Relatório do Laboratório 03 - Organização de Computadores

Grupo:

- Henrique Mateus Teodoro - 23100472
- Rodrigo Schwartz - 23100471

Exercício 1

O exercício 1 consiste em implementar a multiplicação de inteiros utilizando somas sucessivas, através de recursão. Inicialmente, criamos uma função em linguagem C que realiza essa operação, para que fosse possível analisar e entender melhor o comportamento da função:

```c
int multiplicacao(int a, int b){
    if(b == 0){
        return 0;
    }else{
        return a + multiplicacao(a, b-1);
    }
}
```

Iniciamos então o código em assembly declarando as variáveis que seriam necessárias em nosso código:

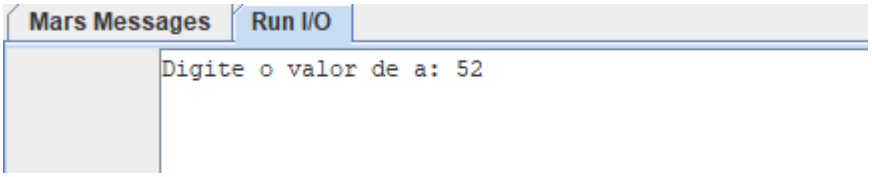
```
.data
        stringa: .asciiz "Digite o valor de a: "
        stringb: .asciiz "Digite o valor de b: "
        stringresult: .asciiz "A mutiplicação de a * b é: "
```

As variáveis **stringa** e **stringb** serão utilizadas para mostrarem as mensagens de input ao usuário no console, e a variável **stringresult** será utilizada para mostrar o resultado do programa no final.

Iniciamos então a parte das instruções realizando a operação de impressão da string **stringa** no console, e em seguida a operação de leitura de um inteiro, para que o usuário possa passar o valor de a desejado. Essas operações são feitas pelo seguinte trecho de código:

```
.text

        li $v0, 4               # seta a operação de impressão de string
        la $a0, stringa         # passa o endereço da variável a ser impressa
        syscall                 # realiza a impressão da stringa

        li $v0, 5               # seta a operação de leitura de um int pelo teclado
        syscall                 # faz a leitura no console
        move $t0, $v0           # move o valor lido pelo console para o reg $t0
```

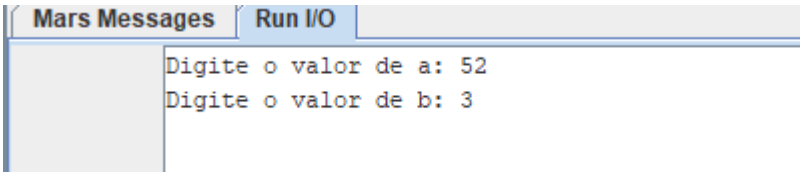
Após a execução desse trecho de código, obtém-se o seguinte resultado:



O mesmo é feito com a **stringb**, para que o usuário passe o valor do inteiro b:

```
        li $v0, 4               # seta a operação de impressão de string
        la $a0, stringb         # passa o endereço da variável a ser impressa
        syscall                 # realiza a impressão da string

        li $v0, 5               # seta a operação de leitura de um int pelo teclado
        syscall                 # faz a leitura no console
        move $t1, $v0           # move o valor lido pelo console para o reg $t1
```

Após a execução desse trecho de código, obtém-se o seguinte resultado:



Com isso, temos os valores dos inteiros a e b que queremos multiplicar, nos registradores **$t0** e **$t1**, respectivamente. Como teremos que utilizar o valor de a e de b como parâmetros de nossa função recursiva, passamos a e b para os registradores de argumento **$a0** e **$a1**, respectivamente:

```
        li $v0, 0               # carrega o valor 0 para o reg $v0, que será o reg de r
        move $a0, $t0           # passa o valor de a que está em $t0 para o reg $a0
        move $a1, $t1           # passa o valor de b que está em $t1 para o reg $a1
```

É feito ainda nesse trecho de código o carregamento do valor 0 para o registrador **$v0**, que será utilizado como o registrador de retorno da função, visto que o reg **$v0** foi carregado com o valor 5, para realizar a operação de leitura de um inteiro anteriormente.

