Instituto Federal de Minas Gerais - Campus Avançado Ouro Branco
Curso: Bacharelado em Sistemas de Informação
Disciplina: Arquitetura e Organização de Computadores
Professor: Saulo Henrique Cabral Silva (saulo.cabral@ifmg.edu.br)

Trabalho Prático 3

Assembly - MIPS

Nome: Felipe Leal Vieira RA: 0034372

João Victor Dutra Martins Silva RA: 0076873

1.Introdução:

Neste trabalho, exploraremos conceitos fundamentais de programação por meio do desenvolvimento de um programa utilizando linguagem Assembly-MIPS, uma das mais básicas e próximas do hardware. A tarefa proposta se baseia em um cenário real e histórico da ferrovia, onde o veterano funcionário Seu José, conhecido por sua habilidade única de reorganizar vagões, enfrenta um novo desafio na era da modernização ferroviária: automatizar a tarefa de organizar os vagões de trens em ordem crescente.

A narrativa se passa em uma antiga estação ferroviária, onde uma ponte giratória de 90 graus era utilizada para permitir a passagem de barcos. Seu José descobriu que, utilizando essa ponte, era possível inverter a posição de dois vagões adjacentes com um giro de 180 graus, facilitando assim o processo de reorganização, dessa forma podemos implementar o algoritmo “Bubble Sort”. No entanto, com o avanço da tecnologia, a empresa ferroviária busca uma solução automatizada para determinar o número mínimo de trocas necessárias para organizar os vagões de forma eficiente.

A missão deste trabalho é desenvolver uma rotina em linguagem Assembly-MIPS que, dado um trem com vagões desordenados, determine o número mínimo de trocas de dois vagões adjacentes necessárias para ordená-lo corretamente. A entrada para o programa consiste em uma série de casos de teste, onde cada caso descreve a quantidade de vagões e a ordem atual deles. O objetivo final é garantir que, após as trocas, os vagões estejam ordenados de forma que o vagão número 1 seja o primeiro, o número 2 seja o segundo, e assim por diante, até o último vagão.

Este exercício não só revisa conceitos básicos de lógica e algoritmos, como também desafia a capacidade de implementação em uma linguagem de baixo nível, promovendo uma compreensão mais profunda da interação entre software e hardware, essencial para a construção de soluções eficientes e eficazes em sistemas computacionais.

2. Documentação:

2.1. Main :

Este trecho do código em Assembly prepara o ambiente para a coleta de dados e interação com o usuário. Primeiro, exibe uma mensagem inicial com instruções sobre o programa. Em seguida, carrega o endereço do array onde os números dos vagões serão armazenados e define o tamanho máximo do array como 8 elementos. Um contador é inicializado para monitorar quantos números foram inseridos, garantindo que a ordenação só comece quando todos os dados estiverem prontos ou o usuário decidir encerrar a entrada. O endereço do array é carregado em um registrador, que servirá como ponteiro para o armazenamento dos dados.

Após a exibição da mensagem inicial, o código prepara o programa para a entrada dos dados do usuário. O endereço de início do array, onde serão armazenados os números dos vagões, é carregado no registrador $t0. Este registrador atuará como um ponteiro para o array, facilitando o armazenamento e acesso aos elementos que serão inseridos.

Em seguida, o tamanho do array, que foi previamente definido como 8 e armazenado na palavra tamanho_array, é carregado no registrador $t1. Esse valor determina o número máximo de elementos que podem ser inseridos no array, garantindo que o usuário não insira mais números do que o array pode comportar.

Finalmente, o registrador $t2 é inicializado com o valor 0, atuando como um contador para acompanhar o número de elementos já inseridos no array. Este contador será utilizado para controlar o fluxo de entrada dos números e para assegurar que o programa só prossiga para a ordenação após todos os números terem sido inseridos ou até que o usuário decida encerrar a entrada de dados.

