Laboratório 1 - Assembly RISC-V

Grupo A8:

Gabriel de Medeiros Matos - 242001491

Gabriel Fonseca Pimenta - 242043428

Giovanni Daldegan - 232002520

Ricardo de Carvalho Nabuco - 231021360

Tauã Valentim de Albuquerque Martins Frade - 231021389

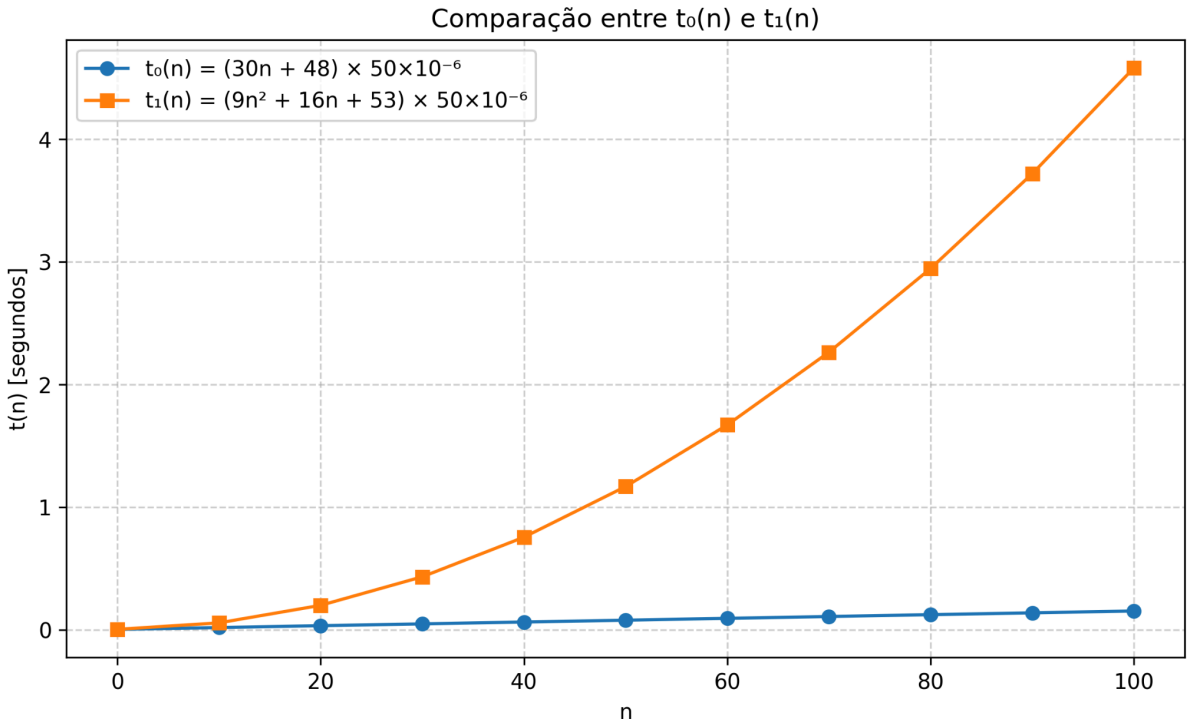
1. Simulador/Montador RARS

1.2. Ordenamento com sort.s e análise de

Usando método algébrico de interpolação polinomial com base na contagem de instruções no RARS temos:

$t_0(n) = (30n + 48) \times 50 \times 10^{-6}$

$t_1(n) = (9n^2 + 16n + 53) \times 50 \times 10^{-6}$



Com base no gráfico, é possível notar que t_0 tem crescimento linear, enquanto t_1 tem crescimento quadrático. O algoritmo de *bubble sort* implementado tem esse comportamento – sua complexidade temporal no melhor caso (o vetor já está ordenado) é O(N) e, no pior caso (o vetor está ordenado inversamente), é $O(N^2)$.