Chama-se então a instrução **jal**, que desvia para a label função, e faz com que o registrador **$ra** (register address) armazene o endereço de retorno, para quando a função terminar, a execução do programa continuar a partir da próxima instrução (nesse caso a instrução **j exit**):

```
jal funcao              # vai para a função, e $
j exit
```

Dentro da label **funcao**, têm-se o seguinte trecho de código:

```
funcao:

        addi $sp, $sp, -4       # aumenta a stack
        sw $ra, 0($sp)          # dá um push no endereço de retorno da função

        beq $a1, $zero, return  # verifica se o segundo parametro é zero, para começar a dar o retorno das chamadas de funções

        addi $a1, $a1, -1       # decrementa o $a1 para chamar a função recursivamente
        jal funcao              # chamando a função recursivamente
        add $v0, $v0, $a0       # efetua a soma dos retornos de cada chamada de função
```

Inicialmente, como a função será chamada de forma recursiva, é necessário armazenar o endereço de retorno para cada chamada de função na stack, para quando começarmos a retornar o valor de cada soma sucessiva, podermos retornar para o ponto correto da execução. Para isso, é incrementado o valor -4 no registrador **$sp** (stack pointer), que aponta para a pilha. O valor precisa ser negativo, visto que a stack é implementada invertida. Salva-se então na stack após ela ser incrementada, o valor de **$ra** no endereço base de **$sp**.

É feito então uma instrução de **beq**, que verifica se o valor b que está em **$a1** já chegou em 0. Caso b não seja 0, o valor de b é decrementado em uma unidade, e então a função é chamada recursivamente. Caso b seja 0, a execução do código é desviada para a label **return**:

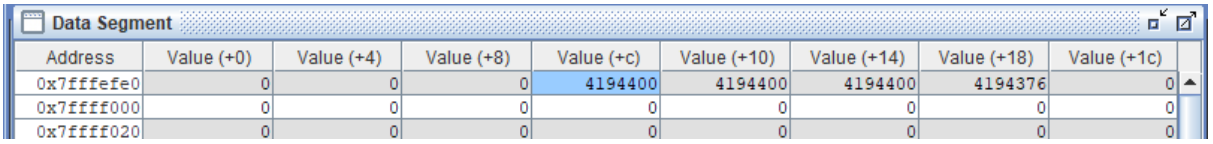
```
return:
        lw $ra, 0($sp)          # carrega o endereço de retorno da chamada atual para o $ra, para a
        addi $sp, $sp, 4        # diminui a stack, incrementa o stack pointer
        jr $ra                  # retorna para o endereço de execução de $ra
```

Na label **return**, a stack começa a ser desempilhada, carregando então o conteúdo do topo (o endereço de retorno da última chamada) para o registrador **$ra**, e então o ponteiro **$sp** é incrementado em 4 unidades, fazendo com que ocorra um pop na stack, e então retorna para o endereço de execução em **$ra**, através da

instrução **jr**. Assim, como **$sp** foi incrementado, agora **$sp** aponta para o endereço de retorno da penúltima chamada da função, e assim, sucessivamente, até a primeira chamada.

```
funcao:

        addi $sp, $sp, -4       # aumenta a stack
        sw $ra, 0($sp)          # dá um push no endereço de retorno da função

        beq $a1, $zero, return  # verifica se o segundo parametro é zero, para começar a dar o ret

        addi $a1, $a1, -1       # decrementa o $a1 para chamar a função recursivamente
        jal funcao              # chamando a função recursivamente
        add $v0, $v0, $a0       # efetua a soma dos retornos de cada chamada de função

return:
        lw $ra, 0($sp)          # carrega o endereço de retorno da chamada atual para o $ra, para
        addi $sp, $sp, 4        # diminui a stack, incrementa o stack pointer
        jr $ra                  # retorna para o endereço de execução de $ra
```

É importante notar que, após b ser igual a 0 (ou seja o **beq** ser satisfeito), e ocorrer o desvio para a label **return**, quando a stack começar a ser desempilhada e então retornar para o endereço da próxima execução com o pc apontando pro endereço armazenado em **$ra**, a instrução **add $v0, $v0, $a0** será chamada, ou seja, **$v0** receberá 1 soma do conteúdo de **$a0**, e seguirá a execução executando a label **return** novamente. Isso fará com que a pilha seja totalmente desempilhada, e assim, todas as somas b somas serão feitas.



Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7fffefe0	0	0	0	4194400	4194400	4194400	4194376	0
0x7ffff000	0	0	0	0	0	0	0	0
0x7ffff020	0	0	0	0	0	0	0	0

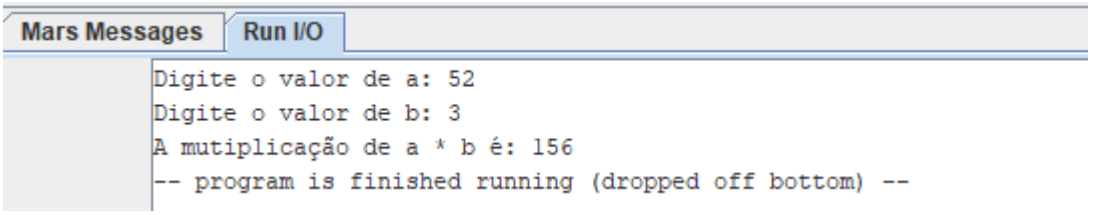
Na memória, é possível ver a stack, onde os endereços de retorno da última chamada até a primeira são armazenados da esquerda para a direita. Os endereços **4194400** representam o endereço para onde pc deve apontar quando a instrução **jr $ra** for efetuada. Por tanto, como o valor **4194400** será carregado para **$ra** quando a stack começar a ser desempilhada, o pc irá apontar para a instrução **add, $v0, $v0, $a0,** que faz as somas sucessivas. Por fim, o último valor a ser desempilhado da stack será o endereço de retorno da chamada da função, que é o endereço **41937**6. Assim, a ocorre a saída da função e retorna-se para a instrução que vem depois da chamada da função:

```
jal funcao              # vai para a função, e $
j exit
```

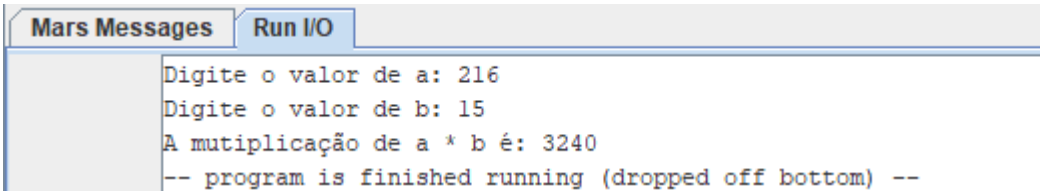
Que é um jump para a label **exit**. Em **exit**, é feita a impressão da **stringresult**, e depois, é feito a impressão do resultado da multiplicação de **a * b**, e então o programa é encerrado.

```
exit:
        move $t0, $v0

        li $v0, 4 # seta a operação de impressão de string
        la $a0, stringresult # passa o endereço da variável a ser impressa
        syscall

        li $v0, 1 # seta a operação de impressão de int
        move $a0, $t0
        syscall
```

Após a execução desse trecho de código, obtém-se o seguinte resultado:



Os valores passados para a e b foram 52 e 3 respectivamente, e o produto de 52 * 3 resulta em 156. Com isso, percebe-se o adequado funcionamento do programa.



Nesse outro caso de teste, foram passados para a e b os valores 216 e 15 e o produto de 216 * 15 resulta em 3240. Portanto, para esse outro caso de teste, percebe-se novamente o funcionamento adequado do programa.

Exercício 2

O exercício dois consiste em implementar um código em assembly que realiza a soma dos elementos de um array de forma recursiva, utilizando uma função. Inicialmente, foi implementada essa função em uma linguagem de alto nível, para que fosse possível identificar o comportamento da função utilizando a recursão:

```
int somaArray(int array[], int tamanho) {
    if (tamanho == 0) {
        return 0;
    } else {
        return array[tamanho - 1] + somaArray(array, tamanho - 1);
    }
}
```

Após isso, iniciou-se o programa em assembly, declarando as variáveis que seriam utilizadas no segmento .data:

```
.data
        stringresult: .asciiz "A soma dos elementos do array é: "
        B: .word 20 10 50
```

Foram declaradas duas variáveis, a primeira chamada **stringresult**, é uma string que será utilizada para mostrar uma mensagem no console no final do programa, juntamente com o resultado da operação. E por fim, a variável B, que consiste em um vetor de inteiros. Para o primeiro teste, o vetor possui 3 elementos (20, 10 e 50), e a soma total esperada desses elementos é 80.

