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)

14 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.

Objetivos

- Familiarizar o aluno com o Simulador/Montador RARS;
- Desenvolver a capacidade de codificação de algoritmos em linguagem Assembly;
- Desenvolver a capacidade de análise de desempenho de algoritmos em Assembly;
- Familiarizar o aluno com a compilação C para Assembly RISC-V RV32IMF.

(2.5) 1) Simulador/Montador RARS

Faça o download e deszipe o arquivo `Lab1.zip` disponível no Moodle.

(0.0) 1.1) No diretório Arquivos, abra o `Rars16_Custom1` e carregue o programa de ordenamento `sort.s`.

Dado o vetor:

$$V[30] = \{9, 2, 5, 1, 8, 2, 4, 3, 6, 7, 10, 2, 32, 54, 2, 12, 6, 3, 1, 78, 54, 23, 1, 54, 2, 65, 3, 6, 55, 31\}$$

(a) ordená-lo em ordem crescente e contar o número de instruções por tipo e o número total exigido pelo procedimento `sort`. Qual o tamanho em bytes do código executável? E da memória de dados usada?

(b) Modifique o programa para ordenar o vetor em ordem decrescente e contar o número de instruções por tipo e o número total exigido pelo procedimento `sort`.

(c) Usando os contadores de instruções e tempo do Banco de Registradores CSR (veja no final), meça novamente a quantidade de instruções executadas e o tempo de execução dos itens (a) e (b).

(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.

Resposta à Questão 1.2

A seguir são apresentadas as respostas para os itens da questão 1.2, com base no relatório fornecido.

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.

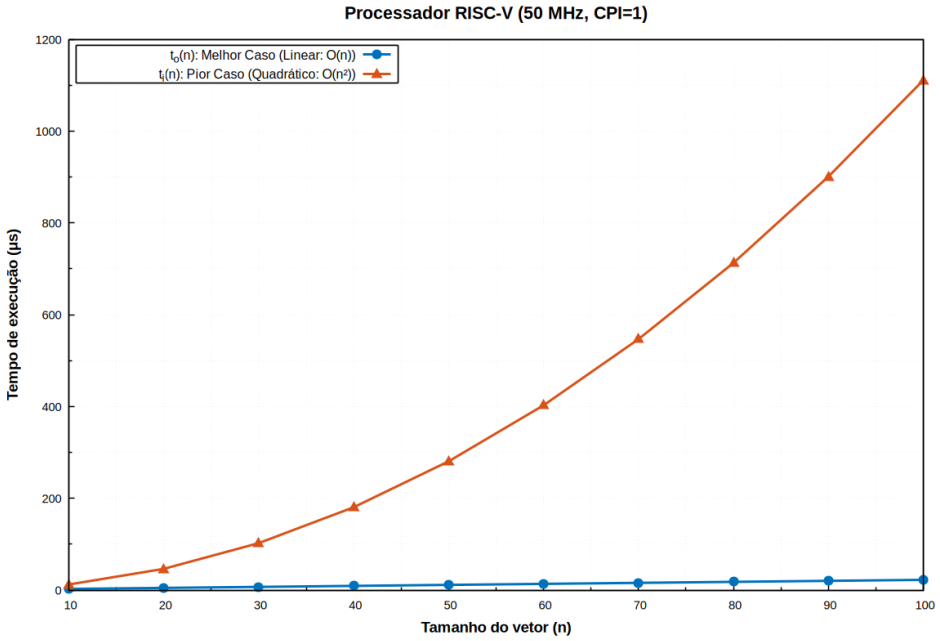

Figura 1: Tempo de execução vs. tamanho do vetor n para melhor e pior caso.

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 con-

traste, 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 ($22.28\,\mu s$).

Código em GNU Octave para Geração do Gráfico

A seguir é apresentado o código em GNU Octave utilizado para gerar o gráfico da Figura 1.

```
% Valores de n (tamanho do vetor)
n = 10:10:100;

% Equacoes de ciclos para melhor e pior caso
C_best = 11*n + 14;
C_worst = 0.5*11*n.^2 + 0.5*11*n + 3;

% Conversao para tempo em microssegundos (us)
t_best = C_best / 50;
t_worst = C_worst / 50;

% Plotagem do grafico
figure;
plot(n, t_best, 'bo-', 'LineWidth', 2, 'DisplayName', 't_o(n): melhor caso');
hold on;
plot(n, t_worst, 'rx-', 'LineWidth', 2, 'DisplayName', 't_i(n): pior caso');
hold off;

% Configuracao do grafico
xlabel('Tamanho do vetor n');
ylabel('Tempo de execucao (\mus)');
title('Tempo de Execucao vs. n');
legend('show');
grid on;
```

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.

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

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.