2. Compilador cruzado GCC

2.2. Compilação de `sort.c` e adaptação do código ao RARS

Usando a ferramenta Compiler Explorer para compilar o programa `sort.c`, as seguintes modificações são necessárias para que ele seja corretamente executado pelo RARS:

- Separar trechos de dados e programa em segmentos `.data` e `.text`, respectivamente;
- Mover o procedimento `main` para o início do programa, para que os demais sejam executados apenas quando chamados;
- Remover "`*`" das *labels*, pois as tornam inválidas para o RARS;
- Guardar as strings "`\t`" e "`\n`" em *labels* e adaptar as chamadas da função `printf` da linguagem C para chamadas do sistema operacional que apresentem o resultado esperado: para cada número do vetor, imprimir seu valor no console, seguido da string "`\t`". Ao final do vetor, imprimir "`\n`";
- Chamar a *syscall* 10 ao fim do programa para encerrá-lo sem erros.

2.3. Compilação de `sortc_mod.c`, otimização e análise de resultados

A compilação e montagem de `sort_mod.c` sem otimização, usando a *flag* `-O0`, foi possível realizando apenas os passos descritos no item 2.2. Porém, ao inserir as *flags* `-O3` e `-Os`, o GCC mostrou novos comportamentos não suportados na montagem.

Para tornar o carregamento de endereços mais eficiente, surgiu a "âncora de seção" `.LANCHOR0`, útil para referenciar vários endereços de dados em relação a um único endereço simbólico. Porém, a instrução `.set`, usada para definir o endereço simbólico de `.LANCHOR0`, não existe na ISA RV32IMF, sendo ignorada na montagem do programa pelo RARS. A solução encontrada foi substituir todas as ocorrências da âncora pela label do vetor "`v`" explicitamente.

Além disso, especificamente no programa otimizado com a flag `-O3`, o GCC repetiu o código do procedimento `show` sempre que utilizado, ao invés de chamá-lo pela pseudo-instrução `call`, o que acabou inserindo 3 pares de *labels* `loop1` e `fim1`. Isso se deu pela injeção de código *assembly* diretamente no código em linguagem *C*, e foi solucionado pela indexação das labels repetidas: `loop1`, `fim1`, `loop2`, `fim2`, etc.

Todas as alterações de código explicadas acima foram documentadas nos programas `sort_mod_O3.s` e `sort_mod_Os.s` , adaptados e montáveis pelo RARS, em comentários para cada linha alterada.

Comparando os resultados do item 2.3 (programas `sort_mod_O*.s` compilados pelo GCC com as respectivas *flags* de otimização) com os resultados do item 1.1 do laboratório (`sort.s` codificado diretamente em assembly), temos a seguinte tabela.

	Total	Tipo R	Tipo I	Tipo S	Tipo B	Tipo U	Tipo J	Tamanho (B)
`sort.s`	3746	1001	1558	317	462	3	405	296
`sort_mod_O0.s`	10103	1987	6078	1323	462	162	91	536
`sort_mod_O3.s`	2175	69	1265	314	462	4	61	380
`sort_mod_Os.s`	4406	535	2371	631	462	161	246	332

Uma observação é que o programa compilado `sort_mod_O0.s` possui linhas com a instrução `nop`, significando que o tamanho total e contagem de instruções do programa poderiam ser levemente reduzidos se fossem removidas essas instruções que não produzem nenhum efeito aparente na execução.

2.4. Níveis de otimização do GCC

Cada nível de otimização inclui um conjunto de flags na compilação, sendo cumulativo à medida que sobe o nível. Os níveis principais são:

- `-O0` desabilita a maioria do processos de otimização;
- `-O` ou `-O1` configuram o nível mais simples de otimização, no qual o compilador tenta reduzir o tamanho do código gerado sem aumentar tanto o tempo de compilação, evitando otimizações de maior complexidade algorítmica;
- `-O2` engloba as flags de `-O1` e adiciona mais outras, permitindo um maior uso de memória e tempo na otimização, gerando um código mais eficiente;
- `-O3` otimiza o programa ainda mais que `-O2`, com métodos ainda mais complexos, por exemplo, o reuso de dados já computados em loops;
- Já `-Os` utiliza todas as flags de `-O2` exceto algumas que frequentemente incrementam o tamanho do código gerado, focando na redução da memória ocupada pelo programa.

Ainda há outras configurações, como `-Ofast` que possui todas as flags de `-O3` e ativa flags de aceleração de cálculos, que podem gerar imprecisões em resultados a depender da

implementação do padrão IEEE 754-2008 de aritmética de ponto flutuante e especificações ISO para funções matemáticas.

