Compiladores: Conceitos e Técnicas

Arthur Henrique F. Queiros,
Carlos Magnus Carlson Filho

e-mail:
arthur.queiros@fatec.sp.gov.br
carlos.carlson@fatec.sp.gov.br

Resumo: Este trabalho apresenta o desenvolvimento de um assembler para um processador de 8 bits, capaz de traduzir instruções de linguagem de montagem para código de máquina binário. O assembler foi implementado em Python e possui funcionalidades para interpretar diferentes formatos de instruções, mapeando operações e registradores para seus códigos binários correspondentes. A ferramenta também inclui um modo de depuração e foi projetada para processar arquivos de entrada e gerar uma saída binária pronta para execução. O projeto atende aos requisitos do processador de 8 bits, permitindo que instruções sejam validadas e codificadas eficientemente. O assembler foi testado com uma variedade de instruções para garantir precisão e usabilidade, sendo uma contribuição relevante para o desenvolvimento e execução de programas em arquiteturas simples de computação.

Palavras-chave: assembler, linguagem de montagem, código de máquina, Python, depuração, arquitetura de computação.

Abstract: *This paper presents the development of an assembler for an 8-bit processor, capable of translating assembly language instructions into binary machine code. The assembler was implemented in Python and includes functionalities to interpret different instruction formats, mapping operations and registers to their corresponding binary codes. The tool also features a debug mode and was designed to process input files and generate binary output ready for execution. The project meets the requirements of the 8-bit processor, enabling efficient validation and encoding of instructions. The assembler was tested with a variety of instructions to ensure accuracy and usability, making it a relevant contribution to the development and execution of programs in simple computing architectures.*

Keywords: *assembler, 8-bit processor, assembly language, machine code, Python, debugging, computing architecture.*

1 Introdução

Este projeto trata do desenvolvimento de um assembler simples para um processador de 8 bits, projetado com foco educacional e destinado a traduzir instruções de uma linguagem de montagem para código binário que possa ser executado pela unidade de processamento. A escolha do conjunto de instruções e dos registradores visa manter a simplicidade, facilitando o entendimento dos conceitos fundamentais de montagem e codificação binária.

O assembler foi desenvolvido em Python e tem a capacidade de traduzir instruções básicas, como adição, subtração e operações lógicas (AND, NOT, deslocamento de bits, etc.), para representações binárias. Ele inclui funcionalidades para ler e interpretar arquivos de texto que contêm instruções em linguagem de montagem, convertendo-as em código de máquina em formato binário.

Esse projeto faz parte de um trabalho acadêmico que explora a arquitetura de um processador simples, com o objetivo de aplicar conceitos de linguagens formais e de construção de compiladores. Além de seu valor educativo, o assembler serve como base para experimentos com instruções de baixo nível, permitindo uma compreensão prática de como os computadores processam operações básicas em nível binário.

2 Justificativa

O desenvolvimento deste assembler para um processador de 8 bits é motivado pela necessidade de consolidar conhecimentos teóricos e práticos em diversas áreas da computação, como arquitetura de computadores, linguagens de programação e compiladores. Em cursos de formação em Tecnologia da Informação, conceitos fundamentais como representação de dados em binário, funcionamento de registradores e execução de instruções em baixo nível são frequentemente abordados de forma teórica. No entanto, implementar um assembler proporciona uma compreensão prática e aprofundada desses conceitos.

Este projeto permite explorar o processo de tradução de código de montagem para código de máquina, facilitando o entendimento dos passos que um compilador ou interpretador executa para converter uma linguagem de alto nível em instruções que o hardware entende. Ao manipular um conjunto reduzido de instruções e registradores, os alunos podem observar de forma clara como operações aritméticas e lógicas básicas são processadas pelo hardware.

Além disso, o assembler oferece uma base para desenvolver habilidades importantes, como análise e criação de instruções de máquina, utilização de padrões de formatação binária e depuração de código em baixo nível. Essas habilidades são essenciais para o desenvolvimento de sistemas mais complexos, como compiladores e interpretadores, e para compreender a relação entre software e hardware.

Este projeto não apenas aprofunda o entendimento dos conceitos mencionados, mas também ajuda a desenvolver uma visão mais ampla da operação interna dos computadores, contribuindo para a formação de profissionais mais completos na área de sistemas e desenvolvimento de software.

3 Objetivo(s)

Desenvolver um assembler para um processador de 8 bits que converta instruções em linguagem de montagem para código de máquina binário, com o intuito de aplicar e consolidar conhecimentos em arquitetura de computadores, montagem e compilação, contribuindo para o entendimento prático desses conceitos.

4 Fundamentação Teórica

A criação de um assembler para um processador de 8 bits exige o entendimento de conceitos teóricos fundamentais em arquitetura de computadores e linguagens formais, especialmente no que se refere à construção de compiladores e interpretadores. A seguir, são apresentados alguns dos principais conceitos teóricos que sustentam o desenvolvimento deste projeto:

1. **Arquitetura de Computadores e Processadores de 8 Bits**
   A arquitetura de computadores descreve a organização e a estrutura de sistemas de processamento de dados. Um processador de 8 bits possui uma unidade central de processamento (CPU) capaz de processar dados e instruções com comprimento de até 8 bits. Isso significa que cada instrução ou dado ocupa apenas um byte de memória. Este tipo de arquitetura é comum em sistemas embarcados e aplicações educacionais devido à sua simplicidade, que facilita o estudo de operações básicas e a construção de unidades de controle e processamento.

Em uma CPU, os dados são manipulados por meio de registradores, que são pequenas áreas de armazenamento de alta velocidade. Este assembler trabalha com um conjunto limitado de registradores (como r0, r1, r2, r3), cada um representado por um código binário de 2 bits, o que permite operar em conjunto com instruções de 4 bits, adequando-se ao formato de 8 bits do processador.

2. **Linguagem de Montagem e Instruções de Máquina**
   A linguagem de montagem (assembly language) é um tipo de linguagem de baixo nível que se aproxima das instruções executáveis pela CPU, utilizando um conjunto reduzido de instruções que correspondem diretamente às operações de máquina. Cada instrução de montagem é traduzida para um código de máquina, que é uma representação binária que a CPU pode entender e executar. No contexto deste projeto, o assembler traduz instruções como `add`, `sub`, `mov` e `nand` para códigos binários específicos, com base em um conjunto de instruções previamente mapeado.

A linguagem de montagem facilita a manipulação direta dos recursos do hardware, como registradores e operações aritméticas e lógicas em nível de bits. Essa abordagem permite ao programador controlar os detalhes da execução de operações, o que é essencial para entender a interação entre o software e o hardware.

3. **Compiladores e Assemblers**

   O assembler desenvolvido neste projeto pode ser considerado um compilador de uma única etapa, que traduz código de montagem diretamente para código de máquina. Compiladores, em geral, são programas que transformam uma linguagem de alto nível (como C ou Python) em uma linguagem de máquina, mas envolvem múltiplas etapas, como análise léxica, análise sintática e geração de código. No caso de um assembler, essas etapas são simplificadas, pois a linguagem de montagem é projetada para ter uma correspondência direta com o código de máquina.

   A construção de um assembler, embora simplificada, se baseia nos mesmos fundamentos apresentados em obras como *Compiladores: Princípios, Técnicas e Ferramentas* (Aho et al.), que explora detalhadamente cada etapa do processo de compilação, incluindo análise léxica, geração de código e otimização. Esses conceitos fundamentam a tokenização das instruções, o mapeamento para binário e a formatação para a arquitetura do processador, permitindo um aprendizado sólido sobre a transformação de linguagens para níveis mais baixos e próximos ao hardware.

4. **Representação Binária e Codificação de Instruções**

   Em um processador, todas as operações são realizadas em formato binário. Cada instrução de montagem é convertida em uma sequência binária que representa uma operação específica e os registradores envolvidos. Por exemplo, a instrução add r1, r2 é traduzida para um opcode binário que indica uma operação de adição, seguido por códigos binários que representam os registradores r1 e r2. A representação binária é essencial para o funcionamento do processador, pois é o único formato compreensível em nível de hardware.

   Esse processo de codificação binária envolve converter valores numéricos em bits e assegurar que o tamanho da instrução não exceda o limite de bits do processador (neste caso, 8 bits). A precisão na codificação é crucial para garantir que as operações sejam interpretadas corretamente pela CPU.

5. **Paradigmas de Depuração e Teste em Baixo Nível**

   Depurar programas em linguagens de baixo nível, como a linguagem de montagem, exige uma abordagem sistemática. No contexto deste assembler, foi implementado um modo de depuração que permite visualizar cada etapa da tradução em binário. Esse recurso é útil para identificar problemas na tradução das instruções e na execução do código de máquina gerado. A depuração em baixo nível ajuda a entender como cada instrução afeta o estado dos registradores e a memória, fornecendo uma visão clara de como as operações de software impactam o hardware.

5 Trabalhos Similares

A análise de trabalhos relacionados é fundamental para contextualizar o projeto desenvolvido e evidenciar sua relevância no campo da computação. Entre os estudos que se destacam por abordarem temas semelhantes ao presente trabalho, encontram-se pesquisas

focadas em otimizações e design de compiladores para diferentes contextos, incluindo alto desempenho e transformações específicas.

1. *Compiler Transformations for High-Performance Computing*

No trabalho de **David F. Bacon, Susan L. Graham e Oliver J. Sharp**, o foco principal é a aplicação de transformações em compiladores voltadas para computação de alto desempenho. Essas transformações exploram técnicas como paralelismo, otimizações específicas de hardware e estratégias de escalabilidade, buscando extrair o máximo de desempenho das arquiteturas modernas. Embora o objetivo do presente trabalho seja mais didático do que performático, a compreensão das limitações e potencial do assembler criado pode ser guiada por princípios abordados neste estudo, como a análise do impacto de otimizações no ciclo de execução.

2. *Compiler Design Theory*

A obra **Compiler Design Theory**, publicada pela Addison-Wesley Longman, apresenta uma visão abrangente das teorias fundamentais para o design de compiladores. Com ênfase em estruturas formais, como gramáticas e autômatos, o livro discute o papel de cada componente de um compilador, desde o analisador léxico até o gerador de código. Este trabalho foi uma inspiração significativa na estruturação do assembler, sobretudo na definição clara de regras para validação e tradução das instruções, além de influenciar o tratamento de erros e a modelagem de registradores e valores imediatos.

3. *Compilers and Staging Transformations*

O estudo de **Ulrik J. Stirring e William D. Scherlis**, desenvolvido na Carnegie Mellon University, explora transformações de estágio em compiladores, com foco na adaptação a diferentes contextos de execução. Embora a abordagem seja mais avançada e voltada para linguagens de programação dinâmicas, as ideias sobre flexibilidade e modularidade apresentadas neste trabalho dialogam com os desafios de expandir o conjunto de instruções e registradores do assembler aqui desenvolvido. As reflexões desse estudo inspiram potenciais melhorias futuras, como a introdução de mecanismos para distinguir tipos de dados e otimizar a codificação binária.

Conexão com o Projeto

Esses trabalhos similares evidenciam a amplitude e a profundidade do campo de pesquisa em compiladores. Enquanto estudos como os mencionados tratam de transformações avançadas e sistemas de alta complexidade, este trabalho posiciona-se como um ponto de partida mais acessível, com foco didático, mas conectado aos fundamentos que sustentam pesquisas mais avançadas. Assim, a análise dessas obras reforça a relevância do projeto tanto como uma introdução prática ao tema quanto como base para aprofundamentos futuros.

6 Metodologia

O desenvolvimento deste assembler seguiu uma abordagem estruturada, que envolve várias etapas de planejamento, implementação e validação. A metodologia adotada busca garantir que o assembler funcione corretamente, traduza as instruções com precisão e ofereça uma base para testes e ajustes. Abaixo estão as etapas principais da metodologia utilizada:

1. **Definição dos Requisitos e Mapeamento de Instruções**
   Inicialmente, foram definidos os requisitos funcionais do assembler, incluindo o conjunto de instruções suportado (como `add`, `sub`, `mov`, `nand`, `shl`, `shr`, e `not`), o mapeamento dos registradores e as restrições de tamanho das instruções para o processador de 8 bits. Esta etapa envolveu a análise das operações fundamentais que o assembler precisaria traduzir e a criação de uma tabela de mapeamento de cada instrução para seu código binário correspondente.

Para representar cada instrução e registrador de maneira eficiente, foi estabelecido um formato padrão de 8 bits, dividindo o opcode e os operandos em seções específicas. Esse mapeamento fornece uma estrutura clara para a tradução e facilita o desenvolvimento do assembler.

2. **Desenvolvimento do Parser para Análise de Instruções**
   A próxima etapa envolveu o desenvolvimento de um parser, responsável por interpretar cada linha do código de montagem e identificar a estrutura de cada instrução. O parser lê as instruções, remove vírgulas e espaços desnecessários, e separa as instruções válidas de comentários e linhas em branco.

Esse processo de tokenização permite decompor cada instrução em seus componentes básicos (operação, registradores, e valores imediatos). O parser foi projetado para identificar instruções válidas e sinalizar erros em caso de instruções malformadas, garantindo que apenas instruções corretas sejam processadas.

3. **Codificação das Instruções para Formato Binário**
   Com o parser em funcionamento, a etapa seguinte foi implementar a função de codificação, que converte cada instrução para seu formato binário. Esse processo envolve obter o opcode e os códigos binários dos registradores e valores imediatos, quando aplicáveis.

Foram criadas funções específicas para formatar e validar valores imediatos (conversão para 4 bits) e instruções baseadas em registradores, assegurando que a instrução completa resultante tivesse 8 bits. Para instruções que não envolvem registradores (como `not`), foram introduzidos valores binários fictícios que ajudam a preencher os bits necessários.

4. **Implementação do Modo de Depuração**
   Para auxiliar no desenvolvimento e garantir a precisão da tradução, foi implementado um modo de depuração opcional, que exibe o resultado de cada instrução em um formato mais legível. Esse modo permite visualizar as instruções separadas por

espaços, facilitando a análise e a verificação do código binário gerado antes da remoção de espaços para a saída final.

O modo de depuração é especialmente útil para entender como cada instrução é interpretada e verificar se o assembler está realizando a tradução corretamente. Essa funcionalidade também facilita a identificação de problemas nas fases de teste.

5. **Testes de Validação**
   Uma série de testes foram conduzidos para validar o funcionamento do assembler. As instruções foram testadas em diferentes cenários, como operações com registradores, instruções com valores imediatos e operações unárias (como `not`). A metodologia de testes incluiu a criação de arquivos de entrada com diferentes combinações de instruções e a comparação dos resultados gerados com os resultados esperados.

Além disso, o assembler foi testado com instruções inválidas para assegurar que ele identificasse erros corretamente e exibisse mensagens de erro informativas. Essa etapa garantiu a robustez do assembler e ajudou a identificar quaisquer ajustes necessários antes da entrega final.

6. **Documentação e Preparação da Apresentação**
   Por fim, toda a implementação foi documentada, descrevendo a estrutura e o funcionamento do assembler, assim como o processo de tradução de instruções para binário. A documentação inclui exemplos de uso e explicações detalhadas dos componentes, facilitando a compreensão do projeto por outros usuários.

Foi também preparada uma apresentação didática, que destaca o propósito do assembler, o processo de desenvolvimento e as decisões de design envolvidas. Essa apresentação visa explicar cada etapa do projeto e demonstrar o funcionamento do assembler de forma prática.

7 Desenvolvimento

O desenvolvimento deste assembler envolveu uma série de passos técnicos que foram fundamentais para criar uma ferramenta de tradução funcional e eficiente para o processador de 8 bits. O processo de desenvolvimento foi dividido em várias etapas, cada uma com objetivos específicos e implementações cuidadosas para garantir que o assembler fosse capaz de interpretar, validar e converter instruções de montagem em código de máquina binário. Abaixo estão descritas as principais fases do desenvolvimento:

1. **Configuração Inicial e Estrutura do Projeto**
   O projeto foi estruturado em Python, uma escolha feita pela facilidade de manipulação de strings e pela simplicidade na leitura e escrita de arquivos. A classe principal, `Assembler`, foi projetada para encapsular todas as funcionalidades, com métodos

dedicados para mapeamento de instruções, validação de dados, e conversão para binário.

Dentro da classe, foram definidos dois dicionários principais: instructions_map para mapear as instruções para seus respectivos opcodes e registers_map para mapear os registradores para seus códigos binários. A estrutura inicial incluiu variáveis e métodos para lidar com o modo de depuração, o que permitiu o teste e ajuste contínuo durante o desenvolvimento.

2. **Implementação da Lógica de Codificação de Instruções**
   Com a estrutura inicial pronta, foi implementada a lógica para codificar as instruções. Cada instrução foi mapeada para um formato binário específico, conforme as regras do processador de 8 bits.

Foram criados métodos para interpretar instruções de diferentes tipos:

- **Instruções com valores imediatos:** Onde o valor numérico é convertido em binário (4 bits) e anexado ao opcode e código do registrador.
- **Instruções com registradores:** Onde ambos os registradores são convertidos para seus códigos binários e combinados ao opcode.
- **Instruções unárias:** Como a operação not, que envolve apenas um registrador e requer um valor de preenchimento binário fictício (00) para completar o formato de 8 bits.

Essa lógica foi testada extensivamente para garantir que cada instrução fosse corretamente interpretada e que as saídas correspondessem ao formato binário esperado.

3. **Leitura e Processamento de Arquivos**
   Para facilitar o uso do assembler, foi desenvolvida uma função que lê instruções diretamente de um arquivo de entrada. A função compile_from_file foi implementada para abrir o arquivo, ignorar linhas vazias e comentários (linhas iniciadas com #), e processar cada instrução válida.

Após o processamento, o assembler escreve a saída em um novo arquivo de texto. Cada instrução convertida é armazenada em uma lista e gravada no arquivo de saída linha por linha, facilitando a visualização do código de máquina gerado. Essa funcionalidade permite que o assembler seja usado de maneira prática, processando arquivos de entrada e gerando saídas binárias rapidamente.

4. **Gestão de Erros e Validação de Instruções**
   Para assegurar que o assembler identificasse e tratasse instruções inválidas, foram implementadas rotinas de validação em cada etapa de interpretação. Essas rotinas verificam se:

   - A operação está definida no mapa de instruções.
   - Os registradores utilizados estão definidos no mapa de registradores.
   - Os valores imediatos estão no formato correto e dentro do intervalo de 4 bits.

Em caso de erro, o assembler retorna uma mensagem informativa, permitindo ao usuário identificar a causa do problema. Esta gestão de erros foi crucial para o desenvolvimento do assembler, pois garante que ele lide corretamente com entradas inesperadas e instrua o usuário sobre como corrigir instruções malformadas.

5. **Modo de Depuração**
   Durante o desenvolvimento, o modo de depuração foi utilizado para acompanhar o processo de codificação das instruções, exibindo cada parte da instrução formatada separadamente (opcodes e registradores). Esta função auxilia o usuário a entender como cada instrução é traduzida e a depurar possíveis falhas na tradução.

A depuração tornou-se um recurso opcional, acessível ao usuário durante o uso do assembler, permitindo que ele visualize a saída formatada e identifique com mais facilidade quaisquer problemas que possam surgir. Este recurso é particularmente útil para o desenvolvedor e usuários que desejam analisar cada etapa da tradução.

6. **Testes e Ajustes Finais**
   Por fim, o assembler foi submetido a uma série de testes, tanto com instruções válidas quanto inválidas, para garantir que todas as funcionalidades estivessem operando conforme esperado. Foram testados arquivos de entrada com combinações variadas de instruções, valores imediatos e registradores.

Esse processo de teste permitiu que ajustes finais fossem feitos, como a melhoria das mensagens de erro, a otimização do modo de depuração, e a verificação dos formatos binários. Após os testes, o assembler foi documentado, e um manual de uso foi incluído para guiar o usuário sobre as funcionalidades e limitações da ferramenta.

8 Resultados e Discussões

O desenvolvimento e a implementação do assembler para o processador de 8 bits resultaram em uma ferramenta funcional capaz de traduzir instruções de montagem para código binário. Abaixo estão os principais resultados e reflexões sobre o projeto:

Resultados Obtidos

1. **Funcionamento da Tradução**
   O assembler conseguiu interpretar e traduzir instruções de diferentes formatos (aritméticas, lógicas, movimentação de dados e unárias) para código binário, conforme o esquema de codificação especificado pelo processador.

Exemplo de tradução:
Entrada: add r1, 4
Saída: 0000 01 0100

2. **Gestão de Erros**
   Entradas inválidas foram identificadas corretamente, exibindo mensagens claras para o usuário, demonstrando a robustez do sistema.

3. **Compatibilidade com Arquivos**
   O assembler processou arquivos .txt, ignorou comentários e linhas vazias, e gerou um arquivo de saída estruturado.

4. **Modo de Depuração**
   O modo de depuração auxiliou na visualização das etapas da tradução, destacando opcodes, registradores e valores imediatos separadamente.

5. **Ambiguidade entre Números Inteiros e Registradores**
   Uma limitação identificada foi a ambiguidade entre valores inteiros e registradores. Atualmente, ambos compartilham o mesmo espaço de codificação (4 bits), o que impede a expansão do número de registradores sem causar conflitos. Como melhoria, sugere-se a inclusão de um bit indicador de tipo (registrador ou valor imediato) ou a redefinição dos intervalos de codificação, permitindo maior flexibilidade e evitando ambiguidade.

Discussões

1. **Limitações do Assembler**
   A simplicidade da arquitetura limita o número de registradores disponíveis e a ausência de suporte para macros ou pseudoinstruções.

2. **Desempenho e Escalabilidade**
   O desempenho foi satisfatório para os testes realizados, mas expansões no modelo atual, como o suporte a mais registradores, exigiriam alterações significativas no esquema de codificação.

3. **Impacto Educacional**
   O assembler demonstrou ser uma ferramenta didática, facilitando o entendimento prático de conceitos fundamentais em arquitetura de computadores.

4. **Futuras Melhorias**
   Além de resolver a ambiguidade mencionada, futuras versões podem incluir mais registradores, suporte a macros e integração com simuladores para execução direta das instruções.

9 Conclusões

O ponto de partida deste trabalho foi a necessidade de criar um assembler capaz de traduzir instruções de montagem para código binário, destinado a um processador de 8 bits desenvolvido como parte de um projeto acadêmico. Justificou-se a relevância do estudo pela

contribuição prática ao aprendizado de conceitos fundamentais em arquitetura de computadores, montagem e compiladores. Assim, o objetivo principal foi projetar e implementar uma ferramenta funcional e didática que facilitasse o entendimento do funcionamento de processadores simples, além de explorar as possibilidades e limitações da arquitetura proposta.

Para atingir esses objetivos, foi desenvolvido um assembler utilizando a linguagem Python, com foco em simplicidade e eficiência. A ferramenta foi estruturada para interpretar instruções em linguagem de montagem, validar sua conformidade com as especificações do processador e convertê-las para binário. O projeto contemplou recursos como o processamento de arquivos, gerenciamento de erros e suporte a diferentes tipos de instruções (aritméticas, lógicas, unárias e de movimentação de dados). A implementação seguiu uma metodologia incremental, com testes contínuos para validar a funcionalidade e a robustez da aplicação.

Os principais resultados alcançados incluem um assembler funcional que traduz com precisão diferentes formatos de instruções, identifica e informa erros de forma clara e organiza a saída em formato adequado para uso no processador. Além disso, o projeto revelou limitações importantes, como a ambiguidade entre números imediatos e registradores devido ao espaço de codificação restrito. Apesar disso, o assembler demonstrou ser eficaz em seu propósito educacional, fornecendo uma base sólida para o aprendizado de montagem e permitindo reflexões sobre a expansão e a evolução da arquitetura.

Com base neste trabalho, várias melhorias e aplicações futuras podem ser exploradas. Uma delas é a expansão do conjunto de instruções e o aumento do número de registradores, com ajustes na codificação binária para evitar ambiguidade. Outra possibilidade é a integração do assembler com simuladores, permitindo a execução direta das instruções traduzidas. Além disso, o projeto pode servir como ponto de partida para estudos mais avançados em compiladores, incluindo a criação de novas linguagens de alto nível para o processador e a exploração de técnicas de otimização. Assim, este trabalho representa tanto uma realização significativa quanto uma base para inovações futuras.

Agradecimentos

Gostaria de expressar minha profunda gratidão, primeiramente, ao meu orientador, **Carlos Magnus Carlson Filho**, por aceitar me guiar neste trabalho, oferecendo seu conhecimento, paciência e incentivo ao longo do processo. Minha sincera apreciação também vai à professora **Valéria Maria Volpe**, que desempenhou um papel fundamental ao me orientar nos primeiros passos deste projeto e por ter me apresentado à professora **Luciana Pavani de Paula Bueno**, cuja valiosa orientação foi essencial na criação do compilador, complementando e enriquecendo os resultados alcançados.

A contribuição de cada um foi imprescindível para o desenvolvimento e sucesso deste trabalho, e sou extremamente grato pelo apoio, dedicação e aprendizado proporcionados ao longo dessa jornada.

Referências

AHO, Alfred V. *et al*. **Compiladores**: princípios técnicas e ferramentas. Rio de Janeiro: Ltc, 1985. 350 p.

BACON, David F.; GRAHAM, Susan L.; SHARP, Oliver J. Compiler Transformations for High-Performance Computing. University of California, Berkeley, 1994. ACM Computing Surveys, v. 26, n. 4, p. 345-420.

FITZGERALD, Michael. **Introdução às Expressões Regulares**: desvendando as expressões regulares, passo a passo autor. São Paulo: Novatec Editora, 2019. 160 p.

JØRRING, Ulrik; SCHERLIS, William L. Compilers and Staging Transformations. In: *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*. 1986. p. 86-96.

LEWIS, Philip M.; ROSENKRANTZ, Daniel J.; STEARNS, Richard E. Compiler Design Theory. United States: Addison-Wesley Longman Publishing Co., 1976. 672 p.