Escola Politécnica da Universidade de São Paulo
PCS 3216 – Sistemas de Programação

Simulador de Máquina de Von Neumann, Loader e Montador

Bernardo Marcelino do Nascimento – 9836197
Professor João José Neto

São Paulo
2020

Sumário

1. Introdução

O objetivo deste projeto é construir uma máquina de Von Neumann e seus mecanismos relacionados, como montador e loader, para aprofundar o entendimento de sistemas de programação e do funcionamento de computadores no geral.

O projeto se baseia na estrutura definida na Figura 1, que mostra os 3 componentes principais: o montador, o loader e a máquina de Von Neumann (MVN). O montador é o responsável por pegar um código escrito em linguagem Assembly e transformá-lo em um código binário compatível com a máquina. O loader então é o responsável por carregar esse código binário na memória da máquina, e iniciar a execução do programa, feita pela MVN.

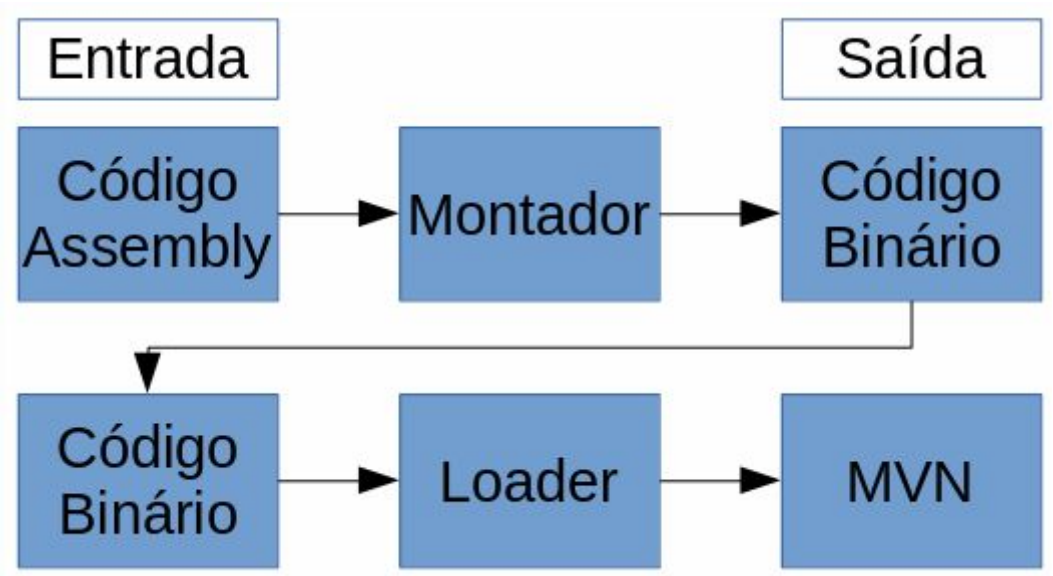


Figura 1 – Visão geral do projeto

A implementação da MVN foi feita na linguagem Python, versão 3.7.3, e toda a interação com o programa é feita através do terminal. O código está disponível em https://sites.google.com/usp.br/2020-pcs3216-9836197/ e em https://github.com/bmnascimento/von-neumann-simulator.

A MVN foi implementada no arquivo vm.py e para utilizar a máquina, basta chamar o programa através do terminal com o comando "python3 vm.py <arquivo_hexa>" onde arquivo_hexa é o nome do arquivo em hexadecimal.

O loader é carregado automaticamente na memória da MVN dentro do arquivo vm.py, mas seu código está descrito no arquivo loader.asm para melhor entendimento.

O montador foi implementado no arquivo montador.py e sua utilização é pelo comando "python3 montador.py <arquivo_input> <arquivo_output>" onde

arquivo_input é um arquivo em Assembly e arquivo_output é o nome do arquivo em hexadecimal gerado pelo montador.

2. Máquina de Von Neumann

2.1. Definição

A máquina de Von Neumman é um modelo de máquina capaz de executar instruções e de realizar computações. A máquina consiste de uma memória, onde são armazenados os programas a serem executados assim como os dados que os programas utilizam. Além disso, também há uma unidade que decodifica as instruções e as executa e por fim há os registradores, que auxiliam a execução das instruções.

O funcionamento da máquina consiste em ler um endereço de memória, executar a instrução lida, ler o próximo endereço e executar a instrução e assim por diante, até atingir uma instrução de HALT.

2.2. Motor de Eventos

No projeto, será feita uma simulação de uma MVN e para isso será o utilizado o conceito de motor de eventos. O motor de eventos está esquematizado na Figura 2 e consiste em um loop que extrai uma instrução da memória, decodifica essa instrução e a executa de acordo com o procedimento específico, altera o ponteiro para a próxima instrução e repete o loop.

