

UNIVERSIDADE FEDERAL DE MINAS GERAIS

DEPARTAMENTO DE ENGENHARIA ELETRÔNICA



ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

Pedro Henrique de Menezes Cosme - 2021075677

Kaique V. Santos Silva - 2022061351

28 de maio de 2024

Trabalho prático de Arquitetura e Organização de Computadores

Median of 1D array using functions in C

28 de maio de 2024

1 Introdução

A **mediana** é uma medida estatística que representa o valor central de um conjunto de dados ordenados. Ao contrário da média, que pode ser influenciada por valores extremos, a mediana proporciona uma representação mais robusta do centro dos dados, especialmente em distribuições assimétricas.

Para encontrar a mediana de um conjunto de dados unidimensional (array 1D), os passos básicos são:

1. **Ordenar os dados:** Colocar todos os elementos do array em ordem crescente.
2. **Determinar o valor central:**
 - Se o número de elementos (n) for ímpar, a mediana é o elemento central do array ordenado.
 - Se n for par, a mediana é a média dos dois elementos centrais do array ordenado.

Exemplo

Considere o seguinte array de inteiros: [3, 1, 4, 1, 5, 9, 2]

1. **Ordenando os dados:** [1, 1, 2, 3, 4, 5, 9]
2. **Valor central:** Como há 7 elementos (número ímpar), a mediana é o quarto elemento, que é 3.

Se o array tivesse um número par de elementos, por exemplo, [3, 1, 4, 1, 5, 9]:

1. **Ordenando os dados:** [1, 1, 3, 4, 5, 9]
2. **Valor central:** Como há 6 elementos, a mediana é a média dos dois elementos centrais:

$$\text{Mediana} = \frac{3 + 4}{2} = 3.5$$

Para realizar o cálculo da mediana em C usando funções foi utilizado o código fornecido pelo *link* de referência em Find Median of 1D Array Using Functions in C. Algumas correções foram realizadas no código, como a remoção da biblioteca **conio.h**, que é desnecessária para a execução do programa e correção de indentação. O código pode ser conferido abaixo:

```

#include <stdio.h>
#include <stdlib.h>

#define N 10

int main() {
    int i, j, n;
    float mediana, a[N], t;

    // Quantidade de itens no array definido pelo usu rio
    printf("Quantidade_de_itens:_");
    scanf("%d", &n);

    // Lendo os itens no array a
    printf("Insira_os_valores:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%f", &a[i]);
    }

    // Ordenando os valores
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (a[j] >= a[j + 1]) { // Trocando valores
                t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t;
            }
        }
    }

    // Mediana
    if (n % 2 == 0) {
        mediana = (a[n / 2 - 1] + a[n / 2]) / 2.0;
    } else {
        mediana = a[n / 2];
    }

    // Print dos valores
    printf("Valores_ordenados:\n");
    for (i = 0; i < n; i++) {
        printf("%f_", a[i]);
    }
    printf("\n_Mediana:_%f\n", mediana);

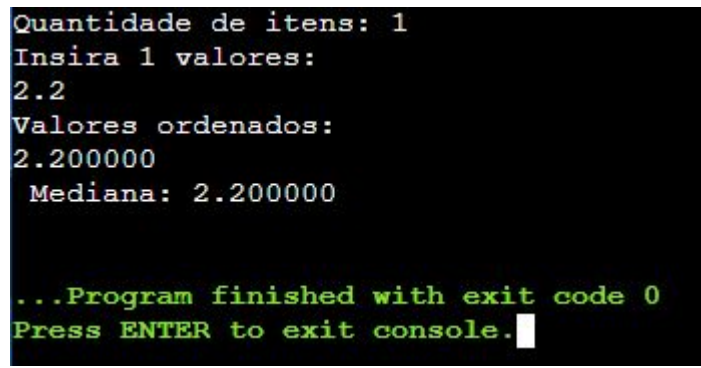
    return 0;
}

```

}

2 Desenvolvimento

A partir do código em C utilizado como base para o trabalho, realizamos três testes que serviriam como parâmetro de correção para o código em RISCV32. Os testes foram os seguintes:



```
Quantidade de itens: 1
Insira 1 valores:
2.2
Valores ordenados:
2.200000
Mediana: 2.200000

...Program finished with exit code 0
Press ENTER to exit console.
```