Um aviso importante é que um programa otimizado pode não produzir os resultados esperados em uma eventual depuração, já que os algoritmos de otimização podem reorganizar, alterar e omitir operações quando for adequado. Isso fica nítido ao comparar os diferentes níveis de otimização na compilação de programas com algoritmos determinísticos simples ou com uso de valores fixos. Já no primeiro nível de otimização há uma grande redução de acessos à pilha e simplificação de operações matemáticas, priorizando o uso de registradores no lugar de memória RAM e a manipulação de potências de 2, por exemplo.

Na compilação de grandes códigos gerados por máquina, a recomendação geral é que se use `-O` (ou `-O1`) para equilibrar tempo e memória gastos em otimização, mas ainda melhorando significativamente o desempenho do programa obtido.

3. Transformada Discreta de Fourier

O código dos seguintes itens está presente no arquivo `main.asm`.

3.1. Procedimento `sincos(float theta)` para aproximação de seno e cosseno

O cálculo das funções seno e cosseno de um ângulo Θ foi feita pela aproximação por Série de Taylor. O programa obtém o resto da divisão de Θ por π por meio de adições ou subtrações sucessivas do ângulo com 2π, operações essas as quais não alteram o valor do seu seno e cosseno, de forma que o resultado final fique no intervalo de $-\pi$ e π. Estando dentro do intervalo, a cálculo da Série de Taylor se torna mais preciso e evita *overflow*/*underflow* na representação *float* de precisão simples Ao final, será obtido uma aproximação de seus valores de seno e cosseno.

3.2. Procedimento `DFT` para um vetor de N pontos

O cálculo da Transformada Discreta de Fourier é dado pela equação.

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

Transf. Discreta de Fourier– Parte 1

$$X(m) = \sum_{n=0}^{N-1} x(n)\left[cos\left(2\pi n\frac{m}{N}\right) - jsen\left(2\pi n\frac{m}{N}\right)\right]$$

- Separamos a exponencial complexa em duas partes: real e imaginária
- X(m): m-ésima componente da TDF. Ex) X(0), X(1),X(2), ...
- m: índice no domínio da frequência da TDF. m=0,1,2...,N-1
- x(n): sequência amostrada. x(0), x(1), x(2), ...
- n: índice no domínio do tempo das amostras (discreto). n=0,1,2,3,...,N-1

Fornecidos o endereço do vetor `x` em `a0`, o endereço do vetor `X_Real` em `a1`, o endereço do vetor `X_Img` em `a2` e o inteiro N em `a3`, o programa calcula um vetor resultante por meio da potência de *e* da fórmula, a qual é decomposta em uma soma de funções trigonométricas, como na Fórmula de Euler, para então calcular o somatório explicitado. Para os vetores fornecidos, os resultados têm, no pior dos casos, apenas 1 dígito significativo por item, seja na parte real ou na parte imaginária do resultado, mas em geral têm entre 3 e 6 (exceto nos casos mais raros em que o resultado é exatamente o esperado).

3.3. Programa `main` para calcular a Transformada Discreta de Fourier para um vetor

O programa `main.asm` possui todos os vetores fornecidos nas orientações do laboratório, basta "comentar"/"descomentar" os vetores e instruções como instruído nos comentários iniciais do arquivo. O procedimento `main` chama o procedimento `DFT` para um vetor, que por sua vez chama `sincos` para calcular cada entrada dos vetores `X_real` e `X_imag` resultantes.

