LEONARDO LEONZI D'ALESSANDRO - 8942242

PROJETO DE SISTEMAS DE PROGRAMAÇÃO

Projeto da disciplina PCS3216 – Sistemas de Programação, do Curso de Engenharia Elétrica da Escola Politécnica da Universidade de São Paulo.

Professor
João José Neto

São Paulo
2019

SUMÁRIO

1. INTRODUÇÃO

Neste trabalho foi projetado e implementado um sistema de programação capaz de, a partir da leitura de um conjunto de instruções descritas em codificação *assembly*, interpretar tais comandos, armazená-los em sua memória interna, traduzi-los para uma linguagem de máquina específica e enfim executá-los.

Para tal foram projetados os componentes fundamentais ao fluxo de processos desejado. Em linhas gerais, o sistema de programação projetado é composto por **4** componentes. Em primeiro lugar, um montador (ou *assembler*), responsável por traduzir os códigos de linguagem *assembly* para uma linguagem de máquina. Uma memória, encarregada por armazenar um conjunto de instruções já traduzidas pelo montador. Um carregador (ou *loader*), responsável por carregar na memória o código traduzido pelo montador. Um motor de eventos, cuja função é interpretar o fluxo de instruções armazenadas na memória e executar as sub-rotinas adequadas.

A seguir, um diagrama que explicita a organização geral do sistema, retirada do material teórico da disciplina PCS3216.

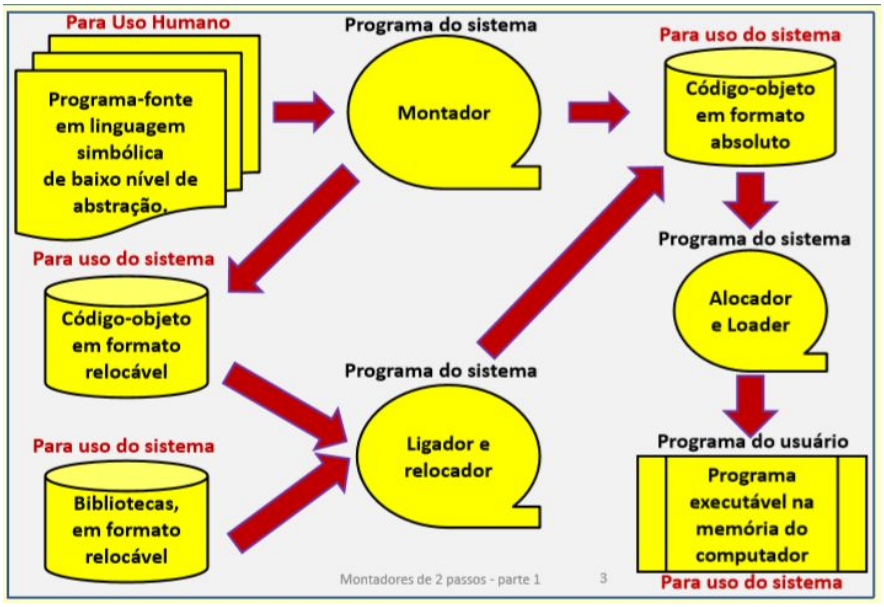


Figura 1

Resume-se aqui informações sobre a memória principal e registradores utilizados:

Memória

- **64** bits de palavra;
- **12** bits de endereçamento, **4096** posições de memória

Registradores

- **Program Counter**
- **Endereço de retorno**
- **Acumulador**
- **Tabela de símbolos**

O sistema exposto neste projeto foi desenvolvido por meio da utilização da linguagem de programação *Python,* e das bibliotecas por ela disponibilizadas.

2. MONTADOR - ASSEMBLER

A fim de simular-se um sistema de programação coerente com a realidade, optou-se pelo projeto de um *assembler* que respeitasse a lógica dos montadores de dois passos.

Dessa forma, o montador projetado realiza em sua primeira etapa a leitura do código-fonte e a montagem de uma tabela dos símbolos encontrados no código. Além disso, no primeiro passo, o montador é responsável por buscar incoerências no código introduzido.

Na segunda etapa de operação, o *assembler* projetado realiza a releitura do código fonte e gera enfim um programa objeto para ser executado.

O conjunto de instruções escolhido para se simular no presente sistema está expresso na **Figura 2**, retirada do material teórico da disciplina PCS3216.

```
; MNEMÔNICOS    CÓDIGO    INSTRUÇÃO DA MÁQUINA VIRTUAL
;                  OBS.1: y=número do banco de memória (hexadecimal,4 bits)
;                  OBS.2: x=dígito (hexadecimal, 4 bits)
;   JP    J      /0xxx     JUMP (UNCONDITIONAL) desvia para endereço /yxxx
;   JZ    Z      /1xxx     JUMP IF ZERO idem, se acumulador contém zero
;   JN    N      /2xxx     JUMP IF NEGATIVE idem, se acumulador negativo
;   CN    C      /3x       * instruções de controle (/x = operação desejada)
;   +     +      /4xxx     ADD soma ac. + conteúdo do endereço /yxxx (8bits)
;   -     -      /5xxx     SUBTRACT idem, subtrai (operação em 8 bits)
;   *     *      /6xxx     MULTIPLY idem, multiplica (operação em 8 bits)
;   /     /      /7xxx     DIVIDE idem, divide (operação em 8 bits)
;   LD    L      /8xxx     LOAD FROM MEMORY carrega ac. com dado de /yxxx
;   MM    M      /9xxx     MOVE TO MEMORY move para /yxxx o conteúdo do ac.
;   SC    S      /Axxx     SUBROUTINE CALL guarda end.de retorno e desvia
;   OS    O      /Bx       * OPERATING SYSTEM CALL - 16 chamadas do sistema
;   IO    I      /Cx       * INPUT/OUTPUT - entrada, saída, interrupção
;                /D        * combinação disponível
;                /E        * combinação disponível
;                /F        * combinação disponível
;
; MNEMÔNICOS    OPERANDO  PSEUDO-INSTRUÇÃO DO MONTADOR
;   @     @      /yxxx     ORIGIN define end. inicial do código a seguir
;   #     #      /yxxx     END define final físico do prog. e end. de partida
;   $     $      /xxx      ARRAY define área de trabalho c/tamanho indicado
;   K     K      /xx       CONSTANT preenche byte corrente c/ constante
```

