Desenvolvimento de Aplicações em Assembly MIPS
Laboratório 1

Gustavo Joshua de Faria Costa de Azevedo
190014105
190014105@aluno.unb.br

Júlia Passos Pontes
190057904
pontesjulia70@gmail.com

Maria Clara Oliveira Fortes
190017503
maria.clara.of@hotmail.com

Dep. Ciência da Computação – Universidade de Brasília (UnB)
CIC0099 - Organização e Arquitetura de Computadores - Turma B
17 de Março de 2022

Abstract

This report refers to Laboratory 1 of practical activities related to Computer Organization and Architecture discipline. Based on theoretical knowledge about the Assembly language MIPS, a code was implemented that reads a ".asm" file in the input and generates two ".mif" files in the output, through the support of Python programming language. In this sense, it was possible to generate the files in hexadecimal in the output according to what was requested in the work script, consolidating the behavior's understanding of the MIPS Assembly language most diverse instructions.

Resumo

Este relatório se refere ao Laboratório 1 das atividades práticas da disciplina de Organização e Arquitetura de Computadores. Baseando-se nos conhecimentos teóricos acerca da linguagem Assembly MIPS, implementou-se um código que lê um arquivo ".asm"na entrada e gera dois arquivos ".mif"na saída, por meio do apoio da linguagem de programação Python. Nesse sentido, foi possível gerar os arquivos em hexadecimal na saída de acordo com o que foi solicitado no roteiro do trabalho, consolidando-se a compreensão acerca do funcionamento das mais diversas intruções da linguagem Assembly MIPS.

1. Introdução

A fim de se controlar o hardware de uma máquina é necessário saber a sua linguagem, na qual as palavras são as instruções e o vocabulário é o conjunto de instruções [1]. Assim sendo, uma arquitetura é um conjunto de instruções que faz a interface entre o hardware e o software de baixo nível de abstração. Nesse sentido, a seguir serão explicados conceitos que englobam a arquitetura MIPS e a análise de códigos de baixo nível.

1.1. Assembly

Assembly é uma representação simbólica das instruções de máquina [1]. Assim sendo, essa linguagem fornece uma linguagem de baixo nível mais humana e conveniente do que a linguagem de máquina puramente. Nesse sentido, programar em Assembly acaba sendo muito mais simples do que programar utilizando apenas números binários, já que pode-se escrever o código de máquina em formato de texto. Contudo, o Assembly ainda está longe de ser a linguagem mais amigável e utilizada pelos programadores, pois ela obriga que seja escrita uma linha para cada instrução que a máquina seguirá [1].

1.2. MIPS

MIPS, nesse caso, é uma arquitetura do conjunto de instruções desenvolvida pela empresa MIPS Computer Systems, atualmente denominada de MIPS Technologies [1]. Dessa forma, na aritmética da arquitetura MIPS, a instrução é capaz de realizar apenas uma operação por vez e em todos os casos necessita de três variáveis, como na instrução "add a, b, c", por exemplo, na qual o computador soma "b"com "c"e atribui o valor obtido à variável "a"[1]. Vale ressaltar, ainda, que o tamanho padrão de um registrador nesse tipo de

arquitetura é de 32 bits e esse grupo de bits recebe o nome de word [1].

1.3. Tempo de Execução

O tempo de execução, também chamado de tempo de resposta, representa o tempo total que um computador exige para a execução de determinada tarefa [1]. Desse modo, ele é a soma do tempo de latência mais o tempo em que a tarefa de fato é executada. Assim, a latência relaciona-se ao tempo que leva desde o início de um processo de execução até esse processo, de fato, começar a ter um efeito que possa ser percebido.

