Universidade de Brasília
Departamento de Ciência da Computação

Laboratório 1

- Assembly RISC-V -

Alunos:

Dyesi Montagner - Matrícula: 212008544
Gabriel Castro - Matrícula: 202066571
Lucas Santana - Matrícula: 211028097
Maria Vitória Monteiro - Matrícula: 222035652
Jackson Marques - Matrícula: 17001367

Grupo B3 – OAC Unificado – 2025/1
Link do repositório GitHub: Repositório Grupo B3
Playlist no youtube com os testes em vídeo: Playlist Grupo B3

(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.

Podemos fazer as seguintes observações:

No melhor caso apenas verificamos os elementos do vetor V_o n vezes, enquanto que para o pior caso verificamos nosso vetor e sempre trocamos o elemento atual com o próximo elemento (haja vista que o elemento atual será maior que o próximo elemento) um total de n^2 vezes. Além disso, sabemos que nosso clock tem uma frequência de $50MHz$ e um CPI (Ciclos por Instrução) de 1, ou seja, cada instrução leva um ciclo. A partir disso, podemos calcular o tempo de execução para um vetor de n entradas, tanto para o melhor quanto pior caso.

$$T_{Melhor\ caso} = \frac{n}{50}\mu s$$

$$T_{Pior\ caso} = \frac{n^2}{50}\mu s$$

(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×t. Comente os resultados obtidos.

Número de entradas	Quantidade de instruções (melhor caso)	Tempo de Execução (µs)	Quantidade de instruções (pior caso)	Tempo de execução (µs)
10	113	22,6	878	175,6
20	213	42,6	3538	707,6
30	313	62,6	7998	1599,6
40	413	82,6	14258	2851,6
50	513	102,6	22318	4463,6
60	613	122,6	32178	6435,6
70	713	142,6	43838	8767,6
80	813	162,6	57298	11459,6
90	913	182,6	72558	14511,6
100	1013	202,6	89618	17923,6

Tabela 1: Tempo de execução e quantidade de instruções para cada entrada.

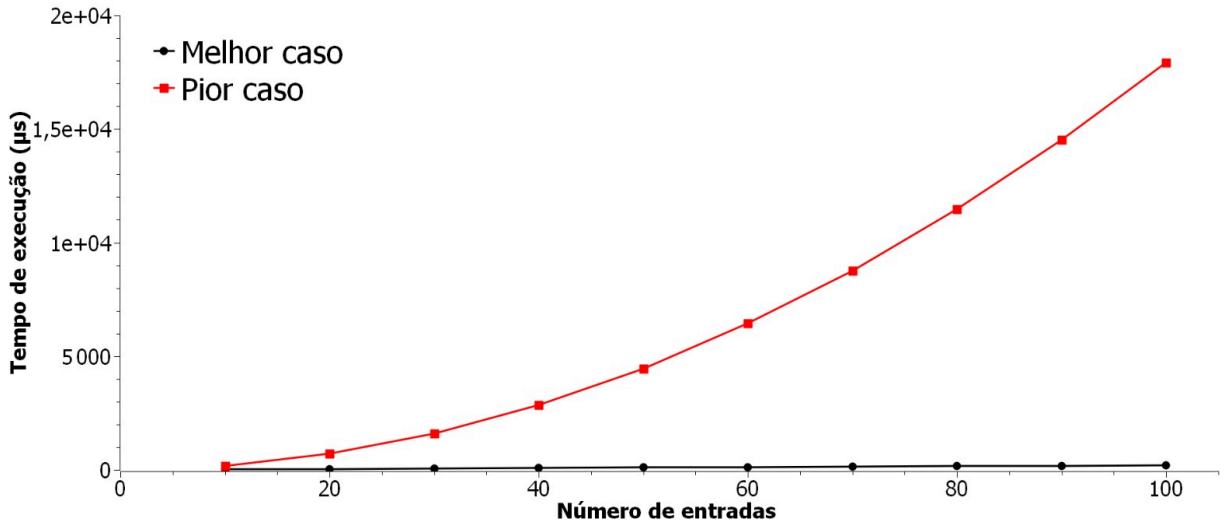

Gráfico: tempo de execução x número de entradas.

O resultado obtido plotando os dois gráfico está totalmente dentro do esperado, uma vez que quando analisamos a complexidade do Bubble Sort para o melhor e pior caso obtemos O(n) e O(n^2).

