Trabalho 1 - Assembly do RISC-V

Guilherme de Abreu Lima Buitrago Miranda - 2018054788

1 Introdução

O trabalho 1 da disciplina de Organização de Computadores I tem como objetivo aprender e praticar os primeiros conceitos vistos em sala de aula, assim como a codificação em Assembly para o conjunto de instruções RISC-V.

Dessa forma, três programas foram criados. O primeiro verifica se um dado número é par ou ímpar e o segundo calcula o fatorial de um dado número n. Por fim, o terceiro recebe dois vetores e checa se um é permutação do outro.

Todo o código desenvolvido encontra-se abaixo e também no github [1] do autor.

2 Problemas

2.1 Problema 1 - Evens and Odds

```
.data
input: .word -43 # Armazena na memória

.text
lw a0, input # Carrega da memória para o registrador

andi x10, a0, 1 # And com 1 faz o mesmo que pegar o último bit do número
#(em binário), que diz se é par ou ímpar.
addi a1, x10, 0 # Coloca o resultado no registrador de saída
```

[1] https://github.com/guilhermealbm/CC-UFMG

```
addi a0, x0, 1 # Fazendo o ecall escrever a resposta no console
ecall                  # Escreve a resposta
```

O programa acima recebe um número e retorna 0 se o número for par ou 1 se o número for ímpar. Para isso, é feito um "andi"com o 1, e, dessa operação lógica deriva-se a resposta desejada. A solução é baseada no fato de que, na representação binária, os números ímpares são sempre terminados em 1 e os pares terminados em 0, independente do sinal.

2.2 Problema 2 - Factorial

```
.data
n: .word 5

.text
lw x5, n
jal x1, fat

addi a1, x10, 0 #Coloca o resultado no registrador de saída
addi a0, x0, 1
ecall # Escreve o Retorno

addi a0, x0, 10
ecall # exit

fat:
    addi sp,sp,-8 #Anda 2 words na memória
    sw   x1,4(sp) #Armazena x1 em memória
    sw   x5,0(sp) #Armazena x5 em memória
    addi x5,x5,-1 #Decrementa x5
    bge  x5,x0,L1
    addi x10,x0,1 #Incrementa x10
    addi sp,sp,8 #Anda com o stack pointer 2 words
    jalr x0,0(x1) # Retorno

L1:
    addi x10,x10,-1 #Decrementa x10
    jal  x1,fat
```

```
    addi x6,x10,0
    lw   x10,0(sp) #Carrega da memória para x10
    lw   x1,4(sp) #Carrega da memória para x1
    addi sp,sp,8 #Incrementa o stack pointer
    mul  x10,x10,x6 #Faz a operação básica do fatorial (n * n-1)
    jalr x0,0(x1) #Retorno
```

O programa acima recebe um inteiro e calcula o fatorial deste. O processamento é feito utilizando a manipulação com o stack pointer. A função "fat"primeiramente percorre do número n a ser calculado até o 0, guardando os elementos e o endereço em memória. Posteriormente, é chamada a função "L1", que é responsável por fazer as múltiplicações de 1 até n, de forma a voltar à "fat"quando o processamento for finalizado. Por fim, o programa exibe no console o resultado do fatorial.

2.3 Problema 3 - Permutation

```
.data #Tamanho dos arrays
array1_size: .word 4
array2_size: .word 4

#Preenchendo os arrays com elementos
array1:
    .word 1, 2, 3, 4
array2:
    .word 4, 2, 1, 3

.text
    la x17, array1_size
    lw x13, 0(x17) # Carregando primeiro tamanho
    la x17, array2_size
    lw x14, 0(x17) # Carregando segundo tamanho
    bne x13, x14, different_sizes

    add x12, x0, x0 #Variável auxiliar (contador 0 .. array1_size)
    addi s4, x0, 1 #S4 sempre = 1

    la t0 array1
```

```
add x15, x14, x0 #Mantendo o tamanho dos arrays em x15
addi x14, x0, -4 #x14 = tamanho da palavra
mul x13, x13, x14 #Tamanho do vetor em bytes
add sp, sp, x13 #Decrementando sp

search_element_array1:
    la t2 array2 #Começo do array2
    add x11, x0, x0 #Variável auxiliar (contador 0 .. array2_size)
    lw t1, 0(t0) # Lê novo valor no array1
    addi x12, x12, 1 #Incrementando contador
    blt x15, x12, exit_success #Se verdadeiro, é permutação
    sub t0, t0, x14 # Avança uma posição no vetor
    jal search_element_array2

    search_element_array2:
    lw t3, 0(t2) # Lê novo valor
    sub t2, t2, x14 # Avança uma posição no vetor
    addi x11, x11, 1 #Incrementando contador
    beq t1, t3, equal_element
    bge x15, x11, search_element_array2
    addi x10, x0, 0 # Coloca em X10 falso
    jal exit #Se chegou até aqui, o elemento não existe no vetor e,
    #portanto, não é uma permutação

    equal_element:
        #Checa se a posição é 1 no vetor auxiliar.
        #Se for = 1, o elemento encontrado já foi usado
        #para chegar à permutação (em caso de elementos repetidos).
    mul x31, x11, x14
    add x31, x31, sp
    lw x30, 0(x31)
    beq x30, s4, search_element_array2 #Se for 1,
    #volta para o search_element_array2
    sw s4, 0(x31)
    jal search_element_array1 #Se chegou até aqui,
    #o elemento existe e pode voltar a pesquisa no array1
```

```
different_sizes:
    addi x10, x0, 0 # Coloca em X10 falso
    jal x0, exit

    exit:
    addi a1, x10, 0 #Coloca o resultado no registrador de saída
    addi a0, x0, 1
    ecall # Escreve o Retorno

    addi a0, x0, 10
    ecall # exit

    exit_success:
    addi x10, x0, 1 # Coloca em X10 verdadeiro
    #(se chegou até aqui, é permutação)
    jal x0, exit
```

O código acima recebe dois vetores (arrays) e retorna se um é permutação do outro. Depois de verificar se o tamanho de ambos são iguais, o programa percorre o primeiro array elemento por elemento, verificando, para cada um deles, todo o segundo array.

Para que não hajam problemas com arrays que não são permutações um do outro retornando 1 em caso de elementos repetidos, foi necessário criar um array auxiliar utilizando o stack pointer. Nesse array auxiliar, na posição relativa ao elemento encontrado no segundo array, é feita uma marcação com 1, indicando que aquele elemento já foi considerado para a comparação de permutações.

Por fim, o programa retorna 1 caso um array seja permutação do outro e 0 caso contrário.

3 Conclusão

A principal dificuldade encontrada nas implementações dos exercícios foi no número 3, pois era preciso criar um vetor auxiliar ou implementar um método de ordenação dos elementos. Ademais, o trabalho foi de grande valia para por em prática os conceitos e comandos aprendidos a respeito do RISC-V.