Figura 2

2.1. PRIMEIRO PASSO DO MONTADOR

Como já comentado, no primeiro passo o montador desenvolvido realiza uma primeira leitura do código fonte introduzido, detectando os comandos utilizados e preenchendo uma tabela de símbolos. A **Figura 3**, retirada do material teórico da disciplina, exibe o fluxo geral do primeiro passo do montador.

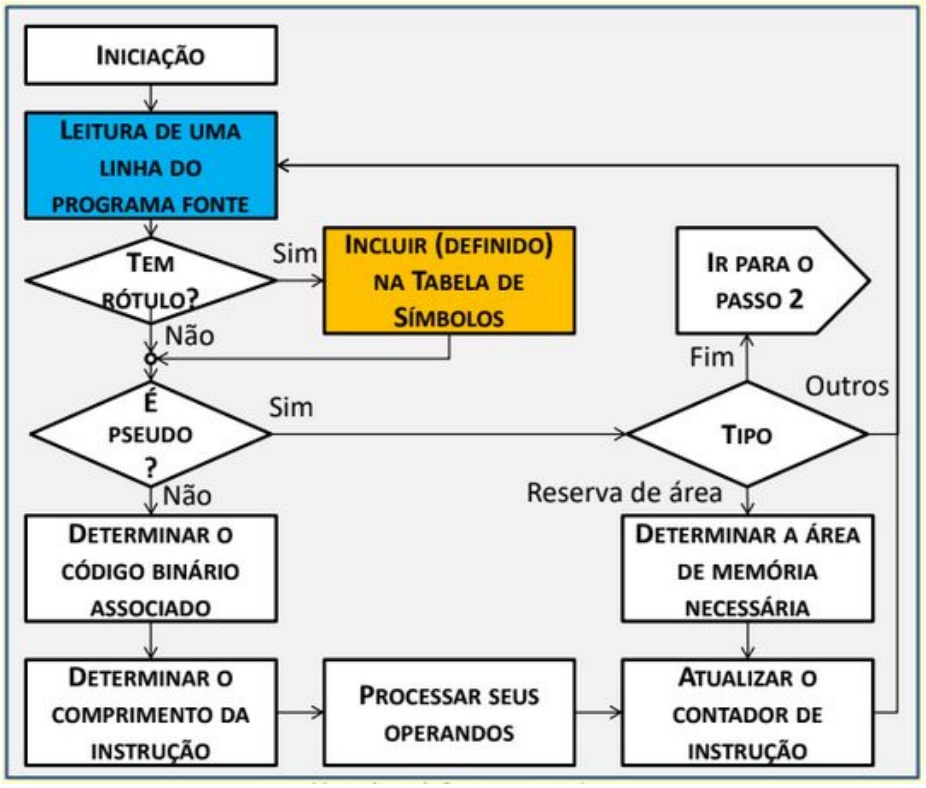


Figura 3

Sendo assim, foi criada uma classe nomeada *Assembler*, que contém como parâmetros uma tabela de símbolos, inicialmente vazia e uma tabela de mnemônicos já preenchida. A tabela de símbolos armazena os símbolos encontrados durante a primeira leitura do código fonte. Ela guarda também o endereço ou valor numérico da instrução em que o símbolo foi detectado.

Já a tabela de mnemônicos armazena um compilado de instruções reconhecidas pelo sistema. Ela guarda o símbolo do mnemônico de determinada instrução, quais tipos de operandos a instrução deve receber e o valor hexadecimal referente ao seu código em linguagem de máquina.

Com essas informações, um método *primeiroPasso*, referente à classe *Assembler,* foi desenvolvido. Tal método recebe um arquivo *.txt* que possui um código, supostamente em linguagem *assembly*. Primeiramente, o montador manipula o arquivo, detectando cada linha do código e armazenando cada em

uma linha de um array denominado *instrucoes,* por meio de uma função denominada *leitura*.

```
def leitura(self,arquivo):
        f=open(arquivo, "r")
        f1 = f.readlines()
        instrucoes = []
        for x in f1:
            instrucoes.append(x)
```

Em seguida, constrói-se a tabela de símbolos, passando por todos os elementos do array *instrucoes.* Utilizando-se a função *findall* da biblioteca *regex* do *Python*, foi possível separar os termos de cada instrução, sempre que um espaço foi localizado. Isso permitiu identificar em uma instrução a parcela que trata do mnemônico utilizado (definindo o tipo de instrução) e a parcela que trata dos parâmetros.

