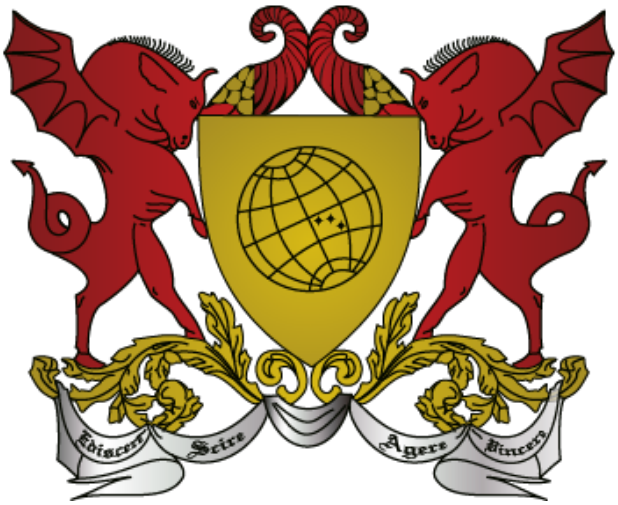


Universidade Federal de Viçosa - Campus Florestal

Ciência da Computação - 3º Período

Trabalho Prático de OC1 - 01

Montador MIPS (versão simplificada)

Bruno Marra - 3029
Gustavo Viegas - 3026

Florestal

2017

1. Introdução

O trabalho envolve a utilização dos conceitos de um montador MIPS, onde usando uma linguagem de programação qualquer podemos desenvolver um montador simplificado de forma a gerar os arquivos binários a partir de um conjunto de instruções do MIPS.

Para a implementação foi necessário o uso de uma linguagem de programação que facilitasse a leitura e escrita dos arquivos, na ocasião foi escolhida Python.

2. Metodologia

A partir disso, precisamos buscar quais são as instruções do MIPS bem como seu OP-code, e também o seu formato, que podem ser dos seguintes tipos:

1. R - op-code, registrador destino, registrador origem 1, registrador origem 2;
2. P - pseudo instrução, formato da pseudo instrução
3. I - op-code, registrador destino, registrador origem, constante

Após isso, tivemos que definir quais instruções seriam aceitas pelo nosso montador, e conseguimos interpretar as seguintes: **add, sub, and, clear, or, ori, move, nor, addi, andi, sll e srl.**

Ao todo, são 2 instruções do tipo P, 3 instruções do tipo I, e 7 instruções do tipo R. O padrão de instruções do tipo R foi o mais fácil de traduzir, visto que são comandos bastante específicos, e usamos apenas uma lógica para conseguirmos representar os registradores. Depois, vem as instruções de tipo P, que são um pouco mais complexas por permitirem a utilização de uma constante em seu último parâmetro, devendo ser tratada de forma diferente. E as mais complexas são as pseudo instruções, que precisam ser traduzidas para instruções do tipo R ou I conhecidas pelo Assembly do MIPS.

A partir disso, o que precisamos fazer foi separar os comandos, e descobrir qual tipo de instrução seria a partir de seu op-code. Um exemplo de arquivo .asm a ser lido é o arquivo exemplificado na Figura 1.

```
add $s0, $s1, $s2
or $s0, $s1, $s2
and $t0, $s1, $zero
andi $t0, $s1, 0
sll $s3, $s2, $t0
srl $s0, $s1, $t0
addi $s4 $s1 29
sub $s0, $s1, $s2
naoexiste $t1, 5, 2, $zero
nor $s1, $s2, $s0
ori $s0, $s1, 0
move $s0, $s1
add $s0, $s1, $zero
clear $t0
add $t0, $zero, $zero
```