1.4. Desempenho

O desempenho pode ser interpretado tomando-se como base diversos fatores a serem analisados, contudo, nesse caso, será relacionado ao tempo de execução de determinada tarefa [1]. À vista disso, tem-se que para maximizar o desempenho é necessário diminuir o tempo de execução, conforme evidencia a **Equação 1**.

$$Desempenho_X = \frac{1}{Tempo\ de\ execução_X} \qquad (1)$$

Tendo em vista o que foi dito, quando discute-se sobre o desenvolvimento de determinado projeto, geralmente o que deseja-se é comparar o desempenho de dois computadores diferentes [1] e, para isso, pode-se utilizar a **Equação 2**, que faz essa relação inversamente proporcional entre desempenho e tempo de execução de máquinas diferentes.

$$\frac{Desempenho_X}{Desempenho_Y} = \frac{Tempo\ de\ execução_Y}{Tempo\ de\ execução_X} = n \qquad (2)$$

Assim, pode-se dizer que ”X é n vezes mais rápido que Y”[1].

1.5. Objetivos

O presente trabalho tem por objetivo desenvolver habilidades e conhecimentos que envolvem a linguagem Assembly MIPS e suas diversas instruções, como lw, sw, j e beq, por exemplo. Assim sendo, a partir da implementação de um programa em Python e da análise dos resultados obtidos, busca-se entender o funcionamento das instruções do Assembly MIPS e adaptar-se ainda mais a essa linguagem, consolidando, assim, o estudo teórico que engloba grande parte da disciplina de Organização e Arquitetura de Computadores.

2. Materiais

Neste experimento foram utilizados os seguintes materiais e equipamentos:

- Software *MARS*
- Editor de texto *Thonny*
- Linguagem de programação *Python*

3. Métodos

Este trabalho foi desenvolvido tomando-se como base a criação de um programa que lê um arquivo de extensão ”.asm”na entrada e devolve na saída dois arquivos ”.mif”diferentes, de acordo com o código da **Figura 1**. À vista disso, para a sua implementação, seguiu-se as seguintes etapas.

3.1. Interface com o usuário

Primeiramente, pede-se que o usuário informe o caminho do arquivo de entrada ”.asm”, conforme evidencia a **Figura 2**.

3.2. Leitura e separação do arquivo de entrada

Em seguida, após o usuário informar o caminho do arquivo de entrada, o programa faz a leitura desse arquivo e separa cada linha do arquivo em uma lista de strings e inteiros, e todas essas listas (cada uma representando uma linha) são colocadas dentro de uma outra lista. Para a parte de .text, isso ocorre de tal maneira que se após a primeira palavra lida tiver o símbolo ”:”como último caractere, o programa entende que essa palavra é uma label e não uma instrução e, portanto, coloca essa label no final da lista que corresponde à linha dessa label e a instrução que a acompanha fica no início dessa lista, de acordo com o código da **Figura 3**.

Além disso, nessa parte, ainda, retirou-se as vírgulas e separou-se os saltos, que ocorrem quando o nome do registrador vem entre parênteses, como no caso das instruções lw e sw, por exemplo. Assim, essa parte pode ser verificada na **Figura 4**. Isso foi feito para isolar as componentes das instruções na lista, sejam eles o mnemônico da instrução, um registrador ou um imediato.

3.3. Manipulações dos dados .data

Durante a execução do programa, após a inserção da documentação ”.asm”ser inserida é chamada uma função nomeada ”escreve_memoria_arquivo”a qual recebe todas as linhas refentes ao ”.data”do arquivo e adiciona em uma lista, que será passada como argumento para a função ”tipo_data”.

A função ”tipo_data”é responsável por verificar qual o tipo de dado é inserido em cada linha para que consiga fazer as conversões de valores necessários para que se insira da forma correta os valores na memória. Nesta função serão chamadas algumas funções de conversão as quais são: ”conv_comp2”, ”conv_bin”, ”conv_hex”, ”conv_bintohex”,

”bin_float_ieee_754”e ”bin_double_ieee_754”, das quais as quatro primeiras podem ter as suas funcionalidades descritas na **Tabela 1**, e as duas últimas fazem a conversão do número float ou double recebido em binário segundo a formatação da norma IEEE754.