Foi considerada também a possível presença de rótulos. Isso foi feito observando-se se o primeiro *"slot"* da primeira instrução lida possui algum termo ou está vazio. Para compreender esta lógica, basta observar o seguinte exemplo de código:

LOOP JP 400

Neste caso o termo "*LOOP",* presente no primeiro *slot* da instrução representa um rótulo. No entanto, no seguinte exemplo:

_ JP 400

Observa-se que o primeiro *slot* se encontra vazio, a portanto a instrução não possui rótulo. A lógica em *Python* que representa a lógica de detecção de rótulos está expressa a seguir.

```
if (self.instr[0] != ' '):
      ehRotulo = 1
      self.tabelaDeSimbolos.append([self.instr[0], numInstrucao, ehRotulo])
```

No código, procura-se no primeiro termo do *string* de instruções um espaço vazio. Caso não seja encontrado, trata-se de uma instrução com rótulo. Nesse caso, adiciona-se à tabela de símbolos tal rótulo, junto do número que define a posição de tal instrução no programa completo e um sinal binário 1 que informa que se trata de um rótulo definido (variável *ehRotulo*).

Por outro lado, caso o primeiro termo da instrução seja um espaço em branco, deduz-se que a instrução não possui um rótulo. Nesse caso, adiciona-se à tabela de símbolos o mnemônico utilizado no *slot* seguinte, assim como o número de posição da instrução no programa e o valor 0 para *ehRotulo* indicando que se trata de uma instrução sem rótulo. Isso foi projetado com o seguinte código.

```
elif(self.instr[0] == ' '):
      ehRotulo = 1
      self.tabelaDeSimbolos.append([self.instr[1], numInstrucao, ehRotulo])
```

Enfim, a primeira parte do código referente ao primeiro passo do montador foi definida como:

```
def primeiroPasso(self,arquivo):
      ehRotulo = 0
      instrucoes = self.leitura(arquivo)
      for numInstrucao in range (len(instrucoes)):
          #Separa os termos da instrução e guarda num array de strings
          self.instr = re.findall(r"[^,.:;' ]+|[,.:;' ]", instrucoes[numInstrucao])
          if (self.instr[0] != ' '):
              ehRotulo = 1
              # montagem tabela de simbolos
              self.tabelaDeSimbolos.append([self.instr[0], numInstrucao, ehRotulo])
          elif(self.instr[0] == ' '):
              self.tabelaDeSimbolos.append([self.instr[1], numInstrucao, ehRotulo])
```

Por meio de tal código constrói-se uma tabela de símbolos com o seguinte formato, com os exemplos das instruções " JP 900" e "LOOP JP 900", supondo que ambas foram encontradas no endereço 400 de memória:

Campo 1	***Campo 2***	***Campo 3***

índice 0 (Símbolo encontrado)	índice 1 (Endereço de memória da instrução)	índice 2 (Determina se é rótulo (1) ou não é rótulo (0))
JP	**400**	**0**
LOOP	**400**	**1**

Em seguida, projetou-se a lógica referente à tabela de mnemônicos e a comparação desta com a tabela de símbolos formada anteriormente.

Inicialmente, verifica-se se o símbolo armazenado é um rótulo ou não, analisando-se o terceiro valor de cada termo armazenado em cada valor da tabela de símbolos. Isso é feito com o seguinte código:

```
if (self.tabelaDeSimbolos[i][2]==0): #não eh rótulo
            (...)
```

Caso o símbolo não seja um rótulo, verifica-se se o tipo de instrução (armazenado na primeira casa de cada termo da tabela de símbolos) se encontra na tabela de mnemônicos. Para tal, é feita uma varredura em ambas as tabelas, paraa cada termo que se deseja encontrar na tabela de mnemônicos. Após a comparação, caso um termo presente na tabela de símbolos não seja identificado na tabela de mnemônicos, o programa simula um erro de identificação, imprimindo a mensagem *"Erro: Instrução Inválida",* e por meio do comando *exit(),* disponibilizado pela biblioteca *sys,* a simulação é interrompida.

Para o correto funcionamento dessa lógica utilizou-se uma variável booleana *encontrouSimbolo* que inicia em estado *False* e quando o termo da tabela de símbolos é detectado na tabela de mnemônicos recebe o valor *True.* Ao final da varredura de todos os elementos da tabela de símbolos, caso a variável *encontrouSímbolo* seja *False*, o programa simula o erro. O código referente a esta lógica está disposto a seguir.

```
    encontrouSimbolo = False
    for i in range (len(self.tabelaDeSimbolos)):
       if (self.tabelaDeSimbolos[i][2]==0): #não eh rótulo
              if (self.tabelaDeSimbolos[i][0] == 'JP' or self.tabelaDeSimbolos[i][0]   ==
'J'):
                encontrouSimbolo = True

              elif (self.tabelaDeSimbolos[i][0] == 'JZ' or self.tabelaDeSimbolos[i][0]   ==
'Z'):
                encontrouSimbolo = True

              elif (self.tabelaDeSimbolos[i][0] == 'JN' or self.tabelaDeSimbolos[i][0]   ==
'N'):
                encontrouSimbolo = True

              elif (self.tabelaDeSimbolos[i][0] == 'CN' or self.tabelaDeSimbolos[i][0]   ==
'C'):
                encontrouSimbolo = True

            elif (self.tabelaDeSimbolos[i][0] == '+'):
                encontrouSimbolo = True

            elif (self.tabelaDeSimbolos[i][0] == '-'):
                encontrouSimbolo = True

            elif (self.tabelaDeSimbolos[i][0] == '*'):
                encontrouSimbolo = True

            elif (self.tabelaDeSimbolos[i][0] == '/'):
                encontrouSimbolo = True

              elif (self.tabelaDeSimbolos[i][0] == 'LD' or self.tabelaDeSimbolos[i][0]   ==
'L'):
                encontrouSimbolo = True

              elif (self.tabelaDeSimbolos[i][0] == 'MM' or self.tabelaDeSimbolos[i][0]   ==
'M'):
                encontrouSimbolo = True

              elif (self.tabelaDeSimbolos[i][0] == 'SC' or self.tabelaDeSimbolos[i][0]   ==
'S'):
                encontrouSimbolo = True

              elif (self.tabelaDeSimbolos[i][0] == 'OS' or self.tabelaDeSimbolos[i][0]   ==
'O'):
                encontrouSimbolo = True

              elif (self.tabelaDeSimbolos[i][0] == 'IO' or self.tabelaDeSimbolos[i][0]   ==
'I'):
                encontrouSimbolo = True

            elif(self.tabelaDeSimbolos[i][0] == '@'):
                encontrouSimbolo = True

            elif(self.tabelaDeSimbolos[i][0] == '#'):
                encontrouSimbolo = True

            elif(self.tabelaDeSimbolos[i][0] == '$'):
                encontrouSimbolo = True

            elif(self.tabelaDeSimbolos[i][0] == 'K'):
                encontrouSimbolo = True
```

```
if (encontrouSimbolo == False):
    print("Erro: Instrução inválida")
    sys.exit() #parar sistema
```