Como já foi falado anteriormente, este trecho do código estabelece a base para a interação com o usuário e a coleta dos dados necessários para a execução do algoritmo de ordenação, preparando o ambiente para que a lógica principal do programa possa ser implementada de forma eficiente e organizada.

```
1  .data
2  array:       .space 32
3  tamanho_array: .word 8
4  contador_trocas: .word 0
5  num_informado: .word 0
6
7  .text
8  .globl main
9
10 main:
11     # Exibe a mensagem inicial
12     li   $v0, 4
13     la   $a0, mensagem_inicial
14     syscall
15
16     # Carrega o endereço do array e o tamanho do array
17     la   $t0, array
18     lw   $t1, tamanho_array
19
20     # Solicita ao usuário que insira os elementos do array
21     addi $t2, $zero, 0
```

2.2. Entrada_Array:

Nessa parte da implementação, a rotina para a entrada dos dados do usuário é implementada. Inicialmente, o código verifica se o número de elementos inseridos ($t2) alcançou o tamanho máximo do array ($t1). Se o limite for atingido, o fluxo é direcionado para a seção de ordenação (bubble_sort).

Se ainda há espaço para mais elementos, o código exibe uma mensagem solicitando ao usuário que insira um número. Para isso, o valor 4 é carregado no registrador $v0, indicando a operação de impressão de string, e o endereço da mensagem (mensagem_prompt) é carregado no registrador $a0. O sistema então exibe a mensagem para o usuário.

Após a exibição da mensagem, o código lê o valor inserido pelo usuário, configurando $v0 para a operação de leitura de um inteiro (5). O valor lido é armazenado em $v0.

O próximo passo é verificar se o valor inserido é 0. Se for, o código salta para a seção de ordenação (bubble_sort), permitindo ao usuário encerrar a entrada de dados. Caso contrário, o valor é armazenado no array no local apontado por $t0, que serve como ponteiro para a posição atual do array.

Em seguida, o contador de números informados (num_informado) é incrementado para refletir o novo número inserido. O ponteiro do array ($t0) é atualizado para a próxima posição disponível e o contador de elementos inseridos ($t2) também é incrementado. O processo é então repetido, retornando ao início da seção de entrada de dados para continuar solicitando e armazenando números até que o usuário decida parar ou o array esteja completo.

```
23  entrada_array:
24      bge  $t2, $t1, bubble_sort
25
26      # Exibe mensagem para inserir um número
27      li   $v0, 4
28      la   $a0, mensagem_prompt
29      syscall
30
31      # Lê o valor do usuário
32      li   $v0, 5
33      syscall
34
35      # Verifica se o valor é 0
36      beq  $v0, $zero, bubble_sort
37
38      # Armazena o valor no array
39      sw   $v0, 0($t0)
40
41      # Incrementa o contador de números informados
42      lw   $t4, num_informado
43      addi $t4, $t4, 1
44      sw   $t4, num_informado
45
46      # Incrementa o ponteiro do array e i
47      addi $t0, $t0, 4
48      addi $t2, $t2, 1
49      j    entrada_array
50
```

2.3. Bubble_Sort:

Esse trecho e bem simples, apenas iniciando e configurando os registradores e as variáveis que serão usadas. Inicia-se a seção de ordenação utilizando o algoritmo Bubble Sort. O código começa carregando o endereço do array ($t0), onde os números dos vagões estão armazenados, e o número total de elementos inseridos ($t1) a partir da variável num_informado. O registrador $t2 é inicializado com o valor 0, atuando como o contador para o número de passes completos pelo array.

Além disso, o código configura o contador de trocas ($t3) que rastreará quantas vezes os elementos foram trocados durante o processo de ordenação. O valor 0 é carregado no registrador $t4 e, em seguida, armazenado na posição de memória correspondente ao contador de trocas. Isso inicializa o contador, garantindo que ele comece a partir de zero antes de iniciar a ordenação.

Esse trecho configura os registradores e as variáveis necessárias para o algoritmo Bubble Sort, preparando o código para executar a ordenação do array e contar o número de trocas realizadas.

```
51  bubble_sort:
52      # Inicia o Bubble Sort
53      la   $t0, array
54      lw   $t1, num_informado
55      addi $t2, $zero, 0
56
57      # Inicializa o contador de trocas
58      la   $t3, contador_trocas
59      li   $t4, 0
60      sw   $t4, 0($t3)
61
```

2.4. Loop_Externo:

Nessa parte, e bem simples, temos o início do Bubble Sort e seu loop externo, inicia-se o loop externo do algoritmo Bubble Sort. O código calcula a diferença entre o número total de elementos ($t1) e o contador de passes completos ($t2) e armazena o resultado em $t5. Em seguida, verifica se $t5 é menor ou igual a zero usando a instrução blez (branch on less than or equal to zero). Se a condição for verdadeira, significa que o array está ordenado e o código salta para a label fim_ordenacao, encerrando o processo de ordenação.