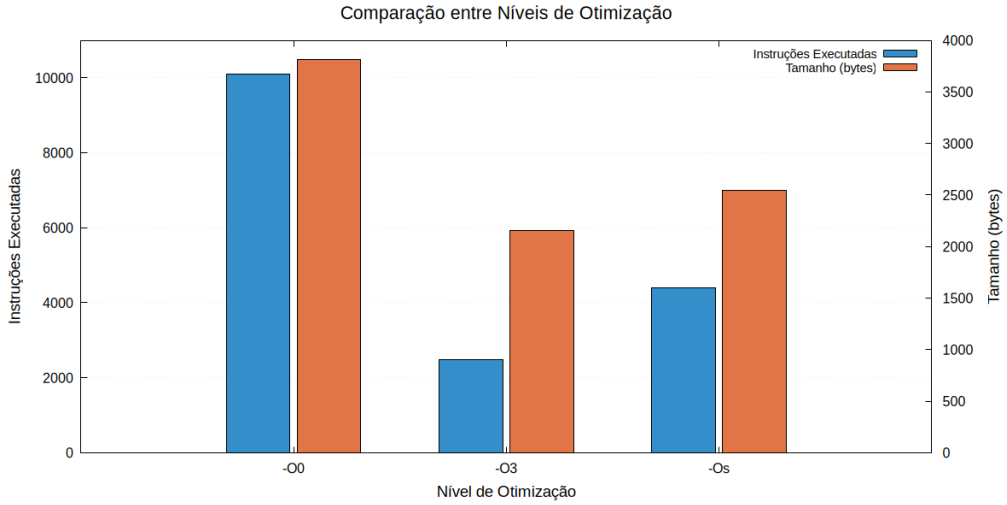

Figura 2: Comparação visual entre níveis de otimização: número de instruções e tamanho do arquivo.

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)
```

```
12 addi s0, sp, 16
13 li a1, 30
14 lui a5, %hi(v)
15 addi a0, a5, %lo(v)
16 call show
17 li a1, 30
18 lui a5, %hi(v)
19 addi a0, a5, %lo(v)
20 call sort
21 li a1, 30
22 lui a5, %hi(v)
23 addi a0, a5, %lo(v)
24 call show
25 li a5, 0
26 mv a0, a5
27 lw ra, 12(sp)
28 lw s0, 8(sp)
29 addi sp, sp, 16
30 li a7, 10
31 ecall
32
33 show:
34 addi sp, sp, -32
35 sw ra, 28(sp)
36 sw s0, 24(sp)
37 addi s0, sp, 32
38 sw a0, -20(s0)
39 sw a1, -24(s0)
40 lw a5, -20(s0)
41 lw a4, -24(s0)
42 mv t0, a5
43 mv t1, a4
44 mv t2, zero
45 loop1:
46 beq t2, t1, fim1
47 li a7, 1
48 lw a0, 0(t0)
49 ecall
50 li a7, 11
51 li a0, 9
52 ecall
53 addi t0, t0, 4
54 addi t2, t2, 1
55 j loop1
56 fim1:
57 li a7, 11
58 li a0, 10
59 ecall
60 nop
61 lw ra, 28(sp)
62 lw s0, 24(sp)
```

```
63  addi sp, sp, 32
64  jr ra
65
66  swap:
67  addi sp, sp, -48
68  sw ra, 44(sp)
69  sw s0, 40(sp)
70  addi s0, sp, 48
71  sw a0, -36(s0)
72  sw a1, -40(s0)
73  lw a5, -40(s0)
74  slli a5, a5, 2
75  lw a4, -36(s0)
76  add a5, a4, a5
77  lw a5, 0(a5)
78  sw a5, -20(s0)
79  lw a5, -40(s0)
80  addi a5, a5, 1
81  slli a5, a5, 2
82  lw a4, -36(s0)
83  add a4, a4, a5
84  lw a5, -40(s0)
85  slli a5, a5, 2
86  lw a3, -36(s0)
87  add a5, a3, a5
88  lw a4, 0(a4)
89  sw a4, 0(a5)
90  lw a5, -40(s0)
91  addi a5, a5, 1
92  slli a5, a5, 2
93  lw a4, -36(s0)
94  add a5, a4, a5
95  lw a4, -20(s0)
96  sw a4, 0(a5)
97  nop
98  lw ra, 44(sp)
99  lw s0, 40(sp)
100 addi sp, sp, 48
101 jr ra
102
103 sort:
104 addi sp, sp, -48
105 sw ra, 44(sp)
106 sw s0, 40(sp)
107 addi s0, sp, 48
108 sw a0, -36(s0)
109 sw a1, -40(s0)
110 sw zero, -20(s0)
111 j .L4
112 .L8:
113 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
la a0, v
li a1, 30
jal ra, sort
# Segunda chamada da funcao show
la a0, v
li a1, 30
jal ra, show
# Retorno da funcao main
li a0, 0
ret

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

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

sort:
ble a1, zero, .L4
li a6, -1
```

```
add a7, a1, a6
mv a1, a6
.L7:
mv a4, a6
mv a5, a0
bne a6, a1, .L6
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
```