São então passados os argumentos que serão utilizados na função recursiva:

```
.text

        la $a0, B               # Carrega o endereço base do vetor para o
        li $a1, 3               # Carrega o valor 3 para o reg $a1 que rep
```

Primeiramente, é passado para o registrador **$a0**, o endereço base do vetor B, ou seja, um ponteiro para B, para que seja possível percorrer esse vetor e fazer a soma de seus elementos corretamente. Já no registrador **$a1**, é passado o tamanho desse vetor,

para que dentro da função seja possível identificar o momento exato de término do vetor. Em seguida, é feita a chamada da função **somaArray**, utilizando a instrução jal:

```
        jal somaArray
        j exit
```

Com isso, a execução do código é desviada para a label **somaArray**, que irá realizar o processo de soma dos elementos do vetor B. Com a chamada da instrução **jal**, o registrador **$ra** guarda o endereço de retorno que aponta para a instrução **j exit**, para quando a função terminar, o código iniciará sua execução partindo dessa instrução.

```
somaArray:
        addi $sp, $sp, -8       # Ajusta a stack reservando espaço para 2 itens
        sw $ra, 4($sp)          # Guarda endereço de retorno
        sw $a1, 0($sp)          # Guarda argumento (n)

        bne $a1, $zero, L1      # Verifica se o tamanho é zero, para começar a dar o retorno das

        li $v0, 0               # Se for igual a zero, devolve 0
        jr $ra                  # Retornar
```

Dentro da label **somaArray**, inicialmente decrementa-se o registador **$sp** (stack pointer) em 8 unidades, para que seja possível armazenar duas variáveis na stack. Essas duas variáveis serão armazenadas em cada chamada recursiva da função, onde uma variável é o endereço de retorno, ou seja, o lugar para onde a execução deve voltar após aquela chamada específica da função **somaArray** tiver terminado e retornado seus valores, e o tamanho do vetor para aquela chamada específica da função. É importante lembrar que para armazenar dados na stack, é necessário decrementar o endereço base (**$sp**), visto que a stack é implementada invertida, de um valor alto na memória, até um valor baixo na memória.

```
        bne $a1, $zero, L1      # Verifica se o tamanho é zero, para começar a dar o retorno das chamadas

        li $v0, 0               # Se for igual a zero, devolve 0
        jr $ra                  # Retornar
```

Durante a execução do programa, é possível ver os endereços de retorno e o tamanho do vetor para cada chamada de função:

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)
0x7fffefc0	0	0	0	0	0	0	0
0x7fffefe0	4194356	1	4194356	2	4194356	3	4194320
0x7ffff000	0	0	0	0	0	0	0
0x7ffff020	0	0	0	0	0	0	0

É então feito um **branch not equal (bne)** para verificar se o tamanho do vetor naquela chamada da função **somaArray** chegou ou não em 0. Se o tamanho não tiver

chego em 0 naquela chamada, o **bne** é satisfeito e a label **L1** é então chamada. Caso o contrário, o registrador **$v0** é carregado com o valor 0, visto que o caso base onde o tamanho do vetor é 0, deve retornar 0. Assim, a instrução **jr** é chamada, para que a função retorne a sua execução conforme aponta o registrador **$ra**. E então os retornos das funções começam a serem feitos.

```
L1:
        addi $a1, $a1, -1       # Nova chamada
        jal somaArray           # Chama somaArray com o novo n decrementado
```

Dentro da label **L1**, que é chamada quando o branch not equal é satisfeito, ou seja, quando o tamanho de vetor ainda não é 0, decrementa-se o registrador **$a1**, que é o registrador de argumentos que a função **somaArray** utiliza para saber o tamanho atual do vetor. Ou seja, é feito um decremento no tamanho do vetor. E então é feita a chamada da função **somaArray** de forma recursiva, que agora utilizará o vetor com um elemento a menos, até que chegue-se no caso base onde o tamanho do vetor é igual a 0.