2.2. SEGUNDO PASSO DO MONTADOR

No segundo passo do montador é feita novamente uma leitura do código fonte, consultando-se a tabela de mnemônicos, que possui o código hexadecimal referente a cada instrução simbólica. A partir disso, o montador realiza a montagem do código objeto e, enfim do programa objeto.

O fluxo do segundo passo do montador está expresso na seguinte imagem, retirada do material teórico da disciplina PCS3216.

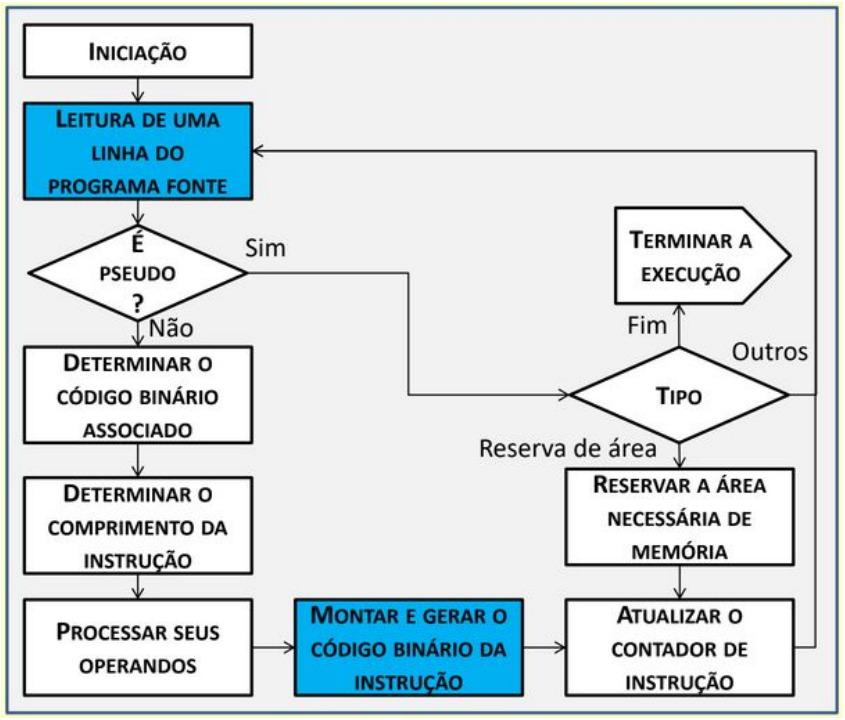


Figura 4

Sendo assim, desenvolveu-se um método denominado *segundoPasso* dentro da classe *Assembler,* que como no *primeiroPasso*, recebe o arquivo *.txt* que contém o código fonte em linguagem simbólica. Tal método tem função de traduzir cada instrução simbólica para a linguagem de máquina adequadamente. Para

este fim, a cada mnemônico lido é feita uma nova comparação com a tabela de mnemônicos a fim de se determinar o código hexadecimal correspondente. Ao se definir tal código, monta-se o código objeto, adicionando-se o código equivalente em um armazenador.

A fim deste trabalho foi utilizado um array denominado *instruCodMaq*, que recebe os códigos hexadecimais de cada uma das instruções já traduzidas. O código em *Python* referente a este passo será expresso adiante.

Para se compreender os índices utilizados nos *arrays,* o esquema abaixo foi produzido. Nele estão expressos o número do *slot* de cada instrução, o índice correspondente no *array* e a aplicação para as instruções " JP 400" e "LOOP JP 400". Note que para instruções sem rótulo, o primeiro *slot* permanece vazio. Note também que os espaços entre termos também são considerados *slots.*

PrimeiroSlot	***SegundoSlot***	***TerceiroSlot***	***QuartoSlot***	***QuintoSlot***
índice 0 (Presença ou ausência de rótulos)	**índice 1 (Espaço)**	**índice 2 (Campo do tipo de instrução)**	**índice 3 (Espaço)**	**índice 4 (Campo do operando)**
		JP		**400**
LOOP		**JP**		**400**

