Nome: Lucas Sousa Nobre
Matrícula: 392030
Nome: Paulo Ricardo da Silva Lopes
Matrícula: 385173

Este trabalho consiste na implementação de um programa em linguagem Python que decodifica um código Thumb em seu respectivo mapa de memória. O decodificador recebe como entrada um arquivo de texto contendo o código a ser executado, em Thumb, e retorna seu respectivo mapa de memória, conforme tabela B.5 do livro **ARM System Developer's Guide**, 1ª edição. O arquivo de saide representa a memória de programa, onde cada linha corresponde a um endereço com alinhamento de 32 bits, no formato $< endereco >:< conteúdo >$.
Como no exemplo abaixo:

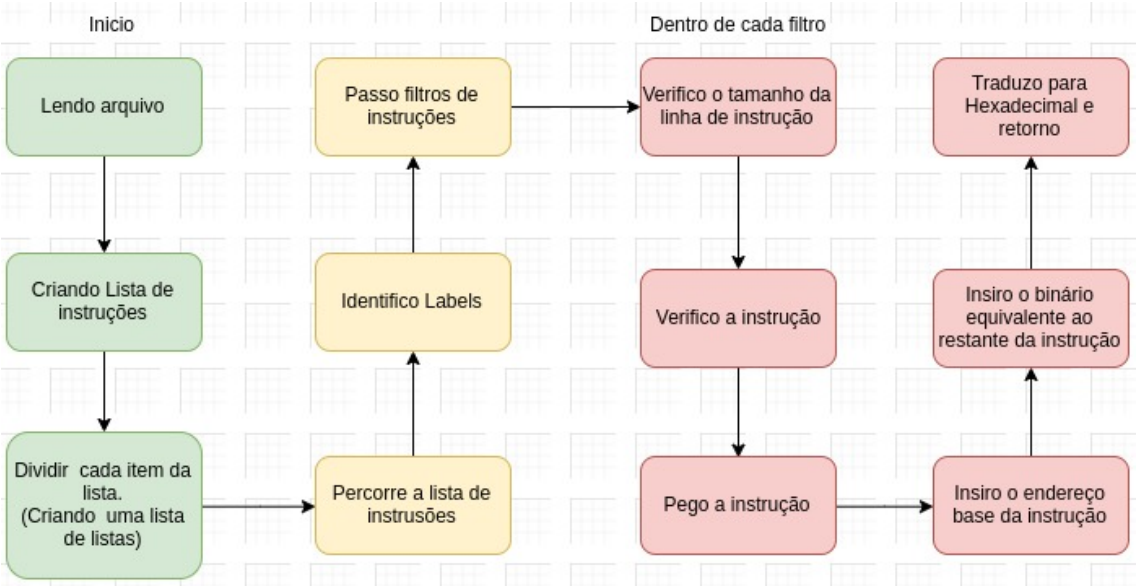
ENTRADA:

```
.thumb
    mov r0, #3
    mov r1, #5

main:
    add r2, r1, r0
    push {r2}
    swi #10
    pop {r2}
    sub r2, #1
    cmp r2, #0
    beq fim
    mov r0, r2
    b main

fim:
    b .
```

SAÍDA:

```
0: 21052003
4: b404180a
8: bc04df0a
c: 2a003a01
10: 1c10d001
14: e7fee7f6
```

1 Fluxograma de Ações



2 Preparando o Ambiente

O trabalho foi feito em cima do *Jupyter-lab* uma ferramenta de *data science*, utilizando a liguagem *Python*. Optamos por usar a linguagem Python devido a facilidade em manipular as strings, arrays, listas e matrizes, assim facilitando o trabalho das decodificação de instruções.

- Anaconda: https://www.anaconda.com/distribution/
- Python3
- Manual de Referência ARM
- Manual ARM7-TDMI-manual-pt3
- ler um arquivo 'entrada.s' e escreve em um 'memMapOut.out'

3 Estrutura do Trabalho

Dentro do trabalho temos um arquivo 'main' e os arquivos de funções/filtros que foram programados em módulos separados da 'main' que iram localizar e trazer uma instrução decodificada que será guardada em um vetor para mais tarde ser reorganizada.

4 Lendo o Arquivo

Primeiro lemos o arquivos com o código abaixo:

```
1  # Lendo arquivo
2  arquivo = open('entrada.txt','r')
3  # Vetor de palavras
4  linhas = []
5
```

```
    # Lendo linha a linha do arquivo
    for linha in arquivo:
        # retirando quebras de linhas
        linha = linha.strip()
        # Separo a linha em palavras
        linhas.append(linha.split())
    #fecho o arquivo
    arquivo.close()
```

Agora que lemos o arquivo temos uma lista1[lista2[]] onde lista1 é o conjunto de todas as linhas que foram lidas do arquivo, e lista2 que está dentro de lista1 é uma lista com todos os caracteres de uma linha que possuem espaço entre si, exemplo:

```
        #Linha que est  no arquivo vamos dizer que    a posi   o 0.
        add r0, r1
        # Mostra linha[0]:
        linhas[0] = ['add' 'r0,' 'r1']
        #Mostra linha[0][0]
        linha[0] = add
```

Com essa lista de listas que contêm as linhas, podemos pegar as informações das instruções de forma mais precisa, e então passar as linhas para as funções transformarem em hexadecimal.

