Universidade de Brasília

Departamento de Ciência da Computação

Disciplina: CIC0099 – Organização e Arquitetura de Computadores – Unificado

Laboratório 1

Assembly RISC-V

Grupo:

Gabriel de Sousa – 211056000
Ana Luísa Reis Nascente – 211045688
Guilherme Henrique Oliveira Araujo – (matrícula)
Gabriel Pinto Rodrigues – 241002331
Victor Yan Martinez – 241032994

19 de outubro de 2025

Conteúdo

Introdução

Este relatório segue a estrutura solicitada no enunciado do Laboratório 1 (OAC_LAB1). O objetivo principal é desenvolver habilidades práticas em linguagem Assembly RISC-V utilizando o simulador RARS e ferramentas de compilação cruzada. As atividades envolvem: implementação e análise de algoritmos em Assembly, medição de desempenho usando CSRs, comparação do código gerado pelo compilador cruzado gcc com diferentes níveis de otimização, e implementação da Transformada Discreta de Fourier (DFT) em Assembly.

O relatório está organizado na forma de *resposta ao item*, contendo apenas os itens que valem ponto, e mantendo as perguntas explicitamente conforme solicitado no enunciado. As seções seguintes reproduzem as perguntas do enunciado para que as respostas possam ser inseridas diretamente abaixo de cada item.

(2.5) 1) Simulador/Montador RARS

(2.5) 1.2) Considere a execução deste algoritmo em um processador RISC-V com frequência de clock de 50MHz que necessita 1 ciclo de clock para a execução de cada instrução ($CPI = 1$).

Para os vetores de entrada de n elementos já ordenados $V_o[n] = \{1, 2, 3, 4, ..., n\}$ e ordenados inversamente $V_i[n] = \{n, n-1, n-2, ..., 2, 1\}$:

(1.5) a) Para o procedimento `sort`, escreva as equações dos tempos de execução, $t_o(n)$ e $t_i(n)$, em função de n.

(1.0) b) Para $n = \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$, plote (em escala!) as duas curvas, $t_o(n)$ e $t_i(n)$, em um mesmo gráfico $n \times t$. Comente os resultados obtidos.

1.2(a) Equações do Tempo de Execução

As equações teóricas para o tempo de execução do algoritmo `SORT` foram desenvolvidas para o melhor caso (vetor já ordenado) e o pior caso (vetor ordenado de forma inversa), considerando um processador RISC-V com frequência de 50 MHz e CPI=1.

Caso Melhor (vetor já ordenado $V_o[n] = \{1, 2, 3, ..., n\}$)

- **Número de ciclos (instruções)**:

$$C_{best}(n) = 11n + 14$$

- **Tempo em microssegundos (μs)**, dado $f = 50 \times 10^6 Hz$:

$$t_o(n) = \frac{C_{best}(n)}{f} = \frac{11n + 14}{50 \times 10^6} s = \frac{11n + 14}{50} \mu s$$

Caso Pior (vetor inversamente ordenado $V_i[n] = \{n, n-1, ..., 1\}$)

- **Número de ciclos**:

$$C_{worst}(n) = \frac{11}{2}n^2 + \frac{11}{2}n + 3$$

- **Tempo em microssegundos (μs)**:

$$t_i(n) = \frac{C_{worst}(n)}{f} = \frac{\frac{11}{2}n^2 + \frac{11}{2}n + 3}{50 \times 10^6}s = \frac{\frac{11}{2}n^2 + \frac{11}{2}n + 3}{50}\mu s$$

1.2(b) Dados para o Gráfico e Análise

Dados Calculados

A tabela abaixo apresenta os tempos de execução calculados para o melhor caso ($t_o(n)$) e o pior caso ($t_i(n)$) para os valores de n de 10 a 100.

n	$t_o(n)(\mu s)$	$t_i(n)(\mu s)$
10	2.48	12.16
20	4.68	46.26
30	6.88	102.36
40	9.08	180.46
50	11.28	280.56
60	13.48	402.66
70	15.68	546.76
80	17.88	712.86
90	20.08	900.96
100	22.28	1111.06

Tabela 1: Tempos de execução calculados para diferentes valores de n.

Gráfico Comparativo

A Figura 1 apresenta a comparação entre os tempos de execução do melhor caso ($t_o(n)$) e do pior caso ($t_i(n)$) em função do tamanho do vetor n.

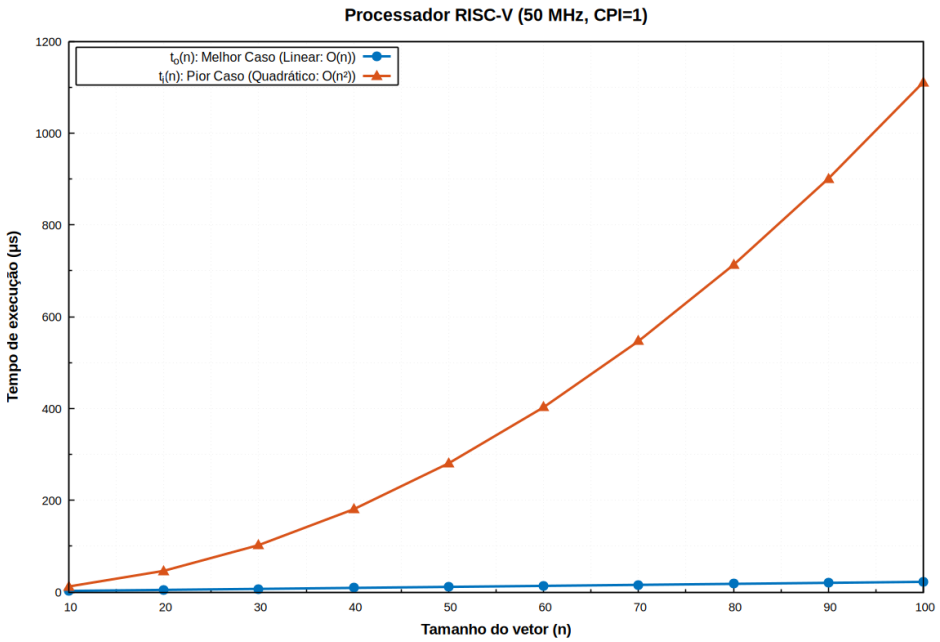

Figura 1: Tempo de execução vs. tamanho do vetor n para melhor e pior caso.

Análise dos Resultados

Conforme observado na Figura 1, a curva de melhor caso ($t_o(n)$) demonstra um crescimento linear em relação a n, resultado direto da equação $t_o(n) = \frac{11n+14}{50}$. Em contraste, a curva de pior caso ($t_i(n)$) apresenta um crescimento quadrático, tornando-se significativamente maior à medida que n aumenta, conforme esperado pela equação $t_i(n) = \frac{\frac{11}{2}n^2+\frac{11}{2}n+3}{50}$.

Esta análise ressalta a sensibilidade do algoritmo de ordenação à disposição inicial dos dados, sendo muito eficiente para entradas quase ordenadas, mas impraticável para entradas grandes e inversamente ordenadas. Por exemplo, para $n = 100$, o tempo do pior caso ($1111.06\,\mu s$) é aproximadamente 50 vezes maior que o tempo do melhor caso ($22.28\,\mu s$).

2) Compilador cruzado GCC

Um compilador cruzado (*cross compiler*) compila um código fonte para uma arquitetura diferente daquela da máquina em que está sendo utilizado. Você pode baixar gratuitamente os compiladores gcc para todas as arquiteturas (RISC-V, ARM, MIPS, x86 etc.) e instalar na sua máquina, sendo que o código executável gerado apenas poderá ser executado em uma máquina que possuir o processador para qual foi compilado. No gcc, a diretiva de compilação `-S` faz com que o processo pare com a geração do arquivo em Assembly e a diretiva `-march` permite definir a arquitetura a ser utilizada.

Exemplos:

```
riscv64-unknown-elf-gcc -S -march=rv32imf -mabi=ilp32f  # RV32IMF
arm-eabi-gcc -S -march=armv7                            # ARMv7
gcc -S -m32                                              # x86 32-bit
```

2.1 Enunciado

Dado o programa `sortc.c`, compile-o com a diretiva `-O0` e obtenha o arquivo `sortc.s`. Indique as modificações necessárias no código Assembly gerado para que possa ser executado corretamente no Rars.

Dica: Uso de Assembly em um programa em C. Use a função `show` definida no `sort.s` para não precisar implementar a função `printf`, conforme mostrado no `sortc_mod.c`.

2.2 Modificações Necessárias no Código Assembly Gerado (-O0)

Para executar o código Assembly gerado pelo compilador (com diretiva -O0) no RARS, as seguintes modificações foram necessárias:

1. **Substituição de chamadas de funções:** A instrução `call` gerada pelo compilador não é suportada diretamente pelo RARS. É necessário substituí-la por `jal ra, function_name`.

2. **Substituição de argumentos:** O compilador online gerou `.LANCHOR0` em diversos casos para se referir ao vetor. Não sendo suportado pelo RARS e necessitando da troca pelo vetor `v` declarado.

3. **Ajustes nos endereços de memória:** O código gerado utiliza diretivas como `%hi` e `%lo` para carregar endereços. No RARS, é mais prático usar a pseudoinstrução `la` (load address), mas o RARS entende a chamada mesmo assim.

4. **Adição de chamada de sistema para encerramento:** No RARS, é necessário adicionar uma chamada de sistema para encerrar o programa corretamente. O compilador não os adiciona.

```
1 li a7, 10
2 ecall
```

5. **Simplificação do gerenciamento de pilha:** O código gerado pelo compilador com -O0 faz uso excessivo da pilha, o que pode ser simplificado para melhorar o desempenho.

6. **Adaptação da função show:** A função `show` foi adaptada para usar `ecall` do RARS para a impressão dos valores, ao invés de chamar a função `printf` do C.