A seguir, o código em *Python* que representa a lógica do segundo passo do montador.

```
def segundoPasso(self, arquivo):
   instrucoes = self.leitura(arquivo)
   instruCodMaq = [0]*4097
   for numInstrucao in range (len(instrucoes)):
       self.instr = re.findall(r"[^,.:;' ]+|[,.:;' ]", instrucoes[numInstrucao]) #Separo os
termos da instrução e guardo num array de strings
       self.tipoInstr = self.instr[2]

       if (self.tipoInstr == 'JP' or self.tipoInstr == 'J'): #JP yxxx
```

```
        comando = '0x0'+ str(self.instr[4])
        instruCodMaq[numInstrucao] = comando

    elif (self.tipoInstr == 'JZ' or self.tipoInstr == 'Z'): #JP if acum = 0
        comando = comando = '0x1'+ str(self.instr[4])
        instruCodMaq[numInstrucao] = comando

    elif (self.tipoInstr == 'JN' or self.tipoInstr == 'N'): #JP if acum<0
        comando = '0x2'+ str(self.instr[4])
        instruCodMaq[numInstrucao] = comando

    elif (self.tipoInstr == 'CN' or self.tipoInstr == 'N'): #Controle
        comando = comando = '0x3'+ str(self.instr[4])
        instruCodMaq[numInstrucao] = comando

    elif (self.tipoInstr == '+'):
        comando = '0x4'+ str(self.instr[4])
        instruCodMaq[numInstrucao] = comando

    elif (self.tipoInstr == '-'):
        comando = '0x5'+ str(self.instr[4])
        instruCodMaq[numInstrucao] = comando

    elif (self.tipoInstr == '*'):
        comando = '0x6'+ str(self.instr[4])
        instruCodMaq[numInstrucao] = comando

    elif (self.tipoInstr == '/'):
        comando = '0x7'+ str(self.instr[4])
        instruCodMaq[numInstrucao] = comando

    elif (self.tipoInstr == 'LD' or self.tipoInstr == 'L'):
        comando = '0x8'+ str(self.instr[4])
        instruCodMaq[numInstrucao] = comando

    elif (self.tipoInstr == 'MM' or self.tipoInstr == 'M'):
        comando = '0x9'+ str(self.instr[4])
        instruCodMaq[numInstrucao] = comando

    elif (self.tipoInstr == 'SC' or self.tipoInstr == 'S'):
        comando = '0xa'+ str(self.instr[4])
        instruCodMaq[numInstrucao] = comando

    elif (self.tipoInstr == 'OS' or self.tipoInstr == 'O'):
        comando = '0xb'+ str(self.instr[4])
        instruCodMaq[numInstrucao] = comando

    elif (self.tipoInstr == 'IO' or self.tipoInstr == 'I'):
        comando = '0xc'+ str(self.instr[4])
        instruCodMaq[numInstrucao] = comando

    elif (self.tipoInstr == '?' or self.tipoInstr == '?'):
        comando = '0xd'+ str(self.instr[4])
        instruCodMaq[numInstrucao] = comando

    elif (self.tipoInstr == '@'):
        comando = '@0x'+ str(self.instr[4])
        instruCodMaq[numInstrucao] = comando

    elif (self.tipoInstr == '#'):
        comando = '#'
        instruCodMaq[numInstrucao] = comando
```

É possível notar que, inicialmente, verifica-se o tipo de instrução a ser tratada. Isso é feito analisando-se a qual mnemônico presente na tabela de mnemônicos a variável *tipoInstr* se identifica. A variável *tipoInstr* representa o segundo termo de cada instrução, dado que o primeiro termo é a presença de um eventual rótulo ou a ausência dele, representada como um ‘ ’. Para cada tipo de instrução, o montador a traduz para um comando em linguagem de máquina específico. Tal especificidade foi descrita na **Figura 2**. Porém, para tornar mais claro, basta analisar o exemplo:

_ JP 400

Nesse caso, no seu segundo passo, o montador realiza pela segunda vez a leitura de tal instrução. Lendo o primeiro *slot,* observa que nenhum rótulo foi implementado. Em seguida, lendo o terceiro *slot* (no código como *instr[2]* e apelidado de *tipoInstr*)*,* detecta uma instrução do tipo **JP**. Consultando a tabela de mnemônicos, define que o primeiro algarismo hexadecimal do comando correspondente em linguagem de máquina é 0. Em seguida, lendo o quinto *slot (instr[4],* no código), define o operando e o acrescenta no código em linguagem de máquina, resultando enfim em **0400.** Todos as instruções são portanto convertidas para código de máquina e armazenadas uma por vez em um *array* denominado *instruCodMaq,* o qual passa a conter em cada linha uma única instrução.

Após essa etapa, o componente *Assembler* retorna o vetor *instruCodMaq,* que será adiante armazenado na memória do sistema e enfim executado.

3. MEMÓRIA

A memória do sistema armazena as instruções traduzidas pelo *Assembler* para notação de linguagem de máquina. Para tal, foi desenvolvido um método

denominado *loader*, cuja função é simular um *Loader* de um sistema de programação. Sendo assim, esse componente é responsável por obter a lista de instruções traduzidas pelo montador e armazená-la na memória interna. O código em *Python* referente a essa lógica está expresso a seguir.

```
def loader(instruCodMaq):
      self.listaDeInstruções = instruCodMaq
```

Além desse método, o componente de memória desenvolvido possui o método *fetch*, que recebe o parâmetro *Program Counter*, advindo do motor de eventos e retorna a ele uma instrução em código de máquina a ser executada. Tal método está descrito a seguir.

```
def fetch(ProgramCounter):
            #retorna o termo indicado pelo PC:
           return self.listaDeInstruções[ProgramCounter]
```

A memória simulada possui **4096** endereços. Isso porque possui 3 algarismos hexadecimais que definem endereços. Sendo assim, possui **12 bits** de endereçamento.

Além disso, a memória simulada possui palavras de até **64 bits**. Isso porque o sistema projetado é compatível com instruções de até 4 algarismos hexadecimais.