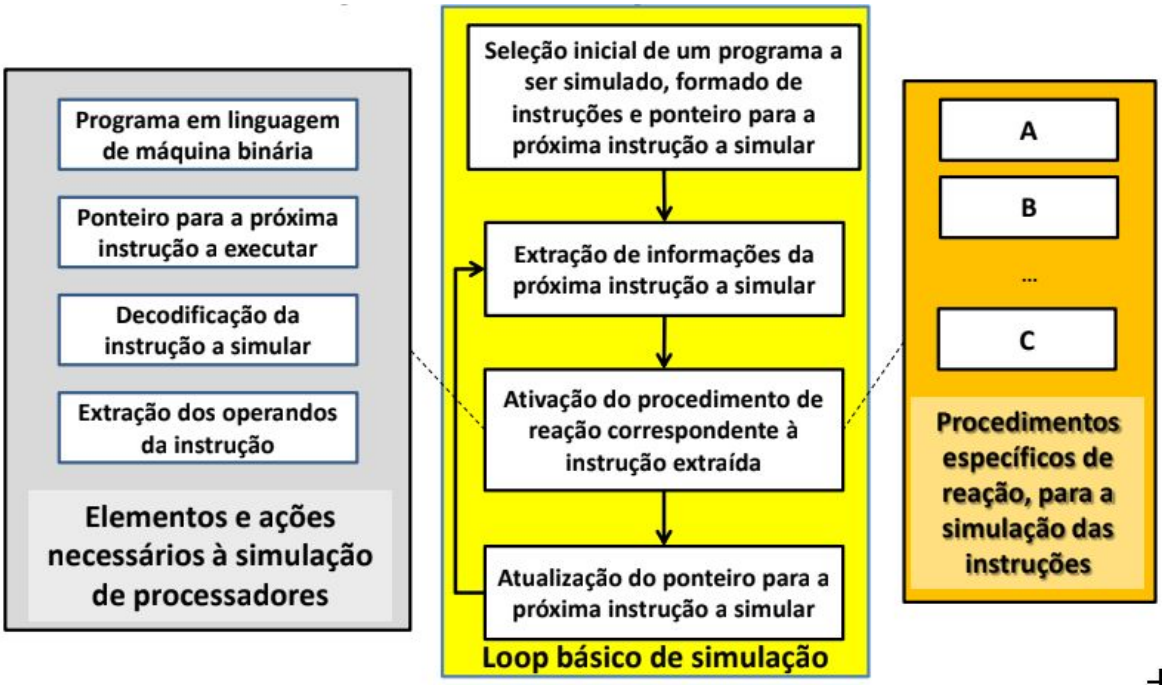
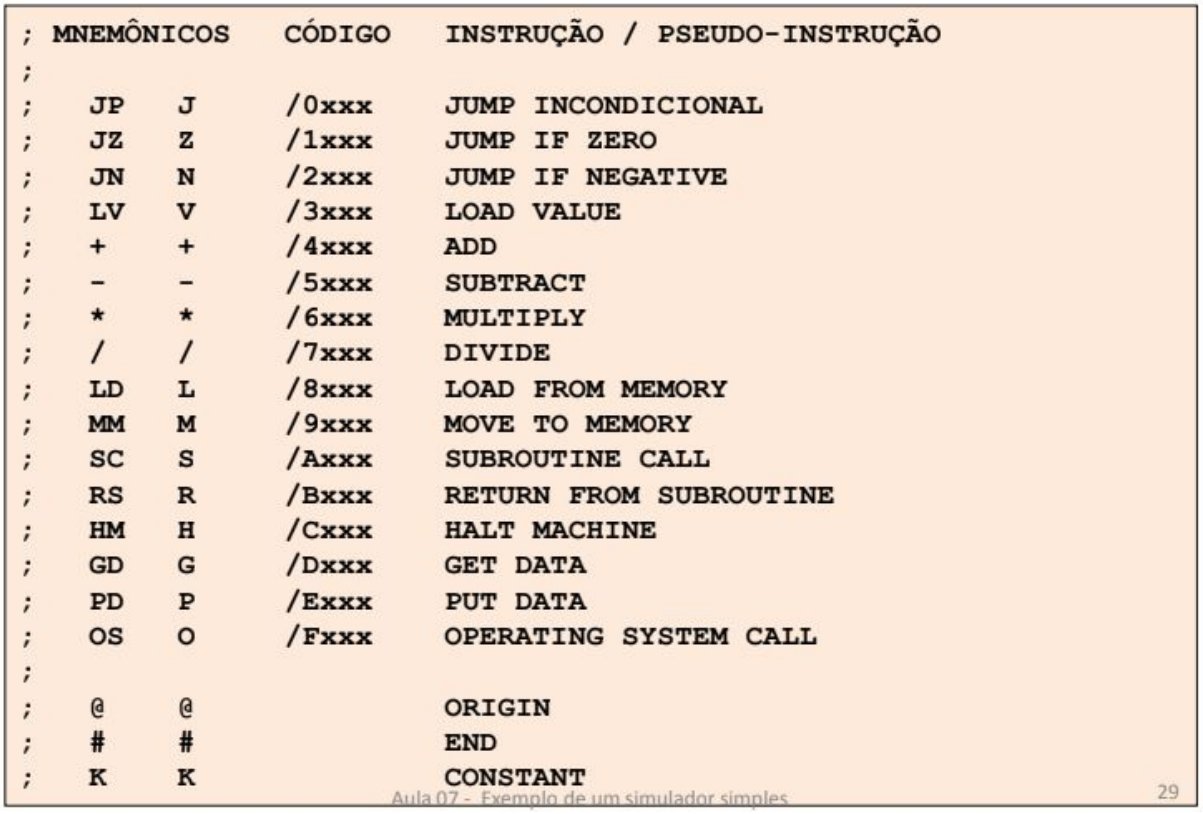


Figura 2 – Loop básico de simulação

2.3. Especificação

A MVN do projeto consiste em uma memória principal de 8 bits com 12 bits de endereçamento (4k x 8 bits), um contador de instruções de 12 bits e um acumulador de 8 bits. Além das memórias são especificadas as instruções que a máquina deve ser capaz de executar, apresentadas na Figura 3. O formato que a instrução deve assumir na memória é 4 bits para o código de operação e 12 bits para o operando, totalizando 16 bits. Portanto, cada instrução ocupa duas posições de memória.



```
; MNEMÔNICOS    CÓDIGO    INSTRUÇÃO / PSEUDO-INSTRUÇÃO
;
;    JP    J     /0xxx     JUMP INCONDICIONAL
;    JZ    Z     /1xxx     JUMP IF ZERO
;    JN    N     /2xxx     JUMP IF NEGATIVE
;    LV    V     /3xxx     LOAD VALUE
;    +     +     /4xxx     ADD
;    -     -     /5xxx     SUBTRACT
;    *     *     /6xxx     MULTIPLY
;    /     /     /7xxx     DIVIDE
;    LD    L     /8xxx     LOAD FROM MEMORY
;    MM    M     /9xxx     MOVE TO MEMORY
;    SC    S     /Axxx     SUBROUTINE CALL
;    RS    R     /Bxxx     RETURN FROM SUBROUTINE
;    HM    H     /Cxxx     HALT MACHINE
;    GD    G     /Dxxx     GET DATA
;    PD    P     /Exxx     PUT DATA
;    OS    O     /Fxxx     OPERATING SYSTEM CALL
;
;    @     @               ORIGIN
;    #     #               END
;    K     K               CONSTANT
```

Aula 07 - Exemplo de um simulador simples 29

Figura 3 – Instruções da MVN

O funcionamento da máquina e a implementação de cada instrução está descrita na Figura 4. Primeiramente o programa é carregado na memória simulada, então o contador de instruções é ajustado para apontar para a instrução inicial, em seguida é feita a extração do código de operação e do operando da instrução, e por fim é feito o tratamento adequado àquela instrução e o loop recomeça.