Se ainda há passes a serem feitos, o código inicializa o registrador $t6 com o valor 0. O registrador $t6 será usado como o índice para percorrer o array durante o loop interno, permitindo a comparação e a troca de elementos adjacentes.

Este trecho configura o loop externo para o Bubble Sort, que controla o número de passes pelo array, e define o índice inicial para o loop interno que realizará as comparações e trocas.

```
62  loop_externo:
63      sub  $t5, $t1, $t2
64      blez $t5, fim_ordenacao
65
66      addi $t6, $zero, 0
67
```

2.5. Loop_Interno:

Neste trecho do código, o loop interno do algoritmo Bubble Sort é implementado. A primeira instrução calcula o número de comparações restantes para o loop interno subtraindo o número de passes completos ($t2) do número total de elementos ($t1) e ajustando o resultado para considerar o índice atual. Esse valor é armazenado em $t7. A instrução bge (branch on greater than or equal) verifica se o índice atual ($t6) é maior ou igual a $t7. Se for, significa que todas as comparações necessárias foram feitas, e o fluxo do código é direcionado para proximo_externo, passando para o próximo passo do loop externo.

Dentro do loop interno, o código calcula o deslocamento necessário para acessar os elementos adjacentes no array. Isso é feito usando a instrução sll (shift left logical) para multiplicar o índice $t6 por 4 (tamanho de um inteiro em bytes). O endereço do elemento atual é obtido somando esse deslocamento ao endereço base do array ($t0), e é carregado no registrador $t9.

```
68  loop_interno:
69      sub  $t7, $t1, $t2
70      addi $t7, $t7, -1
71      bge  $t6, $t7, proximo_externo
72
73      sll  $t8, $t6, 2
74      add  $t9, $t0, $t8
75      lw   $s0, 0($t9)
76      lw   $s1, 4($t9)
77
78      ble  $s0, $s1, pular_troca
79
80      # Troca os valores
81      sw   $s1, 0($t9)
82      sw   $s0, 4($t9)
83
84      # Incrementa o contador de trocas
85      lw   $t4, 0($t3)
86      addi $t4, $t4, 1
87      sw   $t4, 0($t3)
88
```

Os valores dos dois elementos adjacentes são então carregados nos registradores $s0 e $s1. O código compara esses valores para decidir se uma troca é necessária. Se o valor de $s0 for menor ou igual ao valor de $s1, a instrução ble (branch on less than or equal) direciona o fluxo para pular_troca, ignorando a troca.

Se a troca for necessária, os valores são trocados entre os dois locais no array usando as instruções sw (store word). Após a troca, o contador de trocas ($t3) é incrementado. O valor atual do contador é carregado em $t4, incrementado em 1, e armazenado de volta na posição de memória do contador de trocas.

2.6. Pular_trocas e Loop_Exeterno:

Nessa parte do código, e fundamental para garantir que o algoritmo Bubble Sort continue a percorrer e ordenar o array até que todos os elementos estejam na ordem correta.

Quando o código chega ao rótulo pular_troca, significa que os elementos adjacentes no array estão na ordem correta e não precisam ser trocados. Portanto, o índice interno ($t6) é incrementado em 1 para mover para o próximo par de elementos a ser comparado. Em seguida, a instrução j (jump) redireciona o fluxo de execução de volta para o início do loop interno (loop_interno), onde a próxima comparação será realizada.

```
89  pular_troca:
90      addi $t6, $t6, 1
91      j    loop_interno
92
93  proximo_externo:
94      addi $t2, $t2, 1
95      j    loop_externo
96
97  fim_ordenacao:
98      # Exibe o array ordenado
99      la   $t0, array
100     addi $t2, $zero, 0
101
```

No rótulo proximo_externo, o código finalizou todas as comparações para o pass atual do loop externo. Portanto, o contador de passes completos ($t2) é incrementado em 1 para indicar que o loop externo deve avançar para o próximo pass. Após essa atualização, a instrução j redireciona o fluxo de volta para o início do loop externo (loop_externo), onde o próximo pass de Bubble Sort é iniciado.