4. MOTOR DE EVENTOS

O motor de eventos desenvolvido tem a função de executar as instruções de máquinas armazenadas na memória do sistema. Para tal, este componente possui uma lista de eventos, um compilado de instruções a serem executadas. Ele é capaz de detectar o tipo de instrução a ser executada e acionar as rotinas de tratamento adequadas. Portanto, foi preciso inicialmente acionar o método *fetch*

da classe *Memoria,* a fim de se obter a instrução referenciada pelo *Program Counter* armazenada na memória.

Além da lista de eventos e do *Program Counter,* o motor de eventos implementado possui um acumulador, responsável por armazenar resultados das operações.

A fim de se detectar o tipo de instrução adequadamente, é preciso realizar a leitura do primeiro byte da instrução em linguagem de máquina. Para cada valor encontrado, de 0 a F, o motor de eventos é encaminhado a uma rotina de tratamento específica. Na simulação desenvolvida em *Python,* as instruções hexadecimais estão expressas no formato '0x****', em que "*" são os parâmetros de cada instrução. Em geral, o primeiro "*" se refere ao tipo de instrução, e os demais indicam endereços de memória que serão utilizados para executar o comando. Sendo assim, no código em *Python* para se alcançar o tipo de instrução é necessário se obter seu terceiro algarismo. Isso é feito com `instrAtual[2]`. Já para obter-se os seguintes termos pode-se fazer `instrAtual[3], instrAtual[4] e instrAtual[5].`

A seguir, detalhes de cada uma das rotinas de tratamento:

1. Se o primeiro byte lido for '0', a rotina referente a instrução de *Jump* é acionada. Nela, o registrador *Program Counter* é desviado para o endereço de memória descrito nos bytes seguintes da instrução.

```
if (instrAtual[2] == '0'): # É um Jump
   endereco = int(instrAtual[3]+instrAtual[4]+instrAtual[5])
   self.programCounter = endereco
```

2. Se o primeiro byte lido for '1', a rotina referente a instrução de *Jump if zero* é acionada.

```
elif (str(instrAtual)[2] == '1'): # Jump if zero
   if (self.acumulador == 0):
       endereco = int(instrAtual[3]+instrAtual[4]+instrAtual[5])
```

```
            self.programCounter = endereco
        else:
            self.programCounter += 1
```

É importante notar nesse caso que, caso o acumulador seja diferente de zero, o *Program Counter* deve ser adicionado de 1, indo para a próxima instrução armazenada na memória principal.

3. Se o primeiro byte lido for ‘2’, a rotina referente a instrução de *Jump if negative* é acionada.

```
elif (str(instrAtual)[2] == '2'): # Jump if negative
    if (self.acumulador < 0):
        endereco = int(instrAtual[3]+instrAtual[4]+instrAtual[5])
        self.programCounter = endereco
    else:
        self.programCounter += 1
```

Semelhantemente ao anterior, caso o acumulador não seja menor que zero o contador de instruções é acrescido de 1.

4. Se o primeiro byte lido for ‘4’, a rotina referente a instrução de *Adição* é acionada.

```
elif (str(instrAtual)[2] == '4'): # add
    endereco = int(instrAtual[3]+instrAtual[4]+instrAtual[5])
    self.acumulador = self.acumulador + memoria.memoria[endereco]
    self.programCounter += 1
```

5. Se o primeiro byte lido for ‘5’, a rotina referente a instrução de *Subtração* é acionada.

```
elif (str(instrAtual)[2] == '5'): # sub
    endereco = int(instrAtual[3]+instrAtual[4]+instrAtual[5])
    self.acumulador = self.acumulador - memoria.memoria[endereco]
    self.programCounter += 1
```

6. Se o primeiro byte lido for ‘6’, a rotina referente a instrução de *Multiplicação* é acionada.

```
elif (str(instrAtual)[2] == '6'): # mult
    endereco = int(instrAtual[3]+instrAtual[4]+instrAtual[5])
    self.acumulador = self.acumulador * memoria.memoria[endereco]
    self.programCounter += 1
```

7. Se o primeiro byte lido for ‘7’, a rotina referente a instrução de *Divisão* é acionada.

```
elif (str(instrAtual)[2] == '7'): # div
    endereco = int(instrAtual[3]+instrAtual[4]+instrAtual[5])
    self.acumulador = self.acumulador / memoria.memoria[endereco]
    self.programCounter += 1
```

8. Se o primeiro byte lido for ‘8’, a rotina referente a instrução de *Load from Memory* é acionada. Nessa rotina o acumulador recebe o valor de um endereço específico de memória.

```
elif (str(instrAtual)[2] == '8'): # load from memory
    endereco = int(instrAtual[3]+instrAtual[4]+instrAtual[5])
    self.acumulador = memoria.memoria[endereco]
    self.programCounter += 1
```

9. Se o primeiro byte lido for ‘9’, a rotina referente a instrução de *Move to Memory* é acionada. Nessa rotina, um endereço de memória especificado na instrução recebe o valor armazenado no acumulador.

```
elif (str(instrAtual)[2] == '9'): #move to memory
    endereco = int(instrAtual[3]+instrAtual[4]+instrAtual[5])
    memoria.memoria[endereco] = self.acumulador
    self.programCounter += 1
```

10. Se o primeiro byte lido for ‘A’, a rotina referente a instrução de *Subroutine Call* é acionada.

```
elif (str(instrAtual)[2] == 'a'):   #SUBROUTINE CALL
   self.endRetorno = self.programCounter + 1
   self.programCounter = int(instrAtual[3]+instrAtual[4]+instrAtual[5])
```

Nessa rotina de tratamento é utilizado o registrador *endRetorno* que tem a função de armazenar o endereço seguinte ao da própria instrução, servindo para um possível e posterior retorno.

