UNIVERSIDADE DO VALE DO ITAJAÍ

ESCOLA DO MAR, CIÊNCIA E TECNOLOGIA

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

CC4189 – ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

Professores,

Cesar Albanes Zeferino, Dr.

Eduardo Alves da Silva, MSc.

PROGRAMAÇÃO EM LINGUAGEM DE MONTAGEM DO MIPS

Acadêmicos,

Israel Efraim de Oliveira,

José Carlos Zancanaro.

Itajaí (SC), 19 de abril de 2018.

PROGRAMAÇÃO EM LINGUAGEM DE MONTAGEM DO MIPS

Israel Efraim de Oliveira

José Carlos Zancanaro

Abril / 2018

Professores: Cesar Albanes Zeferino, Eduardo Alves da Silva.

Curso: Bacharelado em Ciência da Computação.

Palavras-chave: Linguagem de montagem, assembly, MIPS, processador tipo RISC.

Número de páginas: 9.

"I have always wished for my computer to be as easy to use as my telephone; my wish has come true because I can no longer figure out how to use my telephone"

(Sempre desejei que meu computador fosse tão fácil de usar quanto meu telefone. Meu desejo tornou-se realidade, porque não consigo mais descobrir como utilizar meu telefone)

(Bjarne Stroustrup)

1. Introdução

Este relatório aborda a resolução de dois problemas propostos na disciplina de Arquitetura e Organização de Computadores do curso de bacharelado em Ciência da Computação. Os problemas consistem em operações para serem implementadas na linguagem de montagem do processador MIPS (*Microprocessor without interlocked pipeline stages*), baseado na arquitetura RISC. Tratamos ambos os problemas utilizando o MARS (*MIPS assembly and runtime simulator*), uma IDE para programação na linguagem de montagem do MIPS.

O **programa um** requer a leitura de um vetor de tamanho fornecido pelo usuário e a ordenação crescente deste. O tamanho do vetor é variante e precisa estar de 2 à 8, isto é, maior que um e menor que 9. Além disto, ambos os estágios (pré-ordenação e pós-ordenação) do vetor devem ser imprimidos na tela.

O **programa dois** sugere a separação de um vetor de tamanho também fornecido pelo usuário em dois outros vetores: vetor de números pares e outro de números ímpares. O tamanho do vetor de entrada deve também ser validado com o mesmo requisito do programa um e, o resultado também precisa ser exibido ao usuário.

O relatório consistirá na construção e explicação da resolução abordada pelos autores, mesclando a explicação de procedimentos realizados nos programas em suas interseções, isto é, nas operações similares entre ambos programas.

2. Construção

Para a abordagem dos problemas, procuramos criar soluções flexíveis que possibilitassem a alteração de parâmetros sem afetar o desenvolvimento do programa em si, isto significa evidenciar a separação das variáveis (como tamanho mínimo e máximo) do resto do código.

O primeiro processo em ambos os programas diz respeito à realização do carregamento dos endereços de memória em registradores para prosseguir com a parametrização das operações, isto é, limitar o tamanho mínimo e máximo que o usuário pode escolher.

Em seguida, precisamos solicitar e validar um valor inteiro até que este esteja dentro do tamanho permitido. Esta operação é realizada através de duas comparações e a soma de seus resultados, ou seja, utilizamos a instrução de *set less than* para verificar se o valor inserido pelo usuário é menor que o valor mínimo, caso positivo, a instrução retornará o valor 1, senão 0; o mesmo acontece com a segunda comparação, porém utilizamos a instrução de *set greater than*, que realiza a operação contrária. Definimos o valor como válido se e somente se a soma da primeira comparação com a segunda for igual a 0.

```
add     $s3, $zero, $v0                 # $s3 (tamanhoMin) = $v0
slt     $t0, $s3, $s0                   # $t0 = $s3 < $s0 ? 1 : 0
sgt     $t1, $s3, $s1                   # $t1 = $s3 > $s1 ? 1 : 0
add     $t2, $t0, $t1                   # $t2 = $t0 + $t1

beq     $t2, $zero, do_while_tamVet_fim # Pula pro final se o input for > 1 e < 9.
```