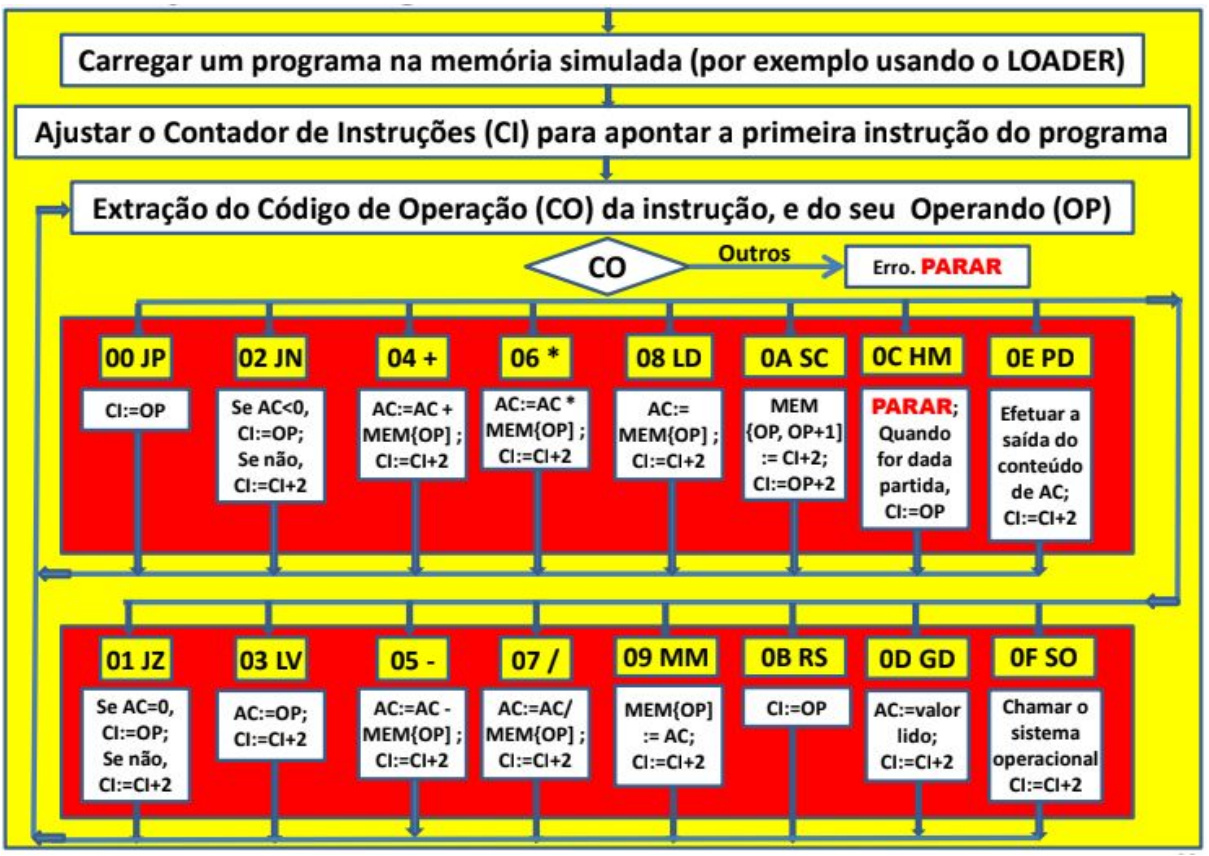


Figura 4 – Descrição das Instruções

2.4. Implementação

Na implementação da máquina, primeiramente é chamada a função boot, que vai inicializar o contador de instruções (CI), o acumulador (AC) e a memória (MEM). Logo em seguida é descrito o loop principal do motor de eventos, que executa infinitamente, primeiro chamando a função executeInstruction que vai decodificar e executar a instrução e se houver um erro ou um comando de halt, o loop é finalizado.

```
CI, AC, MEM = boot(code, debug)
while True:
    CI, AC, status = executeInstruction(CI, AC, MEM, debug)
if status == 'halt':
    print('\nMáquina parada')
break
elif status == 'error':
    print('\nErro: Instrução desconhecida')
break
```

Na função boot AC e CI são inicializados em 0, memória é inicializada com o código do loader e o arquivo de entrada é lido e armazenado em uma variável interna da máquina virtual para ser interpretado.

```
MEM = [0x0f, 0x00]
MEM.extend([0] * 0xefe)
MEM.extend([0xd0, 0x00, 0x9f, 0x44, 0x4f, 0x4b, 0x9f, 0x46, 0xd0, 0x00, 0x9f, 0x45,
0x9f, 0x47, 0xd0, 0x00, 0x9f, 0x4a, 0xd0, 0x00, 0x0f, 0x46, 0x8f, 0x47, 0x4f, 0x4c,
0x1f, 0x3a, 0x9f, 0x47, 0x8f, 0x4a, 0x5f, 0x4c, 0x9f, 0x4a, 0x1f, 0x28, 0x0f, 0x12,
0xd0, 0x00, 0x4f, 0x4b, 0x1f, 0x42, 0x9f, 0x46, 0xd0, 0x00, 0x9f, 0x47, 0xd0, 0x00,
0x9f, 0x4a, 0x0f, 0x12, 0x8f, 0x46, 0x4f, 0x4c, 0x9f, 0x46, 0x0f, 0x1c, 0x30, 0x00,
0x00, 0x00, 0x00, 0x00, 0x0f, 0x16, 0x00, 0x90, 0x01])
MEM.extend([0] * 0xb4)

AC = 0
CI = 0

FILE = []
for line in code.splitlines():
    for byte in line.split():
        FILE.append(int(byte, 16))

return (CI, AC, MEM, FILE)
```