Após passar pela função ”tipo_data”terá uma lista de hexadecimais de cada dado recebido, mas este não estará no formato de 8 hexadecimais, para isso passará pela função ”ajuste_memoria”, que fará a formatação dos valores que serão retornados para a função ”escreve_memoria_arquivo”, com a finalidade de escrever o arquivo referente ao ”data”.

3.4. Escrita dos dados .data em um arquivo

Após a execução das funções necessárias para o ajuste dos dados a serem inseridos no arquivo, retorna-se para a função ”escreve_memoria_arquivo”, a qual pega a lista ajustada dos dados e insere os dados um a um, em cada linha, conforme o endereço da memória.

3.5. Manipulações dos dados .text

Nesta parte, considera-se uma lista grande de funções criadas para ser possível a manipulação dos dados da seção ”.text”do arquivo de entrada. Nesse sentido, pode-se observar nas **Tabelas 1 e 2** a listagem de todas as funções utilizadas para o manuseio desses dados.

À vista disso, é nessa etapa em que ocorre, de fato, a transformação da linguagem Assembly MIPS para a linguagem de máquina, a partir das conversões vistas nas tabelas comentadas.

Além disso, vale ressaltar que a função ”opcode”foi implementada de tal forma que ela recebe o nome da instrução e retorna os bits que diferenciam as instruções similares, sejam eles os bits do opcode (diferenciam um lw de um sw, por exemplo) ou os do funct (diferenciam um add de um sub, por exemplo).

Para as instruções que lidam com um valor imediato como li, addiu e xori, por exemplo, quando esses valores de imediato são muito grandes (maiores que 0x10000), essas instruções acabam sendo executadas por mais de uma instrução, portanto se fez necessário o desenvolvimento de várias funções capazes de identificar quando esses casos ocorrem e gerar o código binário das respectivas instruções.

3.6. Escrita dos dados .text em um arquivo

Posteriormente, depois de feitas as manipulações e as conversões necessárias dos dados da seção ”.text”, a função ”escreve_instrucoes_arquivo”recebe a lista dos dados a serem escritos no arquivo de saída e os insere na ordem correta, conforme expõe a **Figura 5**.

4. Resultados

Tendo em vista o supracitado, ao utilizar um arquivo ”.asm”na entrada do programa, surgem dois novos arquivos ”.mif”na pasta em que está localizado o programa principal. Nesse sentido, se utilizarmos um arquivo ”.asm”(example_saida.asm) com o conteúdo conforme evidencia a **Figura 6**, são gerados os arquivos expostos nas **Figuras 7 e 8**, as quais representam o arquivo data ”.mif”e o arquivo text ”.mif”, respectivamente. E esse resultado bate com o esperado conforme mostra a **Tabela 3**.

Além disso, se colocarmos na entrada do programa o arquivo ”example_saidateste1.asm”com o conteúdo de acordo com as **Figuras 9, 10 e 11**, que juntas formam um mesmo arquivo, obtém-se dois arquivos na saída com os endereços, códigos e mnemônicos correspondentes de cada instrução e com a formatação certa. Esse arquivo ”.asm”foi desenvolvido apenas para testar o funcionamento do programa e se todas as instruções pedidas foram atendidas, portanto não necessariamente o código MIPS desse arquivo é funcional.

Com a execução desse arquivo, foi possível notar que para instruções que envolvem imediatos muito grandes, como é o caso da instrução ”xori $t6, $t5, -65500”da linha 35, são necessárias mais de uma instrução para executá-las. Nesse caso específico, para esse ”xori”ser executado é necessário um ”lui”e um ”ori”antes, apenas para ser possível colocar o valor do imediato completo em um registrador ($at), depois fazer um xor entre $at e $t5 e, por fim, armazenar o resultado em $t6.