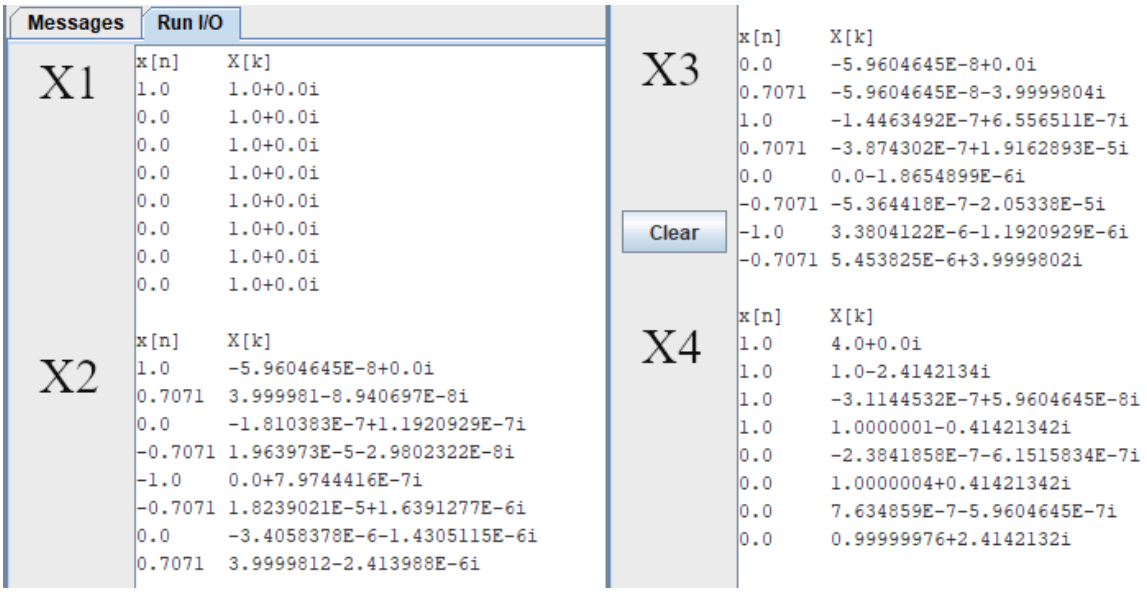
```
1  .data
2
3  N: .word 8
4  X: .float 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0
5  #X1: .float 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
6  #X2: .float 1.0, 0.7071, 0.0, -0.7071, -1.0, -0.7071, 0.0, 0.7071
7  #X3: .float 0.0, 0.7071, 1.0, 0.7071, 0.0, -0.7071, -1.0, -0.7071
8  #X4: .float 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0
9  X_real: .float 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
10 X_imag: .float 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
```

```
x[n]    X[k]
1.0     8.0+0.0i
1.0     0.0+5.9604645E-8i
1.0     -4.306546E-7-5.9604645E-7i
1.0     1.7881393E-7-1.7881393E-7i
1.0     -4.7683716E-7+1.2921903E-6i
1.0     7.1525574E-7+8.940697E-7i
1.0     -3.2922997E-6+1.4305115E-6i
1.0     -6.7949295E-6+6.556511E-7i

-- program is finished running (0) --
```

3.4. Cálculo da DFT para os vetores `X1`, `X2`, `X3` e `X4`, com N=8

Foram obtidos os seguintes resultados para os vetores:



3.5. Análise de resultados da DFT e plotagem

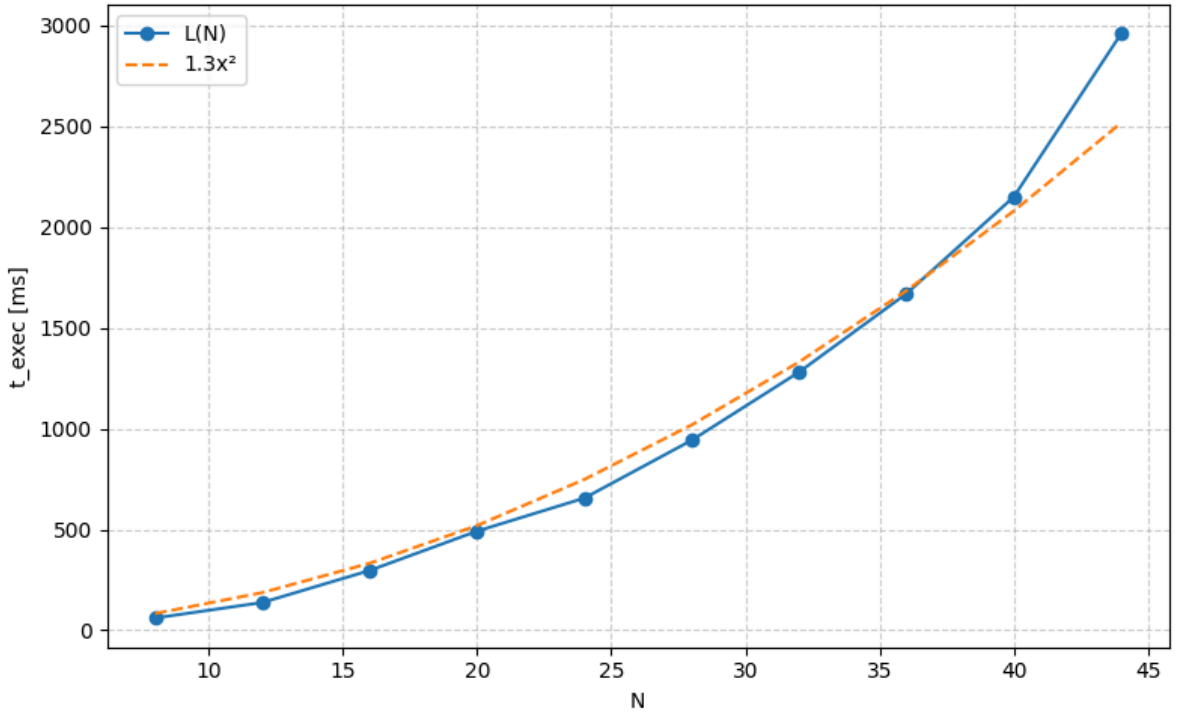
```
12  N: .word 8
13  X: .float 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0
14  X_real: .float 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
15  X_imag: .float 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
16
17  #N: .word 12
18  #X: .float 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
19  #X_real: .float 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
20  #X_imag: .float 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
21
22  #N: .word 16
23  #X: .float 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
24  #X_real: .float 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
25  #X_imag: .float 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
26
27  #N: .word 20
28  #X: .float 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
29  #X_real: .float 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
30  #X_imag: .float 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
31
```

