UNIVERSIDADE FEDERAL DE SANTA CATARINA
EDUARDO BISCHOFF GRASEL - 22200355

Relatório Lab 6 Organização de Computadores

Florianópolis - SC
2024

Objetivo geral

Avaliação de desempenho de acordo com o número de instruções executadas durante o código, trabalhar de maneira a evitar escrever código desnecessário e treinar a manipulação de floats, principalmente com eles sendo passados para procedimentos.

Exercício 1:

Iniciaremos a descrição do exercício no .data, onde teremos apenas algumas as mensagens a serem mostradas na saída e a alocação de um espaço da memória, este sendo definido como o valor máximo possível de alocação.

```
.data
    promptN: .asciiz "Digite N: "
    promptA: .asciiz "Elementos de A : "
    promptB: .asciiz "Elementos de B: "
    avgAmsg: .asciiz "Média de A: "
    avgBmsg: .asciiz "Média de B: "
    newline: .asciiz "\n"

    # Valor maximo da alocação:
    maxN: .word 1000
```

No início do .text simplesmente printamos a mensagem para digitar N no terminal e recebemos o valor e armazenamos em um registrador, $t0 nesse caso, em seguida carregamos o maxN e verificamos se o valor está dentro do limite esperado (1000). Agora que temos as informações, prosseguimos para alocar os arrays. Primeiramente calculamos e espaçamento entre os arrays (4 bytes), podendo usar um sll com constante 2, então utilizamos uma lógica simples para alocar memória no heap e transferir o endereço para $s0.

```
allocate_arrays:
    # Calcula o espaço dos arrays (4 bytes float)
    sll $t1, $t0, 2   # t1 = N * 4

    # Aloca o espaço arrayA
    move $a0, $t1
    li $v0, 9   # alocar no heap
    syscall
    move $s0, $v0     # salva o endereço de arrayA em $s0
```

Tal lógica será idêntica para a locação do array B, a única diferença sendo que o endereço será salvo em $s1.

Precisamos também ler as arrays, novamente como a lógica entre as duas será extremamente parecida, explicarei apenas a array A, tendo em vista que a B terá o mesmo padrão, apenas será executada depois. Carregamos a mensagem na tela para que o usuário esteja ciente que precisa escrever, então entramos em um loop, o qual pedirá ao usuário o float e guardará no array, para isso precisamos sempre atualizar o offset para o acesso e armazenamento correto na memória, quando a condição de parada for atingida ($t2 >= N), seguiremos para ler o array B, o qual seguirá o mesmo padrão.

```
    # Ler os elementos do arrayA
    li $v0, 4
    la $a0, promptA
    syscall
    move $t2, $zero   # index arrayA

read_arrayA:
    bge $t2, $t0, read_arrayB  # if index >= N, vai para arrayB

    # Ler float
    li $v0, 6
    syscall
    sll $t3, $t2, 2   # offset = index * 4
    add $t4, $s0, $t3 # address = base + offset
    swc1 $f0, 0($t4)  # guarda float no arrayA

    addi $t2, $t2, 1
    j read_arrayA
```

Pronto, já temos as duas arrays necessárias, agora basta chamar o procedimento para a execução do cálculo de média, esta será chamada assim que a leitura do array B for concluída.

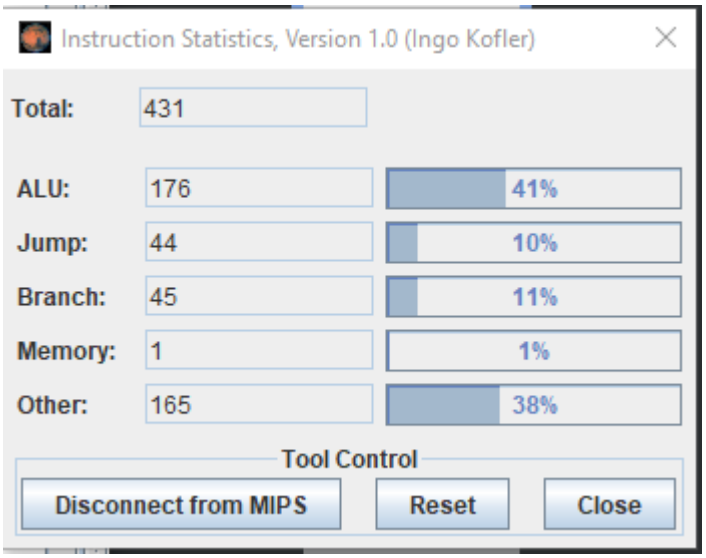
Dentro do procedimento, $a0 e $a1 contém o endereço do array e tamanho, então basta fazer a chamada do procedimento que basicamente criará um index, e entrará num loop que: Calcula o offset da memória, adicionar ao endereço o offset, carregar o valor da posição do array, somar esse valor a um registrador acumulativo. assim que todo array for percorrido e somado, iremos para o bloco responsável por calcular a média, basicamente convertemos N para um valor double, dividimos pelo total calculado acima e retornamos o resultado que estará em $f0.

```
calculate_avg:
    move $t2, $zero   # index
    li $t5, 0         # registrador temporario com 0
    mtc1 $t5, $f2     # sum do array agora □ 0.0
    cvt.s.w $f2, $f2  # Converte 0 para float e guarda em $f2

    calculate_avg_loop:
        bge $t2, $a1, finalize_avg  # if index >= N, finaliza

        sll $t3, $t2, 2   # offset = index * 4
        add $t4, $a0, $t3 # address = base + offset
        lwc1 $f4, 0($t4)  # carrega float do array
        add.s $f2, $f2, $f4 # sum += array[index]

        addi $t2, $t2, 1
        j calculate_avg_loop

finalize_avg:
    mtc1 $a1, $f6     # move n para $f6
    cvt.s.w $f6, $f6  # converte N para float
    div.s $f0, $f2, $f6  # media = sum / N
    jr $ra            # return
```