2.7. Fim_ordenacao:

Neste trecho do código, o rótulo `fim_ordenacao` marca o início da fase de exibição dos resultados após a conclusão da ordenação do array. Aqui, o código está configurado para exibir os elementos do array ordenado.

Primeiro, o endereço base do array (`$t0`) é carregado novamente para começar a acessar os elementos ordenados. Em seguida, o contador de passes (`$t2`) é reiniciado para `0`, preparando-o para percorrer o array e exibir cada elemento.

A lógica completa para exibir o array e qualquer outra etapa pós-ordenamento continuaria a partir desse ponto, mas este trecho prepara o ambiente para começar a exibição dos resultados após a ordenação.

```
 97  fim_ordenacao:
 98      # Exibe o array ordenado
 99      la   $t0, array
100      addi $t2, $zero, 0
101
```

2.8. Imprimir_array:

Neste trecho, o rótulo imprimir_array exibe os elementos ordenados do array. O código verifica se todos os elementos foram exibidos comparando o contador de elementos ($t2) com o total ($t1). Se todos foram exibidos, o código vai para imprimir_contador_trocas. Caso contrário, calcula o endereço do próximo elemento e o exibe. Para separar os números, um espaço é adicionado após cada valor. O contador de elementos é então incrementado, e o processo continua até que todos os elementos sejam mostrados.

Nessa parte do código, por mais que não foi solicitado o trabalho pratico, achamos fundamental adicionar essa parte, para que vemos o vetor final, totalmente ordenado.

```
imprimir_array:
    bge  $t2, $t1, imprimir_contador_trocas

    sll  $t8, $t2, 2
    add  $t9, $t0, $t8
    lw   $s0, 0($t9)

    li   $v0, 1
    move $a0, $s0
    syscall

    li   $v0, 11
    li   $a0, 32
    syscall

    addi $t2, $t2, 1
    j    imprimir_array

```

2.9. Imprimir_array:

Neste trecho do código, o rótulo imprimir_contador_trocas trata da exibição do número total de trocas realizadas durante a ordenação do array. O processo começa configurando o registrador $v0 para a operação de exibição de uma string, que é indicada pelo código 4. O endereço da mensagem que informa sobre o número de trocas, armazenado em mensagem_trocas, é carregado no registrador $a0. Em seguida, o syscall é chamado para exibir essa mensagem ao usuário.

Depois de exibir a mensagem, o código carrega o número total de trocas realizadas a partir do endereço armazenado no registrador $t3. Este valor é carregado no registrador $a0, que é utilizado pelo syscall para exibir um inteiro. O registrador $v0 é configurado para a operação de exibição de um inteiro, que é identificada pelo código 1. Finalmente, o syscall é chamado novamente para exibir o número total de trocas realizadas.

Esse trecho é responsável por fornecer ao usuário a contagem total de trocas feitas pelo algoritmo Bubble Sort, após a ordenação completa do array.

```
120 imprimir_contador_trocas:
121     # Exibe o contador de trocas
122     li   $v0, 4
123     la   $a0, mensagem_trocas
124     syscall
125
126     lw   $a0, 0($t3)
127     li   $v0, 1
128     syscall
129
```

2.10. Imprimir_Contador_Trocas:

Neste trecho do código, o rótulo imprimir_contador_trocas trata da exibição do número total de trocas realizadas durante a ordenação do array como foi solicitado no trabalho prático. O processo começa configurando o registrador $v0 para a operação de exibição de uma string, que é indicada pelo código 4. O endereço da mensagem que informa sobre o número de trocas, armazenado em mensagem_trocas, é carregado no registrador $a0. Em seguida, o syscall é chamado para exibir essa mensagem ao usuário.

Depois de exibir a mensagem, o código carrega o número total de trocas realizadas a partir do endereço armazenado no registrador $t3. Este valor é carregado no registrador $a0, que é utilizado pelo syscall para exibir um inteiro. O registrador $v0 é configurado para a operação de exibição de um inteiro, que é identificada pelo código 1. Finalmente, o syscall é chamado novamente para exibir o número total de trocas realizadas.