Figura 1: Primeiro Teste - Array de Um Espaço

Nesse primeiro teste, utilizamos um array de um único espaço, a fim de atestar se o código era capaz de identificar que, havendo apenas um valor, esse próprio valor seria a mediana. As entradas, portanto, foram:

Quantidades de Itens: 1
Valores Inseridos: 2.2

E a saída esperada seria:

Valores Ordenados: 2.2
Mediana: 2.2

Como visto na Figura 1, a saída do código condiz com o que era esperado, fazendo com que o primeiro teste seja bem-sucedido.

```

Quantidade de itens: 7
Insira 7 valores:
3.3
5.5
6.6
8.3554
775.3
0.8
09.76
Valores ordenados:
0.800000 3.300000 5.500000 6.600000 8.355400 9.760000 775.299988
Mediana: 6.600000

...Program finished with exit code 0
Press ENTER to exit console.

```

Figura 2: Segundo Teste - Array Ímpar

No segundo teste, usamos um array de tamanho ímpar, especificamente de sete valores; visando observar se o código seria capaz de selecionar o valor mediano correto para esse tipo de array. As entradas foram:

Quantidades de Itens: 7
Valores Inseridos: 3.3 5.5 6.6 8.3554 775.3 0.8 09.76

E a saída esperada seria:

Valores Ordenados: 0.8 3.3 5.5 6.6 8.3554 9.76 775.3
Mediana: 6.6

Comparando com a Figura 2, temos, novamente; um retorno adequado ao que se esperava.

```

Quantidade de itens: 8
Insira 8 valores:
34.12
4
8.9
5.55
3.6
0.01
3434.55
45.1
Valores ordenados:
0.010000 3.600000 4.000000 5.550000 8.900000 34.119999 45.099998 3434.550049
Mediana: 7.225000

...Program finished with exit code 0
Press ENTER to exit console.

```

Figura 3: Terceiro Teste - Array Par

No terceiro e último teste foi utilizado um array de tamanho par, nesse caso de oito valores, a fim de determinar se o código seria, ou não, capaz de determinar a mediana fazendo a média entre os dois valores centrais. Foram utilizadas as seguintes entradas:

Quantidades de Itens: 8
Valores Inseridos: 34.12 4 8.9 5.55 3.6 0.01 3434.55 45.1

E o retorno esperado é:

Valores ordenados: 0.01 3.6 4.0 5.55 8.9 34.12 45.1 3434.55
Mediana: 7.225

Finalmente, com o terceiro teste tendo êxito, como visto ao observarmos a Figura 3; podemos dizer que o código é capaz de operar corretamente sobre os três tipos de condições sem problemas notáveis.

Com os testes já realizados, é adequado agora que apresentemos o código em RISC-V32 feito a partir de sua contraparte em C, previamente apresentada. O código, já comentado, é o seguinte:

```

.data
.align 2          # Align the following data to a word boundary (4 bytes)
str:
    .string "Insira_os_valores\n"  # String to prompt user for float input
space_char:
    .string "_"  # String for a space character
array_size:

```

```

        .string "Quantidade_de_itens:_\n" # String to prompt user for array size
median_float:
        .string "\nMediana:_" # String to print before the median value
order_values:
        .string "Valores_ordenados:\n"
array:
        .word 4000 # Allocate space for the array, enough for 1000 floats (4 bytes each)

.text
.globl main

main:
        la a1, array # Load the address of the array into a1

        la a0, array_size # Load the address of the array_size string into a0
        li a7, 4 # System call number for print string
        ecall # Execute system call

        li a7, 5 # System call number for read integer from user
        ecall # Execute system call
        li t5, 1
        beq a0, t5, one_space_array # If the array size is 1, jump to one_space_array
        mv t2, a0 # Move the input (array size) from a0 to t2

        li t0, 0 # Initialize the index of the array to 0

        la a0, str # Load the address of the string into a0
        li a7, 4 # System call number for print string
        ecall # Execute system call

# Input loop for reading floats from the user
input_loop:

        li a7, 6 # System call number for read float from user
        ecall # Execute system call

        fsw fa0, 0(a1) # Store the float into the array at the current index

        addi a1, a1, 4 # Move to the next float space in the array
        addi t0, t0, 1 # Increment the index
        blt t0, t2, input_loop # Loop back if more inputs are needed

# Reset the array address for sorting
        la a1, array