11. Se o primeiro byte lido for ‘B’, a rotina referente a instrução de *Operating System Call* é acionada.

```
elif (str(instrAtual)[2] == 'b'): #Fazer 16 chamadas do sistema operacional
   self.programCounter += 1
```

Como o desenvolvimento do sistema operacional não faz parte do escopo desse projeto, a implementação desse caso não foi realizada.

Para que o fluxo de eventos funcione como esperado é necessário que todas as instruções de um determinado programa dispostas na memória principal sejam lidas e executadas corretamente. Sendo assim, é necessário criar um *loop* dentro do motor de eventos, para que o fluxo se inicie e finalize nos momentos adequados. Para esse fim, foram utilizadas as pseudo-instruções ‘@’ e ‘#’. A primeira tem a função de definir o endereço inicial do programa e informar que as instruções em *assembly* serão expostas adiante. Já a segunda pseudo-instrução tem a função de definir o final do programa, indicando ao motor de eventos que sua tarefa foi finalizada.

Para essa finalidade, na codificação em *Python* foram incluídos os seguintes códigos:

```
while ((str(instrAtual)[0] != '#'):
                (...)
```

Tal código tem a função de realizar o *looping* do motor de eventos. Quando uma pseudo-instrução tip "#" é detectada, o programa sai desse *looping* e imprime a mensagem: `FIM DO PROGRAMA`

Além deste, o seguinte código foi utilizado:

```
if (str(instrAtual)[0]=='@'):
                (...)
```

Que tem a função de detectar o início do programa, e a partir dele começar a execução.

5. CÓDIGO MAIN

Após a construção de todos os componentes documentados acima, criou-se um código *main()* que importou-os todos e definiu o fluxo da simulação do sistema de programação.

Já que optou-se pelo desenvolvimento do programa *assembly* a ser simulado em um arquivo tipo *.txt,* o primeiro passo da *main()* é importar tal arquivo, que se encontra no mesmo diretório dos arquivos dos componentes. Isso foi feito por meio do comando: `codigoFonte = './codigoFonte.txt'.`

Em seguida, as instâncias dos componentes foram chamada por meio das instruções:

```
compMemoria = Memoria()
 motor = MotorDeEventos()
  assembler = Assembler()
```

Após isso, foi importado o parâmetro *Program Counter* do motor de eventos, e foram chamadas as funções “primeiro passo” e “segundo passo” do montador, oferecendo-se como parâmetro o código assembly, presente no arquivo `codigoFonte.txt`.

Em seguida, o *loader* foi invocado, com o fim de carregar na memória principal as instruções de máquina traduzidas pelo montador. Enfim, o método “fluxo de eventos” do motor de eventos foi invocado, oferecendo-se como parâmetro a lista de instruções da memória principal. O código que representa a *main()* está expresso a seguir:

```
def main():
    codigoFonte = './codigoFonte.txt'
    compMemoria = Memoria()
    motor = MotorDeEventos()
    assembler = Assembler()

    programCounter = motor.programCounter
    assembler.primeiroPasso(codigoFonte)
    instruCodMaq = assembler.segundoPasso(codigoFonte)
    compMemoria.loader(instruCodMaq)
    motor.fluxoEventos(compMemoria)

main()
```

6. TESTES E RESULTADOS

Inicialmente, a fim de se avaliar o funcionamento dos comandos de soma (‘+’), subtração (‘-’), multiplicação (‘*’) e divisão (‘/’) foram desenvolvidos os códigos dispostos a seguir. Para tal, optou-se por definir o dado do endereço de memória número 0x100 como 3, o endereço de memória 0x110 como 2, o endereço 0x120 como 10 e enfim, o endereço de memória 0x130 como 2. Estes serão os endereços dos operandos a serem utilizados em cada operação. O acumulador foi iniciado com valor 0.

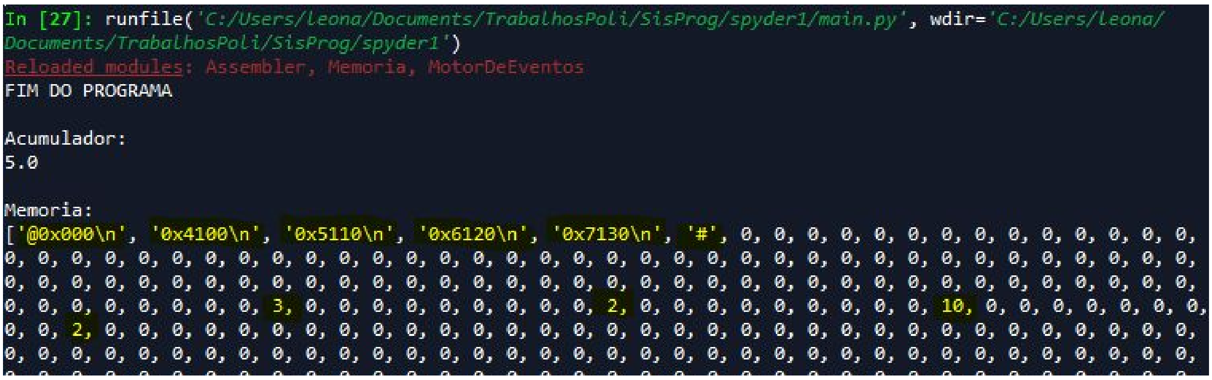
Neste primeiro exemplo de código fonte iniciou-se o programa na posição 0x000 de memória, somou-se o valor do acumulador com o valor armazenado em 0x100, subtraiu-se com o valor armazenado em 0x110, multiplicou-se pelo valor de 0x120 e enfim dividiu-se pelo valor de 0x130. Tal código está expresso a seguir:

```
@ 000
+ 100
- 110
* 120
/ 130
#
```

Para se validar a simulação, na *main(),* imprimiu-se o valor final armazenado pelo acumulador. Além disso, imprimiu-se a memória. Como a memória possui 4096 endereços e a maioria dos endereços não foi utilizada nesta simples simulação, exibe-se a seguir apenas as primeiras centenas de endereços da memória principal.