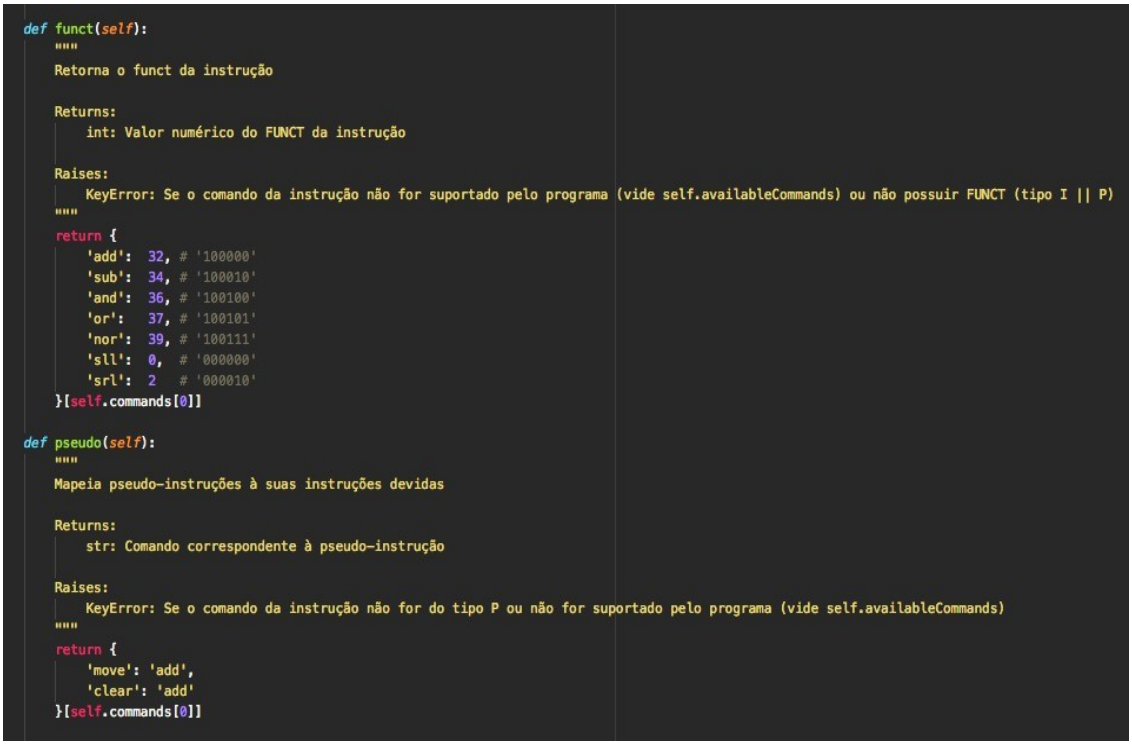
Figura 1 - Arquivo de entrada entrada.asm

Após isso basta converter cada pedaço das instruções em binário e, por último, escrever o arquivo de saída. Como o Python facilita bastante a leitura e escrita de arquivos, precisamos apenas de algumas linhas para fazer esse trabalho, conforme especificado na Figura 2.

```python
from instruction import instruction

def printHeader():
    print('*****************************************************')
    print('*   SUPER MONTADOR MIPS - UFV Florestal             *')
    print('*  por: Bruno Marra (3029), Gustavo Viegas (3026)   *')
    print('*****************************************************')

def proccessAssemblyFile(inputFile, outputFile):
    print('Lendo arquivo de entrada...')

    with open(inputFile) as input:
        lines = input.readlines()

    print('Arquivo lido! Processando entrada...')
    with open(outputFile, 'w') as output:
        output.truncate()
        for line in lines:
            inst = instruction(line.strip()) # Removendo whitespaces com strip
            if(inst.binary()):
                output.write(inst.binary() + '\n')

    print('Montagem feita com sucesso! O arquivo binário está disponivel em output.txt! \n')


if __name__ == "__main__":
    printHeader()
    proccessAssemblyFile('input.asm', 'output.txt')
```

Figura 2 - Função Principal do montador

Para realizar a tradução, temos que fazer algumas verificações e conversões dos termos para seu respectivo valor em decimal, e por último converter em binário. Para isso utilizamos as seguintes definições auxiliares mostradas na Figura 3.



```
def funct(self):
    """
    Retorna o funct da instrução

    Returns:
        int: Valor numérico do FUNCT da instrução

    Raises:
        KeyError: Se o comando da instrução não for suportado pelo programa (vide self.availableCommands) ou não possuir FUNCT (tipo I || P)
    """
    return {
        'add':  32, # '100000'
        'sub':  34, # '100010'
        'and':  36, # '100100'
        'or':   37, # '100101'
        'nor':  39, # '100111'
        'sll':  0,  # '000000'
        'srl':  2   # '000010'
    }[self.commands[0]]

def pseudo(self):
    """
    Mapeia pseudo-instruções à suas instruções devidas

    Returns:
        str: Comando correspondente à pseudo-instrução

    Raises:
        KeyError: Se o comando da instrução não for do tipo P ou não for suportado pelo programa (vide self.availableCommands)
    """
    return {
        'move': 'add',
        'clear': 'add'
    }[self.commands[0]]
```

Figura 3 - Definições auxiliares para montagem

Como podemos observar, as pseudo instruções são transformadas em outras instruções do assembly do MIPS, onde seu código é processado através da mesma lógica, porém o próprio montador que geramos trata essas instruções. A lógica utilizada para conversão de decimal para binário encontra-se no Anexo I.