```



```

        li s1, 0          # Initialize outer loop counter

addi t2, t2, -1          # Force the array being the wanted size

# Outer loop for bubble sort
outer_loop:
    li s2, 0              # Initialize inner loop counter
    la a0, array           # Load address of array into a0

# Inner loop for bubble sort
inner_loop:
    addi a3, a0, 4         # Load address of next float into a3
    flw fa1, 0(a0)         # Load current float into fa1
    flw fa2, 0(a3)         # Load next float into fa2
    flt.s t1, fa1, fa2     # Compare fa1 and fa2 (fa1 < fa2)
    bne t1, zero, no_swap  # No swap needed if fa1 < fa2

    # Swap fa1 and fa2
    fsw fa2, 0(a0)
    fsw fa1, 0(a3)

no_swap:
    addi a0, a0, 4         # Move to the next float in the array
    addi s2, s2, 1         # Increment the inner loop counter
    blt s2, t2, inner_loop # Loop back if more floats are to be swapped

    addi s1, s1, 1         # Increment the outer loop counter
    blt s1, t2, outer_loop # Continue to the next pass

    # Print sorted array
    la a1, array           # Reset array address for printing
    li t0, 0               # Reset index

addi t2, t2, 1

# Loop for printing the sorted array

la a0, order_values # Load address of ordenated_values string
li a7, 4             # System call number for print string
ecall                # Execute system call

print_loop:
    flw fa0, 0(a1)       # Load float from array
    li a7, 2             # System call number to print float
    ecall                # Execute system call

```

```

    la a0, space_char # Load address of space character
    li a7, 4          # System call number to print string
    ecall             # Execute system call

    addi a1, a1, 4     # Move to the next float in the array
    addi t0, t0, 1     # Increment index
    blt t0, t2, print_loop # Loop back if more floats are to be printed

# Calculate median
li t3, 0              # Initialize median index
srai t3, t2, 1        # t3 = t2 / 2

la a0, median_float  # Load address of median_float string into a0
li a7, 4              # System call number for print string
ecall                 # Execute system call

# Check if the array size is even or odd
andi t4, t2, 1        # t4 = t2 & 1 (checks if the array size is odd)
beqz t4, even_array   # If t4 is 0, the array size is even

# If the array size is odd
odd_array:
    la a1, array
    slli t3, t3, 2     # t3 = t3 * 4 -> Convert the integer value into bytes
    add a0, a1, t3     # Gets the median float's address
    flw fa0, 0(a0)     # Load median value

    li a7, 2           # System call number to print float
    ecall              # Execute system call

    # Exit the program
    li a7, 10          # System call number for exit
    ecall

# If the array size is even
even_array:
    la a1, array
    slli t3, t3, 2     # t3 = t3 * 4 -> Convert the integer value into bytes
    add a0, a1, t3     # Get the second median float's address
    flw fa0, 0(a0)     # Load the second median value

    addi t3, t3, -4    # Decrement t3 by 4 to get the previous float's address
    add a0, a1, t3     # Get the first median float's address
    flw fa1, 0(a0)     # Load the first median value

```

```

fadd.s fa0, fa0, fa1 # fa0 = fa0 + fa1 (sum the two median values)
li t1, 0x40000000    # 2 in IEEE754
fmv.s.x fa1, t1      # Move value from t1 to fa1

fdiv.s fa0, fa0, fa1 # Divide by 2

li a7, 2              # System call number to print float
ecall                 # Execute system call

# Exit the program
li a7, 10             # System call number for exit
ecall

one_space_array:
la a0, str            # Load address of string into a0
li a7, 4              # System call number for print string
ecall                 # Execute system call

li a7, 6              # System call number for read float from user
ecall                 # Execute system call

la a0, order_values # Load address of ordered_values string
li a7, 4              # System call number for print string
ecall                 # Execute system call

fsw fa0, 0(a1)        # Store the float into the array at current index
li a7, 2              # System call number to print the float value
ecall                 # Execute system call

la a0, median_float # Load address of median_float string
li a7, 4              # System call number for print string
ecall                 # Execute system call

fsw fa0, 0(a1)        # Store the float into the array at current index
li a7, 2              # System call number to print the float value
ecall                 # Execute system call

li a7, 10             # System call number for exit
ecall                 # Execute system call