É na função executeInstruction que acontece o trabalho de acessar, decodificar e executar a instrução. Primeiramente ocorre a decodificação da instrução, o código de operação (CO) consiste nos 4 bits mais significativos do primeiro endereço de memória e o operando são os 4 bits menos significativos do primeiro endereço concatenados com os 8 bits do segundo endereço. Essa operação foi feita nas linhas abaixo.

```
CO = MEM[CI] >> 4
OP = ((MEM[CI] % 0x10) << 8) + MEM[CI+1]
```

Com o código de operação e o operando decodificados, a simulação vai para um if que executa a instrução de acordo com seu CO, da forma que foi descrita na Figura 4.

```
if CO == 0x0:  # JP
        CI = OP

    elif CO == 0x1: # JZ
        if AC == 0:
            CI = OP
        else:
            CI += 2

    elif CO == 0x2: # JN
        if AC < 0:
            CI = OP
        else:
            CI += 2

    elif CO == 0x3: # LV
        AC = OP
        CI += 2

    elif CO == 0x4: # +
        AC += MEM[OP]
        CI += 2
```

```
    elif CO == 0x5: # -
        AC -= MEM[OP]
        CI += 2

    elif CO == 0x6: # *
        AC *= MEM[OP]
        CI += 2

    elif CO == 0x7: # /
        AC = AC // MEM[OP]
        CI += 2

    elif CO == 0x8: # LD
        AC = MEM[OP]
        CI += 2

    elif CO == 0x9: # MM
        MEM[OP] = AC
        CI += 2

    elif CO == 0xa: # SC
        MEM[OP] = CI >> 8
        MEM[OP+1] = CI % 0x100
        CI = OP + 2

    elif CO == 0xb: # RS
        CI = OP

    elif CO == 0xc: # HM
        CI = OP
        status = 'halt'

    elif CO == 0xd: # GD
        if len(FILE) > 0:
            AC = FILE.pop(0)
        else:
            AC = 0x70
        CI += 2

    elif CO == 0xe: # PD
        FILE.append(AC)
        print(hex(AC)[2:])
        CI += 2

    elif CO == 0xf: # OS
        CI += 2

    else:
        status = 'error'
```

Após executar a instrução, a máquina retorna os novos valores do contador de instruções, do acumulador, limitado a 8 bits em caso de overflow, e do status da máquina (erro ou halt).

```
return (CI, AC % 0x100, status)
```

3. Loader

3.1. Definição

O loader é um programa que é pré-carregado na memória da MVN, como mostra a Figura 5 e é o responsável por ler o código armazenado em um meio externo, que no caso da simulação aqui descrita é um arquivo em hexadecimal, análogo a uma fita perfurada.

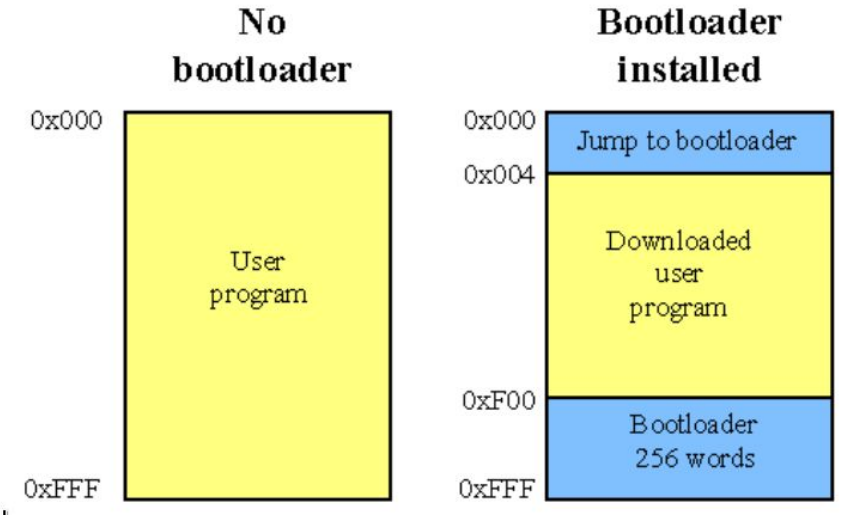


Figura 5 – Loader na memória

A lógica de funcionamento de um loader demonstrada na Figura 6 consiste em ler um byte do arquivo, guardar esse byte no local apropriado na memória da MVN, decrementar o ponteiro que indica o tamanho do arquivo, checar se é o último byte, se for deve mudar o contador de instruções para apontar para o programa carregado, caso contrário o loop recomeça lendo outro byte.

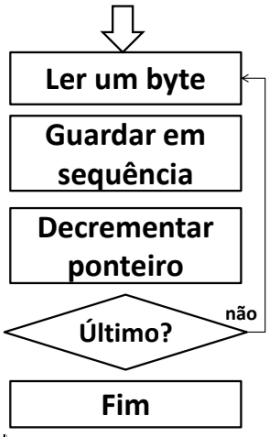


Figura 6 – Loader na memória

3.2. Especificação

O loader do projeto foi feito em linguagem de baixo nível e roda dentro da própria máquina virtual. Na MVN, a posição 0 de memória contém uma instrução de jump incondicional para a posição F00, como mostra a Figura 5, e o código do loader em si se encontra nos endereços de F00 a FFF.

