UNIVERSIDADE FEDERAL DE OURO PRETO

CIÊNCIA DA COMPUTAÇÃO

RELATÓRIO DO TRABALHO PRÁTICO

Integrante:

Andrei Miguel

Yann Eduardo de Souza Silva

Ouro Preto

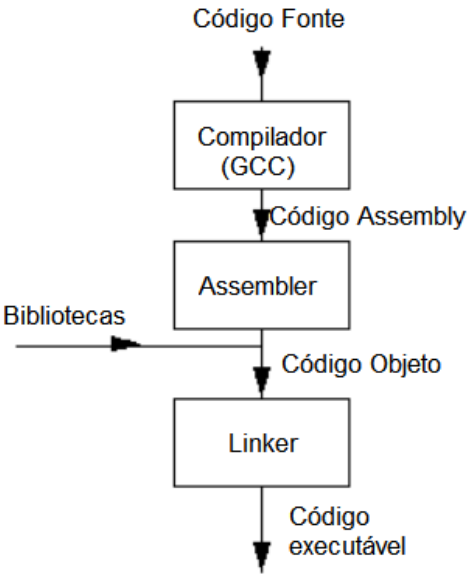
2024

Algoritmo de Ordenação em Mips

I - Introdução

Antes de iniciarmos o trabalho em si, é importante compreender os passos pelos quais o código digitado pelo programador chega até o computador. Os códigos escritos em linguagens de alto nível, como C, Java ou Python, são designados dessa forma para serem mais próximos da linguagem humana do que da linguagem de máquina. Isso permite que os programadores escrevam instruções de forma mais intuitiva e fácil de entender. Após escrever o código em linguagem de alto nível, o programador utiliza um compilador (como o GCC, por exemplo), que traduz o código para assembly, uma linguagem de baixo nível. Em seguida, o Assembler interpreta o código assembly e, com o auxílio de bibliotecas, gera um código objeto, composto por uma sequência de 0s e 1s correspondente ao programa. Após a montagem do código objeto, o Linker reúne todos os códigos objetos do programa, caso haja mais de um, e cria um arquivo compatível (.exe). A Figura 1 ilustra esse processo passo a passo.

Figura 1: Código Fonte até linguagem de máquina



A linguagem de máquina apresenta um conjunto de instruções diretamente executadas pelo processador de um computador. Cada arquitetura de processador possui sua própria linguagem de máquina, composta por um conjunto específico de instruções que o processador pode entender e executar. As instruções em linguagem de máquina são normalmente representadas por códigos numéricos binários, que exigem operações específicas, como adição, subtração, transferência de dados, etc.

Desse modo, o objetivo deste trabalho é construir os algoritmos de ordenação Insertion Sort e Selection Sort em linguagem de baixo nível, os quais ordenam um vetor com tamanho igual a 10. Para isso, o software utilizado para a organização dos métodos de ordenação em linguagem de baixo nível foi software MARS (MIPS Assembly and Runtime Simulator) que é uma ferramenta de desenvolvimento amplamente utilizada para

programação em assembly MIPS, uma vez que, ele fornece um ambiente de simulação onde podemos escrever, compilar e executar programas escritos em assembly MIPS. Já o MIPS (Microprocessor Without Interlocked Pipeline Stages) é uma arquitetura de conjunto de instruções comumente encontrada em CPUs usadas em sistemas embarcados, como roteadores, consoles de videogame e dispositivos de rede. Em resumo, o MARS é um software que permite escrever e executar programas em assembly MIPS, enquanto a linguagem MIPS é uma linguagem de baixo nível composta por um conjunto de instruções binárias entendidas pelo processador MIPS. O MARS facilita o desenvolvimento e a depuração de programas em montagem MIPS, fornecendo um ambiente de simulação intuitivo e recursos de desenvolvimento robustos.

