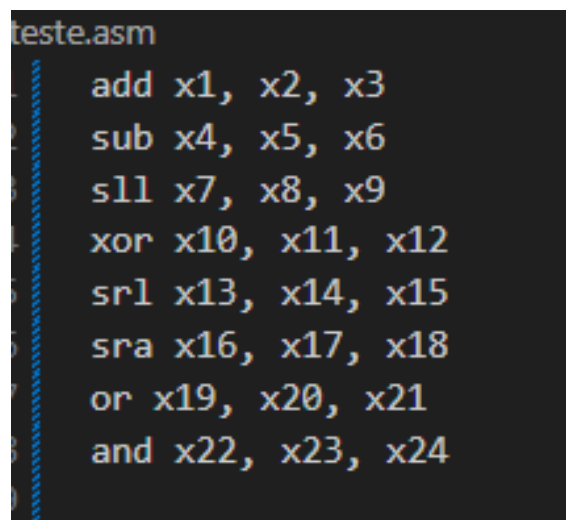
Executados todos os passos, caso a versão do python não seja encontrada ao tentar executar, altere o segmento "python3" do comando para "python".

5. Resultados

Destarte, uma vez que o programa foi compilado com sucesso, é possível testar o seu funcionamento. Sendo assim, nas próximas subseções estarão destacados alguns casos de testes para cada tipo de instrução, além disso será feito o detalhamento da montagem com ênfase nas instruções que compõe o conjunto principal a ser implementado pelo grupo.

5.1. Instruções Tipo R

No que se refere ao processamento de instruções do Tipo R, espera-se receber uma linha no formato "instrução, rd, rs1, rs2", e converter os valores decimais para seu correspondente em binário. Campos como "funct7", "funct3" e "opcode" são previamente estabelecidos. Alguns testes e sua respectiva conversão em linguagem de máquina podem ser observados através das imagens (Figure 6) e (Figure 7).



```
teste.asm
1  add x1, x2, x3
2  sub x4, x5, x6
3  sll x7, x8, x9
4  xor x10, x11, x12
5  srl x13, x14, x15
6  sra x16, x17, x18
7  or x19, x20, x21
8  and x22, x23, x24
```

Figure 6. Conjunto de instruções de tipo R em Assembly

```

saida.asm
1 00000000001100010000000010110011
2 01000000011000101000001000110011
3 00000000100101000001001110110011
4 00000000110001011110010100110011
5 000000001111011101010110110011
6 00000001001010001101100000110011
7 00000001010110100110100110110011
8 00000001100010111111101100110011
9

```

Figure 7. Linguagem de máquina das instruções de Tipo R

Para melhor entendimento, detalharemos a montagem da instrução 'sub x4, x5, x6':

funct7	rs2	rs1	funct3	rd	opcode
0100000	00110	00101	000	00100	0110011

Agora, quanto à instrução 'and x22, x23, x24':

funct7	rs2	rs1	funct3	rd	opcode
0000000	11000	10111	111	10110	0110011

E por fim, a instrução 'srl x13, x14, x15':

funct7	rs2	rs1	funct3	rd	opcode
0000000	01111	01110	101	01101	0110011

5.2. Instruções Tipo I

Quanto às instruções do Tipo I, espera-se receber uma entrada no formato "instrução rd,rs1,imediato", que é convertida para o sistema binário para a montagem em linguagem de máquina, considere que o imediato pode ser recebido na base hexadecimal. Campos como "funct3" e "opcode" são previamente estabelecidos. Alguns testes realizados podem ser observados através das imagens (Figure 8) e (Figure 9).

```

ASM teste.asm
1  lb x1, 0(x2)
2  lh x5, 8(x6)
3  lw x9, 16(x10)
4  ld x13, 24(x14)
5  lbu x17, 32(x18)
6  lhu x21, 36(x22)
7  lwu x25, 40(x26)
8  addi x1, x2, 10
9  slli x5, x6, 3
10 xori x9, x10, 0xF
11 srli x13, x14, 4
12 srai x17, x18, 5
13 ori x21, x22, 0x3C
14 andi x25, x26, 0x0F
15 jalr x1, 0(x2)