5 Percorrendo o Vetor de Linhas

O algoritmo percorre o vetor de linhas para n-linhas vezes para poder identificar as labels e as instruções. Primeiro, quando o algoritmo percorre a lista de linhas, ele procura as labels que podem ter no código e as imprimi na tela. Segundo, aplicamos cada filtro/função a linha, se a linha tiver a instrução de alguma das funções ou filtros o filtro/função retorna o valor da instrução em hexadecimal, caso contrário retorna -1.

6 Filtros/Funções

Filtros/funções são funções criadas em Python que recebem uma linha do arquivo e dentro dessa linha temos uma instrução, os filtros/funções foram criados para verificar e tratar um conjunto específico de instruções cada, ou uma instrução específica, ou seja uma instrução 'A' só será tratada em um filtro 'A1', assim um filtro 'B1' não poderá tratar a instrução 'A'.

Os filtros possuem algumas características semelhantes ao tratar as instruções, para tratar uma linha de instrução, o filtro primeiro deve verificar se o tamanho da linha é compatível com o tamanho esperado, se o tamanho é compatível ele prossegue com o procedimento de identificação verificando se a instrução na primeira posição da linha faz parte do conjunto de instruções esperado na função. Se a instrução pertencer ao conjunto, fazemos *saida* receber o valor da *Base* do conjunto de operação, se a instrução espera um valor imediato, a função/filtro chamará uma função auxiliar chamada *findImmed* para encontrar o imediato na linha, *findImmed* serve tanto para identificar imediatos quanto para realizar as operações em que não se utiliza valores imediatos pois ela tem um retorno *booleano True/False*. após isso verificamos qual a instrução especifica está na linha e definimos seu *offset* junto com o *Base*. Após isso, prosseguimos pegando os valores binários dos registradores com *regSwitch* e os valores de imediato se a instrução utilizar valores imediato. Após temos pegado cada valor binário a função necessário a instrução nos os transformamos em hexadecimal.

Abaixo segue exemplo do filtro/função que procura pelas instruções de *mov*, *cmp*, *add* e *sub* *Rn*, *#immediate* :

- No primeiro if verificamos o tamanho e se a linha não está vazia.
- No segundo if verificamos se a instrução na linha faz parte do conjunto que é tratado no filtro/função.
- Depois verificamos se a algum valor imediato na instrução.
- No terceiro if dependendo do valor do retorno de findImmed fazemos o processo de construção do binário para o hexadecimal.
- Sabendo que a um imediato pegamos o valor do binário em Num.
- Pegamos o binário do registrador que será usado na instrução com o regSwitch.
- Converto o imediato em binário e o preencho com zeros caso ele não seja do tamanho esperado.
- Após concatenar todos os binários fornecidos transformamos em hexadecimal e retornamos.
- Quando qualquer if anterior não for atendido resultará na função retornando -1, assim indicando que a instrução não faz parte do conjunto.

```
# Encontra a intru  o: [ MOV Rn, #immediate ]
def find_Mov_Cmp_Add_Suv_immed(linha):

    #identificador find_Mov_Cmp_Add_Suv_immed
    if len(linha) != 0 and len(linha) == 3:
        if linha[0] == 'mov' or linha[0] == 'cmp' or linha[0] == 'add' or linha
            [0] == 'sub':

            # procura uma '#'
            status = findImmed(linha)

            # 01 - mov com imediato
            if status == True:
                # binario do ADD
                if linha[0] == 'mov':
                    saida = mov_op_im
                elif linha[0] == 'cmp':
                    saida = cmp_op_im
                elif linha[0] == 'add':
                    saida = add_op_im
                elif linha[0] == 'sub':
                    saida = sub_op_im

                # numero imediato a ser usado sem o '#'.
                num = linha[2][1:]

                # Pega o binario dos registradores.
                saida = regSwitch(saida,linha=linha[1][:-1])

```

```
29                  # binario do imediato
30                  num = str(bin(int(num)))[2:]
31
32                  # coloca os 0 faltantes do imediato
33                  while( len(num) < 8):
34                      num = '0'+num
35
36                  saida += num
37                  saida = hex(int(saida,2))
38
39                  if linha[0] == 'mov':
40                      print('   mov imed: ',linha)
41                  elif linha[0] == 'cmp':
42                      print('   cmp imed: ',linha)
43                  elif linha[0] == 'add':
44                      print('   add imed: ',linha)
45                  elif linha[0] == 'sub':
46                      print('   sub imed: ',linha)
47
48                  return saida
49
50      return -1
```

7 INFORMAÇÃO IMPORTANTE:

- Como há muitas instruções os filtros/funções se utilizam do esqueleto da função que foi descrita acima a $def\, find_Mov_Cmp_Add_Suv_imm ed(linha) : como modelo para a identificar a sua s próprias inst$
- regSwitch é uma função que retorna o binário de um registrador concatenado com a saida, ela serve para pegamos o registrador certo que uma instrução muda, e como a findImmed seus valores de parâmetros podem mudar de filtro/função para outra.
- exitem macros já definidas no inicio de cada código, como valores binários para os registradores e valores 'Base' para instruções.

8 Final

No final quando lemos todas as linhas da lista temos um vetor com todas as instruções codificadas em hexadecimal, e o salvamos em um arquivo .out.

9 Debug

O debug acontece no jupter-lab através de prints que foram colocados estrategicamente no nos filtros e podemos acompanhar linha a linha o debug.