Das disposições do trabalho, a seção II possui o referencial teórico usado na solução dos dois algoritmos, a seção III aborda todos os procedimentos da construção do emulador e as métricas avaliativas dos testes, a seção IV traz os resultados dos testes, a seção V mostra uma síntese geral de todo o conteúdo apresentado.

II - Referencial Teórico

A linguagem de baixo nível MIPS é projetada para ser executada em especial MIPS e é composta por um conjunto de instruções que operam em registradores e na memória do sistema. Ao escrever em assembly MIPS interagimos diretamente com instruções, fornecendo as sequências de instruções binárias que realizam operações desejadas, como manipulação de dados, controle de fluxo e acesso à memória.

Para desenvolver os algoritmos, foram utilizados 21 instruções de montagens MIPS:

.DATA – Defina uma seção de dados onde as variáveis globais e constantes são declaradas.

.ASCIIZ "STRING" - Defina uma sequência de caracteres ASCII, terminada por zero.

.TEXT - Define uma seção de texto onde o código do programa está escrito.

.SYSCAL - Indica uma chamada do sistema para realizar uma operação específica. Exemplo:

```
li $v0, 4       # Código de serviço para imprimir string
la $a0, mensagem  # Endereço da string a ser impressa
.syscall        # Chamada do sistema para imprimir a string
```

Neste exemplo, o `.syscall` é usado após configurar o registrador `$v0` com o código de serviço adequado para imprimir uma string na tela. Depois que o `.syscall` sistema for executado, o sistema realizará a operação correspondente, que, neste caso, é impressa na string fornecida.

ADD $DESTINO, $FONTE1, $FONTE2 - É utilizada para realizar a adição de dois

números. Exemplo: `add $t0, $t1, $t2` realize a operação `$t0 = $t1 + $t2`.

BGT $OPERANDO2, $OPERANDO1, ETIQUETA - Realiza um salto condicional se o primeiro operando para maior que o segundo. Exemplo: `bgt $t0, $t1, label1` irá pular para `label1` se o valor contido em `$t0` for maior que o valor contido em `$t1`.

ADDI $DESTINO, $FONTE, VALOR - Realiza a adição de um valor imediato (constante) a um registrador. Exemplo: `addi $t0, $t1, 5` realiza a operação `$t0 = $t1 + 5`, adicionando o valor imediato 5 ao valor contido em `$t1`.

LI $DESTINO, IMEDIATO - Carrega um valor imediatamente em um registrador. Exemplo: `li $s2, 10` carrega o valor imediatamente 10 no registrador `$s2`.

SLL $DESTINO,$REGISTRADORDEORIGEM,SHIFT_AMOUNT - Desloca os bits de um registrador para a esquerda um número específico de vezes. Exemplo: `sll $s0, $v0, 2` desloca os bits do registrador `$v0` para a esquerda duas vezes e armazena o resultado em `$s0`.

J TARGET - Salta para a instrução especificada pelo rótulo `target`.Exemplo: `j insertion_laço_j` é usado para saltar de volta para o início do loop `insertion_laço_j`.

LA $DESTINO, LABEL - Carrega o endereço de memória de um símbolo (geralmente uma variável ou um rótulo) em um registrador. Exemplo: `la $t0, array` é usado para carregar o endereço inicial do array na memória no registrador `$t0`, permitindo o acesso aos elementos do array.

MUL $DEST $RGO1 $RGO2 - Multiplica os valores armazenados em `$RGO1` e `$RGO2` armazena o resultado em `$dest`, Exemplo: `mul $t5, $t3, 4` é usado para multiplicar o valor em `$t3` por 4. O resultado é armazenado em `$t5`.

SUB $DEST $RGO1 $RGO2 - Subtrai o segundo operando do primeiro e armazena o resultado no primeiro operando. Exemplo: `sub $sp, $sp, $s0` subtrai o valor do registrador `$s0` do valor do registrador `$sp` e armazena o resultado em `$sp`.

MOVE $DEST, $RGO - Mover o conteúdo de um registrador para outro. Exemplo: `move $s1, $zero` mova o valor do registrador `$zero` para o registrador $s1.