Para as instruções de salto, como por exemplo é o ”bgezal $t0 LABEL”da linha 33, é necessário saber exatamente o endereço das instruções da memória para conseguir calcular de maneira correta o valor do salto, por isso se fez necessário o desenvolvimento de uma função que corrige o endereço das instruções de acordo com a quantidade de instruções que cada instrução demanda, como por exemplo o ”xori”, discutido anteriormente, que necessita de 3 instruções para ser executado, portanto essa instrução ocupa o tamanho de 3 instruções na memória.

Ademais, vale ressaltar que o tempo de resposta do programa foi bom, contudo ele poderia ser ainda melhor se a leitura dos dados dos arquivos fosse feita apenas uma vez. Desse modo, um ponto que poderia melhorar ainda mais o desempenho do programa seria, justamente, realizando a leitura apenas uma vez.

5. Discussão e Conclusões

Este trabalho teve como objetivo apresentar o funcionamento de um processador MIPS e suas diversas instruções, assim como verificar como estas se transformam de Assembly para a linguagem de máquina compreendida pelo computador, em binário (no caso, em hexadecimal). Dessa forma, conclui-se, a partir de tudo o que foi exposto até aqui, que obteve-se resultados satisfatórios e condizentes com o esperado, visto que foi possível confeccionar um programa que após receber o caminho para um arquivo de entrada ”.asm”, de fato, gera dois arquivos ”.mif”com

as especificações corretas. Além da consolidação do conhecimento em torno da utilização da linguagem Assembly MIPS e da manipulação de suas instruções para diversas aplicações, também desenvolveu-se os aprendizados em relação à linguagem de programação Python.

Referências

[1] D. A. Patterson and J. L. Hennessy. *Organização e Projeto de Computadores: a interface hardware/software*. Elsevier, 5th edition, 2017.

6. Figuras e Tabelas

```
arquivo = input("Insira o caminho do arquivo .asm: ")
nome_arquivo = arquivo[0:-4]
cont_barra = -1
for i in range(len(arquivo)):
    if(arquivo[i]=='/' or arquivo[i]=="\\"):
        cont_barra = i

nome_arquivo = nome_arquivo[cont_barra+1:]
file_leitura = open(arquivo, 'r')
file_text = open(nome_arquivo+'_text.mif', 'w')
file_data = open(nome_arquivo+'_data.mif', 'w')

escreve_memoria_arquivo(file_leitura, file_data)
escreve_instrucoes_arquivo(file_leitura, file_text)

file_leitura.close()
file_text.close()
file_data.close()
```

Figura 1. Visão geral do funcionamento do programa.

```
File  Edit  View  Search  Terminal  Help
julia@julia-note:~/Documents/Lab1_OAC$ python3 Lab1.py
Insira o caminho do arquivo .asm:
```

Figura 2. Representação da interface com o usuário no terminal.

```
    linha.append(numlinha)
    linha.append(aux2)
    linha.append(endereco)
    linha.append([0,''])
    if linha[0][-1] == ':' :      # Label
        linha[-1].append(1)
        linha[-1].append(linha[0][:-1])
        linha[-1].pop(0)
        linha[-1].pop(0)
        linha.pop(0)
```

Figura 3. Separação de cada linha em uma lista dentro de outra lista.

```
while not (linha == '' or linha == '.data\n'): #LEITURA DE TODAS AS LINHAS

    if linha == '\n':
        numlinha+=1
        linha = file_leitura.readline()
    else:
        aux2 = linha
        if aux2[-1] == '\n':
            aux2 = aux2[:-1]
        linha = linha.split()
        for i in range(len(linha)):
            if linha[i][-1] == ',':         # Retirar as vírgulas
                linha[i] = linha[i][:-1]
            if linha[i][-1] == ')':         # Separar os saltos
                linha[i] = linha[i][:-1]
                aux = linha[i]
                aux = aux.split('(')
                linha.pop()
                linha.append(aux[0])
                linha.append(aux[1])

        linha.append(numlinha)
        linha.append(aux2)
        linha.append(endereco)
        linha.append([0,''])
```

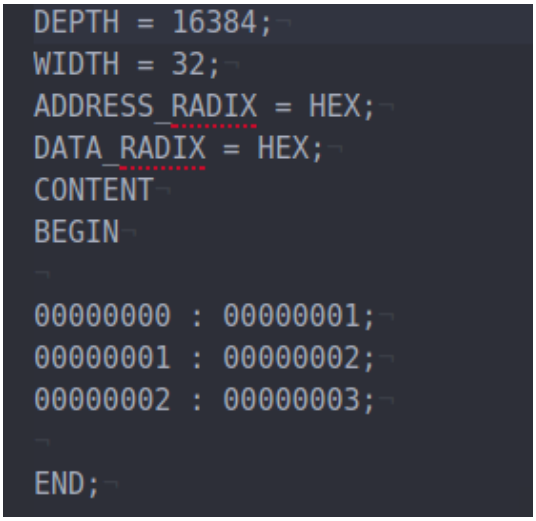
Figura 4. Retirada das vírgulas e separação dos saltos.

```
for i in range(len(lista_de_linhas)):
    diferencia_instrucao(lista_de_linhas, i)

file_text.write('DEPTH = 4096;\n')
file_text.write('WIDTH = 32;\n')
file_text.write('ADDRESS_RADIX = HEX;\n')
file_text.write('DATA_RADIX = HEX;\n')
file_text.write('CONTENT\n')
file_text.write('BEGIN\n\n')

adress = 0
adresshex = hex(adress)[2:]
while len(adresshex) < 8:
    adresshex = '0' + adresshex
for i in range(len(lista_de_linhas)):
    for j in range(len(lista_de_linhas[i][-1][-1])):
        file_text.write(adresshex + ' : ' +lista_de_linhas[i][-1][-1][j]+';')
        if j == 0:
            file_text.write('  % '+ str(lista_de_linhas[i][-4])+ ': '+lista_de_linhas[i][-3]+' %')
        adress+=1
        adresshex = hex(adress)[2:]
        while len(adresshex) < 8:
            adresshex = '0' + adresshex
        file_text.write('\n')

file_text.write('\nEND;\n')
```

Figura 5. Escrita dos dados no arquivo ".text"de saída.

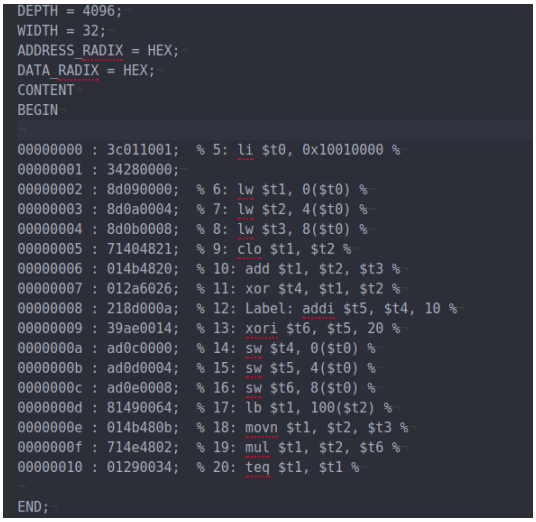
```
.data
a: .word 1, 2, 3

.text
li $t0, 0x10010000
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t3, 8($t0)
clo $t1, $t2
add $t1, $t2, $t3
xor $t4, $t1, $t2
Label: addi $t5, $t4, 10
xori $t6, $t5, 20
sw $t4, 0($t0)
sw $t5, 4($t0)
sw $t6, 8($t0)
lb $t1, 100($t2)
movn $t1, $t2, $t3
mul $t1, $t2, $t6
teq $t1, $t1
```

Figura 6. Conteúdo do arquivo example_asm.