Posteriormente, solicitamos os valores para preencher o vetor a ser ordenado (Programa 01) ou separado (Programa 02). O procedimento é simples e resume-se a um laço de repetição do tipo *for*, em que a variável de controle (neste caso associada ao registrador *$t0*) é utilizada também para endereçar os elementos na memória, respeitando a restrição de alinhamento de uma *word*, ou seja, 4 bytes. Para tanto, é necessário realizar o deslocamento do valor em *$t0* por 2^2 (4) e somar ao endereço base do vetor.

```
sll     $t1, $t0, 2                     # $t1 = $t0 (i) * 4
add     $t1, $s0, $t1                   # $t1 += &Vetor_Entrada
```

Subsequentemente, realizamos operações distintas entre os programas, no primeiro programa, realizamos a ordenação crescente dos valores no vetor de entrada; no segundo, separamos os valores pares dos ímpares em dois outros vetores.

Em relação ao primeiro programa, implementamos a ordenação bubblesort no modo tradicional, com dois laços de repetição do tipo *for* para varrer os elementos do vetor. No laço externo, temos a repetição limitada pela quantidade de elementos do vetor, isto é, deverá ser executada enquanto a variável de controle for menor que o tamanho do vetor.

```
        add     $t0, $zero, $zero               # $t0 (i) = 0
for_bubble_extno_inicio:
        beq     $t0, $s3, for_bubble_extno_fim  # Pula pro fim se i = tamanhoVet
        .
        .
        .
        addi    $t0, $t0, 1                     # [i++] $t0 = $t0 + 1
        j       for_bubble_extno_inicio         # Pula para o inicio do for externo
for_bubble_extno_fim:
```

No laço interno, a repetição é limitada não apenas pela quantidade de elementos iteráveis do vetor (*tamanho – 1*), mas também pela contagem realizada no laço externo (*tamanho – i – 1*).

```
        add     $t1, $zero, $zero               # $t1 (j) = 0
        sub     $t2, $s3, $t0                   # $t2 = $s3 (tamanhoVet) - $t0 (i)
        addi    $t2, $t2, -1                    # $t2 -= 1
for_bubble_intno_inicio:
        beq     $t1, $t2, for_bubble_intno_fim  # Pula pro fim se j = tamanhoVet - i - 1
        .
        .
        .
        addi    $t1, $t1, 1                     # [j++] $t1 = $t1 + 1
        j       for_bubble_intno_inicio         # Pula para o inicio do for interno
for_bubble_intno_fim:
```

Após criar as estruturas de repetição, é necessário implementar a operação que verifica se um par de valores (*vetor[j]*, *vetor[j+1]*) precisa ser ordenado. Assim como na inserção de valores no vetor, utilizamos o deslocamento lógico na variável de controle do laço para endereçar os elementos na memória de acordo com o alinhamento. Após a consulta dos valores, utiliza-se uma operação de if-then para realizar ou não a troca dos elementos.

Para buscar os elementos j e j+1 na memória:

```
sll     $t3, $t1, 2                     # $t3 (&vet[j]) = $t1 (j) * 4
add     $t3, $s4, $t3                   # $t3 (&vet[j]) += &vetor

addi    $t4, $t1, 1                     # $t4 = j+1;
sll     $t4, $t4, 2                     # $t4 (&vet[j+1]) = $t1 * 4
add     $t4, $s4, $t4                   # $t4 (&vet[j+1]) += &vetor

lw      $t5, 0($t3)                     # $t5 = vet[j]
lw      $t6, 0($t4)                     # $t6 = vet[j+1]
```

Para efetuar a troca quando necessário:

```
        slt     $t7, $t6, $t5                   # $t7 = vet[j+1] < vet [j] ? 1 : 0
        beq     $t7, $zero, bubble_nao_ordenar  # Não ordena se vet[j+1] > vet[j]
        sw      $t6, 0($t3)                     # $t6 (vet[j]) = vet[j+1]
        sw      $t5, 0($t4)                     # $t5 (vet[j+1]) = vet[j]
bubble_nao_ordenar:
        addi    $t1, $t1, 1                     # [j++] $t1 = $t1 + 1
        j       for_bubble_intno_inicio         # Pula para o inicio do for interno
```