```

Com o código em RISCV32 já apresentado, mostraremos os resultados correspondentes a cada um dos testes realizados em C:

```
Quantidade de itens:
1
Insira os valores
2.2
Valores ordenados:
2.2
Mediana: 2.2
-- program is finished running (0) --
```

Figura 4: Caso de Array de Um Espaço, em Assembly

Como pode ser visto comparando as figuras 1 e 4, sob as mesmas entradas, as saídas foram, essencialmente, iguais; salvo pequenas diferenças como a quantidade de 0 após os valores retornados em C, uma particularidade que não afeta a validade dos resultados.

```
Quantidade de itens:
7
Insira os valores
3.3
5.5
6.6
8.3554
775.3
0.8
09.76
Valores ordenados:
0.8 3.3 5.5 6.6 8.3554 9.76 775.3
Mediana: 6.6
-- program is finished running (0) --
```

Figura 5: Caso de Array Ímpar, em Assembly

Novamente, diante de uma comparação entre as figuras 2 e 5, temos resultados semelhantes e relativamente satisfatórios, com diferenças pequenas no que tange à quantidade de 0 após o último significativo, ou aproximações dos valores ordenados em C, o que pode se dever tanto à linguagem em si, quanto ao compilador utilizado, mas ainda assim não compromete o resultado final.

Por fim, comparando as figuras 3 e 6, mais uma vez obtivemos um resultado semelhante e bem-sucedido, com, basicamente, as mesmas diferenças notadas no caso do array ímpar; e que, novamente, não comprometem o êxito dos códigos.

```
Quantidade de itens:
8
Insira os valores
34.12
4
8.9
5.55
3.6
0.01
3434.55
45.1
Valores ordenados:
0.01 3.6 4.0 5.55 8.9 34.12 45.1 3434.55
Mediana: 7.225
-- program is finished running (0) --
```

Figura 6: Caso de Array Par, em Assembly

Utilizando o compilador GNU-GCC configurado para o RISC-V 32-bit, o código foi compilado gerando arquivos em Assembly em dois níveis de otimização: com a opção `-O0`, que não aplica otimizações, e depois com a opção `-O3`, que aplica otimizações máximas. É crucial analisar as diferenças entre os três códigos: o escrito manualmente pelo grupo e os dois gerados automaticamente pelo compilador. Uma observação inicial relevante é que os códigos gerados pelo compilador tendem a ser significativamente mais extensos do que o código elaborado manualmente. Isso se deve às inserções automáticas que o compilador realiza para otimizar o desempenho ou a eficiência do código, o que frequentemente resulta em uma quantidade maior de instruções Assembly, mesmo quando não há otimizações explicitamente aplicadas (`-O0`).

A maior quantidade de linhas no código Assembly gerado pelo compilador, em comparação com o desenvolvido pelo grupo, pode ser atribuída a várias técnicas de compilação e otimizações automáticas que são aplicadas, mesmo na ausência de otimizações explícitas. Compiladores modernos, como o GNU-GCC, incorporam uma série de padrões e práticas predefinidos para garantir a compatibilidade, a eficiência de execução e a segurança do código.

Por exemplo, o compilador pode introduzir verificações adicionais de segurança, como prevenção de estouro de buffer, e pode alocar variáveis em registradores de forma a otimizar o acesso à memória e a reduzir a latência, o que não foi priorizado no desenvolvimento manual do código, onde o grupo teve uma abordagem focada mais na funcionalidade direta e na clareza do código. Além disso, o compilador tenta fazer previsões inteligentes sobre o fluxo de controle do programa para inserir instruções que podem melhorar a previsão de ramificação e o paralelismo a nível de instrução. Essas inserções resultam em um código mais longo, mas potencialmente

mais rápido ou mais robusto em certas condições de operação.