Esse trecho é responsável por fornecer ao usuário a contagem total de trocas feitas pelo algoritmo Bubble Sort, após a ordenação completa do array.

```
120 imprimir_contador_trocas:
121     # Exibe o contador de trocas
122     li   $v0, 4
123     la   $a0, mensagem_trocas
124     syscall
125
126     lw   $a0, 0($t3)
127     li   $v0, 1
128     syscall
129
```

2.11. Fim_Programa:

Por fim, temos o código que inclui o rótulo fim_programa e a seção de dados, o rótulo fim_programa é responsável por encerrar a execução do programa. Para isso, o código configura o registrador $v0 com o valor 10, que é o código para a operação de término de programa. Em seguida, o syscall é chamado para finalizar a execução e retornar ao sistema operacional.

Na seção de dados, são definidas três mensagens que são usadas em diferentes partes do programa. A mensagem_prompt solicita ao usuário que insira um número de vagão e é exibida para guiar o usuário na entrada de dados. A mensagem_trocas é exibida após a ordenação do array, mostrando o número total de trocas realizadas durante o processo de Bubble Sort, com uma descrição que antecede o valor.

Por fim, a mensagem_inicial fornece informações ao usuário sobre o limite de vagões, que é 8, e orienta que, se o usuário desejar encerrar a entrada de dados, deve digitar ‘0’. Essas mensagens são essenciais para a interação do usuário com o programa e para fornecer feedback sobre as operações realizadas.

```
130 fim_programa:
131     li   $v0, 10
132     syscall
133
134 .data
135 mensagem_prompt: .asciiz "Insira um número do vagão: "
136 mensagem_trocas: .asciiz "\nNúmero de trocas de vagões: "
137 mensagem_inicial: .asciiz "O limite de vagões são '8'.\nInforme '0' caso já tenha informado todas os vagões.\n "
138
```

3. Conclusão:

A conclusão deste trabalho prático envolveu a implementação de uma rotina em Assembly-MIPS para ordenar vagões desordenados e calcular o número mínimo de trocas de vagões adjacentes necessárias para alcançar a ordenação. Utilizando o algoritmo de Bubble Sort, que é relativamente simples e apropriado para este tipo de tarefa, o código foi desenvolvido para ler a entrada do usuário, processar a lista de vagões e realizar a ordenação.

O programa foi projetado para solicitar ao usuário a entrada dos números dos vagões, armazená-los em um array, e então aplicar o algoritmo de Bubble Sort para ordená-los. Durante a ordenação, o código conta o número de trocas realizadas, fornecendo um indicador da eficiência do processo. Após a conclusão da ordenação, o programa exibe tanto o array ordenado quanto o número total de trocas realizadas.

A realização deste exercício permitiu uma compreensão mais profunda dos conceitos fundamentais da programação em Assembly, como manipulação de registros e controle de fluxo. Além disso, destacou a importância e a aplicabilidade dos conhecimentos adquiridos na resolução de problemas práticos, como a organização eficiente de elementos. A experiência proporcionou uma visão prática sobre como técnicas de baixo nível pode ser aplicadas para resolver problemas reais e otimizar processos.

Ao implementar o Bubble Sort em Assembly foi desafiador devido à necessidade de gerenciar diretamente registros e memória. Embora o Bubble Sort seja um algoritmo relativamente simples em termos de lógica, a implementação em Assembly exigiu uma manipulação cuidadosa dos índices e trocas entre elementos, da mesma forma, que quando aumentamos o tamanho do array, mais difícil fica a implementação do código.

Cada comparação e troca precisou ser gerenciada manualmente, o que era propenso a erros e demandava atenção detalhada ao controle de fluxo e armazenamento de dados. Enfrentei dificuldades adicionais ao tentar otimizar o código para minimizar o número de instruções e evitar acessos incorretos à memória. Essa experiência me fez perceber a complexidade de trabalhar com Assembly e a importância de entender bem a arquitetura do processador e as operações de baixo nível.

4. Referência:

As imagens utilizadas no trabalho, foram retiradas do link abaixo:

https://github.com/Joao-Dutra/AOC-TrabalhoPratico3