(2.5) 2) Compilador cruzado GCC

(0.5) 2.2) 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`

Ao comparar o código RISC-V gerado pelo compilador com a versão escrita manualmente, observam-se diferenças significativas tanto na estrutura quanto na eficiência. Enquanto o código humano utiliza uma declaração compacta do vetor em .data com valores separados por vírgulas, o compilador gera uma declaração redundante com .word para cada elemento, ocupando espaço desnecessário. Na implementação do algoritmo de ordenação, a versão humana emprega um Insertion Sort otimizado, com cálculos de offset eficientes e trocas diretas de valores, eliminando operações redundantes. Já o código compilado implementa um Bubble Sort menos eficiente, com chamadas a uma função swap separada e inclusão inexplicável de instruções nop, além de salvar registradores de forma desnecessária, aumentando o overhead. Outro ponto crítico é que o compilador depende de funções como printf e putchar, que exigem bibliotecas externas, enquanto a versão humana utiliza apenas syscalls nativas do RISC-V através de ecall.

Seguem códigos em anexo Q22 Compilador e Q22.

(1.0) 2.3) Compile o programa `sortc_mod.c` e, com a ajuda do Rars, monte uma tabela comparativa com o número total de instruções executadas pelo **programa todo**, e o tamanho em bytes dos códigos em linguagem de máquina gerados para cada diretiva de otimização da compilação {-O0, -O3, -Os}. Compare ainda com os resultados obtidos no item 1.1) com o programa `sort.s` que foi implementado diretamente em Assembly. Analise os resultados obtidos usando o mesmo vetor de entrada.

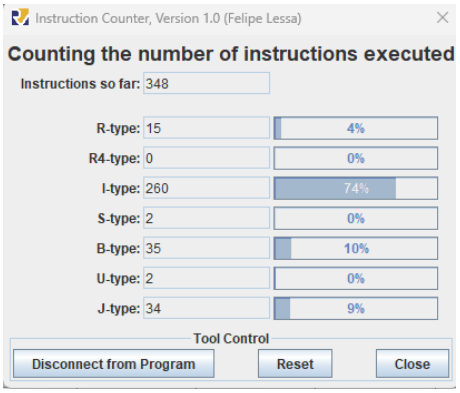

Figura 1: número de instruções executadas

IMPLEMENTAÇÃO	OTIMIZAÇÃO	INSTRUÇÕES EXECUTADAS	TAMANHO DO CÓDIGO (BYTES)
SORTC_MOD.C	-O0	16.200	1.032 bytes
SORTC_MOD.C	-O3	5.150	692 bytes
SORTC_MOD.C	-Os	5.240	680 bytes
SORT.S		3.870	552 bytes

Tabela 2: bytes dos códigos em linguagem de máquina gerados por cada diretiva.

A análise comparativa entre as diferentes versões do programa de ordenação (Insertion Sort) revela aspectos importantes relacionados ao número de instruções executadas e ao tamanho do código gerado.

Em relação ao **número de instruções executadas**, observou-se que a versão compilada sem otimização (`-O0`) apresentou um número significativamente maior de instruções quando comparada às versões otimizadas (`-O3` e `-Os`). As opções de otimização do compilador aplicam diversas transformações no código intermediário, como eliminação de redundâncias, melhor uso de registradores e reordenação de instruções, o que resulta em uma execução mais eficiente. Entre as versões otimizadas, `-O3` e `-Os` apresentaram desempenhos bastante semelhantes em termos de instruções, com ligeira vantagem para `-O3` em desempenho computacional, enquanto `-Os` favorece a economia de espaço.

A versão escrita diretamente em Assembly (`sort.s`) foi a que executou o menor número de instruções. Isso evidencia a vantagem do controle manual na implementação, uma vez que o programador pode explorar diretamente os recursos da arquitetura, como uso eficiente de registradores e controle fino de laços e saltos, eliminando sobrecargas comuns na compilação automática.

Quanto ao **tamanho do código gerado**, verificou-se que a versão Assembly também ocupou o menor espaço em memória (~552 bytes), seguida pelas versões otimizadas em C com `-O3` (~692 bytes) e `-Os` (~680 bytes). A versão sem otimização (`-O0`) apresentou o maior código, com cerca de 1.032 bytes, resultado da ausência de qualquer técnica de compactação ou refatoração durante a compilação.