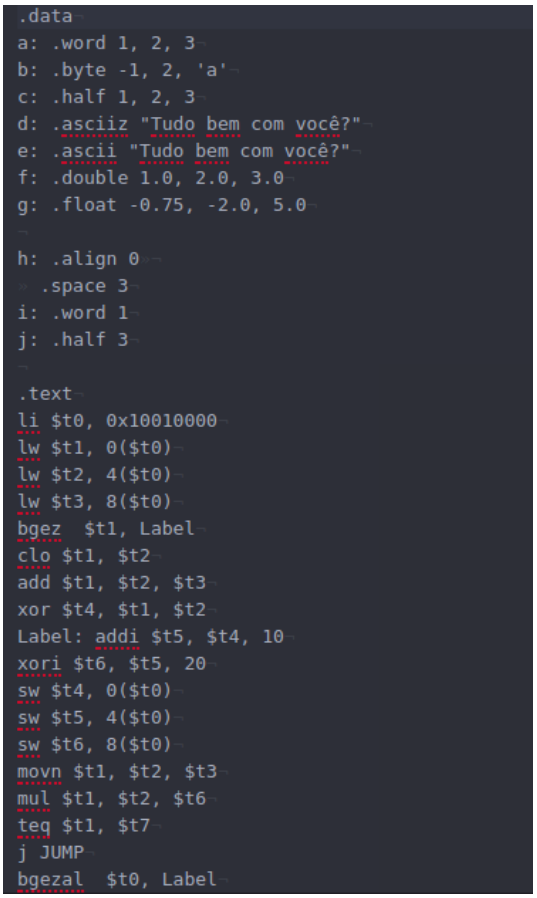
```
DEPTH = 16384;
WIDTH = 32;
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;
CONTENT
BEGIN

00000000 : 00000001;
00000001 : 00000002;
00000002 : 00000003;

END;
```

Figura 7. Conteúdo gerado do arquivo example_saida_data.mif.



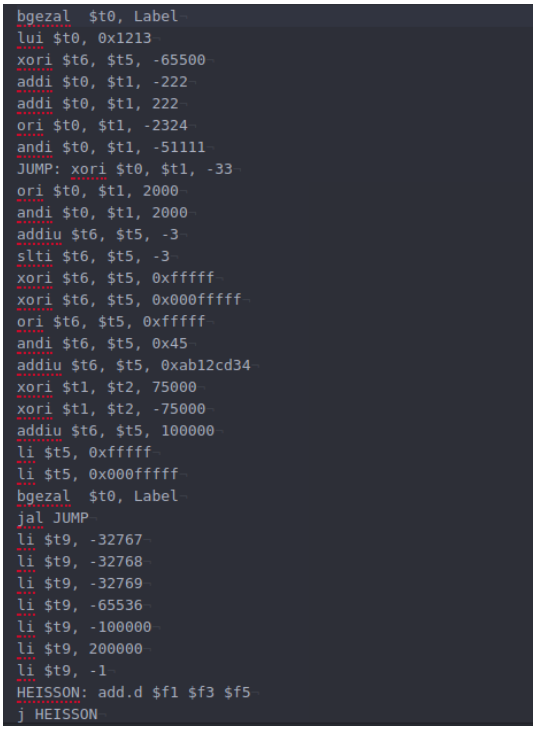
```
DEPTH = 4096;
WIDTH = 32;
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;
CONTENT
BEGIN

00000000 : 3c011001;  % 5: li $t0, 0x10010000 %
00000001 : 34280000;
00000002 : 8d090000;  % 6: lw $t1, 0($t0) %
00000003 : 8d0a0004;  % 7: lw $t2, 4($t0) %
00000004 : 8d0b0008;  % 8: lw $t3, 8($t0) %
00000005 : 71404821;  % 9: clo $t1, $t2 %
00000006 : 014b4820;  % 10: add $t1, $t2, $t3 %
00000007 : 012a6026;  % 11: xor $t4, $t1, $t2 %
00000008 : 218d000a;  % 12: Label: addi $t5, $t4, 10 %
00000009 : 39ae0014;  % 13: xori $t6, $t5, 20 %
0000000a : ad0c0000;  % 14: sw $t4, 0($t0) %
0000000b : ad0d0004;  % 15: sw $t5, 4($t0) %
0000000c : ad0e0008;  % 16: sw $t6, 8($t0) %
0000000d : 81490064;  % 17: lb $t1, 100($t2) %
0000000e : 014b480b;  % 18: movn $t1, $t2, $t3 %
0000000f : 714e4802;  % 19: mul $t1, $t2, $t6 %
00000010 : 01290034;  % 20: teq $t1, $t1 %

END;
```

Figura 8. Conteúdo gerado do arquivo example_saida_text.mif.



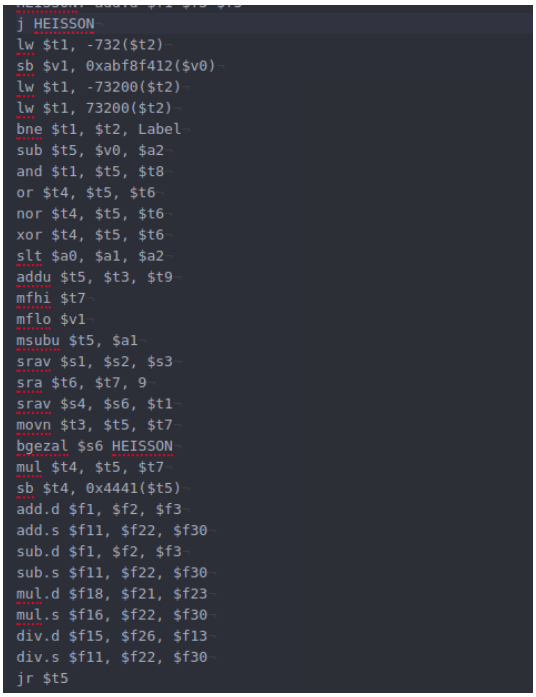
```
.data
a: .word 1, 2, 3
b: .byte -1, 2, 'a'
c: .half 1, 2, 3
d: .asciiz "Tudo bem com você?"
e: .ascii "Tudo bem com você?"
f: .double 1.0, 2.0, 3.0
g: .float -0.75, -2.0, 5.0

h: .align 0
   .space 3
i: .word 1
j: .half 3

.text
li $t0, 0x10010000
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t3, 8($t0)
bgez  $t1, Label
clo $t1, $t2
add $t1, $t2, $t3
xor $t4, $t1, $t2
Label: addi $t5, $t4, 10
xori $t6, $t5, 20
sw $t4, 0($t0)
sw $t5, 4($t0)
sw $t6, 8($t0)
movn $t1, $t2, $t3
mul $t1, $t2, $t6
teq $t1, $t7
j JUMP
bgezal  $t0, Label
```

Figura 9. Conteúdo do arquivo example_saidateste1.asm.



```
bgezal  $t0, Label
lui $t0, 0x1213
xori $t6, $t5, -65500
addi $t0, $t1, -222
addi $t0, $t1, 222
ori $t0, $t1, -2324
andi $t0, $t1, -51111
JUMP: xori $t0, $t1, -33
ori $t0, $t1, 2000
andi $t0, $t1, 2000
addiu $t6, $t5, -3
slti $t6, $t5, -3
xori $t6, $t5, 0xfffff
xori $t6, $t5, 0x000fffff
ori $t6, $t5, 0xfffff
andi $t6, $t5, 0x45
addiu $t6, $t5, 0xab12cd34
xori $t1, $t2, 75000
xori $t1, $t2, -75000
addiu $t6, $t5, 100000
li $t5, 0xfffff
li $t5, 0x000fffff
bgezal  $t0, Label
jal JUMP
li $t9, -32767
li $t9, -32768
li $t9, -32769
li $t9, -65536
li $t9, -100000
li $t9, 200000
li $t9, -1
HEISSON: add.d $f1 $f3 $f5
j HEISSON
```

Figura 10. Continuação do conteúdo do arquivo example_saidateste1.asm.



```
j HEISSON
lw $t1, -732($t2)
sb $v1, 0xabf8f412($v0)
lw $t1, -73200($t2)
lw $t1, 73200($t2)
bne $t1, $t2, Label
sub $t5, $v0, $a2
and $t1, $t5, $t8
or $t4, $t5, $t6
nor $t4, $t5, $t6
xor $t4, $t5, $t6
slt $a0, $a1, $a2
addu $t5, $t3, $t9
mfhi $t7
mflo $v1
msubu $t5, $a1
srav $s1, $s2, $s3
sra $t6, $t7, 9
srav $s4, $s6, $t1
movn $t3, $t5, $t7
bgezal $s6 HEISSON
mul $t4, $t5, $t7
sb $t4, 0x4441($t5)
add.d $f1, $f2, $f3
add.s $f11, $f22, $f30
sub.d $f1, $f2, $f3
sub.s $f11, $f22, $f30
mul.d $f18, $f21, $f23
mul.s $f16, $f22, $f30
div.d $f15, $f26, $f13
div.s $f11, $f22, $f30
jr $t5
```

Figura 11. Fim do conteúdo do arquivo example_saidateste1.asm.

Função	Funcionalidade
conv_comp2	Recebe um número inteiro e retorna esse número transformado em complemento de 2
conv_bin	Recebe um número hexadecimal ou inteiro e retorna um número binário
conv_hex	Recebe um número decimal em formato de string e retorna esse número convertido em hexadecimal
conv_bintohex	Recebe um número binário e retorna esse número convertido em hexadecimal
rbin	Recebe o nome dos registradores e retorna os 5 bits binários que representam esse registrador
checar_subinstrucao	Checa se a instrução é nativa ou pseudo-instrução e retorna o endereço ajustado
opcode	Recebe o nome da instrução e retorna os bits do opcode ou do funct
li_addiu	Utilizada para gerar os códigos binários de instruções que são compostas por mais de uma instrução
li_luiori	Utilizada para gerar os códigos binários de instruções que são compostas por mais de uma instrução

Tabela 1. Funções utilizadas na manipulação dos dados da seção .text do arquivo de entrada.

Função	Funcionalidade
andiorixori	Utilizada para gerar os códigos binários de instruções que são compostas por mais de uma instrução
tipoi	Utilizada para gerar os códigos binários de instruções que são compostas por mais de uma instrução
loadstore	Utilizada para gerar os códigos binários de instruções que são compostas por mais de uma instrução
diferencia_instrucao	Recebe uma lista contendo as informações de todas as linhas do .text e retorna uma lista que contém os 32 bits de cada parte de uma instrução

Tabela 2. Mais funções utilizadas na manipulação dos dados da seção .text do arquivo de entrada.

Endereço	Código	Mnemônico
00000000	3c011001	li
00000001	34280000	
00000002	8d090000	lw
00000003	8d0a0004	lw
00000004	8d0b0008	lw
00000005	71404821	clo
00000006	014b4820	add
00000007	012a6026	xor
00000008	218d000a	addi
00000009	39ae0014	xori
0000000a	ad0c0000	sw
0000000b	ad0d0004	sw
0000000c	ad0e0008	sw
0000000d	81490064	lb
0000000e	014b480b	movn
0000000f	714e4802	mul
00000010	01290034	teq

Tabela 3. Resultados esperados de acordo com o arquivo de entrada example_saida.asm