3.3. Implementação

Primeiramente, o código indica o jump na posição 0 de memória e o resto do código se encontra da posição F00 em diante.

```
              @  0
              JP F00
```

Os valores nos endereços de 1 a EFF são 0 e do F00 adiante se encontra o código do loader em si:

```
              @  F00
              ; carrega o endereco inicial do programa
              GD 0
              ; INICIAL_MSB e INICIAL_LSB vao ficar 0xxx, indicando
              ; jump para endereco inicial do programa
              MM INICIAL_MSB
              ; soma 90 para POS_MEM_MSB e POS_MEM_LSB ficarem 9xxx,
              ; indicando MM da posicao inicial do bloco
              +  OP_SOMA
              MM POS_MEM_MSB

              GD 0
              MM INICIAL_LSB
              MM POS_MEM_LSB

              ; carrega tamanho do bloco
              GD 0
              MM SIZE

              ; le um byte da fita
LER_BYTE      GD 0
              JP POS_MEM_MSB
              ; soma um no endereco da memoria
RETURN_MM   LD POS_MEM_LSB
              +  UM
              ; checa se deu overflow
              JZ SOMA_UM_MSB
RETURN_OF   MM POS_MEM_LSB
              ; subtrai um do size
              LD SIZE
              -  UM
              MM SIZE
              ; checa se chegou em zero, se chegou recomeca bloco
              JZ START_BLOCK
              ; se nao chegou, le proximo byte da fita
              JP LER_BYTE

              ; carrega a posicao inicial do bloco
```

```
START_BLOCK GD 0
            ; essa soma transforma o endereco em uma operacao MM.
            ; se for end of file, GD responde com 70, logo a soma
            ; com 90 dá 00 por causa do overflow
            +  OP_SOMA
            ; se for end of file da jump pro inicio do programa
            JZ START_PROG
            MM POS_MEM_MSB

            GD 0
            MM POS_MEM_LSB

            ; carrega tamanho do bloco
            GD 0
            MM SIZE

            JP LER_BYTE


SOMA_UM_MSB LD POS_MEM_MSB
            +  UM
            MM POS_MEM_MSB
            JP RETURN_OF

            ; zera o AC
START_PROG  LV 0
INICIAL_MSB K  0
INICIAL_LSB K  0

POS_MEM_MSB K  0
POS_MEM_LSB K  0
JP          RETURN_MM

SIZE        K  0

OP_SOMA     K  90
UM          K  1
```

4. Montador

4.1. Definição

O montador é um programa escrito em linguagem de alto nível que converte um código escrito em mnemônicos Assembly em seus respectivos valores em binário, que podem então ser carregados na máquina virtual pelo loader e executados.

O montador deve ser capaz de associar rótulos a certas instruções para marcar sua posição na memória, facilitando o trabalho do programador. Além disso, ele deve oferecer a opção de pseudo instruções que definem o endereço inicial de um bloco de código e o final desse bloco.

4.2. Implementação

Na implementação do montador são feitas duas passadas no código em Assembly para se gerar o código em binário que pode ser carregado pelo loader.

A primeira passada captura os rótulos e define seus endereços correspondentes em uma tabela de símbolos. Na segunda passagem, os rótulos são substituídos por seus endereços correspondentes e a saída, chamada de código objeto, é gerada.

O código objeto pode ser formado por vários blocos de código diferentes que se iniciam em endereços diferentes. Esses blocos se iniciam com 2 bytes que indicam o endereço inicial, seguidos por 1 byte que contém o tamanho do bloco, então há o conteúdo do bloco e por fim há um checksum para detecção de possíveis erros.

Na implementação do montador, primeiramente o código lê um arquivo e elimina as linhas em branco e os comentários, que se iniciam com ;. O código que faz isso é:

```
def make_listing(code):
    lines = code.splitlines()

    listing = ''
    for line in lines:
        line = line.strip()
        if line and not line.startswith(';'):
            listing += line.partition(';')[0] + '\n'

    return listing.upper()
```

Há definido no código, uma tabela de mnemônicos que relaciona cada mnemônico ao valor binário da instrução.

```
tabela_mnemonicos = {
    'JP': 0x0,
    'JZ': 0x1,
    'JN': 0x2,
    'LV': 0x3,
    '+':  0x4,
    '-':  0x5,
    '*':  0x6,
    '/':  0x7,
    'LD': 0x8,
    'MM': 0x9,
    'SC': 0xA,
    'RS': 0xB,
    'HM': 0xC,
    'GD': 0xD,
    'PD': 0xE,
    'OS': 0xF
}
```