```

Figure 8. Conjunto de instruções de tipo I em Assembly

```

ASM saida.asm
1  00000000000000010000000010000011
2  00000000100000110001001010000011
3  00000000100000101001001001000011
4  00000000110000111001101101000011
5  00000010000010010100100010000011
6  00000010010010110101101010000011
7  00000010100011010110110010000011
8  00000000101000010000000010010011
9  00000000001100110001001010010011
10 00000000111101010100010010010011
11 00000000010001110101011010010011
12 01000000010110010101100010010011
13 00000011110010110110101010010011
14 000000001111101011110010010011
15 0000000000000010000000011100111

```

Figure 9. Linguagem de máquina das instruções de Tipo I

Segue o detalhamento da montagem da instrução 'lb x1, 0(x2)':

imm[11:0]	rs1	funct3	rd	opcode
000000000000	00010	000	00001	0000011

E por fim, a instrução 'ori x21, x22, 0x3C', que também utiliza a funcionalidade de reconhecimento do imediato na base hexadecimal.

imm[11:0]	rs1	funct3	rd	opcode
000000111100	10110	110	10101	0010011

5.3. Instruções Tipo S

Quanto às instruções do Tipo S, o formato recebido segue o padrão "instrução rs2, offset(rs1)". Campos como "funct3" e "opcode" são previamente estabelecidos. Seguem nas (Figure 10) e (Figure 11) os testes realizados e resultados obtidos.

```

ASM teste.asm
1  sb x5, 12(x10)
2  sh x3, -8(x6)
3  sw x1, 0(x2)
4  sd x7, 100(x8)
5

```

Figure 10. Conjunto de instruções de tipo S em Assembly

```

ASM saida.asm
1  00000000010101010000011000100011
2  1111110001100110001110000100011
3  0000000000100010010000000100011
4  0000011001101000111001000100011
5

```

Figure 11. Linguagem de máquina das instruções de Tipo S

Detalhando a montagem da instrução 'sb x5,12(x10):

imm[11:5]	rs2	rs1	funct3	im[4:0]	opcode
0000000	00101	01010	000	01100	0100011

5.4. Instruções Tipo SB

Em seguida, quanto às instruções do TipoSB, espera-se receber uma linha no formato "instrução rs1,rs1,imediato" que também suporta que o imediato esteja na base hexadecimal. Campos como "funct3" e "opcode" são previamente estabelecidos. Alguns testes e seus resultados podem ser observados nas (Figure 12) e (Figure 13).

```

ASM teste.asm
1  beq x1, x2, 16
2  bne x3, x4, -4
3  blt x5, x6, 32
4  bge x7, x8, -20
5  bltu x9, x10, 12
6  bgeu x11, x12, -8
7

```

Figure 12. Conjunto de instruções de tipo SB em Assembly

```

ASM saída.asm
1 00000000001000001000100001100111
2 01111110010000011001111011100111
3 00000010011000101100000001100111
4 01111110100000111101011011100111
5 00000000101001001110011001100111
6 011111101100010111111110011100111
7

```

Figure 13. Linguagem de máquina das instruções de Tipo SB

Agora, observe a estrutura de montagem da instrução 'beq x1,x2,16':

imm[12,10:5]	rs2	rs1	funct3	im[4:1,11]	opcode
0 000000	00010	00001	000	1000 0	1100111

5.5. Instruções Tipo U

Quanto à instrução do Tipo U, espera-se que a linha de entrada siga o padrão "lui rd,imediato". Nesse caso apenas o opcode é previamente estabelecido, além disso o imediato pode estar na base decimal ou hexadecimal. Seguem nas (Figure 14) e (Figure 15) a realização de testes.

```

ASM teste.asm
1 lui x5, 0x12345
2 lui x0, 0xFFFF
3 lui x31, 0x7FFFF
4 lui x10, 0x1
5 lui x16, 0x80000
6

```

Figure 14. Conjunto de instruções de tipo U em Assembly

```

ASM saída.asm
1 00010010001101000101001010110111
2 111111111111111111111000000110111
3 0111111111111111111111110110111
4 00000000000000000001010100110111
5 1000000000000000000100000110111
6

```

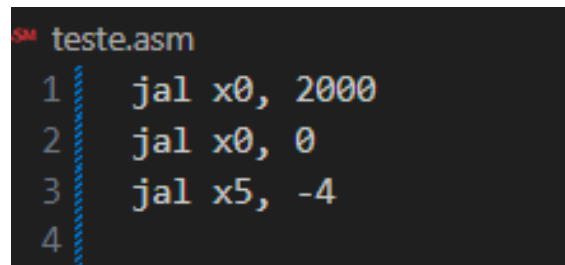
Figure 15. Linguagem de máquina das instruções de Tipo U