SW $RGO, OFFSET($BASE) - Armazena uma palavra da memória em um endereço especificado. Exemplo: `sw $a0, 0($t1)` armazena o valor do registrador `$a0` no endereço calculado pela soma de `$t1` com o valor imediato 0.

JAL TARGET - Realiza um salto para a etiqueta especificada e armazena o endereço de retorno no registrador `$ra`. Exemplo:`jr $ra` realize um salto para o endereço armazenado no registrador `$ra`.

BGE $RGO1, $RGO2, LABEL - Salta para a etiqueta especificada se o primeiro operando for maior ou igual ao segundo operando. Exemplo: `bge $s1, $s2, saída_display` salta para a etiqueta.

LW $DEST $OFFSET($BASE) - Carrega uma palavra da memória para um registrador. Exemplo: `lw $t7, 0($t6)` carrega o valor armazenado no endereço calculado pela soma de `$t6` com o valor imediato 0 para o registrador `$t7`.

BLE $RGO1 $RGO2 - Realiza um salto condicional se o primeiro operando for menor ou igual ao segundo operando. Exemplo: `ble $t0, $t1, loop`, salta para a etiqueta 'loop' se o valor em `$t0` for menor ou igual ao valor em `$t1`

SUBI $DEST $RGO $IMEDIATO - Subtrai um valor imediatamente de um registrador e armazena o resultado no registrador de destino. Exemplo: `subi $t0, $t0, 1`. Subtrai 1 do valor em `$t0` e armazena o resultado em `$t0`.

JR $REGISTRADOR - Realiza um salto para o endereço armazenado em um registrador. Exemplo: `jr $ra`. Salta para o endereço armazenado no registrador `$ra`.

O algoritmo de ordenação Selection Sort é um método simples e intuitivo para ordenar uma lista de elementos. Funciona da seguinte maneira:

1. **Seleção do menor(ou maior) elemento:** O algoritmo percorre a lista em busca do menor elemento e o posiciona na primeira posição.
2. **Seleção do segundo menor elemento:** Em seguida, ele continua a percorrer a lista, procurando o segundo menor elemento e o posicionando na segunda posição.
3. **Repetição:** Este processo é repetido até que todos os elementos estejam ordenados.

Estabilidade: O Selection Sort não é um algoritmo estável. Isso significa que, se houver elementos iguais, a ordem relativa entre eles pode não ser preservada após a ordenação.
In-situ: O Selection Sort é um algoritmo in-situ, o que significa que ele não usa memória extra além daquela usada para armazenar a lista original. Ele trabalha reorganizando os elementos dentro da própria lista, sem exigir espaço adicional.
Complexidade de tempo: O Selection Sort tem uma complexidade de tempo de O(n^2), onde 'n' é o número de elementos na lista. Isso ocorre porque, para cada elemento na lista, o algoritmo precisa percorrer toda a lista novamente para encontrar o próximo menor elemento. Assim, o tempo de execução aumenta quadraticamente com o tamanho da lista. Embora seja simples de implementar e entender, o Selection Sort não é eficiente para grandes conjuntos de dados devido à sua alta complexidade de tempo.

Já o algoritmo de ordenação Insertion Sort é outro método simples e eficaz para ordenar uma lista de elementos. Funciona da seguinte maneira:

1. **Partição inicial:** O algoritmo divide a lista em duas partes: uma parte ordenada e uma parte não ordenada.

2. **Iteração:** Ele então percorre a parte não ordenada da lista, um elemento de cada vez.
3. **Inserção:** Para cada elemento na parte não ordenada, ele é comparado com os elementos na parte ordenada.
    a. Se o elemento for menor que o elemento atualmente sendo comparado, ele é inserido na posição correta na parte ordenada da lista, movendo os elementos maiores para a direita.
    b. Se o elemento for maior ou igual, ele é deixado no lugar e o próximo elemento na parte não ordenada é considerado.