Em seguida há o código que realiza todo o processo de montagem, incluindo os dois passos descritos anteriormente, a formatação do código objeto de saída de acordo com as pseudo instruções.

```
def montador(code, com_checksum):
    listing = make_listing(code)
```

```
    # tabela de símbolos
    simbolos = {}

    print('DEBUG listing:')
    print(listing)

    # executa os dois passos da montagem
    for i in range(2):
        lines = listing.splitlines()

        object_code = ''
        size = 0
        checksum = 0

        for line in lines:
            line_elements = line.split()

            if len(line_elements) == 3:
                rotulo = line_elements.pop(0)
                simbolos[rotulo] = hex(current_line_address)[2:]

            mnemonico = line_elements[0]

            if len(line_elements) > 1:
                try:
                    operando = hex(int(line_elements[1], 16))[2:]
                except ValueError:
                    if line_elements[1] in simbolos:
                        operando = simbolos[line_elements[1]]
                    else:
                        operando = simbolos[line_elements[1]] = '000'
            else:
                operando = ''

            if mnemonico in tabela_mnemonicos:
                instrucao = hex(tabela_mnemonicos[mnemonico])[2:] + operando.zfill(3)
                object_code += instrucao
                print(hex(current_line_address) + ':', instrucao)
                current_line_address += 2
                size += 2
                checksum += int(instrucao[:2], 16) + int(instrucao[2:], 16)

            elif mnemonico == '@':
                if com_checksum:
                    size += 1
                    checksum += size
                    object_code += hex(((-checksum) % 0x100) & 0xff)[2:].zfill(2)

                object_code = object_code.replace('size', hex(size)[2:].zfill(2))

                current_line_address = int(line_elements[1], 16)
                endereco_inicial = hex(current_line_address)[2:].zfill(4)

                object_code += endereco_inicial
                #print(hex(current_line_address) + ':',
hex(current_line_address)[2:].zfill(4))
                object_code += 'size'

                size = 0
                checksum = int(endereco_inicial[:2], 16) + int(endereco_inicial[2:], 16)

            elif mnemonico == '#':
                pass

            elif mnemonico == 'K':
                instrucao = operando.zfill(2)
                object_code += instrucao
                print(hex(current_line_address) + ':', instrucao)
                current_line_address += 1
                size += 1
```

```
                checksum += int(instrucao, 16)

    if com_checksum:
        size += 1
        checksum += size
        object_code += hex((~(checksum%0x100)+1) & 0xff)[2:].zfill(2)
        object_code = object_code[2:]

    object_code = object_code.replace('size', hex(size)[2:].zfill(2))

    print('DEBUG Tabela de símbolos:')
    for simbolo in simbolos:
        print(simbolo + ':', simbolos[simbolo])
    print()
    print('DEBUG Tabela código objeto:')

    output_code = ''
    for i in range(0, len(object_code), 32):
        for j in range(0, 32, 2):
            output_code += object_code[i+j:i+j+2] + ' '
            print(object_code[i+j:i+j+2], end=' ')
        output_code += '\n'
        print()

    return output_code
```

5. Interpretador de comandos

5.1. Definição

Para realizar a interface com o montador e a máquina virtual, foi desenvolvido um interpretador de comandos, onde o usuário pode definir os arquivos que serão usados como entrada e saída, como disco da máquina virtual e pode chamar todos os programas, a máquina virtual, o montador e o interpretador de linguagem de alto nível que será visto mais adiante.

Os comandos que esse interpretador de comandos reconhece estão descritos na Figura 7.

comando	formato	ação
Inicia operação	**$JOB nome**	Login - inicia um novo job, reinicia todo o sistema
Escolhe pasta	**$DISK pasta**	Simula disco do sistema nesta pasta do hospedeiro
Arquivos existentes	**$DIRECTORY**	Lista o conteúdo da pasta do hospedeiro
Cria arquivo	**$CREATE nome**	Cria no disco um novo arquivo, se ainda não existir
Apaga arquivo	**$DELETE nome**	Remove do disco o arquivo indicado, se existir
Mostra conteúdo	**$LIST nome**	Apresenta o conteúdo do arquivo indicado, se existir
Mídia de entrada	**$INFILE nome**	Adota o arquivo indicado como a fita de entrada
Mídia de saída	**$OUTFILE nome**	Adota o arquivo indicado como a fita de saída
Arquivo em disco	**$DISKFILE nome**	Adota o arquivo indicado como arquivo em disco
Executa programa	**$RUN nome**	Executa o programa indicado (de sistema ou ususário)
Encerra operação	**$ENDJOB nome**	Finaliza pendências e termina o job corrente

Figura 7 - Descrição dos comandos

5.2. Implementação

O interpretador de comandos foi implementado com um motor de eventos, onde cada input do usuário é considerado um evento que aciona uma ação específica como criar um arquivo, definir um arquivo entrada ou saída e chamar o montador ou a máquina virtual. Se for acionado um comando não existente, uma mensagem de erro é mostrada ao usuário.

```
job = ''
pwd = os.path.join(os.getcwd(), 'teste')
infile = 'teste_in'
outfile = 'teste_out'
diskfile = 'teste_disk'

while (True):
    event = input('> ').split(' ')
    event[0] = event[0].lower()
    #print(event)

    if event[0] == '$job':
        try:
            job = event[1]
            pwd = os.getcwd()
            infile = ''
            outfile = ''
            diskfile = ''
            print('Iniciado o job', job)
        except IndexError:
            print('Especifique um job')

    elif event[0] == '$disk':
        try:
            pwd = os.path.join(pwd, event[1])
        except IndexError:
            print('Especifique um diretório')

    elif event[0] == '$directory':
        for item in os.listdir(pwd):
            print(item)

    elif event[0] == '$create':
        try:
            f = open(os.path.join(pwd, event[1]), 'x')
            f.close()
        except FileExistsError:
            print('Arquivo já existe')
        except IndexError:
            print('Especifique o nome do arquivo')

    elif event[0] == '$delete':
        try:
            os.remove(os.path.join(pwd, event[1]))
        except IndexError:
            print('Especifique o nome do arquivo')
        except FileNotFoundError:
            print('Arquivo não encontrado')
        except IsADirectoryError:
            print('Nome especificado é um diretório e não um arquivo')

    elif event[0] == '$list':
        try:
            with open(os.path.join(pwd, event[1]), 'r') as f:
                for line in f.readlines():
```

```
                    print(line, end='')
                print()
        except IndexError:
            print('Especifique o nome do arquivo')
        except FileNotFoundError:
            print('Arquivo não encontrado')

    elif event[0] == '$infile':
        try:
            infile = event[1]
            print('Infile:', infile)
        except IndexError:
            print('Especifique um arquivo')

    elif event[0] == '$outfile':
        try:
            outfile = event[1]
            print('Outfile:', outfile)
        except IndexError:
            print('Especifique um arquivo')

    elif event[0] == '$diskfile':
        try:
            diskfile = event[1]
            print('Diskfile:', diskfile)
        except IndexError:
            print('Especifique um arquivo')

    elif event[0] == '$run':
        try:
            with open(os.path.join(pwd, diskfile), 'r') as diskfile_obj:
                with open(os.path.join(pwd, infile), 'r') as infile_obj:
                    with open(os.path.join(pwd, outfile), 'w') as outfile_obj:
                        if event[1] == 'vm':
                            saida = vm.main(diskfile_obj.read(), infile_obj.read(),
False)
                            outfile_obj.write(saida)
                        elif event[1] == 'montador':
                            saida = montador.montador(infile_obj.read(),
infile_obj.read())
                            outfile_obj.write(saida)
                        elif event[1] == 'interpretador':
                            saida = interpretador.run(diskfile_obj.read(),
infile_obj.read())
                            outfile_obj.write(saida)
        #except IndexError:
        #    print('Especifique um programa')
        except FileNotFoundError:
            print('Arquivo não encontrado')

    elif event[0] == '$endjob':
        print('Até logo!')
        break

    else:
        if len(event[0]) > 0:
            print(event[0], 'não é um programa conhecido')
            if event[0][0] != '$':
                print('Não se esqueça do $')
```