3. Resultados

Fizemos ainda a utilização do complemento de dois para representação dos números binários, o que faz com que possamos representar tantos números positivos quanto negativos, aumentando a abrangência de nossas operações, conforme podemos analisar o código na Figura 4.

```python
def toBin(number, length):
    """
    Converte um inteiro para binário, com complemento de dois

    Args:
        number (int): Inteiro a ser convertido
        length (int): Número de bits do binário retornado

    Returns:
        str: String formatada em binário com tamanho fixo e complemento de dois
    """
    binaryFormat = '0' + str(length) + 'b'
    return format(number if number >= 0 else ((1 << length) + number), binaryFormat)
```

Figura 4 - Função para tratamento do Complemento de 2

Por fim, o montador gera os resultados das instruções que foram identificadas em um arquivo binário conforme a Figura 5. Dado que o importante é a saída correta para interpretação feita pela máquina a partir do arquivo gerado, o fato de instruções não reconhecidas serem ignoradas pelo montador é uma boa saída, visto que não causará um "crash" no montador caso passe alguma instrução num formato não aceitável.

```
1   00000010001100101000000000100000
2   00000010001100101000000000100101
3   00000010001000000100000000100100
4   00110010001010000000000000000000
5   00000000000100101001101000000000
6   00000000000100011000001000000010
7   00100010001101000000000000011101
8   00000010001100101000000000100010
9   00000010010100001000100000100111
10  00110100000100000000000000000000
11  00000010001000001000000000100000
12  00000010001000001000000000100000
13  00000000000000000100000000100000
14  00000000000000000100000000100000
15
```

Figura 5 - Arquivo .bin de saída

4. Conclusão

Esse trabalho mostra principalmente o quão complexo e difícil pode ser o trabalho de um compilador, como GCC (Linguagem C) por exemplo, visto que muito desse trabalho deve ser feito e avaliado da melhor maneira possível para não gerar nenhum tipo de interpretação inviável para o hardware.

Sendo assim, tivemos uma noção mais clara e objetiva das necessidades e cuidados de uma implementação, mesmo que simples, porém que consegue abranger um número concreto

de instruções do assemby, conseguindo até criar programas simples por meio do montador desenvolvido.

5. Referências

Tanenbaum, Andrew S. - **Organização Estruturada de Computadores - 5ª Ed**, Pearson - Addison Wesley; 2006.

Anexo I - Código fonte para conversão binária

```
  def binario(self):
      # confere se comando existe
      if(not self.comandos[0] in self.comands()):
          print("Comando '"+ self.comandos[0] +"' não encontrado")
          return 0
      # verifica se é uma das pseudo instruções e completa parâmetros
      if(self.tipo() == 'p'):
          if(self.comandos[0] == 'clear'):
              self.comandos.append('$zero')
          self.comandos[0] = self.pseudo()
          self.comandos.append('$zero')
      op = '{0:06b}'.format(self.op())
      # O rs é o terceiro nos comandos a seguir, e no SLL, SRL, ele é zero sempre
      if (self.comandos[0] in ['add', 'sub', 'and', 'or', 'nor', 'addi', 'andi']):
          rs = '{0:05b}'.format(binarioRegistrador(self.comandos[2]))
      else:
          rs = '{0:05b}'.format(0)
      # O rt é o quarto nos comandos a seguir
      if (self.comandos[0] in ['add', 'sub', 'and', 'or', 'nor']):
          rt = '{0:05b}'.format(binarioRegistrador(self.comandos[3]))
      elif (self.comandos[0] in ['sll', 'srl']):
          rt = '{0:05b}'.format(binarioRegistrador(self.comandos[2])) # É o terceiro nos
comandos a seguir
      else:
          rt = '{0:05b}'.format(binarioRegistrador(self.comandos[1])) # E o segundo
nos do tipo I
      if (self.tipo() == 'r' or self.tipo() == 'p'):
          # O rd é sempre o segundo, no tipo R
          rd = '{0:05b}'.format(binarioRegistrador(self.comandos[1]))
          # O shamt é o quarto nos SLL e SRL, e 0 nos restante
          if (self.comandos[0] in ['sll', 'srl']):
              shamt = '{0:05b}'.format(binarioRegistrador(self.comandos[3]))
          else:
              shamt = '{0:05b}'.format(0)
          funct = '{0:06b}'.format(self.funct())
          return op + rs + rt + rd + shamt + funct
      else:
          const = '{0:016b}'.format(int(self.comandos[3]))
          return op + rs + rt + const
```