4. **Repetição:** Este processo é repetido até que todos os elementos estejam na parte ordenada da lista.

Estabilidade: O Insertion Sort é um algoritmo estável. Isso significa que, se houver elementos iguais, a ordem relativa entre eles será preservada após a ordenação.
In-situ: Assim como o Selection Sort, o Insertion Sort também é um algoritmo in-situ. Ele reorganiza os elementos dentro da própria lista, sem exigir espaço adicional.
Complexidade de tempo: O Insertion Sort tem uma complexidade de tempo de O(n^2) no pior caso e no caso médio, onde 'n' é o número de elementos na lista. Isso ocorre porque, para cada elemento na parte não ordenada, o algoritmo precisa compará-lo com os elementos na parte ordenada, o que resulta em uma quantidade quadrática de comparações e movimentos. No entanto, para listas quase ordenadas ou pequenas, o Insertion Sort pode ser mais eficiente do que outros algoritmos de ordenação devido à sua natureza de percorrer a lista uma vez e inserir elementos no lugar correto.

III. Metodologia

Utilizamos do método insertion sort e selection sort para ordenar os vetores. Em um dos códigos andamos no vetor usando 2 shifts para a esquerda, já no outro só incrementamos 4 a cada iteração do programa.
Tentamos padronizar o código deixando o mais parecido possível e seguindo a mesma linha de raciocínio neles, o que mais mudou mesmo foram somente as funções de ordenação. Agora fique com alguns trechos do código comentado:

Selection

```
sll $t0, $s1, 2          # DA DOIS SHIFTS PARA A ESQUERDA PARA PULAR 4 ENDEREÇOS          -> ($t0 = i * 4) <-
add  $t1, $t0, $sp       # SOMA O $t0 MAIS O $sp PARA ANDAR DE 4 EM 4 NO ENDEREÇO ATUAL  -> ($t1 = $sp + $t0) <-
li  $a0, 5               # LÊ O NÚMERO (INT) FORNECIDO PELO USUÁRIO
sw $a0, 0($t1)           # ARMAZENA O NÚMERO NO ENDEREÇO $t1 DO VETOR
addi  $s1, $s1, 1        #i++
sll $t0, $s1, 2          # DA DOIS SHIFTS PARA A ESQUERDA PARA PULAR 4 ENDEREÇOS          -> ($t0 = i * 4) <-
add  $t1, $t0, $sp       # SOMA O $t0 MAIS O $sp PARA ANDAR DE 4 EM 4 NO ENDEREÇO ATUAL  -> ($t1 = $sp + $t0) <-
li  $a0, 8               # LÊ O NÚMERO (INT) FORNECIDO PELO USUÁRIO
sw $a0, 0($t1)           # ARMAZENA O NÚMERO NO ENDEREÇO $t1 DO VETOR
addi  $s1, $s1, 1        #i++
```

Primeiro damos dois shifts para andar no vetor a cada iteração e somamos o sp atual com o que está sendo deslocado, após isso lemos o número que iremos adicionar no vetor e no Store Word (sw) armazenamos ele no vetor e incrementamos mais um no i.
Trecho responsável por ler o array fixado na criação do código.

```
li   $s2, 10                  # ARMAZENA O TAMANHO DO VETOR EM $s2          -> ($s2 = n) <-
sll  $s0, $v0, 2              # DA DOIS SHIFTS PARA A ESQUERDA              -> ($s0 = n * 4) <-
sub $sp, $sp, $s0             # CRIA UM ESPAÇO PARA ARMAZENAR O VETOR       -> ($sp - s0) <-
```