$$t_{exec}\left[\frac{segundos}{programa}\right] = I\left[\frac{Instruções}{programa}\right] \times CPI\left[\frac{Ciclos\_clock}{Instrução}\right] \times T\left[\frac{segundos}{Ciclos\_clock}\right]$$

Foram medidos os tempos de execução (t_{exec}) de `DFT` e o número total de instruções (instr.) do procedimento para os vetores dados e calculadas as respectivas frequências do processador RISC-V uniciclo (CPI = 1) simulado pelo RARS, resultando na tabela:

N	8	12	16	20	24	28	32	36	40	44
t_{exec} (ms)	61	138	297	492	656	943	1281	1670	2150	2959
instr.	17262	39147	70525	111886	163600	226382	300437	386430	484861	596053
freq. (MHz)	282,98	283,67	237,46	227,41	249,39	240,07	234,53	231,40	225,52	201,44

Considerando os dados coletados, o tempo médio de execução foi de 799,5 ms, e a frequência média, 235,995 MHz. O gráfico N x t_{exec} a seguir compara a curva do polinômio interpolador de Lagrange $L(N)$ nos pontos da tabela (em azul) com uma curva de crescimento quadrático (em laranja):



Dessa forma, é possível constatar que o algoritmo tem complexidade temporal $O(N^2)$, assim como sugere a fórmula da transformada, na qual cada resultado de índice k, $0 \leq k \leq N-1$, exige cálculos sobre todos os n valores do vetor de entrada, $0 \leq n \leq N-1$.

4. Vídeo da apresentação do laboratório
   https://youtu.be/_OhupgnLO2U

5. Referências bibliográficas

COMPILER Explorer. *In*: **Compiler Explorer**. Gh-16498. [*S. l.*], 11 out. 2025. Disponível em: https://godbolt.org. Acesso em: 12 out. 2025.

FREE Software Foundation, Inc. *In*: **GCC online documentation: Options That Control Optimization**. [*S. l.*], 2025. Disponível em: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Acesso em: 12 out. 2025.

PETER Cordes. *In*: **What is the role of .LANCHOR0 in detecting multiple definitions error?**. [*S. l.*], 30 mar. 2019. Disponível em: https://stackoverflow.com/questions/41877351/what-is-the-role-of-lanchor0-in-detecting-multiple-definitions-error. Acesso em: 17 out. 2025.

UDURU0522. *In*: **Lab2: RISC-V RV32I[MA] emulator with ELF support**. [*S. l.*], 22 dez. 2020. Disponível em: https://hackmd.io/@Uduru0522/S11Mg_Xow. Acesso em: 19 out. 2025.

FREE Software Foundation, Inc. *In*: **GCC online documentation: Anchored Addresses**. [*S. l.*], 2025. Disponível em: https://gcc.gnu.org/onlinedocs/gccint/Anchored-Addresses.html. Acesso em: 19 out. 2025.