6. Linguagem de alto nível

6.1. Definição

Foi implementado um interpretador de linguagem de alto nível, que é capaz de rodar programas sem a necessidade da etapa de montagem como é o caso dos programas que rodam na máquina virtual. Isso simplifica o projeto e permite ao programador utilizar uma linguagem de mais alto nível, acelerando o processo de programação. Essa linguagem é bem simples e está descrita na Figura 8.

Comando	Formato sintático	Operação
Comando com rótulo	**NNNN : comando**	Associa rótulo(s) NNNN ao comando
Comando sem rótulo	**comando**	Este comando não tem rótulo
Comando atribuição	**LET NNNN = expressão**	Associa a NNNN o valor da expressão
Comando desvio incondicional	**GOTO NNNN**	Desvia para o rótulo NNNN
Comando desvio condicional	**GOTO NNNN IF expressão op expressão**	Desvia para o rótulo NNNN apenas se essa comparação for verdadeira
Comando entrada	**READ NNNN**	Dá a NNNN o valor lido na entrada
Comando saída	**WRITE expressão**	Grava na saída o valor da expressão
Expressão	**valor {(+|-|*|/) valor}**	Série de operações sobre valores
valor	**NNNN | DDDD**	Valores são: ou variáveis ou números

NNNN = nome de uma variável ou rótulo; DDDD = inteiro decimal sem sinal;
Parênteses agrupam opções separadas por barras verticais);
Chaves agrupam elementos que podem ocorrer zero ou mais vezes

Figura 8 - Linguagem de alto nível

6.2. Implementação

O interpretador é o programa responsável por ler o código na linguagem de alto nível e executar as ações correspondentes e para isso se utilizou um motor de eventos onde cada símbolo da linguagem é tratada como um evento, onde os símbolos podem ser as palavras-chave como LET, GOTO, READ e WRITE, IF, operações como =, +, -, *, / e os rótulos.

Primeiramente são extraídos esses símbolos do arquivo de entrada e o end of line (\n) são considerados como separadores entre os comandos.

```
def extrair(code):
    code = code + '\n'
    palavras = []
    while len(code) > 0:
        nextChar = ''
        firstWord = ''
        while len(code) > 0:
            nextChar = code[0]
            code = code[1:]
            if nextChar in [':', '+', '-', '*', '/', ';', '=']:
                if firstWord != '':
                    palavras.append(firstWord)
                firstWord = nextChar
                palavras.append(firstWord)
                break
            elif nextChar == ' ':
                if firstWord != '':
                    palavras.append(firstWord)
                break
            elif nextChar == '\n':
                if firstWord != '':
                    palavras.append(firstWord)
                if (len(palavras) > 1) and (palavras[-1] != '\n'):
                    palavras.append(nextChar)
                break
            else:
                firstWord += nextChar

    return palavras
```

Com os símbolos separados, criou-se um motor de eventos onde o tratamento dos eventos segue a Figura 9 de acordo com as entradas.