No segundo programa, realiza-se apenas uma operação de if-then-else para separar os valores pares e ímpares, determinados como tais pelo valor do bit menos significativo (o bit mais a direita). Se este bit for positivo (1), o número é ímpar, pois esta é a única posição (2^0) no sistema de numeração binário que possui um resultado ímpar. Para descobrir se este bit está ligado, utilizamos uma máscara para zerar (*and 0x00000001*) todos os bit à esquerda do bit menos significativo e comparamos o resultado com zero (se zero, então par).

```
        andi    $t2, $t1, 1                     # $t2 = $t1 & 0x00000001
        bne     $t2, $zero, if_valor_impar      # Desvia se é impar
        .
        .
        .
        j       if_valor_impar_fim
if_valor_impar:
        ...
if_valor_impar_fim:
        ...
```

Após determinado se par ou ímpar, devemos adicionar o valor no devido vetor, mas, *como sabemos em que posição do vetor adicionar?* A resposta para isto é, armazenar em registradores separados quantos valores já foram adicionados em cada um dos vetores.

Neste programa, guardamos a quantidade de pares no registrador *$s4* e a quantidade de ímpares em *$s7*, como a seguir:

Adicionar em vetor de números pares:

```
sll     $t2, $s4, 2                    # $t2 = (tamanhoPares) * 4
add     $t2, $s2, $t2                  # $t2 += &(Vetor_Pares)
sw      $t1, 0($t2)                    # Vetor_Pares[tamanhoPares] = $t1 (Vetor_Entrada[i])
addi    $s4, $s4, 1                    # $s4 (tamanhoPares) += 1
```

Adicionar em vetor de números ímpares:

```
sll     $t2, $s7, 2                   # $t2 = (tamanhoImpares) * 4
add     $t2, $s5, $t2                 # $t2 += &(Vetor_Impares)
sw      $t1, 0($t2)                   # Vetor_Impares[tamanhoImpares] = $t1 (Vetor_Entrada[i])
addi    $s7, $s7, 1                   # $s7 (tamanhoImpares) += 1
```

Por último, em ambos os programas necessita-se exibir o resultado, que consiste em uma operação muito simples: iterar o vetor com um laço de repetição e usar o deslocamento lógico pelo alinhamento para consultar os devidos elementos na memória.

No primeiro programa, precisamos apenas exibir um único vetor (o vetor ordenado), enquanto que no segundo programa, devemos exibir ambos os vetores de números pares e ímpares, porém, como temos a quantidade de elementos de cada vetor em ambos os programas, o processo continua sendo simples.

Exibir valores ordenados:

```
for_exibir_inicio:
        beq     $t0, $s3, for_exibir_fim        # Pular se i = tamanhoVet (preenchido pelo usuário)

        sll     $t1, $t0, 2                     # $t1 = $to (i) * 4
        add     $t1, $t1, $s4                   # $t1 = $t1 (i*4) + &vetor

        ...

        lw      $a0, 0($t1)                     # $a0 = vet[i]
        addi    $v0, $zero, 1                   # $v0 = print_int
        syscall                                 # CHAMADA DE SISTEMA (print_int)

        la      $a0, TXT_COM_NOVALINHA          # $a0 = "\n"
        addi    $v0, $zero, 4                   # $v0 = print_string
        syscall                                 # CHAMADA DE SISTEMA (print_string)

        addi    $t0, $t0, 1
        j       for_exibir_inicio               # Volta ao início do for
for_exibir_fim:
```

Exibir vetores de pares (utiliza-se o mesmo processo para o de ímpares).

```
        add     $t0, $zero, $zero               # $t0 (i) = 0
for_exibir_pares_inicio:
        beq     $t0, $s4, for_exibir_pares_fim  # Pular se i = tamanhoPares

        sll     $t1, $t0, 2                     # $t1 = $to (i) * 4
        add     $t1, $t1, $s2                   # $t1 = $t1 (i*4) + &(Vetor_Pares)
        ...
        lw      $a0, 0($t1)                     # $a0 = vet[i]
        addi    $v0, $zero, 1                   # $v0 = print_int
        syscall                                 # CHAMADA DE SISTEMA (print_int)
        ...
        addi    $t0, $t0, 1
        j       for_exibir_pares_inicio         # Volta ao início do for
for_exibir_pares_fim:
```