Trecho que libera o espaço para o array já que não foi liberado antes, como em um dos códigos que liberou espaço automaticamente no .data;

```
min:
      move $t0, $a0           # INICIO DO ARRAY = PRIMEIRA POSIÇÃO
      move $t1, $a1           # MIN_INDEX COMEÇA COMO SENDO O i = COMEÇO
      move $t2, $a2           # FINAL DO ARRAY = FINAL

      sll  $t3, $t1, 2        # SHIFT PARA A ESQUERDA                                  -> (MIN * 4) <-
      add   $t3, $t3, $t0     # ENDEREÇAMENTO IGUAL INICIO DO ARRAY MAIS A POSIÇÃO QUE VOCÊ ANDOU
      lw  $t4, 0($t3)         # MIN_ELEMENTO É IGUAL O PRIMEIRO ELEMENTO               -> (MIN = v[first]) <-

      addi  $t5, $t1, 1       # INICIALIZA O i IGUAL 0
min_laço:
      bgt   $t5, $t2, min_fim # ENCERRA A FUNÇÃO

      sll  $t6, $t5, 2        # DOIS SHIFTS PARA A ESQUERDA -> (i * 4) <-
      add   $t6, $t6, $t0     # ENDEREÇAMENTO IGUAL O INICIO DO VETOR MAIS A QUANTIDADE QUE ANDOU NO ARRAY -> (inicio do a
      lw  $t7, 0($t6)         # V[ENDEREÇAMENTO]
      bge   $t7, $t4, min_if_saída  # PULA PARA A SAÍDA DO IF SE  v[i] FOR MAIOR OU IGUAL O MIN
      move $t1, $t5           # ATRIBUI O i COMO SENDO O MIN  -> (min = i) <-
      move $t4, $t7           # ATRIBUI O V[i] COMO SENDO O MIN  -> (min = V[i]) <-
```

Parte mais importante e elaborada de um dos programas pois retorna a posição de onde está o menor elemento parte crucial no selection sort, pois o algoritmo funciona tendo isso como a base de sua ordenação.
O selection irá ordenar do menor elemento para o maior, diferentemente do bubble e por isso a importância desse trecho.

Insertion

```
array: .word 0 : 200          # um array de palavras, para armazenar valores.
```

Alocação estática de memória, diferentemente do código anterior.

```
la $t0, array                 # carrega o array em $t0.
lw $t1, tam                   # carrega o tamanho em $t1.

li $a0, 5                     # 5 = Número q será adicionado no array.
sw $a0, 0($t0)                # armazena o valor no array.
addi $t0, $t0, 4              # incrementa o ponteiro do array em 4.

li $a0, 2                     # 2 = Número q será adicionado no array.
sw $a0, 0($t0)                # armazena o valor no array.
addi $t0, $t0, 4              # incrementa o ponteiro do array em 4.
```

Leitura dos números e o armazenamento dos mesmos, diferente do programa anterior o vetor é percorrido apenas com o incremento de 4 no ponteiro inicial a cada iteração, mostrando diferentes maneiras de percorrer um vetor/array.

```
insertion_laço_i:
    la  $t0, array                      # carrega o array em $t0.
    bge $t2, $t1, insertion_laço_i_end  # enquanto (t2 < $t1).
    move $t3, $t2                       # copia o valor de $t2 para $t3.

insertion_laço_j:
    la  $t0, array                      # carrega o array em $t0.
    mul $t5, $t3, 4                     # multiplica $t3 por 4 e armazena em $t5.
    add $t0, $t0, $t5                   # adiciona o endereço do array com $t5, que é o índice multiplicado por 4.
    ble$t3, $zero, insertion_laço_j_end # enquanto (t3 > 0).
    lw $t7, 0($t0)                      # carrega array[$t3] em $t7.
    lw $t6, -4($t0)                     # carrega array[$t3 - 1] em $t6.
    bge $t7, $t6, insertion_laço_j_end  # enquanto (array[$t3] < array[$t3 - 1]).
    lw $t4, 0($t0)
    sw $t6, 0($t0)
    sw $t4, -4($t0)
    subi $t3, $t3, 1
    j insertion_laço_j                  # salta de volta para o início de insertion_laço_j.
```