```
72  li s2, -1
73  .L4:
74  blt s1, s3, .L9
75  lw ra, 44(sp)
76  lw s0, 40(sp)
77  lw s1, 36(sp)
78  lw s2, 32(sp)
79  lw s3, 28(sp)
80  addi sp, sp, 48
81  jr ra
82  .L9:
83  slli s0, s1, 2
84  addi a1, s1, -1
85  add s0, a0, s0
86  .L5:
87  bne a1, s2, .L6
88  .L8:
89  addi s1, s1, 1
90  j .L4
91  .L6:
92  lw a4, -4(s0)
93  addi s0, s0, -4
94  lw a5, 4(s0)
95  ble a4, a5, .L8
96  sw a1, 12(sp)
97  sw a0, 8(sp)
98  call swap
99  lw a1, 12(sp)
100 lw a0, 8(sp)
101 addi a1, a1, -1
102 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.

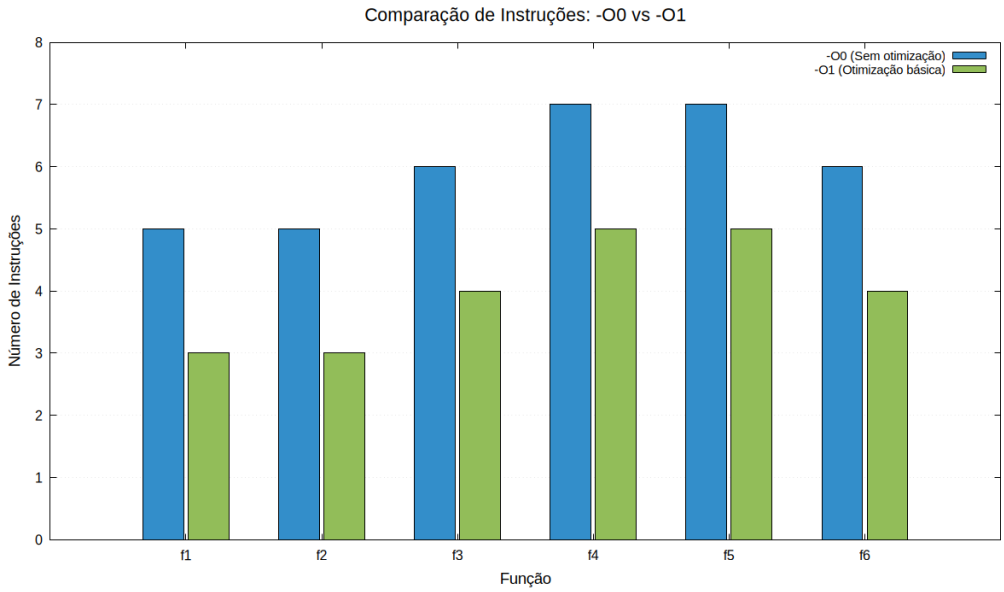

Figura 3: Comparação visual do número de instruções por função nos níveis -O0 e -O1.

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.

(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]`.

(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
...
```

(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
```

(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.

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

Que conclusões podemos tirar desta análise?

Dicas para medir o desempenho

O RISC-V possui o banco de registradores de Status e Controle (CSRs) que armazena informações úteis e pode ser lido pela instrução:

```
csrr t1, fcsr   # Control and Status Register Read
```

Registros úteis:

- `{timeh, time}` = tempo do sistema em ms
- `{instreth, instret}` = número de instruções executadas
- `{cycleh, cycle}` = número de ciclos executados

Exemplo de medição (usar somente os 32 bits menos significativos):

```
main:
    csrr s1, 3074    # instret - instr antes
    csrr s0, 3073    # time   - tempo antes
    jal PROC
    csrr t0, 3073    # time   - tempo depois
    csrr t1, 3074    # instret - instr depois
    sub s0, t0, s0   # tempo de execução (ms)
    sub s1, t1, s1   # número de instruções
```

Tabela 5: Control and Status Registers (exemplo)

Nome	Número	Exemplo
cycle	3072	0x00038946
time	3073	0x9a130c8d
instret	3074	0x00038946

Observações finais

O relatório deve ser escrito na forma de *resposta ao item*, contendo apenas os itens que valem ponto. Ao final inclua a URL clicável do vídeo da apresentação (Teams/YouTube) com a participação em câmera de todos os componentes do grupo.

Estrutura sugerida para preenchimento das respostas:

1. Para cada item que vale ponto, crie uma subseção com: (i) método; (ii) resultados (tabelas e/ou figuras); (iii) discussão / conclusão.
2. Inclua as saídas do RARS (screenshots ou tabelas de contagem de instruções), os arquivos `.s` modificados e referências às linhas alteradas.
3. Para gráficos (item 1.2.b e 3.5.2), guarde as figuras em pasta `figures/` e inclua via `\includegraphics`.

Se quiser, eu já preench o esqueleto das seções de respostas (por exemplo: `1.1(a) resultados`, `1.1(b) resultados`, etc.) com tabelas vazias, espaços para inserir capturas do RARS e comandos exatos de medição — quer que eu faça isso agora?