3. Experimento e análise de instruções

3.1 Programa 01

Entrada e saída de dados para o vetor { 8, 7, 6, 5, 4, 3, 2, 1 }.

```
Entre com o tamanho do vetor (min = 2, máx = 8): 8

Insira os valores para o vetor:
Vetor[0] = 8
Vetor[1] = 7
Vetor[2] = 6
Vetor[3] = 5
Vetor[4] = 4
Vetor[5] = 3
Vetor[6] = 2
Vetor[7] = 1

Realizando bubble sort.
Bubblesort concluído.

Exibindo vetor ordenado...
Vetor[0] = 1
Vetor[1] = 2
Vetor[2] = 3
Vetor[3] = 4
Vetor[4] = 5
Vetor[5] = 6
Vetor[6] = 7
Vetor[7] = 8

-- program is finished running --
```

O quadro estatístico resultante foi:

Classe	**Nº de execuções**	**Percentual**
Aritmética e Lógica (ALU)	455	54%
Desvio incondicional (Jump)	52	06%
Desvio condicional (Branch)	92	11%
Acesso à memória (Memory)	131	16%
Outras	112	13%
Total	842	100%

3.2 Programa 02

Entrada e saída de dados para o vetor { 0, 1, 2, 3, 4, 5, 6, 7 }.

```
Entre com o número de itens do vetor(mín. = 2, máx. = 8): 8

Insira os valores para o vetor de entrada:
Vetor_Entrada[0]: 0
Vetor_Entrada[1]: 1
Vetor_Entrada[2]: 2
Vetor_Entrada[3]: 3
Vetor_Entrada[4]: 4
Vetor_Entrada[5]: 5
Vetor_Entrada[6]: 6
Vetor_Entrada[7]: 7

Separando valores numéricos.
Separação concluída.

Exibindo vetores...

Vetor de números pares:
Vetor_Pares[0]: 0
Vetor_Pares[1]: 2
Vetor_Pares[2]: 4
Vetor_Pares[3]: 6

Vetor de números ímpares:
Vetor_Impares[0]: 1
Vetor_Impares[1]: 3
Vetor_Impares[2]: 5
Vetor_Impares[3]: 7

-- program is finished running --
```

O quadro estatístico resultante foi:

Classe	**Nº de execuções**	**Percentual**
Aritmética e Lógica (ALU)	333	64%
Desvio incondicional (Jump)	28	06%
Desvio condicional (Branch)	37	07%
Acesso à memória (Memory)	39	07%
Outras	88	16%
Total	525	100%

4. Conclusão

Através dos quadros estatísticos obtidos, nota-se a superioridade da complexidade do primeiro programa em comparação ao segundo. Isto deve-se ao fato de que ordenar um vetor é naturalmente mais extenso do que separar um vetor com base em uma determinada condição. Além disto, envolve-se muito mais controle de variáveis de laço de repetição e gerenciamento adequado do deslocamento para acesso de um elemento do vetor na memória.

Em relação à construção das resoluções para os problemas sugeridos, é possível identificar que ambos os programas possuem diversas rotinas repetitivas, isto é, operações similares com parâmetros diferentes. Para estes casos, o uso de procedimentos em linguagens de montagem também é recomendado (instruções de *jal*, *jalr*, *jr*), pois reduz a quantidade de esforço na construção de uma determinada solução através de reusabilidade de código e distribuição da responsabilidade de tarefas.

A resolução de ambos os problemas no estudo da programação em linguagem de montagem permitiu-nos amplificar imensamente o conhecimento das estruturas fundamentais da arquitetura e organização de computadores através do exercício da lógica e do estímulo ao estudo dos conceitos arquiteturais do processador MIPS. Evidencia-se, portanto, a importância da prática como um fator indispensável para o aprendizado destes segmentos.