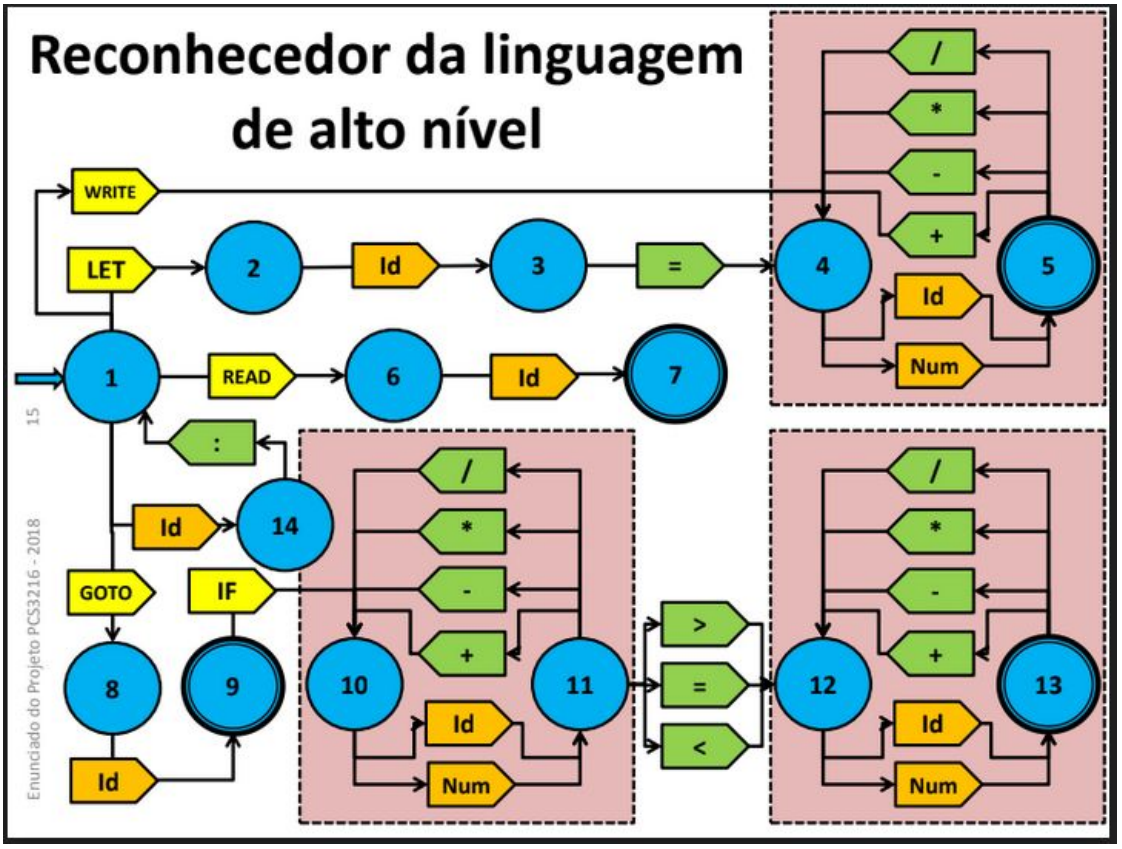


Figura 9 - Estados interpretador de linguagem

```
def run(code, infile):
    palavras = extrair(code)
    #print(palavras)

    Id = ''
    Num = 0
    Op = ''
    Comp1 = 0
    Comp2 = 0
    CompOp = ''
    isWrite = False
    outfile = ''
    inputs = infile.splitlines()

    variaveis = {}
    rotulos = {}
    state = 1

    i = 0
    while i < len(palavras):
        palavra = palavras[i]

        #print(variaveis)
        #print(rotulos)
        #print('state', state, 'palavra', palavra)

        if state == 1:
            if palavra == 'LET':
                state = 2
            elif palavra == 'READ':
                state = 6
            elif palavra == 'GOTO':
                state = 8
            elif palavra == 'WRITE':
                state = 4
                isWrite = True
            else:
                Id = palavra
                state = 14

        elif state == 2:
            Id = palavra
            state = 3

        elif state == 3:
            Op = '='
            state = 4

        elif state == 4:
            if palavra in variaveis:
                Num2 = variaveis[palavra]
            else:
                Num2 = int(palavra)

            if Op == '=':
                Num = Num2
            elif Op == '/':
                Num /= Num2
            elif Op == '*':
                Num *= Num2
            elif Op == '-':
                Num -= Num2
            elif Op == '+':
                Num += Num2
            state = 5

        elif state == 5:
            if palavra == '\n':
                if isWrite:
                    print(Num)
```

```
                outfile += str(Num)
            else:
                variaveis[Id] = Num
            state = 1
        else:
            Op = palavra
            state = 4

    elif state == 6:
        Id = palavra
        state = 7

    elif state == 7:
        variaveis[Id] = int(inputs.pop(0))
        state = 1

    elif state == 8:
        Id = palavra
        state = 9

    elif state == 9:
        if palavra == '\n':
            if Id in rotulos:
                i = rotulos[Id]
            else:
                for key, value in enumerate(palavras):
                    if (value == Id) and (palavras[key+1] == ':'):
                        i = key-1
            state = 1
        elif palavra == 'IF':
            Op = 'IF'
            state = 10

    elif state == 10:
        if palavra in variaveis:
            Num2 = variaveis[palavra]
        else:
            Num2 = int(palavra)

        if Op == 'IF':
            Num = Num2
        elif Op == '/':
            Num /= Num2
        elif Op == '*':
            Num *= Num2
        elif Op == '-':
            Num -= Num2
        elif Op == '+':
            Num += Num2
        state = 11

    elif state == 11:
        if palavra in ['>', '=', '<']:
            Comp1 = Num
            CompOp = palavra
            Op = CompOp
            state = 12
        else:
            Op = palavra
            state = 10

    elif state == 12:
        if palavra in variaveis:
            Num2 = variaveis[palavra]
        else:
            Num2 = int(palavra)

        if Op in ['>', '=', '<']:
            Num = Num2
        elif Op == '/':
```

```
                Num /= Num2
            elif Op == '*':
                Num *= Num2
            elif Op == '-':
                Num -= Num2
            elif Op == '+':
                Num += Num2
            state = 13

        elif state == 13:
            if palavra == '\n':
                Comp2 = Num
                if CompOp == '>':
                    if Comp1 > Comp2:
                        if Id in rotulos:
                            i = rotulos[Id]
                        else:
                            for key, value in enumerate(palavras):
                                if (value == Id) and (palavras[key+1] == ':'):
                                    i = key-1
                elif CompOp == '=':
                    if Comp1 == Comp2:
                        if Id in rotulos:
                            i = rotulos[Id]
                        else:
                            for key, value in enumerate(palavras):
                                if (value == Id) and (palavras[key+1] == ':'):
                                    i = key-1
                elif CompOp == '<':
                    if Comp1 < Comp2:
                        if Id in rotulos:
                            i = rotulos[Id]
                        else:
                            for key, value in enumerate(palavras):
                                if (value == Id) and (palavras[key+1] == ':'):
                                    i = key-1
                state = 1
            else:
                Op = palavra
                state = 12

        elif state == 14:
            rotulos[Id] = i
            state = 1

        i += 1

    return outfile


def extrair(code):
    code = code + '\n'
    palavras = []
    while len(code) > 0:
        nextChar = ''
        firstWord = ''
        while len(code) > 0:
            nextChar = code[0]
            code = code[1:]
            if nextChar in [':', '+', '-', '*', '/', ';', '=']:
                if firstWord != '':
                    palavras.append(firstWord)
                firstWord = nextChar
                palavras.append(firstWord)
                break
            elif nextChar == ' ':
                if firstWord != '':
                    palavras.append(firstWord)
                break
            elif nextChar == '\n':
```

```
                if firstWord != '':
                    palavras.append(firstWord)
                if (len(palavras) > 1) and (palavras[-1] != '\n'):
                    palavras.append(nextChar)
                break
            else:
                firstWord += nextChar

    return palavras
```