A seguir, se encontra a montagem da instrução 'lui x5, 0x12345', que se trata de uma implementação extra.

imm[31:12]	rd	opcode
00010010001101000101	00101	0110111

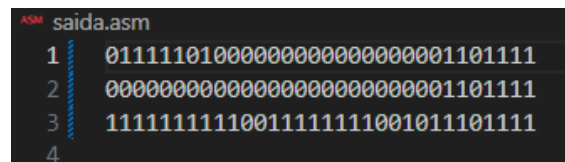
5.6. Instruções Tipo UJ

Para instruções que seguem o formato UJ, o padrão de entrada esperado é "jal rd,imediato". Esse é outro caso em que apenas o opcode é previamente estabelecido. Seguem nas (Figure 16) e (Figure 17) alguns testes realizados e os resultados obtidos.



```
ASM teste.asm
1 jal x0, 2000
2 jal x0, 0
3 jal x5, -4
4
```

Figure 16. Conjunto de instruções de tipo UJ em Assembly



```
ASM saida.asm
1 011111010000000000000000000000001101111
2 000000000000000000000000000000001101111
3 11111111110011111111001011101111
4
```

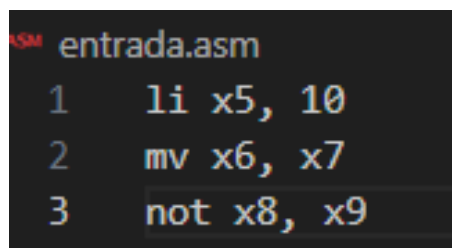
Figure 17. Linguagem de máquina das instruções de Tipo UJ

Segue o detalhamento da montagem da instrução 'jal x0,2000', que se trata de uma implementação extra.

imm[20,10:1,11,19:12]	rd	opcode
011111010000000000000000	00000	1101111

5.7. Pseudo-instruções

No que se refere às três pseudo-instruções implementadas, os resultados obtidos na conversão em linguagem de máquina podem ser observados através das imagens (Figure 18) e (Figure 19).



```
ASM entrada.asm
1 li x5, 10
2 mv x6, x7
3 not x8, x9
```

Figure 18. Conjunto de pseudo-instruções em Assembly

```

saida.asm
1 00000000101000000000001010010011
2 00000000000000111000001100010011
3 11111111111101001100010000010011
4

```

Figure 19. Linguagem de máquina das pseudo-instruções

Por fim, observe a montagem da pseudo-instrução 'li x5,10' que equivale a 'addi,rd,x0,im' e é uma implementação extra.

imm[11:0]	rs1	funct3	rd	opcode
000000001010	00000	000	00101	0010011

Assim, verifica-se que o sistema faz o processamento de maneira correta das instruções implementadas.

6. Conclusão

As etapas de desenvolvimento desse projeto tiveram como foco a implementação de um montador que realiza a conversão de um conjunto de instruções Assembly, da arquitetura RISC-V 32 bits para a linguagem de máquina correspondente. Para isso, foram utilizados artifícios da linguagem Python e implementadas funções específicas para cada tipo de instrução, além de funções auxiliares necessárias ao funcionamento da arquitetura.

Diante disso, os resultados obtidos foram satisfatórios e coerentes com a especificação fornecida. A combinação de elementos implementados proporcionou a criação de um sistema que é capaz de identificar formatos como R,I,S,SB,U e UJ, e realizar a montagem de cada segmento de maneira correta como observado na seção anterior. Além disso, o programa reconhece algumas pseudo-instruções e também oferece suporte para base decimal e hexadecimal.

Dessa forma, o desenvolvimento do projeto não apenas atendeu aos requisitos estabelecidos, mas também possibilitou a aplicação prática dos conceitos aprendidos na disciplina de Organização de Computadores I, caracterizando-se como uma experiência enriquecedora por meio da interação com problemas reais.

7. Referências

Materias disponibilizados pelo Professor Nacif e notas de aula.

Livro texto da disciplina [Patterson and Hennessy 2011]

References

Patterson, D. and Hennessy, J. (2011). *Computer Organization and Design*. Editora Morgan Kaufmann, 4th edition.