Basicamente esse trecho faz a mesma coisa do algoritmo em C, ele percorre o vetor a cada iteração e a cada passo ele faz a verificação de onde o elemento deverá ser encaixado, enquanto ele for menor que o elemento comparado o algoritmo irá trocando os dois até chegar na condição de parada (elemento maior que elemento que o ponteiro do laço estiver apontando).

```
display:
    la $t0, array
    lw $t1, tam

display_laço:
    bge $t2, $t1, display_saída

    lw $a0, 0($t0)
    syscall

    li $v0, 4
    la $a0, array3
    syscall

    addi $t0, $t0, 4
    addi $t2, $t2, 1

    j display_laço

display_saída:

    li $v0, 4
    la $a0, array2
    syscall
    jr $ra
```

E por final, basicamente o um laço que vai printar todo o vetor com base no tamanho.

IV. Resultados

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	0
$v1	3	0
$a0	4	5
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	0

Armazena um número no reg $a0

Value (+1c)
5

Salva o número que está no reg auxiliar no começo do vetor

$zero	0	0
$at	1	0
$v0	2	0
$v1	3	0
$a0	4	8
$a1	5	0
$a2	6	0
$a3	7	0

Armazena um número no reg $a0

Value (+0)
0
8

Salva o número que está no reg auxiliar na próxima posição do vetor

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	0
$v1	3	0
$a0	4	2

Armazena um número no reg $a0

Value (+4)
0
2

Salva o número que está no reg auxiliar na próxima posição do vetor

E assim continua a execução do código até acabar o armazenamento do vetor, o que resulta nesse array (Após ordenado):

0	0	4194556	0	10	10	0	1
2	3	4	5	6	7	8	9
10	0	0	0	0	0	0	0

|1|2|3|4|5|6|7|8|9|10|

Fizemos vários testes com diferentes arrays todos eles deram exatamente o planejado.

Alguns testes realizados:

```
Vetor Ordenado:
10 12 21 39 40 52 64 79 81 95
```

```
Vetor Ordenado:
3 22 39 42 51 56 83 90 104 112
```

```
Vetor Ordenado:
0 21 32 33 36 53 66 72 99 104
```

```
Vetor Ordenado:
1 7 9 9 34 45 59 66 67 104
```

```
Vetor Ordenado:
1 1 2 2 3 5 7 8 9 10
```

V. Conclusão

Concluímos, através deste trabalho, a implementação e análise do algoritmo de ordenação por Insertion Sort e Selection Sort em linguagem assembly MIPS. Nosso objetivo foi compreender e demonstrar o funcionamento deste algoritmo em um contexto de programação de baixo nível, fornecendo uma base sólida para compreensão das estruturas e operações fundamentais dessa arquitetura.

Na metodologia adotada, começamos definindo variáveis e arrays necessários para armazenar valores e controlar o fluxo do programa. Em seguida, implementamos o algoritmo de ordenação por Insertion Sort e Selection Sort, demonstrando passo a passo como ele organiza os elementos em ordem crescente. Através da análise de cada linha de código, exploramos conceitos como cabeamento de endereços, manipulação de registradores e instruções de controle de fluxo.

Os resultados obtidos revelaram a correta ordenação dos elementos no array, conforme esperado pelo algoritmo de inserção. No entanto, observamos que, embora o algoritmo de ordenação por Insertion Sort e Selection Sort tenha sido corretamente

implementado, sua eficiência pode estar comprometida em grandes conjuntos de dados devido à sua complexidade de tempo quadrática. Portanto, recomendamos considerar algoritmos de ordenação mais eficientes, como o mergesort ou quicksort, para lidar com conjuntos de dados maiores e melhorar o desempenho do programa em contextos de produção.

Em geral, este trabalho proporcionou uma valiosa experiência de aprendizado sobre programação em montagem MIPS e algoritmos de ordenação. A análise crítica dos resultados destacou a importância da escolha adequada de algoritmos para diferentes cenários e a necessidade contínua de buscar otimizações para garantir a eficiência e a escalabilidade dos programas desenvolvidos em linguagem assembly.