Ao analisar o código assembly gerado pela versão -O0 e compará-lo com o código `median.asm` programado manualmente, várias diferenças estruturais e de conteúdo se destacam:

- **Prólogo e Epílogo da Função:** No código gerado pelo compilador com -O0, observamos instruções detalhadas para o gerenciamento do stack frame no início e no final da função `main`. Isso inclui ajustar o ponteiro de pilha (`sp`), salvar e restaurar o endereço de retorno (`ra`) e o ponteiro de frame (`s0`). Essas operações garantem que a chamada e retorno de funções sejam manuseadas corretamente, preservando o estado do programa. O código manual, por outro lado, assume um controle mais direto e menos automático sobre o stack.

Como podemos ver abaixo:

Código Gerado ‘-O0’:

```
# Prologo
addi    sp, sp, -80
sw      ra, 76(sp)
sw      s0, 72(sp)
addi    s0, sp, 80

# Epilogo
lw      ra, 76(sp)
lw      s0, 72(sp)
addi    sp, sp, 80
jr      ra
```

Código Manual ‘median.asm’:

Retorno simples, sem manipulação detalhada do stack

```
li      a7, 10          # System call number for exit
ecall                   # Execute system call
```

- **Strings e Seções de Dados:** O código -O0 inclui múltiplas strings armazenadas na seção `.rodata` e utiliza instruções para carregar endereços base para essas strings antes das chamadas a funções como `printf` e `scanf`. Essas são técnicas típicas para facilitar a manipulação de dados constantes e seu uso repetido. No código manual, strings e dados são também definidos, mas o método de acesso e utilização é mais direto.

- **Chamadas de Biblioteca:** No código -O0, há chamadas a funções de biblioteca como `printf`, `scanf`, e outras operações de ponto flutuante (`__gesf2`, `__addsf3`, `__divsf3`), que são gerenciadas automaticamente pelo compilador para lidar com operações complexas. No código manual, as operações são direcionadas através de chamadas de sistema específicas do ambiente de execução (por exemplo, `ecall` no RISC-V), o que requer um controle mais explícito dos registros e do comportamento do sistema.

Código Gerado ‘-O0’:

```
call    printf
call    scanf
```

Código Manual ‘median.asm’:

```
li      a7, 4          # System call number for print string
ecall                   # Execute system call
li      a7, 5          # System call number for read integer
ecall                   # Execute system call
```

- **Instruções de Loop e Controle de Fluxo:** O compilador insere verificações e loops com uma abundância de instruções de controle de fluxo, como `blt`, `bne`, e saltos condicionais e incondicionais (`j`). Estas são otimizadas apenas para a corretude, sem considerar a minimização do número de instruções. Em contraste, o código manual apresenta uma estrutura de loop mais compacta, focada na eficiência do uso de instruções específicas e no controle direto do fluxo de execução.

Essas diferenças destacam como o compilador aborda a geração de código para garantir a generalidade, segurança e compatibilidade, enquanto o código assembly manual foi feito com um foco na otimização específica e no controle direto, adequado ao contexto específico da sua utilização e tradução do código em C fornecido.

3 Conclusão

Durante a realização desse projeto, ocorreu um notável e importante aprendizado acerca do uso da linguagem de programação Assembly: ainda que o ensino teórico seja imprescindível para se obter noções básicas sobre o assunto, desenvolver um código com uma aplicação compreensível e próxima da realidade de um estudante de exatas, como descobrir uma mediana, confere uma noção diferente e especial sobre a linguagem aplicada.

Em essência, ainda que a Assembly não seja tão popular e utilizado nos dias de hoje, em partes devido ao surgimento de outras linguagens de programação mais simples e eficientes; ter um conhecimento, ainda que básico, dela é imprescindível para a engenharia de controle e automação, haja visto que essa linguagem se caracteriza por seu baixo nível e boa comunicação com máquinas, sendo elas extremamente relevantes na jornada daqueles que cursam a modalidade em pauta.

Em termos finais, o projeto não foi fácil, longe disso; porém, teve um resultado satisfatório, e os conhecimentos adquiridos com ele serão de grande importância ao longo do futuro do curso de engenharia de controle e automação.