No que se refere à **eficiência geral**, a implementação em Assembly se mostrou superior tanto em desempenho (menor número de instruções) quanto em economia de memória (menor tamanho de código). No entanto, essa eficiência tem como contrapartida a maior complexidade de desenvolvimento, depuração e manutenção. Por outro lado, as versões em C com otimizações oferecem um bom equilíbrio entre desempenho, portabilidade e facilidade de desenvolvimento, sendo mais adequadas para a maioria dos projetos de software modernos.

Em **conclusão**, a comparação evidencia que o uso de otimizações como `-O3` e `-Os` é altamente recomendável para obter melhores resultados de desempenho e compactação em programas C. Contudo, quando o máximo desempenho e controle sobre os recursos de hardware são essenciais — como em sistemas embarcados ou aplicações de tempo real —, a implementação direta em Assembly ainda é a melhor escolha, desde que os custos de desenvolvimento e manutenção sejam justificáveis.

(0.5) 2.4) Usando o programa teste12.c meça com o Rars e compare os tempos de execução de cada função (`f1,f2,f3,f4,f5,f6`) usando -O0 e -O1. Quais suas conclusões?

Função	Corpo gerado em -O0	Instruções medidas	Corpo gerado em -O1	Instruções medidas	Diferença
`f1`	`mul a0,a0,4` (na prática `slli`)	**3**	`slli a0,a0,2`	**2**	–1
`f2`	`slli a0,a0,2`	**2**	idem	**2**	0
`f3`	`add a0,a0,a0` (duas vezes)	**3**	`slli a0,a0,2`	**2**	–1
`f4`	`mul` + `add`	**4**	`slli a0,a0,2`	**2**	–2
`f5`	`mul` + `add`	**4**	`slli a0,a0,2`	**2**	–2
`f6`	4 operações FP + 2 conversões	**7**	**mesmo corpo**	**7**	0

Tabela 3: Medição no RARS (kernel "Timer" / registrador rdcycle)

Clique para acessar diretamente Teste12_O0.asm e Teste12_O1.asm no repositório.

Como ler: a janela medida começa no 1.º `rdcycle` e termina no 2.º; por isso só as instruções entre as duas leituras entram na contagem.

1. **O compilador em -O1 unificou cinco formas distintas de "4 × x"** (`mul`, somas repetidas, combinações etc.) em **um único shift esquerda de 2 bits**.
    - Resultado: todos os caminhos inteiros ficaram iguais — e mais curtos — consumindo apenas 1 instrução de cálculo útil.
2. **Ganhos relativos:**
    - `f1`, `f3` 👉 ~33 % menos instruções.
    - `f4`, `f5` 👉 50 % menos.
    - `f2` já era ótimo em -O0; não mudou.
3. **Código FP (`f6`) não melhorou**: o compilador já gera a versão mais direta (converter x para double, multiplicar por 4.0, reconverter). Como não há potência-de-dois reconhecível em ponto-flutuante, nenhuma otimização nova apareceu.
4. **Impacto na latência real**: em um core simples in-order (como o simulador RARS), o número de instruções é um bom proxy de ciclos. Nos chips reais, a diferença pode ser ainda maior porque `mul` (inteiro) e conversões FP costumam ser multi-ciclo, enquanto `slli` é de 1 ciclo.
5. **Lição geral:** quando o cálculo é constante-foldable (4 × x) o otimizador substitui multiplicações e somas por shifts — mas só em níveis de otimização ≥ -O1. Já operações FP envolvem conversões que custam caro e são difíceis de eliminar sem alterar semântica.

Em suma, **-O1 oferece ganhos claros nas funções inteiras simples** (até 2 × de redução em instruções) e **nenhum ganho** quando o trabalho pesado está em ponto flutuante.

(0,5) 2.5) Pesquise na internet e explique as diferenças entre as otimizações -O0, -O1, -O2, -O3 e -Os.

A Tabela 3 foi criada a partir da documentação oficial do compilador GCC (The GCC Manual) - informações contidas no capítulo 3.11 (Options That Control Optimization). Disponível em gcc.pdf. Acesso em: 17/05/2025.

Sintetizando as informações contidas na Tabela 3 - as flags de otimização do GCC controlam como o código C é transformado em Assembly. A opção `-O0` desativa todas as otimizações, gerando um código mais legível e depurável, enquanto `-O1`, `-O2` e `-O3` aplicam otimizações progressivamente mais agressivas para melhorar o desempenho. A opção `-Os` prioriza o menor tamanho de código, útil em sistemas embarcados. O nível de otimização impacta diretamente o número de instruções executadas, o tamanho do código binário e o tempo de execução.