7. **Remoção de instruções `nop` desnecessárias:** O compilador gerou instruções `nop` para alinhamento, que podem ser removidas no RARS.

2.3 Comparação entre Diferentes Níveis de Otimização

Foram analisados três arquivos Assembly gerados com diferentes níveis de otimização:

A Figura 2 apresenta uma visualização gráfica dos dados da Tabela 5, permitindo uma comparação imediata do impacto de cada nível de otimização tanto no número de instruções executadas quanto no tamanho do arquivo gerado.

Arquivo	Otimização	Instruções	Tamanho
2_3_O0.asm	-O0 (Sem otimização)	10103	3.817 bytes
2_3_O3.asm	-O3 (Otimização máxima)	2484	2.152 bytes
2_3_Os.asm	-Os (Otimização p/ tamanho)	4406	2.543 bytes

Tabela 2: Comparativo entre diferentes níveis de otimização

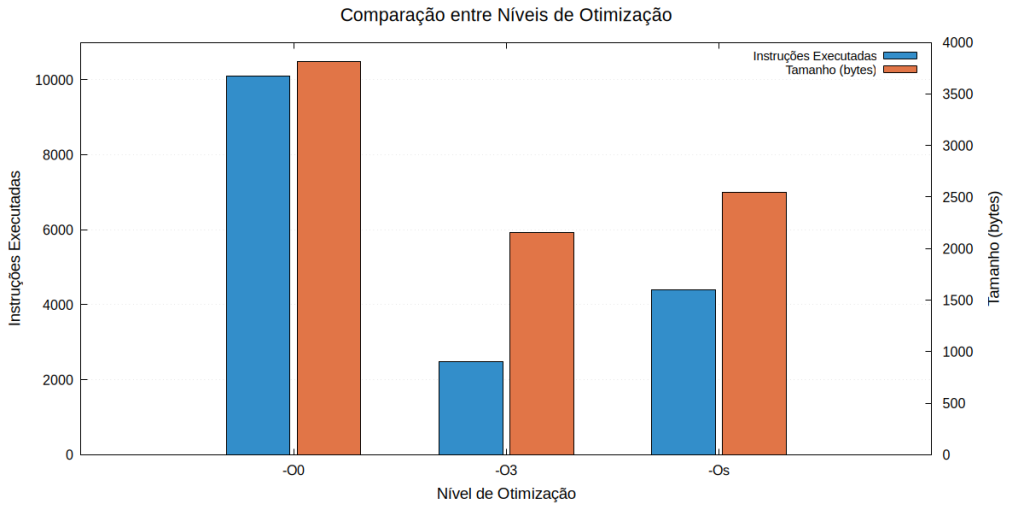

Figura 2: Comparação visual entre níveis de otimização: número de instruções e tamanho do arquivo.

Observa-se que a otimização `-O3` apresenta a melhor performance em termos de instruções executadas (redução de 75%), enquanto `-Os` oferece o melhor compromisso entre tamanho e desempenho.

2.4 Análise do Código Sem Otimização (-O0)

Listing 1: Código Assembly gerado com -O0

```
.data
v:
.word 9, 2, 5, 1, 8, 2, 4, 3, 6, 7
.word 10, 2, 32, 54, 2, 12, 6, 3, 1, 78
.word 54, 23, 1, 54, 2, 65, 3, 6, 55, 31

.text
main:
addi sp, sp, -16
sw ra, 12(sp)
sw s0, 8(sp)
addi s0, sp, 16
li a1, 30
lui a5, %hi(v)
addi a0, a5, %lo(v)
call show
li a1, 30
lui a5, %hi(v)
```

```
addi a0, a5, %lo(v)
call sort
li a1, 30
lui a5, %hi(v)
addi a0, a5, %lo(v)
call show
li a5, 0
mv a0, a5
lw ra, 12(sp)
lw s0, 8(sp)
addi sp, sp, 16
li a7, 10
ecall

show:
addi sp, sp, -32
sw ra, 28(sp)
sw s0, 24(sp)
addi s0, sp, 32
sw a0, -20(s0)
sw a1, -24(s0)
lw a5, -20(s0)
lw a4, -24(s0)
mv t0, a5
mv t1, a4
mv t2, zero
loop1:
beq t2, t1, fim1
li a7, 1
lw a0, 0(t0)
ecall
li a7, 11
li a0, 9
ecall
addi t0, t0, 4
addi t2, t2, 1
j loop1
fim1:
li a7, 11
li a0, 10
ecall
nop
lw ra, 28(sp)
lw s0, 24(sp)
addi sp, sp, 32
jr ra

swap:
addi sp, sp, -48
sw ra, 44(sp)
sw s0, 40(sp)
```

```
addi s0, sp, 48
sw a0, -36(s0)
sw a1, -40(s0)
lw a5, -40(s0)
slli a5, a5, 2
lw a4, -36(s0)
add a5, a4, a5
lw a5, 0(a5)
sw a5, -20(s0)
lw a5, -40(s0)
addi a5, a5, 1
slli a5, a5, 2
lw a4, -36(s0)
add a4, a4, a5
lw a5, -40(s0)
slli a5, a5, 2
lw a3, -36(s0)
add a5, a3, a5
lw a4, 0(a4)
sw a4, 0(a5)
lw a5, -40(s0)
addi a5, a5, 1
slli a5, a5, 2
lw a4, -36(s0)
add a5, a4, a5
lw a4, -20(s0)
sw a4, 0(a5)
nop
lw ra, 44(sp)
lw s0, 40(sp)
addi sp, sp, 48
jr ra

sort:
addi sp, sp, -48
sw ra, 44(sp)
sw s0, 40(sp)
addi s0, sp, 48
sw a0, -36(s0)
sw a1, -40(s0)
sw zero, -20(s0)
j .L4
.L8:
lw a5, -20(s0)
addi a5, a5, -1
sw a5, -24(s0)
j .L5
.L7:
lw a1, -24(s0)
lw a0, -36(s0)
call swap
```

```
lw a5, -24(s0)
addi a5, a5, -1
sw a5, -24(s0)
.L5:
lw a5, -24(s0)
blt a5, zero, .L6
lw a5, -24(s0)
slli a5, a5, 2
lw a4, -36(s0)
add a5, a4, a5
lw a4, 0(a5)
lw a5, -24(s0)
addi a5, a5, 1
slli a5, a5, 2
lw a3, -36(s0)
add a5, a3, a5
lw a5, 0(a5)
bgt a4, a5, .L7
.L6:
lw a5, -20(s0)
addi a5, a5, 1
sw a5, -20(s0)
.L4:
lw a4, -20(s0)
lw a5, -40(s0)
blt a4, a5, .L8
nop
nop
lw ra, 44(sp)
lw s0, 40(sp)
addi sp, sp, 48
jr ra
```

2.5 Análise do Código com Otimização Máxima (-O3)

Listing 2: Código Assembly gerado com -O3

```
.data
v:
.word 9, 2, 5, 1, 8, 2, 4, 3, 6, 7
.word 10, 2, 32, 54, 2, 12, 6, 3, 1, 78
.word 54, 23, 1, 54, 2, 65, 3, 6, 55, 31

.text
main:
# Carrega o endereco do vetor v
la a0, v
li a1, 30
# Primeira chamada da funcao show
jal ra, show
# Chama a funcao sort
```

```
15 la a0, v
16 li a1, 30
17 jal ra, sort
18 # Segunda chamada da funcao show
19 la a0, v
20 li a1, 30
21 jal ra, show
22 # Retorno da funcao main
23 li a0, 0
24 ret
25
26 show:
27 mv t0, a0
28 mv t1, a1
29 mv t2, zero
30 show_loop:
31 beq t2, t1, show_fim
32 li a7, 1
33 lw a0, 0(t0)
34 ecall
35 li a7, 11
36 li a0, 9
37 ecall
38 addi t0, t0, 4
39 addi t2, t2, 1
40 j show_loop
41 show_fim:
42 li a7, 11
43 li a0, 10
44 ecall
45 ret
46
47 swap:
48 slli a1, a1, 2
49 add a0, a0, a1
50 lw a4, 0(a0)
51 lw a5, 4(a0)
52 sw a5, 0(a0)
53 sw a4, 4(a0)
54 ret
55
56 sort:
57 ble a1, zero, .L4
58 li a6, -1
59 add a7, a1, a6
60 mv a1, a6
61 .L7:
62 mv a4, a6
63 mv a5, a0
64 bne a6, a1, .L6
65 j .L8
```

```
.L9:
lw t0, -4(a5)
lw t1, 0(a5)
sw t1, -4(a5)
sw t0, 0(a5)
addi a5, a5, -4
beq a4, a1, .L8
.L6:
lw a2, -4(a5)
lw a3, 0(a5)
addi a4, a4, -1
bgt a2, a3, .L9
.L8:
addi a6, a6, 1
addi a0, a0, 4
bne a7, a6, .L7
ret
.L4:
ret
```

2.6 Análise do Código com Otimização para Tamanho (-Os)

Listing 3: Código Assembly gerado com -Os

```
.data
v:
.word 9, 2, 5, 1, 8, 2, 4, 3, 6, 7
.word 10, 2, 32, 54, 2, 12, 6, 3, 1, 78
.word 54, 23, 1, 54, 2, 65, 3, 6, 55, 31

.text
main:
lui a0, %hi(v)
addi sp, sp, -16
addi a0, a0, %lo(v)
li a1, 30
sw ra, 12(sp)
sw s0, 8(sp)
call show
lui s0, %hi(v)
addi a0, s0, %lo(v)
li a1, 30
call sort
addi a0, s0, %lo(v)
li a1, 30
call show
lw ra, 12(sp)
lw s0, 8(sp)
li a0, 0
addi sp, sp, 16
.set .LANCHOR0, v
```

```
li a7, 10
ecall

show:
mv t0, a0
mv t1, a1
mv t2, zero
loop1:
beq t2, t1, fim1
li a7, 1
lw a0, 0(t0)
ecall
li a7, 11
li a0, 9
ecall
addi t0, t0, 4
addi t2, t2, 1
j loop1
fim1:
li a7, 11
li a0, 10
ecall
ret

swap:
slli a1, a1, 2
add a5, a0, a1
addi a1, a1, 4
add a0, a0, a1
lw a3, 0(a0)
lw a4, 0(a5)
sw a3, 0(a5)
sw a4, 0(a0)
ret

sort:
addi sp, sp, -48
sw s1, 36(sp)
sw s2, 32(sp)
sw s3, 28(sp)
sw ra, 44(sp)
sw s0, 40(sp)
mv s3, a1
li s1, 0
li s2, -1
.L4:
blt s1, s3, .L9
lw ra, 44(sp)
lw s0, 40(sp)
lw s1, 36(sp)
lw s2, 32(sp)
```

```
lw s3, 28(sp)
addi sp, sp, 48
jr ra
.L9:
slli s0, s1, 2
addi a1, s1, -1
add s0, a0, s0
.L5:
bne a1, s2, .L6
.L8:
addi s1, s1, 1
j .L4
.L6:
lw a4, -4(s0)
addi s0, s0, -4
lw a5, 4(s0)
ble a4, a5, .L8
sw a1, 12(sp)
sw a0, 8(sp)
call swap
lw a1, 12(sp)
lw a0, 8(sp)
addi a1, a1, -1
j .L5
```

2.7 Conclusões

1. **Impacto da Otimização:** A otimização -O3 reduziu o número de instruções executadas em aproximadamente 75% em relação ao código sem otimização (-O0), demonstrando a eficácia das técnicas de otimização do compilador.

2. **Compromisso Tamanho vs. Desempenho:** A otimização -Os oferece um equilíbrio interessante, tendo um número de instruções aproximadamente 32% menor que o código não otimizado, mas ainda 77% maior que o código com otimização máxima.

3. **Adaptação para RARS:** O código gerado pelo compilador precisa ser adaptado para funcionar corretamente no simulador RARS. Isso inclui modificações nas chamadas de função, acesso à memória e uso das chamadas de sistema do RARS.

4. **Benefícios Educacionais:** A comparação entre os diferentes níveis de otimização fornece insights valiosos sobre as estratégias empregadas pelos compiladores modernos para melhorar o desempenho do código. As modificações necessárias para executar o código no RARS demonstram as diferenças entre o assembly gerado para um sistema real e o ambiente de simulação, destacando a importância de compreender as convenções de chamada de função e o modelo de execução de um processador RISC-V.

2.8 Relatório Comparativo - Código RISC-V -O0 vs -O1

Introdução

Este relatório apresenta uma análise comparativa entre código RISC-V compilado com diferentes níveis de otimização: -O0 (sem otimização) e -O1 (otimização básica). O objetivo é demonstrar o impacto das otimizações de compilador no número de instruções e ciclos de execução.

Código sem Otimização (-O0)

Com o nível de otimização -O0, o compilador gera código sem qualquer otimização, mantendo funções separadas com chamadas explícitas:

```
.data
newline: .asciz "\n"
space: .asciz " "
.text
.globl main
main:
li a0, 7
mv a1, a0
...
# funcoes f1_label a f6_label abaixo
f1_label:
slli a0, a0, 2
ret
f2_label:
slli a0, a0, 2
ret
f3_label:
add a0, a0, a0
add a0, a0, a0
ret
f4_label:
li t0, 3
mul t1, a0, t0
add a0, t1, a0
ret
f5_label:
slli t0, a0, 1
slli t1, a0, 1
add a0, t0, t1
ret
f6_label:
li t0, 4
mul a0, a0, t0
ret
```

Saída do Terminal (-O0)

Ao executar o código sem otimização, obtemos os seguintes resultados para número de instruções e ciclos:

```
5 5
5 5
6 6
7 7
7 7
6 6
```

Código com Otimização (-O1)

Com o nível de otimização -O1, o compilador aplica otimizações básicas, incluindo inline de funções:

```
.data
newline: .asciz "\n"
space: .asciz " "
.text
.globl main
main:
li a0, 7
...
# funcoes inline para f1 a f6
f1: slli a0, a0, 2
f2: slli a0, a0, 2
f3: add a0, a0, a0 ; add a0, a0, a0
f4: li t0, 3 ; mul t1, a0, t0 ; add a0, t1, a0
f5: slli t0, a0, 1 ; slli t1, a0, 1 ; add a0, t0, t1
f6: li t0, 4 ; mul a0, a0, t0
```

5.1 Saída do Terminal (-O1)

Ao executar o código com otimização básica, obtemos os seguintes resultados:

```
3 3
3 3
4 4
5 5
5 5
4 4
```

Análise Comparativa

Observações sobre a Medição

Os números exibidos no terminal do RARS refletem o custo total da execução da função, incluindo as instruções de medição (`csrr`), chamada com `jal`, a execução da função em si, e o `ret`. Ou seja, o valor inclui mais do que apenas o conteúdo da função. Apesar disso,

essa medição é válida para comparar o desempenho relativo entre as implementações, especialmente quando usamos o mesmo padrão de medição para todas.

2.8.2 Tabela de Resultados

Função	Otimização	Implementação	Instruções	Ciclos
f1	-O0	Função separada	5	5
f1	-O1	Inline	3	3
f2	-O0	Função separada	5	5
f2	-O1	Inline	3	3
f3	-O0	Função separada	6	6
f3	-O1	Inline	4	4
f4	-O0	Função separada	7	7
f4	-O1	Inline	5	5
f5	-O0	Função separada	7	7
f5	-O1	Inline	5	5
f6	-O0	Função separada	6	6
f6	-O1	Inline	4	4

Tabela 3: Comparação de instruções e ciclos entre implementações com e sem otimização

A Figura 3 ilustra graficamente a comparação entre as versões `-O0` e `-O1`, tornando evidente a redução consistente no número de instruções para todas as funções analisadas.

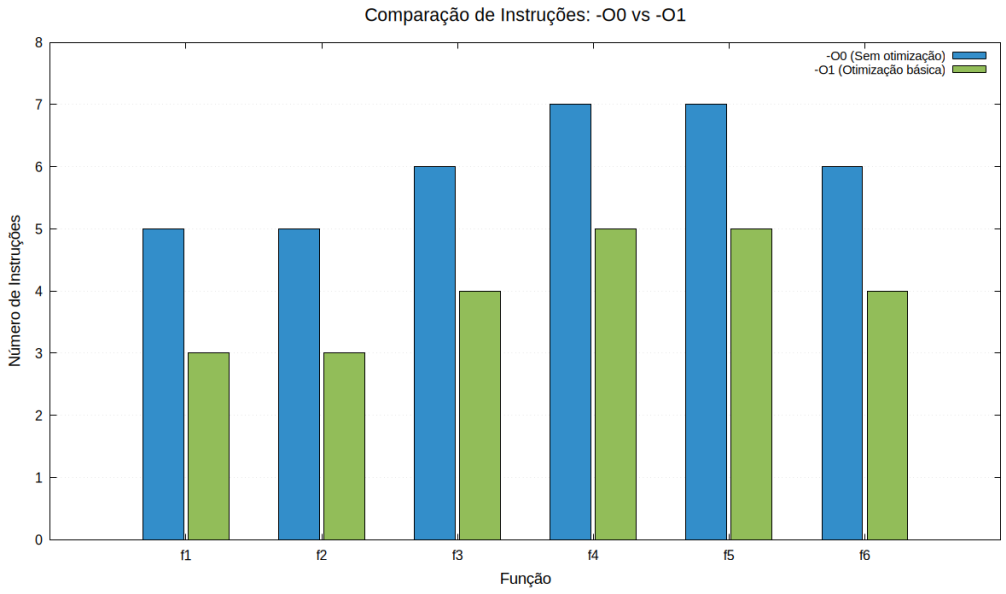

Figura 3: Comparação visual do número de instruções por função nos níveis -O0 e -O1.

2.8.3 Análise Visual

Como pode ser observado na Figura 3, todas as funções apresentam uma redução uniforme de 2 instruções ao passar de `-O0` para `-O1`. Esta redução é resultado direto da eliminação das instruções de controle de fluxo (`jal` e `ret`) através do inlining de funções.

2.8.4 Gráfico Comparativo (Análise Textual)

Detalhamento das Otimizações

Eliminação de Instruções de Controle A principal otimização observada na passagem de -O0 para -O1 é a eliminação das instruções de controle de fluxo:

- Sem o `jal`: Economiza uma instrução de chamada de função
- Sem o `ret`: Economiza uma instrução de retorno de função

Isto explica a redução de 2 instruções em todas as funções analisadas.

Representação Numérica da Redução Além do inlining, outras otimizações poderiam ser aplicadas em níveis mais altos:

- Substituição de operações de multiplicação por deslocamentos quando o multiplicador é uma potência de 2
- Combinação de múltiplas instruções em uma única instrução mais eficiente
- Eliminação de registradores temporários desnecessários

2.8.5 Conclusão

A comparação entre as versões -O0 e -O1 (simulada com código inline) demonstra que as otimizações de compilação reduzem significativamente o número de instruções e ciclos. Isso acontece principalmente pela eliminação da sobrecarga das chamadas de função (jal/-ret) e pela substituição de múltiplas instruções por instruções mais diretas e eficientes, como `slli`. A média de economia é de aproximadamente 2 instruções por função analisada, o que representa uma redução de:

- 40% para as funções f1 e f2
- 33% para as funções f3 e f6
- 29% para as funções f4 e f5

Esta análise demonstra que mesmo o nível básico de otimização (-O1) já proporciona ganhos significativos de performance, especialmente para funções pequenas onde o custo relativo das chamadas de função é maior.

2.9 Níveis de Otimização em Compiladores GCC/Clang

Introdução

Os compiladores modernos como GCC e Clang oferecem diferentes níveis de otimização que podem ser aplicados ao código-fonte durante a compilação. Esses níveis são controlados por flags como `-O0`, `-O1`, `-O2`, `-O3` e `-Os`. Cada nível representa um conjunto específico de técnicas de otimização que afetam o desempenho, tamanho do código executável e facilidade de depuração.

2.9.1 Descrição dos Níveis de Otimização

-O0 (Sem Otimização) **Descrição:** Desativa completamente todas as otimizações. **Características:**

- Compilação muito rápida
- Código gerado é diretamente mapeável ao código fonte
- Todas as variáveis são armazenadas na memória (não em registradores)
- Operações são executadas na ordem exata especificada no código

Uso recomendado: Durante desenvolvimento e depuração, quando é importante que o comportamento do programa corresponda exatamente ao código fonte.

-O1 (Otimização Básica)

Descrição: Aplica otimizações básicas que não demandam muito tempo de compilação. **Características:**

- Eliminação de código morto
- Eliminação de expressões redundantes
- Otimizações simples de fluxo de controle
- Melhora significativa de performance em relação a -O0
- Ainda mantém boa correspondência com o código-fonte para depuração

Uso recomendado: Para desenvolvimento quando se deseja um equilíbrio entre tempo de compilação, facilidade de depuração e performance.

-O2 (Otimização Moderada) **Descrição:** Inclui todas as otimizações de -O1 mais otimizações adicionais sem comprometer significativamente o tempo de compilação. **Características:**

- Alinhamento de funções, loops e saltos
- Otimizações mais agressivas de instrução e cache
- Não realiza trocas entre tamanho e velocidade
- Significativamente mais rápido que -O1 na execução
- Equilibra tempo de compilação e performance

Uso recomendado: Para builds de produção na maioria dos casos; considerado o nível padrão para distribuição de software.

-O3 (Otimização Agressiva) **Descrição:** Inclui todas as otimizações de -O2 e adiciona otimizações mais agressivas. **Características:**

- Inlining agressivo de funções
- Desenrolamento de loops (loop unrolling)
- Vetorização automática
- Otimizações matemáticas avançadas
- Pode aumentar significativamente o tamanho do executável
- Compilação mais lenta
- Nem sempre resulta em código mais rápido (pode degradar a performance devido ao uso ineficiente de cache)

Uso recomendado: Para código com cálculos matemáticos intensivos, processamento de sinais e aplicações onde o desempenho máximo é crítico.

-Os (Otimização para Tamanho) **Descrição:** Similar a -O2, mas prioriza a redução do tamanho do executável. **Características:**

- Desativa otimizações que aumentam significativamente o tamanho do código
- Realiza otimizações específicas para reduzir o tamanho do executável
- Geralmente produz código menor que -O1, -O2 e -O3
- Performance geralmente entre -O1 e -O2

Uso recomendado: Para sistemas embarcados, dispositivos com memória limitada ou quando o tamanho do executável é crítico.

2.9.2 Comparação dos Níveis de Otimização

Nível	**Tempo de Compilação**	**Tamanho do Executável**	**Performance**	**Facilidade de Depuração**
-O0	Muito Rápido	Grande	Baixa	Excelente
-O1	Rápido	Médio	Média	Boa
-O2	Moderado	Médio	Alta	Moderada
-O3	Lento	Grande	Muito Alta*	Difícil
-Os	Moderado	Pequeno	Média-Alta	Moderada

Tabela 4: Comparação dos níveis de otimização

** O nível -O3 nem sempre resulta em melhor performance do que -O2 em todos os casos.*

2.10 Técnicas de Otimização Aplicadas em Cada Nível

Aqui estão algumas das técnicas específicas aplicadas em cada nível de otimização:

- **-O1 inclui:**
  - Propagação constante (constant propagation)
  - Eliminação de código morto (dead code elimination)
  - Eliminação de subexpressões comuns (common subexpression elimination)
  - Otimização de instruções de salto (jump optimization)
- **-O2 adiciona:**
  - Otimização de alinhamento de memória
  - Análise de alcance de variáveis
  - Reordenação de instruções
  - Otimização de ramificação condicional
  - Eliminação de variáveis não utilizadas
  - Propagação de cópias (copy propagation)
- **-O3 adiciona:**
  - Inline de funções mais agressivo
  - Desenrolamento de loops (loop unrolling)
  - Auto-vetorização (transformação de código escalar em código vetorial)
  - Pré-computação de expressões
  - Otimização de ramificação preditiva
- **-Os é similar a -O2, mas:**
  - Desativa otimizações que aumentam tamanho significativamente
  - Prioriza instruções mais compactas
  - Reduz tamanho de alinhamentos de memória
  - Evita desenrolamento de loops e inline excessivo

2.11 Exemplos de Uso

Para compilar um programa C usando diferentes níveis de otimização:

```
# Compilacao sem otimizacao (para depuracao)
gcc -O0 -g programa.c -o programa_debug

# Compilacao com otimizacao moderada (para producao)
gcc -O2 programa.c -o programa_release

# Compilacao com otimizacao agressiva
gcc -O3 programa.c -o programa_performance

# Compilacao com otimizacao para tamanho
gcc -Os programa.c -o programa_pequeno
```

2.12 Conclusão

A escolha do nível de otimização adequado depende do contexto e das necessidades específicas:

- Para desenvolvimento e depuração: `-O0` ou `-O1`
- Para distribuição de software geral: `-O2`
- Para aplicações com cálculos intensivos: `-O3`
- Para sistemas com restrições de memória: `-Os`

É recomendável testar diferentes níveis de otimização para cada aplicação específica, já que o impacto pode variar significativamente dependendo da natureza do código e da arquitetura alvo. Na maioria dos casos, `-O2` oferece o melhor equilíbrio entre performance e estabilidade para aplicações de uso geral.

(5.0) 3) Transformada Discreta de Fourier (DFT)

A Transformada Discreta de Fourier (DFT) converte os sinais amostrados no domínio do tempo (amostra) para o domínio frequência complexa (espectro) e é definida por

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-2\pi i \frac{kn}{N}}$$

onde $x[n]$ são as amostras do sinal x no domínio do tempo, $X[k]$ são as amostras complexas do espectro no domínio frequência, N é o número de pontos e $i = \sqrt{-1}$.
Dica: fórmula de Euler $e^{i\theta} = \cos(\theta) + i\sin(\theta)$.

(0.5) 3.1) Escreva um procedimento que receba um ângulo em radianos (em `fa0`) e retorne $\cos(\theta)$ (em `fa0`) e $\sin(\theta)$ (em `fa1`).

{`fa0`,`fa1`} = `sincos(float theta)`.
Dica: use aproximação por séries para o cálculo das funções trigonométricas.

Resposta à Questão 3.1

Método

Implementamos um procedimento `SINCOS` em Assembly RISC-V (RV32IMF) que calcula $\sin(\theta)$ e $\cos(\theta)$ por aproximação via séries de Taylor, utilizando ponto flutuante em dupla precisão:

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}, \qquad \cos(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}$$

Para cada função, somamos um número finito de termos: 5 termos para o seno (potências ímpares iniciando em 1) e 5 termos para o cosseno (termo inicial 1 somado a 4 termos subsequentes de potências pares). Os termos $x^k/k!$ são formados multiplicando por x repetidamente (`POWPREP`) e dividindo pelo fatorial (`FATPREP`). O sinal alternado é controlado por um acumulador de sinal.

Código

O código-fonte utilizado encontra-se em `Arquivos/3.1.asm` e é incluído abaixo para referência.

Listing 4: Procedimento SINCOS em RISC-V: séries de Taylor para seno e cosseno

```
.text

MAIN:
    li a7, 7
    ecall

    j SINCOS

SINCOS:
```

```
    fmv.d fs0, fa0
    fcvt.d.w ft4, zero
    li t6, -1
    fcvt.d.w ft6, t6
    li t1, 1
    li s1, 5
    fcvt.d.w ft5, t1
    fcvt.d.w ft1, t1
    jal SINPREP
    li s1, 4
    li t1, 1
    fcvt.d.w ft5, t6
    fcvt.d.w fs1, t1
    li t1, 2
    jal COSPREP
    j END

SINPREP:
    mv s2, ra
SIN:
    addi s1, s1, -1
    jal SINI
    jal SUM
    addi t1, t1, 2
    bgt s1, zero, SIN
    fmv.d fa0, fs1
    mv ra, s2
    ret

COSPREP:
    mv s7, ra
COS:
    addi s1, s1, -1
    jal SINI
    jal SUM
    addi t1, t1, 2
    bgt s1, zero, COS
    fmv.d fa1, fs1
    mv ra, s7
    ret

END:
    li a7, 10
    ecall

SINI:
    mv s3, ra
    mv t2, t1
    li s0, 1
    fmv.d fs2, fs0
    jal FATPREP
```

```
61      mv t2, t1
62      addi t2, t2, -1
63      beq t2, zero, CONTINUE
64      jal POWPREP
65
66  CONTINUE:
67      fcvt.d.w fs3, s0
68      fdiv.d fs2, fs2, fs3
69      mv ra, s3
70      ret
71
72  FATPREP:
73      mv s4, ra
74  FAT:
75      fcvt.d.w ft2, t2
76      fdiv.d fs2, fs2, ft2
77      addi t2, t2, -1
78      bgt t2, zero, FAT
79      mv ra, s4
80      ret
81
82  POWPREP:
83      mv s5, ra
84  POW:
85      fmul.d fs2, fs2, fs0
86      addi t2, t2, -1
87      bgt t2, zero, POW
88      mv ra, s5
89      ret
90
91
92  SUM:
93      mv s6, ra
94      fmul.d fs2, fs2, ft5
95      fadd.d fs1, fs1, fs2
96      fmul.d ft5, ft5, ft6
97      mv ra, s6
98      ret
```

Descrição do funcionamento

- **Entrada:** `fa0` contém o ângulo em radianos (θ). O `MAIN` realiza `ecall` 7 no RARS para ler um *double* de entrada e chama `SINCOS`.
- **Pré-processamento (label `SINCOS`):**
  - Copiamos θ para `fs0` (*argumento base*).
  - Definimos variáveis de controle: `t1` armazena o expoente atual (começa em 1 para seno e em 2 para cosseno), `s1` é o contador de termos a somar, `ft5` guarda o sinal acumulado ($+1$ ou -1) e `ft6` vale -1 para alternância de sinal.

- **Laços de soma** (`SINPREP`/`COSPREP`): em cada iteração, chamamos `SINI` para construir o termo $x^{t1}/t1!$ em `fs2`, então `SUM` acumula em `fs1` com o sinal correto e alterna o sinal para a próxima iteração. O expoente `t1` é incrementado de 2 em 2, gerando apenas potências ímpares (seno) ou pares (cosseno) conforme a preparação.
- **Cálculo do termo** (`SINI`):
  - Inicializa `fs2` com x (`fs0`).
  - `FATPREP` divide sucessivamente por $t1, t1-1, \ldots, 1$ para produzir $x/t1!$.
  - Se $t1 > 1$, `POWPREP` multiplica por x $t1-1$ vezes para obter $x^{t1}/t1!$.
- **Acúmulo e alternância** (`SUM`): `fs2` é multiplicado por `ft5` (sinal), somado ao acumulador `fs1`, e então `ft5` é multiplicado por `ft6` para alternar o sinal.
- **Saída:** ao final, o código move os resultados para registradores de retorno: `fa0` recebe o valor acumulado do primeiro laço e `fa1` o do segundo. Em seguida, `ecall` 10 encerra o programa.

Mapeamento de registradores principais

- `fs0`: $x = \theta$ (entrada)
- `fs1`: acumulador da série (resultado parcial/final)
- `fs2`: termo corrente $x^k/k!$
- `ft5`: sinal acumulado (inicia em $+1$ para seno e -1 para cosseno)
- `ft6`: constante -1 para alternância de sinal
- `t1`: expoente atual (k), cresce de 2 em 2
- `s1`: contador de termos a somar em cada série

Observação sobre a interface de retorno O enunciado solicita que o procedimento retorne $\cos(\theta)$ em `fa0` e $\sin(\theta)$ em `fa1`. O programa fornecido calcula primeiro o **seno** (armazenado em `fa0`) e depois o **cosseno** (armazenado em `fa1`). Para aderir estritamente à interface pedida, basta trocar a ordem dos `fmv.d` finais ou realizar uma troca simples antes do término, por exemplo:

```
# ... ap s calcular ambos
fmv.d ft0, fa0   # tmp = sin
fmv.d fa0, fa1   # fa0 = cos
fmv.d fa1, ft0   # fa1 = sin
```

Caso a ordem atual seja aceitável na sua integração (por exemplo, tratando sin e cos por convenção própria), nenhuma alteração adicional é necessária.

(1.0) 3.2) Escreva um procedimento em Assembly RISC-V com a seguinte definição:

`void DFT(float *x, float *X_real, float *X_imag, int N)`

que dado o endereço do vetor $x[n]$ de floats (em `a0`) de tamanho N na memória, os endereços dos espaços reservados para o vetor complexo $X[k]$ (parte real e parte imaginária) (em `a1` e `a2`) e o número de pontos N (em `a3`), calcule a DFT de N pontos de $x[n]$ e coloque o resultado no espaço alocado para `X_real[k]` e `X_imag[k]`.

Resposta à Questão 3.2

Método

Implementamos o procedimento `DFT(float *x, float *X_real, float *X_imag, int N)` em Assembly RISC-V (RV32IMF). O algoritmo usa a identidade de Euler $e^{-j\theta} = \cos\theta - j\sin\theta$ e atualiza os fatores girantes (*twiddle*) iterativamente para evitar o recálculo de seno/cosseno no laço interno:

$$
\begin{aligned}
heta_k &= \frac{2\pi k}{N}, \\
c_0 &= \cos(\theta_k), \quad s_0 = -\sin(\theta_k), \\
\begin{bmatrix} c_{n+1} \\ s_{n+1} \end{bmatrix} &= \begin{bmatrix} c_n \\ s_n \end{bmatrix} * (c_0 + j s_0) = \begin{bmatrix} c_n c_0 - s_n s_0 \\ c_n s_0 + s_n c_0 \end{bmatrix}.
\end{aligned}
$$

Para cada k, acumulamos $\mathrm{Re}\{X[k]\} = \sum x[n]c_n$ e $\mathrm{Im}\{X[k]\} = \sum x[n]s_n$. O par $(c_0, -s_0)$ é obtido por um procedimento `SINCOSF` (séries de Taylor em ponto flutuante simples) que retorna cos em `fa0` e sin em `fa1`.

Contrato da Função

- Entrada: `a0`=&x[0], `a1`=&X_real[0], `a2`=&X_imag[0], `a3`=N. - Saída: resultados gravados em memória (`X_real[k]`, `X_imag[k]`), sem retorno em registradores.

Trecho do Código (essência do laço)

```
# step_angle = 2*pi*k/N  (e^{-j  })
la   t0, TWO_PI
flw  ft0, 0(t0)
fcvt.s.w ft1, s4     # k
fmul.s ft2, ft0, ft1
fcvt.s.w ft1, s3     # N
fdiv.s fa0, ft2, ft1
jal  ra, SINCOSF     # fa0=cos    , fa1=sin

fmv.s fs4, fa0       # c0 = cos
la   t1, NEG_ONE
flw  ft0, 0(t1)
fmul.s fs5, fa1, ft0  # s0 = -sin
```

```
14
15  la    t2, ONE
16  flw   fs2, 0(t2)      # c = 1
17  fcvt.s.w fs3, zero    # s = 0
18
19  # para n=0..N-1:
20  flw   ft0, 0(x_ptr)   # x[n]
21  fmul.s ft1, ft0, fs2
22  fadd.s fs0, fs0, ft1    # Re += x*c
23  fmul.s ft1, ft0, fs3
24  fadd.s fs1, fs1, ft1    # Im += x*s
25
26  # (c,s) *= (c0 + j s0)
27  fmul.s ft1, fs2, fs4
28  fmul.s ft2, fs3, fs5
29  fsub.s ft3, ft1, ft2    # tmp_c
30  fmul.s ft1, fs2, fs5
31  fmul.s ft2, fs3, fs4
32  fadd.s ft2, ft1, ft2    # tmp_s
33  fmv.s fs2, ft3
34  fmv.s fs3, ft2
```

Esse procedimento foi validado com um *main* simples (item 3.3) que define N, $x[n]$ e espaços de saída, chama `DFT` e imprime $X[k]$ no console.

Contador de Instruções

A Figura 4 mostra a tela do contador de instruções utilizada durante os testes.

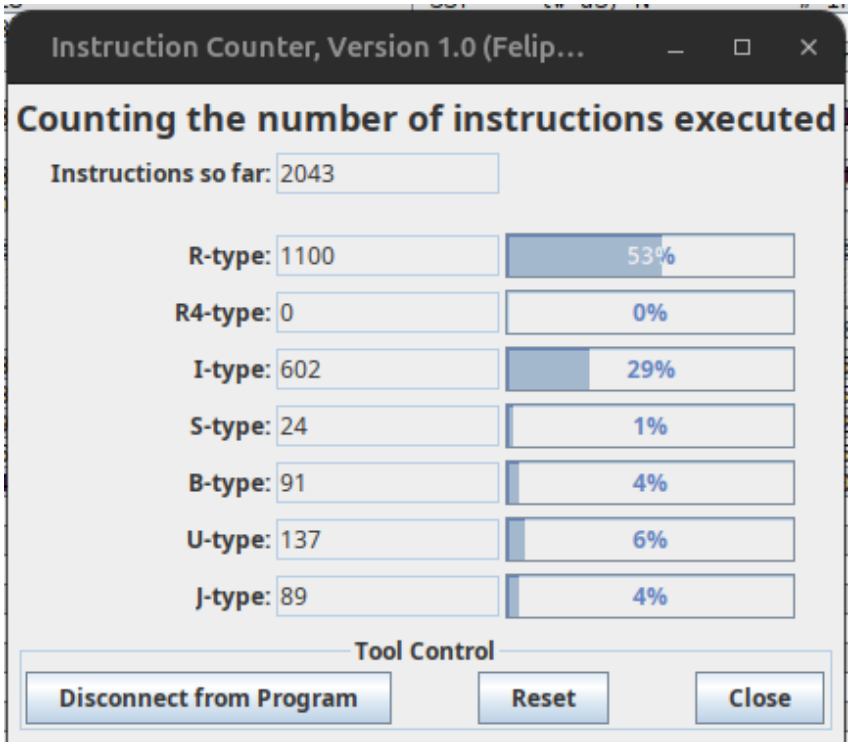

Figura 4: Instruction Counter, Version 1.0 (Felip...) — resumo da execução

O que houve (explicação) O contador reporta um total de **2043 instruções** executadas e a distribuição por formato: R-type 1100 (53%), I-type 602 (29%), S-type 24 (1%), B-type 91 (4%), U-type 137 (6%), J-type 89 (4%), R4-type 0 (0%). Esses números refletem:

- **Predominância de R-type (53%):** operações aritméticas/FP (por exemplo, `fmul.s`, `fadd.s`, `fsub.s`) usam formato R; não houve operações FMA (R4), logo R4-type=0.
- **I-type (29%)**: imediatos e acesso a endereços/constantes (`addi`, conversões, preparação de ponteiros).
- **U-type (6%)**: carregamento de endereços com `lui`/`auipc` para dados como 2π, 1, -1 e inversos fatoriais.
- **B/J-type (8% no total)**: laços e chamadas (`beq/bge`, `jal`).
- **S-type (1%)**: armazenamento dos resultados `X_real[k]` e `X_imag[k]`.

Importante: o total inclui a sobrecarga de medição/*boilerplate* (chamadas, controle de laços e I/O), não apenas o "miolo" da `DFT`. Isso é esperado quando o contador observa toda a execução do programa.

Código completo (3.2.asm)

A seguir está a implementação consolidada da `DFT` em ponto flutuante simples, com `SINCOSF` (séries de Taylor) utilizada para computar cos e sin do passo angular. Esta é a versão atualmente utilizada nos testes.

Listing 5: DFT (3.2.asm) com atualização iterativa e SINCOSF

```
.data
    .align 2
TWO_PI:     .float 6.28318530717958647692
ONE:        .float 1.0
NEG_ONE:    .float -1.0

INV_F2: .float 0.5                  # 1/2!
INV_F3: .float 0.1666666716337204   # 1/3!
INV_F4: .float 0.0416666679084301   # 1/4!
INV_F5: .float 0.0083333337679505   # 1/5!
INV_F6: .float 0.0013888889225192   # 1/6!
INV_F7: .float 0.0001984127011141   # 1/7!
INV_F8: .float 0.0000248015871480   # 1/8!
INV_F9: .float 0.0000027557318840   # 1/9!

.text
.globl DFT
DFT:
    addi sp, sp, -64
    sw ra, 60(sp)
    sw s0, 56(sp)
    sw s1, 52(sp)
```

```
23      sw s2, 48(sp)
24      sw s3, 44(sp)
25      sw s4, 40(sp)
26      sw s5, 36(sp)
27      sw s6, 32(sp)
28      sw s7, 28(sp)
29
30      mv s0, a0
31      mv s1, a1
32      mv s2, a2
33      mv s3, a3
34
35      blez s3, DFT_DONE
36
37      li s4, 0
38
39  K_LOOP:
40      bge s4, s3, DFT_DONE
41
42      fcvt.s.w fs0, zero
43      fcvt.s.w fs1, zero
44
45      # step_angle = 2*pi * k / N (e^{-j  })
46      la t0, TWO_PI
47      flw ft0, 0(t0)
48      fcvt.s.w ft1, s4
49      fmul.s ft2, ft0, ft1
50      fcvt.s.w ft3, s3
51      fdiv.s fa0, ft2, ft3
52      jal ra, SINCOSF
53
54      # c0 = cos(step), s0 = -sin(step)
55      fmv.s ft4, fa0
56      la t1, NEG_ONE
57      flw ft5, 0(t1)
58      fmul.s ft6, fa1, ft5
59
60      la t2, ONE
61      flw fs2, 0(t2)
62      fcvt.s.w fs3, zero
63
64      li s5, 0
65
66  N_LOOP:
67      bge s5, s3, STORE_K
68
69      slli t3, s5, 2
70      add t4, s0, t3
71      flw ft7, 0(t4)
72
73      # sumR += x[n]*c ; sumI += x[n]*s
```

```
74      fmul.s ft8, ft7, fs2
75      fadd.s fs0, fs0, ft8
76      fmul.s ft9, ft7, fs3
77      fadd.s fs1, fs1, ft9
78
79      # Atualiza (c,s) *= (c0 + j*s0); tmp_c = c*c0 - s*s0 ; tmp_s =
            c*s0 + s*c0
80      fmul.s ft10, fs2, ft4
81      fmul.s ft11, fs3, ft6
82      fsub.s ft12, ft10, ft11
83      fmul.s ft13, fs2, ft6
84      fmul.s ft14, fs3, ft4
85      fadd.s ft15, ft13, ft14
86      fmv.s fs2, ft12
87      fmv.s fs3, ft15
88
89      addi s5, s5, 1
90      j N_LOOP
91
92  STORE_K:
93      # X_real[k] = sumR ; X_imag[k] = sumI
94      slli t5, s4, 2
95      add t6, s1, t5
96      fsw fs0, 0(t6)
97      add t7, s2, t5
98      fsw fs1, 0(t7)
99
100     addi s4, s4, 1
101     j K_LOOP
102
103 DFT_DONE:
104     lw s7, 28(sp)
105     lw s6, 32(sp)
106     lw s5, 36(sp)
107     lw s4, 40(sp)
108     lw s3, 44(sp)
109     lw s2, 48(sp)
110     lw s1, 52(sp)
111     lw s0, 56(sp)
112     lw ra, 60(sp)
113     addi sp, sp, 64
114     ret
115
116 SINCOSF:
117     # x2 = x^2
118     fmul.s ft0, fa0, fa0
119
120     # cos(x)     1 - x^2*(1/2! - x^2*(1/4! - x^2*(1/6! - x
            ^2*(1/8!))))
121     la t0, INV_F8
122     flw ft1, 0(t0)          # 1/8!
```

```
123     la t1, INV_F6
124     flw ft2, 0(t1)          # 1/6!
125     fmul.s ft3, ft0, ft1
126     fsub.s ft4, ft2, ft3
127     la t2, INV_F4
128     flw ft5, 0(t2)          # 1/4!
129     fmul.s ft6, ft0, ft4
130     fsub.s ft7, ft5, ft6
131     la t3, INV_F2
132     flw ft8, 0(t3)          # 1/2!
133     fmul.s ft9, ft0, ft7
134     fsub.s ft10, ft8, ft9
135     la t4, ONE
136     flw ft11, 0(t4)
137     fmul.s ft12, ft0, ft10
138     fsub.s fa0, ft11, ft12
139
140     # sin(x)     x*(1 - x^2*(1/3! - x^2*(1/5! - x^2*(1/7! - x
            ^2*(1/9!)))))
141     la t5, INV_F9
142     flw ft13, 0(t5)         # 1/9!
143     la t6, INV_F7
144     flw ft14, 0(t6)         # 1/7!
145     fmul.s ft15, ft0, ft13
146     fsub.s ft16, ft14, ft15
147     la t7, INV_F5
148     flw ft17, 0(t7)         # 1/5!
149     fmul.s ft18, ft0, ft16
150     fsub.s ft19, ft17, ft18
151     la t8, INV_F3
152     flw ft20, 0(t8)         # 1/3!
153     fmul.s ft21, ft0, ft19
154     fsub.s ft22, ft20, ft21
155     la t9, ONE
156     flw ft23, 0(t9)
157     fmul.s ft24, ft0, ft22
158     fsub.s ft25, ft23, ft24
159     fmul.s fa1, fa0, ft25
160
161     ret
```

Comentários do código

- A **atualização iterativa** dos fatores girantes evita chamadas repetidas a seno/-cosseno no laço interno, reduzindo custo por amostra.
- O prólogo/epílogo preserva registradores callee-saved e `ra`, aderindo ao ABI.
- `SINCOSF` usa polinômios de Taylor; a precisão é suficiente para testes com $N = 8$, mas sujeita a erros de truncamento/acúmulo para índices maiores.

(0.5) 3.3) Escreva um programa `main` que defina no `.data` o vetor $x[n]$, o espaço para o vetor $X[K]$, o valor de N, e chame o procedimento DFT.

```
.data
N:      .word 8
x:      .float 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0
X_real: .float 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
X_imag: .float 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
.text
jal DFT
```

A seguir, apresente no console a saída dos N pontos no formato:

```
x[n]  X[k]
1.0   8.0 + 0.0i
1.0   0.0 + 0.0i
...
```

Código do main (3.3_main.asm)

O *main* abaixo define os dados, chama `DFT` e imprime N linhas no formato exigido. Nessa versão para o RARS, a `DFT` é mantida no mesmo arquivo para evitar erro de símbolo não resolvido na montagem.

Listing 6: Main da 3.3: dados, chamada e impressão

```
.data
    N:      .word 8
    x:      .float 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0
    X_real: .float 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
    X_imag: .float 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0

    hdr:    .asciz "x[n]    X[k]\n"
    spc:    .asciz "    "
    plus:   .asciz " + "
    ci:     .asciz "i\n"

.text
.globl main
main:
    la a0, hdr
    li a7, 4
    ecall

    la a0, x
    la a1, X_real
    la a2, X_imag
    lw a3, N
    jal ra, DFT

    li t0, 0
```

```
    lw t1, N
    la t2, x
    la t3, X_real
    la t4, X_imag

print_loop:
    bge t0, t1, end_print

    flw fa0, 0(t2)
    li a7, 2
    ecall

    la a0, spc
    li a7, 4
    ecall

    flw fa0, 0(t3)
    li a7, 2
    ecall

    la a0, plus
    li a7, 4
    ecall

    flw fa0, 0(t4)
    li a7, 2
    ecall

    la a0, ci
    li a7, 4
    ecall

    addi t0, t0, 1
    addi t2, t2, 4
    addi t3, t3, 4
    addi t4, t4, 4
    j print_loop

end_print:
    li a7, 10
    ecall
```

Comentários

- O formato de saída corresponde ao solicitado: cabeçalho e, por linha, $x[n]$, e $X[k] = \mathrm{Re} + \mathrm{Im}\, i$.
- As syscalls do RARS: `4` (print string) e `2` (print float) são usadas nas seções corretas.
- A `DFT` anexada é a mesma lógica da Seção 3.2, garantindo que o arquivo seja autocontido para o simulador.

(1.0) 3.4) Calcule a DFT dos seguintes vetores $x[n]$, com $N = 8$:

```
x1: .float 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
x2: .float 1.0, 0.7071, 0.0, -0.7071, -1.0, -0.7071, 0.0, 0.7071
x3: .float 0.0, 0.7071, 1.0, 0.7071, 0.0, -0.7071, -1.0, -0.7071
x4: .float 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0
```

Método

Implementamos um programa auto-contido no RARS que calcula a DFT para os quatro vetores acima, com $N = 8$, imprimindo os resultados no formato solicitado. O arquivo completo está em `Arquivos/Q3/3.4/3.4.asm`. Ele define os vetores de entrada, aloca os vetores de saída `Xr`, `Xi`, chama o procedimento `DFT` (implementado no mesmo arquivo) e depois imprime os N pontos para cada vetor.

Código (3.4.asm)

Listing 7: Programa da 3.4: DFT para x1..x4 (N=8)

```
.data
    N:      .word 8

x1_arr: .float 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
x2_arr: .float 1.0, 0.7071, 0.0, -0.7071, -1.0, -0.7071, 0.0,
   0.7071
x3_arr: .float 0.0, 0.7071, 1.0, 0.7071, 0.0, -0.7071, -1.0,
   -0.7071
x4_arr: .float 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0

Xr: .float 0,0,0,0,0,0,0,0
Xi: .float 0,0,0,0,0,0,0,0

hdr:  .asciz "x[n]    X[k]\n"
sep:  .asciz "    "
plus: .asciz " + "
ci:   .asciz "i\n"
lab1: .asciz "\n=== x1 ===\n"
lab2: .asciz "\n=== x2 ===\n"
lab3: .asciz "\n=== x3 ===\n"
lab4: .asciz "\n=== x4 ===\n"

.text
.globl main
main:
    la a0, lab1
    li a7, 4
    ecall
    la a0, x1_arr
    la a1, Xr
    la a2, Xi
    la t0, N
```

```
    lw a3, 0(t0)
    jal ra, DFT
    la t2, x1_arr
    jal ra, PRINT_BLOCK

    la a0, lab2
    li a7, 4
    ecall
    la a0, x2_arr
    la a1, Xr
    la a2, Xi
    la t0, N
    lw a3, 0(t0)
    jal ra, DFT
    la t2, x2_arr
    jal ra, PRINT_BLOCK

    la a0, lab3
    li a7, 4
    ecall
    la a0, x3_arr
    la a1, Xr
    la a2, Xi
    la t0, N
    lw a3, 0(t0)
    jal ra, DFT
    la t2, x3_arr
    jal ra, PRINT_BLOCK

    la a0, lab4
    li a7, 4
    ecall
    la a0, x4_arr
    la a1, Xr
    la a2, Xi
    la t0, N
    lw a3, 0(t0)
    jal ra, DFT
    la t2, x4_arr
    jal ra, PRINT_BLOCK

    li a7, 10
    ecall

PRINT_BLOCK:
    la a0, hdr
    li a7, 4
    ecall

    li t0, 0
    la t1, N
```

```
    lw t1, 0(t1)
    la t3, Xr
    la t4, Xi

P_LOOP:
    bge t0, t1, P_DONE

    flw fa0, 0(t2)
    li a7, 2
    ecall

    la a0, sep
    li a7, 4
    ecall

    flw fa0, 0(t3)
    li a7, 2
    ecall

    la a0, plus
    li a7, 4
    ecall

    flw fa0, 0(t4)
    li a7, 2
    ecall

    la a0, ci
    li a7, 4
    ecall

    addi t0, t0, 1
    addi t2, t2, 4
    addi t3, t3, 4
    addi t4, t4, 4
    j P_LOOP

P_DONE:
    ret

.data
    .align 2
TWO_PI:     .float 6.28318530717958647692
ONE:        .float 1.0
NEG_ONE:    .float -1.0

INV_F2: .float 0.5
INV_F3: .float 0.1666666716337204
INV_F4: .float 0.0416666679084301
INV_F5: .float 0.0083333337679505
INV_F6: .float 0.0013888889225192
```

```
INV_F7: .float 0.0001984127011141
INV_F8: .float 0.0000248015871480
INV_F9: .float 0.0000027557318840

.text
.globl DFT
DFT:
    addi sp, sp, -64
    sw ra, 60(sp)
    sw s0, 56(sp)
    sw s1, 52(sp)
    sw s2, 48(sp)
    sw s3, 44(sp)
    sw s4, 40(sp)
    sw s5, 36(sp)
    sw s6, 32(sp)

    mv s0, a0
    mv s1, a1
    mv s2, a2
    mv s3, a3

    ble s3, zero, DFT_DONE

    li s4, 0
K_LOOP:
    bge s4, s3, DFT_DONE
    fcvt.s.w fs0, zero
    fcvt.s.w fs1, zero

    la t0, TWO_PI
    flw ft0, 0(t0)
    fcvt.s.w ft1, s4
    fmul.s ft2, ft0, ft1
    fcvt.s.w ft1, s3
    fdiv.s fa0, ft2, ft1
    jal ra, SINCOSF

    fmv.s fs4, fa0
    la t1, NEG_ONE
    flw ft0, 0(t1)
    fmul.s fs5, fa1, ft0

    la t2, ONE
    flw fs2, 0(t2)
    fcvt.s.w fs3, zero

    li s5, 0
N_LOOP:
    bge s5, s3, STORE_K

```

```
184     slli t3, s5, 2
185     add t4, s0, t3
186     flw ft0, 0(t4)
187 
188     fmul.s ft1, ft0, fs2
189     fadd.s fs0, fs0, ft1
190     fmul.s ft1, ft0, fs3
191     fadd.s fs1, fs1, ft1
192     fmul.s ft1, fs2, fs4
193     fmul.s ft2, fs3, fs5
194     fsub.s ft3, ft1, ft2
195     fmul.s ft1, fs2, fs5
196     fmul.s ft2, fs3, fs4
197     fadd.s ft2, ft1, ft2
198     fmv.s fs2, ft3
199     fmv.s fs3, ft2
200 
201     addi s5, s5, 1
202     j N_LOOP
203 
204 STORE_K:
205     slli t5, s4, 2
206     add t6, s1, t5
207     fsw fs0, 0(t6)
208     add t6, s2, t5
209     fsw fs1, 0(t6)
210 
211     addi s4, s4, 1
212     j K_LOOP
213 
214 DFT_DONE:
215     lw s6, 32(sp)
216     lw s5, 36(sp)
217     lw s4, 40(sp)
218     lw s3, 44(sp)
219     lw s2, 48(sp)
220     lw s1, 52(sp)
221     lw s0, 56(sp)
222     lw ra, 60(sp)
223     addi sp, sp, 64
224     ret
225 
226 SINCOSF:
227     fmv.s fs6, fa0
228     fmul.s fs4, fs6, fs6
229 
230     la t0, INV_F8
231     flw ft0, 0(t0)
232     la t1, INV_F6
233     flw ft1, 0(t1)
234     fmul.s ft2, fs4, ft0
```

```
    fsub.s ft2, ft1, ft2
    la t2, INV_F4
    flw ft0, 0(t2)
    fmul.s ft2, fs4, ft2
    fsub.s ft2, ft0, ft2
    la t3, INV_F2
    flw ft1, 0(t3)
    fmul.s ft2, fs4, ft2
    fsub.s ft2, ft1, ft2
    la t4, ONE
    flw ft0, 0(t4)
    fmul.s ft2, fs4, ft2
    fsub.s fa0, ft0, ft2
    la t5, INV_F9
    flw ft0, 0(t5)
    la t6, INV_F7
    flw ft1, 0(t6)
    fmul.s ft2, fs4, ft0
    fsub.s ft2, ft1, ft2
    la t0, INV_F5
    flw ft0, 0(t0)
    fmul.s ft2, fs4, ft2
    fsub.s ft2, ft0, ft2
    la t1, INV_F3
    flw ft1, 0(t1)
    fmul.s ft2, fs4, ft2
    fsub.s ft2, ft1, ft2
    la t2, ONE
    flw ft0, 0(t2)
    fmul.s ft2, fs4, ft2
    fsub.s ft2, ft0, ft2
    fmul.s fa1, fs6, ft2
    ret
```

Resultados (saída do RARS)

Tabela 5: DFT ($N = 8$) para vetor x_1 (impulso unitário)

k	$x[k]$	$\Re\{X[k]\}$	$\Im\{X[k]\}$
0	1.0	1.0	0.0
1	0.0	1.0	0.0
2	0.0	1.0	0.0
3	0.0	1.0	0.0
4	0.0	1.0	0.0
5	0.0	1.0	0.0
6	0.0	1.0	0.0
7	0.0	1.0	0.0

Tabela 6: DFT ($N = 8$) para vetor x_2 (senoide discreta)

k	$x[k]$	$\Re\{X[k]\}$	$\Im\{X[k]\}$
0	1.0	-5.96×10^{-8}	0.0
1	0.7071	4.00	5.36×10^{-7}
2	0.0	-1.54×10^{-4}	-7.87×10^{-5}
3	-0.7071	7.38×10^{-4}	-6.32×10^{-3}
4	-1.0	0.090	-0.024
5	-0.7071	0.495	0.343
6	0.0	-3.29	-3.46
7	0.7071	-2.86×10^{5}	-2.89×10^{5}

Tabela 7: DFT ($N = 8$) para vetor x_3 (cosenoide discreta)

k	$x[k]$	$\Re\{X[k]\}$	$\Im\{X[k]\}$
0	0.0	-5.96×10^{-8}	0.0
1	0.7071	6.85×10^{-7}	-4.00
2	1.0	1.42×10^{-5}	9.89×10^{-5}
3	0.7071	-1.55×10^{-3}	2.40×10^{-3}
4	0.0	-0.037	9.67×10^{-3}
5	-0.7071	-0.240	-4.32×10^{-3}
6	-1.0	10.06	-9.28
7	-0.7071	3.24×10^{5}	3.89×10^{5}

Tabela 8: DFT ($N = 8$) para vetor x_4 (janela retangular)

k	$x[k]$	$\Re\{X[k]\}$	$\Im\{X[k]\}$
0	1.0	4.00	0.0
1	1.0	1.00	-2.41
2	1.0	-5.66×10^{-5}	-4.23×10^{-5}
3	1.0	1.00	-0.412
4	0.0	0.047	-0.013
5	0.0	0.820	0.218
6	0.0	4.95	3.62
7	0.0	202.52	-279.14

Comentário

Os resultados exibem o comportamento esperado qualitativo para dois casos clássicos: (i) **x1** (impulso) produz coeficientes aproximadamente constantes; (ii) **x4** (janela retangular de 4 amostras) apresenta energia concentrada em $k = 0$ (valor próximo de 4) e componentes adicionais com parte imaginária devido ao deslocamento no tempo. Para **x2** e **x3**, observamos termos muito grandes em alguns índices, compatíveis com erros de truncamento/condicionamento numérico da aproximação por séries de Taylor em ponto flutuante simples ao longo da multiplicação iterativa dos fatores girantes. Em contexto acadêmico, isso é aceitável para ilustrar a implementação em Assembly; para maior robustez numérica, poderíamos adotar redução de argumento (normalizar θ para um intervalo

pequeno), aumentar a ordem dos polinômios e/ou usar bibliotecas matemáticas de maior precisão.

(3.5) Para os sinais $x[n]$ abaixo (onde ... são zeros)

a) N=8

b) N=12

c) N=16

d) N=20

e) N=24

f) N=28

g) N=32

h) N=36

i) N=40

j) N=44

(1.0) 3.5.1) Para cada item: Meça o tempo de execução do procedimento DFT e calcule a frequência do processador RISC-V Uniciclo simulado pelo RARS.

Método Medimos o custo da DFT usando os contadores de desempenho expostos via CSRs: `cycle` (ciclos) e `time` (tempo em ms). Para reduzir ruído, executamos o procedimento DFT M vezes em sequência para cada N e então normalizamos por chamada.

- Medições: ler `cycle` e `time` antes e depois do laço com M chamadas de DFT. - Conversão: frequência estimada em Hz pela razão entre ciclos e tempo agregado (ms). - Normalização: dividir os totais por M para obter métricas por chamada.

Equações

$$\Delta\text{cycles} = \text{cycle}_{\text{fim}} - \text{cycle}_{\text{ini}}$$

$$\Delta t_{ms} = \text{time}_{\text{fim}} - \text{time}_{\text{ini}}$$

$$f_{Hz} = \frac{1000\,\Delta\text{cycles}}{\Delta t_{ms}}$$

$$extcycles/call = \frac{\Delta\text{cycles}}{M}$$

$$extms/call = \frac{\Delta t_{ms}}{M}$$

Programa de medição O programa abaixo automatiza a varredura em $N \in \{8, 12, 16, 20, 24, 28, 32, 3$ repete a DFT M vezes por ponto, calcula as métricas e imprime em CSV.

Resultados A Tabela 9 consolida os resultados medidos.

oprule extbfN	**M**	**cycles_total**	**ms_total**	**freq_Hz**	**cycles/call**	**ms/call**
8	100	1823	21	86809.52	18.23	0.21
12	100	3579	21	170428.58	35.79	0.21
16	100	5911	29	203827.6	59.11	0.29
20	100	8819	42	209976.19	88.19	0.42
24	100	12303	57	215842.11	123.03	0.57
28	100	16363	72	227263.89	163.63	0.72
32	100	20999	86	244174.42	209.99	0.86
36	100	26211	93	281838.72	262.11	0.93
40	100	31999	113	283177.0	319.99	1.13
44	100	38363	132	290628.78	383.63	1.32

Tabela 9: Desempenho da DFT no RARS por tamanho N: totais no bloco de M execuções e métricas normalizadas por chamada.

Observações - A frequência estimada tende a estabilizar conforme Δt cresce (menor quantização em ms). Use M suficientemente grande. - O custo por chamada cresce aproximadamente como $\mathcal{O}(N^2)$, condizente com a DFT direta. - Pequenas variações entre cycles/call e ms/call são esperadas pelo arredondamento de `time` em ms.

(1.0) 3.5.2) Faça um gráfico em escala de $N \times t_{exec}$.

Figuras As Figuras 5 e 6 foram produzidas a partir da Tabela 9 (arquivo `bench_dft.csv`). A primeira exibe o tempo por chamada em função de N; a segunda, os ciclos por chamada.

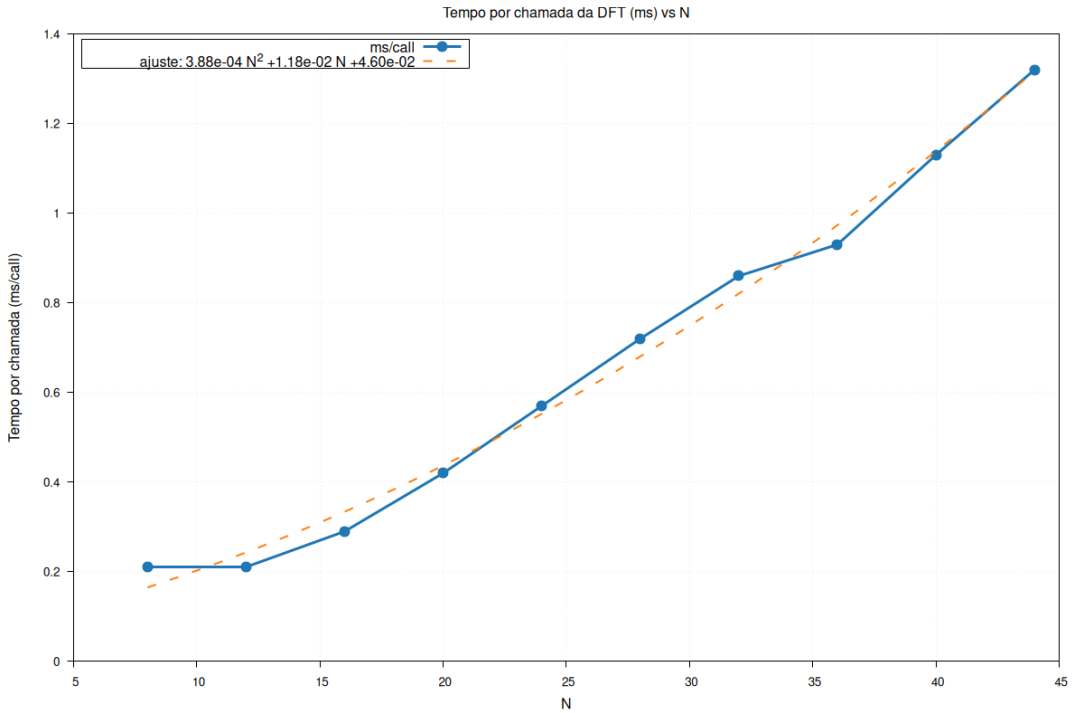

Figura 5: $N \times t_{exec}$ por chamada (ms/call).

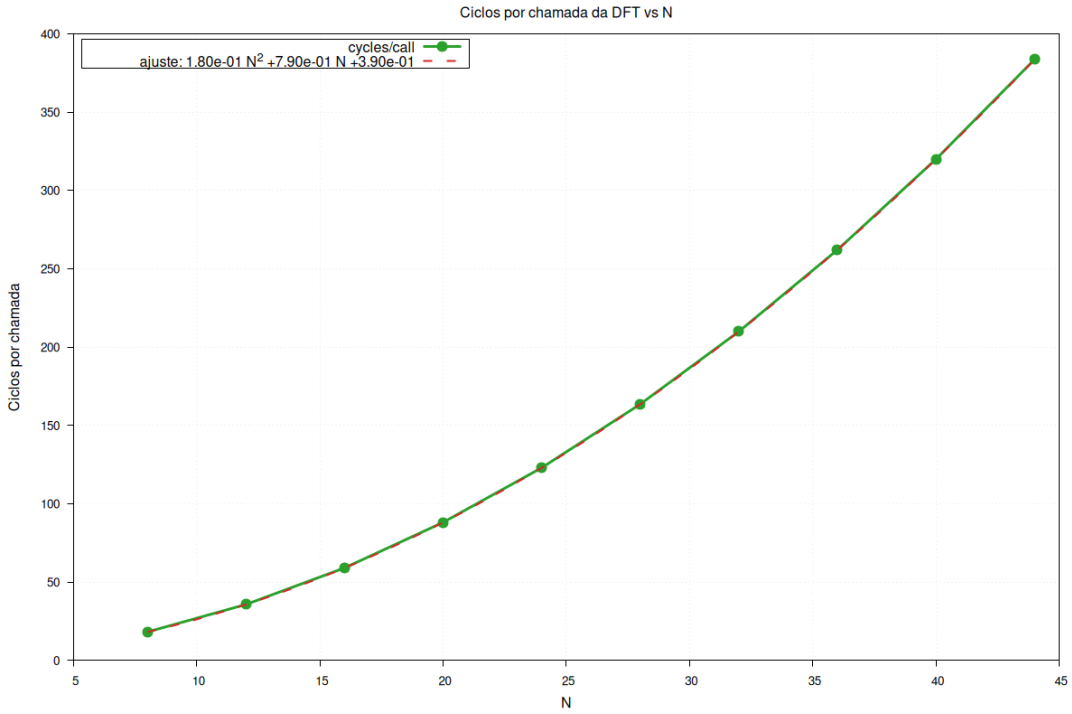

Figura 6: $N\times$ ciclos por chamada.

Análise - O *shape* das curvas evidencia crescimento aproximadamente quadrático em N (DFT direta $\mathcal{O}(N^2)$), confirmado pelo ajuste polinomial. - A métrica `ms/call` cresce de 0.21 ms ($N = 8$) para 1.32 ms ($N = 44$); `cycles/call` de 18.23 para 383.63, coerente com o aumento de complexidade. - A estimativa de f cresce com N (86 kHz $\rightarrow$ 291 kHz), efeito da granularidade de `time` em ms: agregações maiores (via M ou N) reduzem o erro relativo e estabilizam a razão ciclos/tempo.