Com tal programa, o resultado esperado era que o acumulador, iniciando-se em zero se somasse com 3, do endereço 0x100, fosse subtraído por 2 do endereço 0x110, multiplicado por 10 do endereço 0x120, dividido por 2 do endereço 0x130, alcançando enfim o valor 5 (ou 5.0 para um número de tipo flutuante). Em respeito a memória do programa, esperava-se que a posição 0x000 houvesse o comando hexadecimal @0x000, indicando a pseudo-instrução de início de programa, na posição 0x001 a instrução 0x4100, referente à soma pedida. As posições 0x002, 0x003 e 0x004 deveriam armazenar respectivamente as instruções 0x5110, 0x6120 e 0x7130, referentes aos processos de subtração, multiplicação e divisão. A posição 0x005 enfim deveria apresentar o símbolo ‘#’ indicando o final do programa.

Todos os resultados ocorreram conforme o esperado e estão dispostos a seguir. Os valores relevantes estão destacados em amarelo.



Em seguida, testou-se as instruções *JP (Jump), JZ (Jump if zero) e JN (Jump if negative).* Para tal, desenvolveu-se o seguinte código:

```
 @ 000
JN 200
JP 210
+ 230
JZ 220
 #
```

Esperava-se que inicialmente, o *Program Counter* fosse levado até a instrução armazenada na posição 0x200 caso o acumulador fosse zero. Arbitrariamente, a fim de se avaliar o funcionamento das instruções, foi guardada na posição 0x200 a instrução 0x0002, ou seja *Jump* para a posição 0x002 da memória, para que o restante do código seja executado. Algo semelhante foi feito para a posição 0x210 que recebeu como dado a instrução *Jump* 0x0003 e para a posição 0x220 que recebeu 0x0005, para ler '#' e finalizar o programa. Para que todos os *jumps* fossem realizados, determinou-se o valor do acumulador inicial como -3 e o valor armazenado pelo endereço 0x230 como 3, a fim de que o *jump* em JZ 220 ocorresse.

Como esperado, o valor final do acumulador é 0. A seguir, segue os parâmetros de memória definidos na *main()* e a execução do código no terminal.

```
motor.acumulador = -3
compMemoria.memoria[200] = '0x0002'

compMemoria.memoria[210] = '0x0003'
compMemoria.memoria[220] = '0x0005'
```

```
compMemoria.memoria[230] = 3
```

```
FIM DO PROGRAMA

Acumulador:
0

Memoria:
['@0x000\n', '0x2200\n', '0x0210\n', '0x4230\n', '0x1220\n', '#', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
'0x0002', 0, 0, 0, 0, 0, 0, 0, 0, 0, '0x0003', 0, 0, 0, 0, 0, 0, 0, 0, 0, '0x0005', 0, 0, 0, 0, 0, 0, 0, 0,
0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

Além dessas, as instruções MM (*Move to Memory)* e LD (*Load from Memory)* também foram testadas com sucesso. Para tal, os códigos a seguir foram projetados:

@ 000
MM 100
#

Tal código carrega no endereço 0x100 o valor do acumulador. Arbitrariamente definimos o valor do acumulador como 5. O resultado está exibido a seguir:

```
In [4]: runfile('C:/Users/leona/Documents/TrabalhosPoli/SisProg/spyder1/main.py', wdir='C:/Users/leona/
Documents/TrabalhosPoli/SisProg/spyder1')
Reloaded modules: Assembler, Memoria, MotorDeEventos
FIM DO PROGRAMA

Acumulador:
5

Posição 0x100 da memória:
5

Memoria:
['@0x000\n', '0x9100\n', '#', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

Para testar a instrução LD foi desenvolvido o seguinte código:

@ 000
LD 100
#

Na posição 0x100 de memória armazenou-se 7. Sendo assim, esperava-se que o acumulador obtivesse o valor 7. Isso foi alcançado com sucesso e está exibido a seguir:

```
In [5]: runfile('C:/Users/leona/Documents/TrabalhosPoli/SisProg/spyder1/main.py', wdir='C:/Users/leona/
Documents/TrabalhosPoli/SisProg/spyder1')
Reloaded modules: Assembler, Memoria, MotorDeEventos
FIM DO PROGRAMA

Acumulador:
7

Posição 0x100 da memória:
7

Memoria:
['@0x000\n', '0x8100\n', '#', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, '0x0002', 0, 0, 0, 0, 0, 0,
0, 0, 0, '0x0003', 0, 0, 0, 0, 0, 0, 0, 0, 0, '0x0005', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

7. CONSIDERAÇÕES FINAIS

O sistema de programação desenvolvido nesse projeto desempenha satisfatoriamente as atividades requisitadas. O montador é capaz de traduzir instruções em *assembly* para comandos em código de máquina. O *Loader* é capaz de carregar as instruções de máquina na memória principal. A memória principal é capaz de armazenar os códigos de máquina e realizar a função de *fetch,* buscando uma instrução a partir de uma determinado valor do contador de instruções. E enfim, o motor de eventos é capaz de interpretar os comandos em código de máquina e criar um fluxo de execução do programa.