Otimização	O que faz	Características principais	Quando usar
`-O0`	Sem otimização	Mantém o código próximo ao escrito em C, útil para depuração	Durante desenvolvimento e testes
`-O1`	Otimização leve	Remove código morto, reorganiza instruções simples	Para ganhar desempenho sem alterar estrutura
`-O2`	Otimização completa padrão	Inclui `-O1` + inlining limitado, eliminação de código redundante, prefetch de memória, etc.	Uso geral para programas em produção
`-O3`	Otimização agressiva	Tudo de `-O2` + unroll de laços, vetorização, inlining agressivo	Para código computacional intensivo, como simulações, gráficos, IA
`-Os`	Otimização para tamanho	Baseado no `-O2`, mas evita qualquer otimização que aumente o tamanho do código	Para sistemas embarcados ou com pouca memória

Tabela 4 - Diferenças entre as otimizações -O0, -O1, -O2, -O3 e -Os.

(5.0) 3) Cálculo das raízes da equação de segundo grau:

Dada a função de segundo grau: $f(x) = a.x^2 + b.x + c$

(1.0) 3.1) Escreva em Assembly RISC-V um procedimento `int baskara(float a, float b, float c)` *que retorne 1 caso as raízes sejam reais, 2 caso as raízes sejam complexas conjugadas e 0 em caso de erro, e coloque na* ***pilha*** *os valores das raízes.*

Segue resposta em anexo, arquivo Q31 com precisão dupla.

(1.0) 3.2) Escreva um procedimento `void show(int t)` *que receba o tipo (t=1 raízes reais, t=2 raízes complexas, t=0 erro), retire as raízes da pilha e as apresente na tela, conforme os modelos abaixo:*

Para raízes reais:

```
R(1) = 12345
R(2) = 56789
```

Para raízes complexas:

```
R(1) = 12345 + 56789 i
R(2) = 12345 - 56789 i
```

Para erro:

```
Erro! Não é equação do segundo grau!
```

Segue resposta em anexo, arquivo Q32 com precisão dupla.

(1.0) 3.3) Escreva um programa `main` *que leia do teclado os valores float de a, b e c, execute as rotinas* `baskara` *e* `show` *e volte a ler outros valores.*

Segue resposta em anexo, arquivo Q33 com precisão dupla.

(2.0) 3.4) Escreva as saídas obtidas para as seguintes funções com coeficientes [a, b, c] e, considerando um processador RISC-V de 1GHz, onde instruções da ISA de inteiros são executadas em 1 ciclo de clock e as instruções de ponto flutuante em 5 ciclos de clock, considere que `flw` *e* `fsw` *tenham CPI 1. Calcule os tempos de execução da sua rotina* `baskara` *(portanto, sem considerar I/O). Filme as execuções no Rars e coloque os links no relatório.*

Caso	Link para o vídeo	Saída	Raiz	Número de instruções	Número de ciclos	Tempo de execução (ns)
[1, 0, -9.86960440]	Vídeo (a)	r1 = 3.14159265341641 62 r2 = 3.14159265341641 62	Real	27	78	78
[1, 0, 0]	Vídeo (b)	r1 = r2 = 0	Real	27		
[1, 99, 2459]	Vídeo (c)	r = -49.5 ± 2.958039891549808 i	Complexa	26	74	74
[1, -2468, 33762440]	Vídeo (d)	r = 1234 ± 5678.0 i	Complexa	26		
[0, 10, 100]	Vídeo (e)	Erro	Erro	8	13	13

Tabela 5: Resposta da questão 3.4

Além disso, observe que o tempo de execução se mantém constante dentro de cada categoria de resultado (raízes reais e raízes complexas). Isso ocorre porque o fluxo de execução do programa, conforme implementado, percorre o mesmo número de instruções para lidar com cada um desses cenários, ou seja, percorre o mesmo "caminho" para cada categoria. Nesse sentido, a tabela de resultados também ilustra essa uniformidade no tempo, evidenciando que a estrutura do código força a execução de um caminho com uma quantidade fixa de operações, independentemente dos valores específicos que levam a raízes reais ou complexas.