Agora voltamos ao main, salvamos o retorno e mostramos na tela, Agora para o array B, basta chamar o procedimento passando o endereço do array B como parâmetro.

```
Digite N: 10
Elementos de A : 0.11
0.34
1.23
5.34
0.76
0.65
0.34
0.12
0.87
0.56
Elementos de B: 7.89
6.87
9.89
7.12
6.23
8.76
8.21
7.32
7.32
8.22
M?dia de A: 1.0320001
M?dia de B: 7.783

-- program is finished running --
```

O programa funciona de acordo com o esperado, agora gerando o relatório das instruções, tivemos a seguinte distribuição e total:



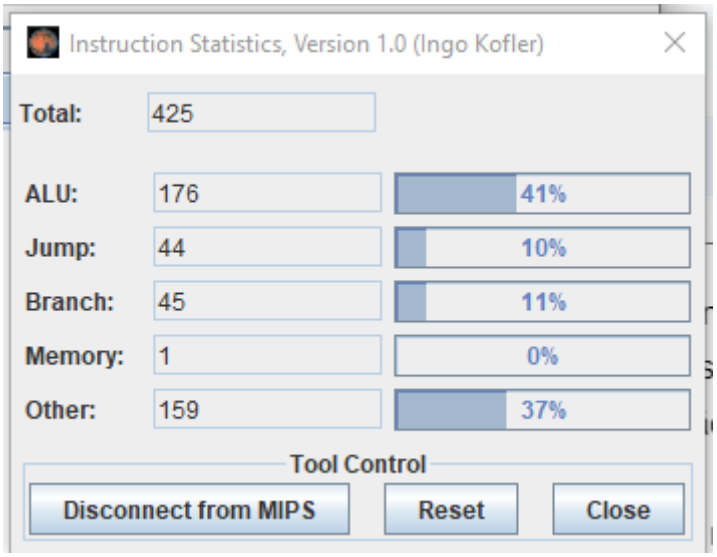
Exercício 2:

Como eu acredito ter feito o código sem muita redundância, foi difícil alterar o código de modo que torna-se ele com menos instruções e sem redundâncias, ou até mesmo mudar a distribuição de tipos de instrução.

No fim segui esses critérios no código:

- remoção de moves duplicados em read_arrayA e read_arrayB
- tentar facilitar os loops, tornando-os mais simples sem precisar redefinir o índice
- remoção de moves duplicados nos procedimentos (move $a0, $s1; move $a1, $t0) já deixando esses valores previamente carregados
- tentativa de refatoração nos syscalls a fim de diminuir as instruções

No fim de tudo a melhora não foi muito significativa, as instruções que precisam da ULA, jumps, branches e memória permaneceram as mesmas, entretanto tivemos uma melhora no “other's” que diminuiu de 165 para 159



Ou seja, como as instruções principais e que ocupam mais tempo permaneceram constantes, a melhora apresentada no código é mínima, mesmo que exista uma melhora, pelo programa ser simples referente ao propósito, essa diferença precisaria de uma execução extremamente longa para mostrar alguma mudança significativa.

Conclusão

A minha maior dificuldade foi trabalhar com float e endereços sendo passados para o procedimento, pois é muito complicado trabalhar com float ou doubles pelo fato de não terem as mesmas instruções disponíveis para inteiros, fora isso na hora de tentar alterar o código foi muito complicado, afinal qualquer mudança quebrava ele completamente achar algo que poderia ser modificado ou removido não foi fácil.