Quando o **branch not equal** não for mais satisfeito, ou seja, quando o tamanho do vetor for 0, o registrador **$ra** da chamada do caso base irá apontar para a instrução seguinte, que consiste na instrução que segue a instrução de chamada recursiva da função **somaArray**. Ou seja, a partir do comentário que consta o ponto de retorno:

```
L1:
        addi $a1, $a1, -1       # Nova chamada
        jal somaArray           # Chama somaArray com o novo n decrementado

        # Ponto de retorno da chamada recursiva (começa a desempilhar)

        lw $a1, 0($sp)          # Recupera o argumento passado
        lw $ra, 4($sp)          # Recupera o endereço de retorno
        add $sp, $sp, 8         # Libera o espaço da pilha
```

A partir daí, a stack começa a ser desempilhada a partir de seu topo, ou seja, o registrador **$a1** passa a receber o tamanho do vetor naquela chamada, e carrega-se para o registrador **$ra**, o endereço de retorno para aquela chamada. E então incrementa-se 8 unidades no stack pointer (**$sp**), para que esses valores sejam deslocados da pilha.

```
li $t1, 4               # Tamanho de cada elemento do array (em bytes)
mul $t2, $a1, $t1       # Multiplica o índice pelo tamanho do elemento
add $t2, $a0, $t2

lw $t3, 0($t2)          # Carrega o valor do último endereço do array[tamanho - 1]
add $v0, $v0, $t3       # Soma o valor de array[tamanho - 1] à soma do restante do array
jr $ra
```

É então feito a multiplicação do índice do vetor pelo valor 4, visto que cada elemento do vetor ocupa 4 bytes. E então soma-se esse deslocamento ao endereço base do vetor. Assim, por exemplo, se deseja-se somar o conteúdo do vetor no índice 2, é preciso acessar o vetor com 8 deslocamentos a partir de seu endereço base, que está em **$a0**.

```
lw $t3, 0($t2)          # Carrega o valor do último endereço do array[tamanho - 1]
add $v0, $v0, $t3       # Soma o valor de array[tamanho - 1] à soma do restante do array
jr $ra                  # Faz o retorno para voltar a execução correta do programa com base em $ra
```

É então feito com base nos últimos cálculos, o carregamento do valor do vetor no endereço base atual. Portanto, o vetor será somado de trás para frente, visto quando a pilha começar a ser desempilhada, para esse caso onde o vetor possui 3 elementos, o primeiro índice a realizar esse cálculo será o índice 2, que foi multiplicado por 8, para que seja possível obter o valor do último elemento do vetor. É então somado o valor desse último elemento do vetor no registrador **$v0**, e é feita a chamada de retorno da penúltima chamada da função **somaArray**, que irá começar a desempilhar a stack, e assim, o processo comentado anteriormente se repetirá até que todos os elementos do vetor sejam somados.

Quando a stack for completamente desempilhada, o registrador **$ra** estará apontado para o endereço de retorno da primeira chamada da função **somaArray**, que foi feita lá no começo do programa:

```
.text

    la $a0, B               # Carrega o endereço base do vetc
    li $a1, 3               # Carrega o valor 3 para o reg $a

    jal somaArray
    j exit
```

O registrador **$ra** irá apontar então para a função que **j exit**, que deve ser a próxima função a ser executada.

```
exit:
    move $t0, $v0           # Passa o resultado retornado no fim da função somaArray para o reg $t0

    li $v0, 4               # Seta a operação de impressão de string
    la $a0, stringresult    # Passa o endereço da variável a ser impressa
    syscall                 # Imprime a string no console

    li $v0, 1               # Seta a operação de impressão de int
    move $a0, $t0           # Passa o inteiro que será impresso no console para o reg $a0
    syscall                 # Faz a impressão do inteiro
```

Nessa label, será impressa a **stringresult** no console e será mostrado o resultado da função **somaArray** em sequência também no console. Com isso, após o fim dessas instruções, o programa é encerrado.

Para o primeiro teste feito, o vetor **B** possui 3 elementos (20, 10 e 50) e se espera que a soma resulte em 80. Após a execução do programa, é possível verificar o resultado

esperado, e confirmar o funcionamento correto do programa:

```
A soma dos elementos do array é: 80
-- program is finished running (dropped off bottom) --
```

Para o segundo teste, o vetor **B** possui 5 elementos (10, 20, 30, 40, 50) e se espera que a soma resulte em 150:

```
.data
        stringresult: .asciiz "A soma dos elementos do array é: "
        B: .word 10 20 30 40 50
.text

        la $a0, B               # Carrega o endereço base do vetor
        li $a1, 5               # Carrega o valor 3 para o reg $a1
```

Após o término do segundo teste, obtém-se o resultado esperado, confirmando novamente o funcionamento adequado do programa: