UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

CENTRO TECNOLÓGICO

COLEGIADO DO CURSO DE ENGENHARIA DE COMPUTAÇÃO

Carlos Daniel Albertino Vieira

Montador e Emulador didáticos para a arquitetura RISC-V 32 bits com suporte a sessões de depuração no GDB

Vitória, ES

2026

Carlos Daniel Albertino Vieira

Montador e Emulador didáticos para a arquitetura RISC-V 32 bits com suporte a sessões de depuração no GDB

Monografia apresentada ao Colegiado do Curso de Engenharia de Computação do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Bacharel em Engenharia de Computação.

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Colegiado do Curso de Engenharia de Computação

Orientador: Prof. Dr. Rodolfo da Silva Villaca

Vitória, ES

2026

Carlos Daniel Albertino Vieira

Montador e Emulador didáticos para a arquitetura RISC-V 32 bits com suporte a sessões de depuração no GDB/ Carlos Daniel Albertino Vieira. – Vitória, ES, 2026-

92 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Rodolfo da Silva Villaca

Monografia (PG) – Universidade Federal do Espírito Santo – UFES
Centro Tecnológico
Colegiado do Curso de Engenharia de Computação, 2026.

1. Montador. 2. Emulador. I. Carlos Daniel Albertino Vieira. II. Universidade Federal do Espírito Santo. IV. Montador e Emulador didáticos para a arquitetura RISC-V 32 bits com suporte a sessões de depuração no GDB

CDU 02:141:005.7

Carlos Daniel Albertino Vieira

Montador e Emulador didáticos para a arquitetura RISC-V 32 bits com suporte a sessões de depuração no GDB

Monografia apresentada ao Colegiado do Curso de Engenharia de Computação do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Bacharel em Engenharia de Computação.

Trabalho aprovado. Vitória, ES, 04 de fevereiro de 2026:

Prof. Dr. Rodolfo da Silva Villaca
Orientador

Prof. Dr. Eduardo Zanbom
Convidado 1

Prof. Dr. Camilo Arturo Rodriguez Diaz
Convidado 2

Vitória, ES
2026

Em memória ao meu avô José Albertino e minhas avós Leanira Albertino e Ivone Vieira.
Amarei vocês para todo sempre.

Agradecimentos

Agradeço a Deus e meus pais pela vida, e todos meus familiares e amigos que me apoiaram e acompanharam por todo trajeto, sem vocês a vida não faria sentido.

Resumo

A arquitetura RISC-V consiste numa arquitetura aberta de computador, que tem ganhado popularidade nas últimas décadas pela sua natureza aberta e livre de royalties. No contexto da popularização da arquitetura RISC-V, surge o interesse de viabilizar e consolidar sua aceitação na esfera acadêmica e técnico-industrial, de modo a incentivar a adoção de soluções baseadas nessa arquitetura e que sejam de domínio público.

O presente trabalho propõe um ferramental didático que sirva principalmente à esfera academica no ensino e aprendizagem da arquitetura, de modo a se alinhar com tendências modernas vistas na área de computação em prol do RISC-V. Para tanto, o sistema desenvolvido permite a escrita de programas em linguagem de montagem RISC-V, sua codificação em formato binário e a execução das instruções em um ambiente de emulação que suporta depuração. A arquitetura do sistema foi projetada de forma modular, que contempla componentes como montador, emulador e depurador para a arquitetura RISC-V.

A robustez e corretude do sistema foi avaliada por meio de testes automatizados e com a validação cruzada com ferramentas consolidadas, que incluem utilitários GNU e o emulador QEMU. Os resultados indicam que o projeto representa uma ferramenta funcional e extensível no ensino da arquitetura RISC-V.

Palavras-chave: RISC-V. 32 bits. Montador. Emulador. Emulação. Depuração. Ensino. Assembly.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ABI	Application Binary Interface
COFF	Common Object File Format
EEI	Executable Environment Interface
ELF	Executable and Linkable Format
ISA	Instruction Set Architecture
RISC	Reduced Instruction Set Computer
XLEN	Register Length

Sumário

1 Introdução

A arquitetura RISC-V consiste numa arquitetura aberta de computador, que se baseia em um conjunto reduzido de instruções (RISC - Reduced Instruction Set Computer). Essa arquitetura tem ganhado popularidade nos últimos 15 anos pela sua natureza aberta e livre de royalties, que serve a interesses acadêmicos e industriais (LU, 2021). Atualmente, a RISC-V International corresponde à organização internacional que reúne as especificações da arquitetura a toda comunidade, o que impulsiona sua difusão e adoção. Dentre essas, a especificação da arquitetura inclui seu Instruction Set Architecture (ISA), que consiste no conjunto de instruções suportadas e como essas instruções estão organizadas a nível binário na memória (WATERMAN et al., 2019a).

No contexto da popularização da arquitetura RISC-V nos tempos atuais e do seu potencial enquanto alvo de estudo na academia, o projeto consiste na construção de um programa que seja capaz de: processar código de montagem RISC-V em executáveis no formato ELF, emular programas da arquitetura RISC-V 32 bits e oferecer suporte a sessões de depuração através do GDB. Com isso, objetiva-se prototipar uma solução que seja capaz de cobrir a execução de um programa desde a sua escrita em linguagem de montagem até a execução em um ambiente emulado real, com suporte a depuração.

Nesse sentido, a seção 1 busca introduzir o cenário em torno da arquitetura RISC-V, de modo a apresentar a motivação por trás desse trabalho e o escopo de atuação definido. A seção 2 trata da explicação de alguns conceitos fundamentais para o entendimento do projeto e que são frequentemente revisitados no decorrer do texto. Em seguida, a seção 3 explica a especificação do sistema, o que inclui a descrição em alto nível de suas partes e de seu funcionamento, os requisitos a serem atendidos e escolhas arquiteturais tomadas. Uma vez fundamentado o aspecto teórico e o planejamento do sistema, a seção 4 progride com a descrição de sua implementação, de modo a analisar a lógica, estratégia, decisões e resultados por trás das unidades construídas. Ao final, a seção 5 apresenta uma análise final do trabalho quanto ao seu êxito no cumprimento da proposta e objetivos traçados, bem como também traz possíveis linhas que trabalhos futuros poderiam explorar para estender ou enriquecer as funcionalidades apresentadas.

1.1 Motivação

A arquitetura RISC-V representa um passo importante na consolidação de uma arquitetura livre, acessível, pragmática e que seja relevante tanto em ambientes acadêmicos como em ambientes industriais/comerciais. Nesse sentido, o estado atual do RISC-V é promissor, dado que boa parte das especificações técnicas em torno da arquitetura

se encontram ratificadas, o que abre margem para que essa seja levada à frente com seriedade por diferentes personagens importantes nessa trama, como estudantes, professores, empresários, investidores, fabricantes e outros (LU, 2021).

De todo modo, a adoção da arquitetura ainda enfrenta desafios, frente a popularidade de outras arquiteturas proprietárias e que atualmente detém muito mais notoriedade no ramo comercial e industrial (RANTAKARI; TESTA, 2025). No ramo acadêmico também é possível perceber um movimento similar, onde geralmente prefere-se pelo uso de arquiteturas já bem conhecidas para estudo e fins didáticos, tais como MIPS e x86-16. Nesse sentido, é importante fomentar o conhecimento, estudo e entendimento da arquitetura na forma de ferramentas que permitam diferentes atores a se familiarizar com o RISC-V.

Atualmente existem outros projetos que seguem na direção de ensino da arquitetura por via de simuladores e emuladores. Alguns desses projetos, como o rars, ripes e interactive risc-v simulator se enquadram nessa categoria e serviram como inspiração para a formalização do projeto em questão, que busca desenvolver do zero boa parte do ferramental necessário para a construção de um montador e emulador de RISC-V 32 bits e que, diferente dos demais projetos, seja capaz de gerar arquivos no formato ELF com suporte a depuração e viabilize a portabilidade de programas dessa arquitetura com outros emuladores, máquinas virtuais e ambientes reais RISC-V 32 bits.

Nesse viés, o presente projeto busca enriquecer o ecossistema em torno da arquitetura RISC-V com um montador e emulador para fins didáticos e que possa auxiliar discentes e docentes na formação de um conceito sólido acerca da arquitetura.

1.2 Objetivos

O presente projeto tem como objetivo geral a construção de um programa que seja capaz de gerar executáveis no formato ELF a partir de código de montagem RISC-V via linha de comando e de servir de emulador da arquitetura RISC-V 32 bits, com suporte a sessões de depuração via GDB.

1.2.1 Objetivos específicos

Com o intuito de atingir o objetivo geral delineado, este trabalho estabelece como objetivos específicos o desenvolvimento de um ambiente capaz de reproduzir, de maneira fidedigna, o ciclo de execução de uma máquina RISC-V de 32 bits, materializado pelo esquema de *fetch-decode-execute*, em conformidade com as especificações formais da arquitetura.

Busca-se ainda possibilitar a inspeção detalhada do estado interno da máquina emulada ao longo da execução dos programas, contemplando registradores, memória e

demais componentes do estado arquitetural, de modo a auxiliar tanto na depuração quanto na validação do comportamento do sistema.

Outro objetivo consiste na implementação de um montador capaz de gerar executáveis finais no formato ELF a partir de código-fonte escrito em linguagem de montagem RISC-V, permitindo sua execução em ambientes Linux compatíveis com a arquitetura alvo.

Adicionalmente, propõe-se a implementação de testes unitários e de integração que assegurem a corretude e a consistência tanto dos binários gerados quanto do ambiente de emulação desenvolvido.

Por fim, pretende-se disponibilizar todas as funcionalidades do sistema por meio de uma interface de linha de comando, de forma a facilitar sua utilização, automação e integração com outras ferramentas do ecossistema de desenvolvimento.

2 Revisão Bibliográfica

2.1 RISC-V 32 bits

Um ambiente computacional que implementa a arquitetura RISC-V é caracterizado pela presença de uma ou mais unidades de processamento, as quais são chamadas de *cores*. Tais unidades possuem um subcomponente independente responsável pela leitura de instruções, ação essa conhecida como *fetch*. Nesse sentido, após a execução do *fetch*, um *core* pode executar instruções em uma ou mais *threads* implementadas fisicamente em *hardware*, que recebem o nome de *harts*. A matéria constituinte da execução de um ou mais *cores* corresponde a instruções que fazem parte da ISA da arquitetura. Adicionalmente, um *core* pode contar com um outro componente conhecido como *coprocessor*, que pode oferecer suporte a sequências de instruções mais personalizáveis e refinadas do que aquelas tipicamente previstas pelo conjunto de instruções base e padrão do RISC-V (WATERMAN et al., 2019a). Nesse viés, um sistema condizente à arquitetura RISC-V fornece às aplicações minimamente um *core*, o qual é responsável pela operação de *fetch* e execução das instruções obtidas em no mínimo um *hart*.

Concomitantemente, um *hart* possui acesso a um espaço de memória linear, endereçável até 4 GiB e que é modular, isto é, o último endereço de memória está ao lado do endereço 0. Desse modo, acessos de memória fruto de *overflow* são ignorados e simplesmente reiniciam a partir do endereço inicial.

Além dos recursos físicos disponíveis, como *cores* e *harts*, o comportamento da execução de programas na arquitetura RISC-V é também determinado pelo ambiente de execução envolto daquela aplicação, que poderá existir na forma de um sistema operacional, *hypervisor*, ambientes de emulação ou não existir (sendo tal modo conhecido como *bare metal*). A interface do ambiente de execução (EEI) será então responsável por: definir o estado inicial do programa, número de *harts* disponíveis, regiões de memória acessíveis, conveções de chamada de primitivas do sistema subjacente e dentre outros.

2.2 ISA do RISC-V

A documentação oficial da arquitetura RISC-V está dividida em diferentes materiais disponíveis digitalmente no site da RISC-V International, os quais explicam tecnicamente aspectos variados da especificação arquitetural. Dentre os materiais divulgados está a documentação da ISA, que está dividida em dois módulos: ISA privilegiada e ISA não-privilegiada. A ISA privilegiada define tratativas referentes ao ambiente de execução e

mecanismos de controle para o gerenciamento de diferentes aspectos da execução de um programa, tais como: níveis de privilégio, gerenciamento de exceções e *traps*, endereçamento, memória virtual e outros tópicos avançados (WATERMAN et al., 2019b). De forma complementar, a ISA não privilegiada cobre os diferentes conjuntos de instruções que fazem parte da arquitetura RISC-V, de forma a descrever os diferentes formatos de instrução, o funcionamento e os comportamentos esperados de cada primitiva (WATERMAN et al., 2019a).

A documentação da ISA não privilegiada traz a definição de que uma ISA do RISC-V consiste mandatoriamente de uma ISA base para números inteiros e de opcionalmente outros conjuntos de instruções que oferecem funcionalidades variadas, as quais são referidas como extensões. Atualmente, a ISA base para números inteiros existe na forma de 4 variantes (RV32I, RV64I, RV32E e RV64E), que se diferenciam no tamanho dos registradores de inteiro (XLEN), no limite do espaço de memória endereçável e no número de registradores de inteiro disponíveis. De modo geral, as variantes RV32I e RV64I contam com 32 registradores e foram pensadas para conformar a aplicações de uso geral, enquanto as variantes RV32E e RV64E dispõem de 16 registradores e são voltadas para sistemas embarcados e/ou com recursos limitados (WATERMAN et al., 2019a).

Sob uma análise mais detalhada da variante base RV32I, essa se refere a arquitetura de 32 bits, que conta com 32 registradores de 32 bits e um espaço de memória endereçável até 4 GiB. Quanto ao conjunto de instruções de números inteiros, a ISA RV32I define 40 instruções básicas para a operação, leitura, armazenamento e controle de fluxo e também 6 formatos de instrução e 5 formatos de imediatos, identificados pelas letras R, I, S, U, B e J. Um formato de instrução consiste em um conjunto de campos que armazenam números que identificam registradores, valores inteiros, opcodes, endereços e tipos de operação, enquanto um formato de imediato consiste na forma como valores numéricos são convertidos para serem armazenados nos seus respectivos formatos de instrução. Cada uma das 40 instruções é montada utilizando apenas um dos formatos de instrução disponíveis, o que varia a depender do tipo de operação em questão. A exemplo de uma dessas instruções, é possível observar na Figura 1 como algumas instruções referentes a operações com imediatos do tipo inteiro são estruturadas de acordo com o formato de instrução R. O formato R aceita ao todo 5 campos: um campo para um imediato entre os valores de -2048 e +2047 (imm), um campo para o registrador de destino (rd), um campo para um registrador de operando (rs1), um campo para identificar o tipo da operação em questão (funct3) e outro campo para identificar o opcode específico da instrução de formato R (funct7) (MEZGER et al., 2022).

De forma complementar a ISA base RV32I, diferentes conjuntos de instruções podem ser adicionados ao repertório do sistema, de forma a oferecer funcionalidades adicionais a serem suportadas pelo ambiente de execução. Nesse aspecto, conjuntos de instruções

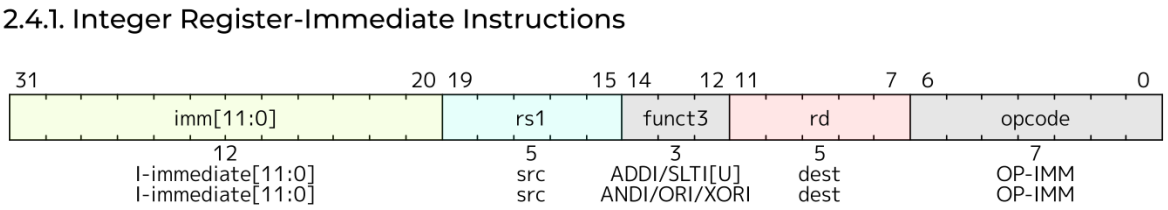

Figura 1 – Conjunto de instruções da ISA RV32I do formato R

que complementam o conjunto base para números inteiros são formalmente referenciados como extensões, algumas das quais são consideradas 'padrão' como as extensões para: divisão e multiplicação (RV32M), números de ponto flutante (RV32F) e operações atômicas (RV32A).

A documentação não privilegiada cobre também diversas outras extensões, de modo a conduzir o detalhamento de cada uma dessas em capítulos separados, que buscam cobrir a codificação de instruções de máquina em inteiros de 32 bits.

2.3 Código de montagem para RISC-V

Ambas as documentações da ISA não privilegiada do RISC-V Waterman et al. (2019a) e o manual de código de montagem para programadores RISC-V RISC-V International (2016) têm um papel fundamental no fomento de uma representação simbólica em linguagem de montagem, ou *assembly*, que seja consistente e compatível com as muitas extensões da arquitetura. Nesse sentido, vale ressaltar a relação direta que o código assembly possui com a representação binária tal qual prevista na ISA, mas que almeja abstrair o detalhismo técnico relacionado ao seu formato, que inclui: opcodes, ordem dos campos, codificação de valores, dentre outros. Tal abstração é importante para viabilizar a escrita de programas que sejam inteligíveis e compreensíveis por humanos, mas que continuem fidedignos a maneira como uma máquina processaria as instruções em formato binário. Um exemplo que demonstra como o assembly pode tornar uma instrução de máquina mais palatável e familiar à semântica humana se encontra na Figura 2.

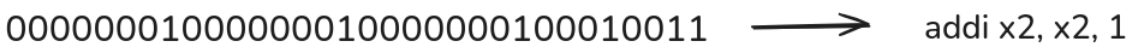

Figura 2 – Conversão de instrução de adição do formato binário para assembly RISC-V

A abstração propiciada pela linguagem assembly também é fundamental para facilitar referências feitas aos registradores, com nomes curtos e de fácil memorização, o que é conveniente na escrita de programas. Nesse aspecto, a convenção tradicional para referenciar algum registrador é utilizar um nome formado pela letra 'x' prefixada ao

número do registrador em questão. Para além disso, também é comum associar apelidos aos registradores de acordo com convenções que remetem a forma como são usados na documentação não privilegiada, como para salvar endereços de retorno, de parâmetros de funções, valores temporários e entre outros. Uma relação completa dos registradores com seus nomes tradicionais, apelidos semânticos e descrições se encontra na tabela 1.

Tabela 1 – Tabela com relação dos nomes, apelidos e descrições dos registradores do RISC-V 32 bits.

Registrador	**Apelido**	**Descrição**
x0	zero	Registrador constante, sempre contém o valor zero. Escritas são ignoradas.
x1	ra	Registrador de endereço de retorno de chamadas de função.
x2	sp	Registrador ponteiro da pilha (*stack pointer*).
x3	gp	Registrador ponteiro global (*global pointer*).
x4	tp	Registrador ponteiro de *thread* (*thread pointer*).
x5 - x7	t0 - t2	Registradores temporários.
x8	s0/fp	Registrador salvo; também utilizado como ponteiro de quadro (*frame pointer*).
x9	s1	Registrador salvo.
x10 - x17	a0 - a7	Registradores de argumento de função (ao todo 8 argumentos); Os registradores a0 e a1 são utilizados para o valor de retorno de funções.
x18 - x27	s2 - s11	Registradores salvos.
x28 - 31	t3 - t6	Registradores temporários.

Para além da relação direta e um-para-um das primitivas assembly relacionadas às instruções de extensões da ISA, é de fundamental importância a lacuna preenchida pelas *pseudo instruções*, que resumem uma sequência de 1 ou mais de outras instruções que frequentemente são utilizadas e repetidas em rotinas comuns para se atingir um determinado objetivo. Pela necessidade de padronização, praticidade e brevitude, as pseudo instruções fornecem uma forma alternativa e mais concisa de se convir a um mesmo significado, como em operações de: *push* e *pop* da pilha, salvamento e recuperação de contexto de uma rotina, carregamento de endereços simbólicos, dentre outros. Um exemplo disso é a instrução *NOP*, que consiste em uma operação que não possui efeitos observáveis (não gera efeitos colaterais) e que é traduzida como uma operação de soma de valores nulos e que são escritos no registrador 0, que deliberadamente não salva os valores nele escritos. Outra vantagem fornecida pelo uso de tais instruções, é a de possibilitar que o montador, ou *assembler*, escolha a sequência de instruções mais apropriada que convenha a uma dessas operações a depender do ambiente alvo de execução, o que pode propiciar a geração de código de máquina mais eficiente e específico, porém sem comprometer a

qualidade descritiva do assembly que o representa.

2.4 Formato ELF

A arquitetura de Von Neumann prevê um sistema composto por uma memória e uma unidade de processamento, as quais atuam em conjunto para a execução de um programa, que é carregado em memória e então executado (TANENBAUM; AUSTIN, 2013, p. 18). Assim como comentado na subseção 2.1, a arquitetura RISC-V conta com um modelo de memória linear e modular, com a presença de *cores* e *harts* para a leitura e execução das instruções, de forma a conformar ao modelo previsto pela arquitetura de Von Neumann. No entanto, é também importante a definição de como um programa pode ser empacotado para posterior carregamento, decodificação e execução. Historicamente, tal tarefa envolve, dentre outras decisões, a escolha de um formato de arquivo apropriado para a estruturação do programa em memória. Nesse contexto, o formato ELF se consolidou no universo Unix como parte integrante da ABI dos sistemas operacionais para permitir o carregamento e execução dos programas. A adesão do ELF nos ambientes Unix foi propulsionada na década de 90, após a formalização desse formato em uma série de revisões e publicações que foram abertas ao público e no incentivo a sua adoção (TOOL INTERFACE STANDARDS COMMITTEE, 1995), o que marcou a predominância desse formato sobre outros que eram comuns a época, como COFF e *a.out*.

Nesse contexto, o ELF corresponde a um formato multi plataforma, cuja especificação define uma estrutura interna que é capaz de representar arquivos executáveis. No contexto da execução de aplicações, programas escritos no formato ELF facilitam a portabilidade do *software* para diferentes ambientes de execução que o suportem, tais como: sistemas operacionais, *hypervisors*, emuladores e simuladores. Para tanto, o formato ELF conta com uma descrição interna formada por cabeçalhos, que contém informações diversas sobre o arquivo e que servem a diferentes ferramentas (como *loaders* e *linkers*), e seções, que se referem aos dados e código do programa em si. De forma geral, um sistema que suporta o formato ELF realiza o carregamento em regiões da memória das estruturas ali descritas e transfere o controle de execução para o ponto de entrada definido no arquivo, que será executado até que finalmente termine.

2.5 Emulação

Um emulador consiste em um programa capaz de fazer um sistema fonte se comportar como algum outro sistema alvo. Nesse sentido, cabe ao sistema original reproduzir de forma fidedigna o funcionamento do sistema alvo, de modo a espelhar o modo como seus componentes operam e interagem entre si. A exemplo disso, a emulação da arquitetura RISC-V consiste em seguir o modelo do funcionamento arquitetural tal como exibido na

subseção 2.1, de modo a reproduzir o ciclo *fetch-decode-execute* e permitir a execução de programas compilados para o ambiente RISC-V. A depender do suporte de diferentes componentes, recursos computacionais e funcionalidades, a emulação pode oferecer um ambiente ainda mais rico, completo e próximo do alvo da emulação.

A grande vantagem da emulação de ambientes RISC-V é a de permitir vislumbrar a execução de programas compilados para a arquitetura, sem a necessidade de possuir um sistema RISC-V real. Essa característica confere uma vantagem significativa ao uso didático, pois permite que o usuário possa se familiarizar com a arquitetura e experimentá-la com uma máquina que já tenha acesso, sem antes haver a necessidade de comprar um computador ou sistema que a implemente. Portanto, ainda que não existam fabricantes que produzam computadores ou sistemas RISC-V em larga escala (salvo algumas exceções de sistemas embarcados), a emulação viabiliza a escrita e teste de programas que funcionem nesse ambiente.

2.5.1 Modelo de Memória

O ambiente de emulação requer um modelo de memória análogo àquele considerado em ambientes RISC-V. Nesse aspecto, a emulação do RISC-V deve buscar a utilização de memória linear, modular e endereçamento a *byte*. Para a arquitetura de 32 bits, o espaço de memória endereçável contempla uma região de 4 GiB com alinhamento de 4 bytes para dados. Quanto ao *endianness*, é convencional a adoção do *little endian*, não sendo usual a utilização do *big endian*.

2.5.2 Modelo de CPU

Assim como comentado previamente, a emulação das unidades de processamento do RISC-V requer reprodução fiel do ciclo de *fetch-decode-execute* feito por 1 ou mais *cores* e *harts*. Um modelo de CPU minimalista da arquitetura RISC-V possui um único *core* e um *hart* para o sustento do ciclo em questão, que foi a estratégia adotada para esse projeto, a fim de evitar problemas e consequências clássicas que surgem ao envolver o suporte a *multithreading*.

No contexto da execução *single-core* e sem *multithreading*, cabe ao único *core* realizar a leitura de uma instrução por vez da seção de memória que contém o código executável, decodificar o tipo de instrução e executar o comando em questão na estrutura interna da máquina.

Paralelamente, a execução das instruções requer o gerenciamento do *program counter*, ou *PC*, que consiste em um registrador especial, aparte dos 32 de propósito geral, o qual referencia o próximo endereço de memória (alinhado em 4 bytes) a ser processado pelo ciclo do *fetch-decode-execute*. Instruções que não alteram o fluxo normal de um

programa requerem simplesmente que o PC aponte para o próximo endereço alinhado em 4 bytes para que o programa continue normalmente. No entanto, instruções que podem potencialmente alterar o fluxo de um programa, como saltos condicionais, saltos incondicionais e retorno de rotinas, requerem um cálculo mais cuidadoso do próximo valor que o PC poderá assumir. Nesse sentido, uma alternativa que pode ser implementada consiste na predição do próximo valor a ser assumido pelo PC, a qual confere a possibilidade de enriquecer sessões de depuração ao permitir informar ao *debugger* quanto ao próximo endereço, cujo fluxo será transferido, sem a necessidade da execução premeditada da instrução.

Por fim, outro cuidado a ser direcionado durante a execução das instruções se refere ao gerenciamento de comportamentos anormais que podem se suceder no ambiente de execução. Tais anomalias consistem, em geral, em uma entre três: exceções, interrupções e *traps*. Segundo a documentação da ISA não privilegiada Waterman et al. (2019a)[p. 18-19], uma exceção consiste em uma condição não usual que ocorre durante a execução de uma instrução em um *hart*, como um erro gerado por divisão por zero. Por outro lado, uma interrupção se refere a um evento assíncrono que pode causar o *hart* em execução a passar por uma transferência de controle, como os sinais POSIX. Por fim, uma *trap* consiste na transferência de controle a uma rotina personalizada sob a ocorrência de uma exceção ou de uma interrupção. Nesse contexto, o ambiente de emulação pode oferecer suporte parcial ou completo a tais anomalias, a depender do grau de fidelidade que almeja com o ambiente real. Neste trabalho, não foram implementados mecanismos de interrupção ou tratamento de exceções, restringindo-se o emulador à execução correta das instruções em fluxo normal.

2.6 Depuração

Um executável escrito no formato ELF pode ser enriquecido com informações de depuração por meio do formato DWARF, que consiste em uma especificação protocolada e em sua quinta revisão que serve para adicionar informações que descrevem programas de diferentes linguagens de programação (C, C++, Cobol e outros) ao formato ELF. Em geral, as informações de depuração são injetadas no arquivo executável final por meio de ferramentas como compiladores, montadores e ligadores (WORKGROUP, 2017)[p. 19], mas a princípio não são restritas a apenas tais programas.

Por meio da informação fornecida pelo formato DWARF, é possível que programas forneçam sessões de depuração ricas em detalhes relacionados a diferentes aspectos da execução de um programa, tais como: definição de pontos de quebra (*breakpoints*), execução passo a passo, inspeção de memória, inspeção de registradores, inspeção da pilha de execução, interação em tempo real com o programa, dentre outros. Para programas

compilados em código de máquina, a capacidade de acompanhar o estado de execução da máquina durante a sua execução confere a possibilidade de examinar cuidadosamente o fluxo de controle do programa, o que é benéfico a programadores de diferentes níveis de familiaridade com linguagem de montagem, ao passo que permite relacionar instruções de montagem com suas representações em binário, o que facilita a detecção de erros e consolida a ligação de ambos.

Assim, ganha relevância o depurador, o qual le as informações de depuração presentes no executável e fornece uma interface ao usuário para que examine e interaja com um programa em execução. Nesse contexto, um depurador conhecido e consolidado para programação em mais baixo nível é o *GNU Debugger*, ou GDB, que fornece muitas funcionalidades adicionais além daquelas esperadas de um depurador simples.

Uma das funcionalidades especiais fornecida pelo GDB consiste no suporte a sessões remotas de depuração. Uma sessão de depuração remota permite a inspeção de um programa que esteja rodando em uma máquina remota, denominada *target* ou alvo, a partir de uma máquina local que esteja executando o GDB, denominada *host* ou hospedeiro. Esse modelo permite a separação e independência dos ambientes de execução e de depuração, os quais podem se basear em máquinas de arquiteturas diferentes. A exemplo dessa funcionalidade, é perfeitamente possível que uma máquina pertencente a arquitetura x86-64 utilize o GDB para inspecionar um programa compilado para a arquitetura RISC-V 32 bits, que será executado em um ambiente de execução remoto compatível, como uma máquina real/virtual ou emulador.

O funcionamento da depuração remota oferecida pelo GDB funciona por meio de uma arquitetura cliente-servidor, na qual a máquina hospedeira troca mensagens com a máquina alvo por meio de comunicação serial e de um canal de rede. Para tanto, as mensagens obedecem ao *GDB Remote Serial Protocol*, ou Protocolo Remoto Serial GDB, que dita as regras que devem ser obedecidas por ambas as partes durante a comunicação, o formato das mensagens e a forma como a comunicação se estrutura. Mais especificamente, esse protocolo pode ser dividido em duas fases: o *handshake* e a troca de mensagens. A fase do *handshake* caracteriza o início da comunicação entre o hospedeiro e o alvo, sendo nesse momento estabelecida a conexão e o envio das informações suportadas pelo ambiente alvo. Enquanto isso, a fase de troca de mensagens caracteriza o momento em que ambas as partes envolvidas já estão cientes das funcionalidades suportadas e oferecidas, de modo a consistir em uma série de comandos codificados, que retratam tanto as ações de depuração executadas pelo usuário, como as respostas fornecidas pelo ambiente alvo aos comandos em questão (FREE SOFTWARE FOUNDATION, 2023).

O fluxo de mensagens trocadas entre a máquina hospedeiro e a máquina alvo conta com a participação de um programa intermediário, denominado *stub*, o qual é responsável por recepcionar as mensagens enviadas pelo programa cliente do GDB (no hospedeiro),

encaminha-las ao ambiente de execução (no alvo) e vice-versa. No campo da arquitetura da depuração remota do GDB, o *stub* se refere ao elemento que media a comunicação e controla o envio e a entrega das mensagens a ambas as partes.

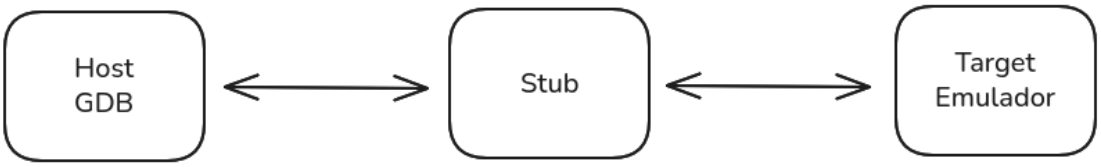

Figura 3 – Diagrama do fluxo de mensagens trocadas entre os elementos presentes na arquitetura de depuração remota do GDB

O entendimento desses mecanismos fundamenta a implementação do stub neste projeto, responsável por emular as operações esperadas pelo GDB ao interagir com um sistema RISC-V.

2.7 Ferramentas utilizadas

O código por trás desse projeto foi desenvolvido na linguagem de programação Rust e contou com a utilização de diferentes ferramentas para testar e validar os resultados provenientes do montador, emulador e da depuração. As ferramentas consistem majoritariamente em programas e utilitários que permitem interagir com programas escritos para a arquitetura RISC-V, entre os quais estão: QEMU, RISC-V 32 bits toolchain, Kernel do linux, Busybox e GDB. Os demais capítulos dessa subseção buscam apresentar a finalidade desssas ferramentas, bem como também apresentar a maneira como foram empregadas nesse projeto.

Quanto a linguagem de programação escolhida neste trabalho, Rust consiste em uma linguagem multiparadigma e que é amplamente conhecida pela segurança no manejo de memória, alta expressividade, tipagem forte e performance. Rust explicita conceitos como tempo de vida e *ownership* para a escrita de programas válidos, que são assegurados pelo mecanismo do compilador conhecido como *borrow checker*. Isso permite a escrita de programas robustos, que são capazes de lidar com problemas típicos que surgem em implementações de baixo nível, como condições de corrida, ponteiros inválidos, *double-free*, paralelismo e entre outros. Para além disso, a linguagem apresenta um ecossistema rico de bibliotecas que oferecem APIs diversas para a manipulação de uma gama de funcionalidades e/ou aplicações, bem como um gerenciador de pacotes próprio (*cargo*), que trata da compilação, distribuição, documentação, testes, dentre outras tarefas que facilitam o gerenciamento de projetos (KLABNIK; NICHOLS, 2024). Nesse contexto, a

linguagem Rust foi escolhida para o desenvolvimento do projeto, em razão de oferecer um controle fino sobre operações diversas de mais baixo nível, um sistema de testes unitários embutidos no gerenciador de pacotes, inúmeras bibliotecas que aceleram o desenvolvimento e mecanismos próprios da linguagem que reforçam boas práticas de código

Entre as ferramentas utilizadas, o programa QEMU consiste em um emulador vastamente utilizado para a emulação de diferentes arquiteturas de computador (BELLARD, 2005). Nesse trabalho, o QEMU foi empregado como uma base comparativa para validar o comportamento dos executáveis produzidos pelo montador. Por se tratar de um emulador popular, consolidado e amplamente utilizado nas esferas acadêmicas e industriais, a utilização desse programa busca tornar a etapa de montagem confiável, de modo a garantir que os programas gerados nesse projeto sejam válidos e também funcionais em outras ferramentas.

No contexto da ambientação de um sistema RISC-V 32 bits real, a emulação via QEMU contou com a utilização da versão 6.18.0 do kernel do Linux para a arquitetura em questão, o qual foi compilado manualmente a partir do código-fonte disponibilizado no repositório do github (TORVALDS; CONTRIBUTORS, 2024). A execução do kernel em ambiente emulado permitiu validar o suporte a instruções, chamadas de sistema e mecanismos básicos de inicialização, além de servir como um cenário mais próximo de um ambiente real de execução. Esse ambiente foi enriquecido por meio do Busybox, que corresponde a um programa que implementa diversos utilitários de linha de comando que são comuns em distribuições Linux, como os comandos *cd* e *ls*, o qual foi compilado também manualmente para a arquitetura RISC-V 32 bits.

Além do ambiente de emulação utilizado, o projeto contou com a utilização de diferentes utilitários de linha de comando que funcionam em computadores da arquitetura x86-64, mas que almejam arquivos e programas preparados para a arquitetura RISC-V 32 bits. Para tanto, foi utilizado o *riscv32 toolchain*, que consiste em um conjunto de ferramentas que oferecem uma base para trabalhar diferentes aspectos da análise, preparo e reprodução de programas da arquitetura RISC-V (RISC-V Collaboration and contributors, 2024). Nesse sentido, foram utilizadas ferramentas essenciais no fluxo de desenvolvimento desse trabalho, como o montador *as*, o ligador *ld*, e utilitários de inspeção e análise de binários, como *readelf* e *objdump*, com o objetivo de consolidar a corretude e validade dos produtos obtidos nesse trabalho.

Quanto a depuração, assim como mencionado na subseção 2.6, foi utilizado o programa GDB como ferramenta de depuração externa, por meio do qual é possível se conectar ao emulador feito e interagir com a execução dos programas construídos. No contexto educacional e de desenvolvimento, o GDB se mostra como um forte aliado, pois permite que os usuários acompanhem a execução dos programas, o que facilita a compreensão do funcionamento interno da arquitetura.

Em suma, as ferramentas apresentadas fomentam a base ferramental utilizada no desenvolvimento do projeto, de modo a cobrirem desde a geração e análise de binários até a execução, emulação e depuração de programas RISC-V. A Tabela 2 apresenta um resumo das principais ferramentas empregadas no escopo desse trabalho.

Tabela 2 – Ferramentas utilizadas no desenvolvimento do projeto

Ferramenta	**Descrição**
Rust	Linguagem principal utilizada na implementação do montador e do emulador
Cargo	Gerenciador de dependências e sistema de build da linguagem Rust
QEMU	Emulador de referência utilizado para validação
Toolchain RISC-V	Conjunto de ferramentas para geração e análise de binários
Linux Kernel	Sistema operacional utilizado como carga de teste
BusyBox	Conjunto de utilitários de espaço de usuário
GDB	Ferramenta de depuração

3 Especificação do Sistema

A especificação do projeto busca estruturar uma solução que seja capaz de lidar com um processo que inclua a montagem, emulação e depuração de programas para a arquitetura RISC-V de 32 bits. No mesmo raciocínio, cabe a definição explícita das interfaces que conectam esses 3 elementos, de forma a tratar a entrada, processamento e saída esperada em cada uma das etapas. Para o cumprimento deste objetivo, faz-se também necessária a definição do escopo do projeto, de modo a determinar quais requisitos, compromissos, restrições e relaxamentos conduziram o design, bem como também esclarecer os motivos por trás das decisões que os cerca.

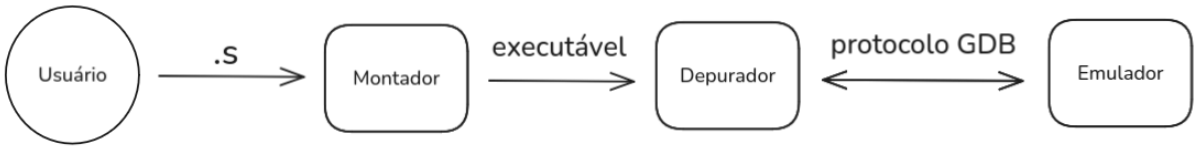

Figura 4 – Panorama geral da arquitetura do sistema

A Figura 4 apresenta um diagrama que exibe os principais elementos que participam da arquitetura da solução, bem como as interfaces de entrada e saída que os conectam. Nesse contexto, é relevante esclarecer que o projeto apresenta autonomia suficiente para funcionar sem que todos os elementos da figura estejam presentes. A exemplo disso, é possível que os executáveis sirvam de entrada do emulador, ao invés do depurador. Da mesma forma, é possível que arquivos executáveis produzidos por outras ferramentas de montagem ou compilação sejam utilizados, de modo a pular a etapa inicial do montador feito especialmente para esse projeto. Esse caso requer que as ferramentas utilizadas compilem os programas com opções que produzam executáveis compatíveis com o emulador feito.

3.1 Montador

O montador consiste num elemento importante no design da arquitetura da solução, uma vez que será responsável pela emissão do executável a ser processado pelo emulador. Nesse aspecto, a partir do código fonte escrito em linguagem de montagem, o montador possui o objetivo de gerar código de máquina compatível com as funcionalidades suportadas pelo emulador, que incluem: as extensões da ISA, as *features* do formato ELF utilizadas, o ponto de entrada do programa, entre outros.

O domínio da etapa de geração de código de máquina a partir de um montador

próprio permite flexibilidade e autonomia nesse projeto, à medida que é possível ajustar ambas as partes simultaneamente para que se entendam e se acoplem como especificado. No entanto, esse mesmo ponto traz desvantagens, como a possibilidade do emulador ficar muito específico e atuar apenas com executáveis produzidos pelo montador feito. Por conta disso, o design do montador deve assegurar que sua saída não distoe gravemente de outras ferramentas de compilação, como o montador *as*, a fim de garantir o compromisso do emulador de funcionar para arquivos no formato ELF produzidos por outros programas de montagem/compilação disponíveis. A independência do montador e do emulador permite também que esses evoluam a passos diferentes, de modo que o emulador possa ser enriquecido para suportar funcionalidades ainda não implementadas na etapa de montagem. Na prática, isso permite que o emulador não se prenda ao ciclo de desenvolvimento do montador feito especialmente para esse projeto, o que traz a vantagem do ambiente de emulação poder ser preparado para outras ferramentas mais avançadas, para que não fique defasado por conta do montador em voga.

Como a etapa de conversão de código de montagem no formato binário está contida no montador, seu design deve permitir que os opcodes disponíveis para uso na linguagem de montagem acompanhem a evolução da ISA do RISC-V, de modo a facilitar a implementação de novas extensões.

Em vista do escopo definido, a arquitetura do montador inclui ao todo 5 entidades, as quais são responsáveis por conduzir o processamento do código fonte até o executável final no formato ELF. A primeira entidade corresponde ao *Lexer*, cuja função é a de simplesmente ler o arquivo que contém o código de montagem e formar palavras a partir dos caracteres ali presentes. A segunda entidade é intitulada *Tokenizer*, a qual classifica as palavras provenientes do *Lexer* de acordo com a função sintática dessas no código e as retorna na forma de *tokens*. Vale destacar que como é papel do *Lexer* identificar uma palavra como um Opcode, é também a responsabilidade dessa entidade de distinguir as instruções que pertencem a diferentes extensões. A terceira entidade corresponde ao *Parser*, cuja função consiste em agrupar os *tokens* da etapa anterior em instruções e também as instruções em grupos. A quarta entidade corresponde ao *Núcleo do Montador*, cujo objetivo é estruturar a saída da etapa anterior de modo a permitir a resolução de símbolos, endereçamento de instruções, formatação de seções e outros detalhes necessários para a conversão de instruções de alto nível para o formato binário previsto pela ISA do RISC-V. Por fim, a quinta e última etapa corresponde a etapa de ligação, que é responsável pela composição de diferentes arquivos objetos para a escrita do arquivo ELF final.

Na Figura 5 é possível observar um diagrama que contém as entidades que fazem parte do Montador e como a entrada e saída está definida para cada elemento. O modelo em questão trata o processo de montagem como sequencial, no qual cada etapa é responsável por aproximar a entrada inicial ao objetivo final.

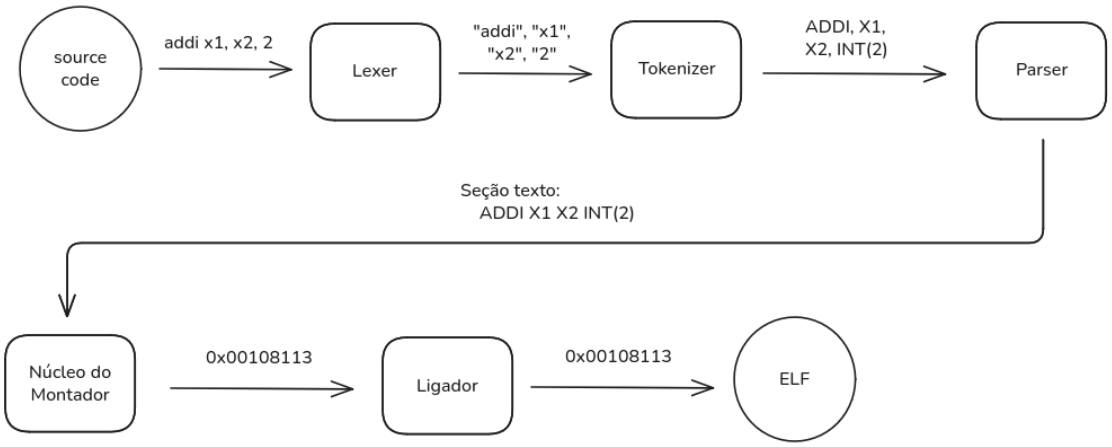

Figura 5 – Panorama geral da arquitetura do montador

A partir do modelo em questão, o montador oferecerá executáveis no formato ELF e com suporte a depuração como produto final, que poderão servir como entrada para o Emulador e/ou Depurador, que respondem pelos procedimentos de execução do programa.

Assim, uma vez definido o processo de geração de executáveis compatíveis com a arquitetura RISC-V, é necessário especificar o ambiente que será responsável pela sua execução. Nesse contexto, a seção seguinte descreve a arquitetura do emulador, cuja função é reproduzir o comportamento funcional da arquitetura alvo a partir do código de máquina gerado ou fornecido externamente.

3.2 Emulador

A especificação do emulador busca estruturar um sistema que seja capaz de executar código de máquina da arquitetura RISC-V, de forma a incluir os componentes de processamento e de memória, que foram revisados na subseção 2.1, bem como também reproduzir o ciclo de *fetch-decode-execute*. Além disso, o sistema deve permitir que a memória da máquina seja carregada com programas provenientes de executáveis e também ser capaz de configurar o ponto de entrada dos programas, por meio da definição do registrador PC.

A Figura 6 contém um modelo na forma de diagrama que exibe as entidades envolvidas no funcionamento básico do emulador descrito anteriormente. Nesse contexto, é possível perceber a presença de 2 entidades principais: a *Máquina* e o *Loader*. A *Máquina* conta com dois elementos internos que abstraem o funcionamento da unidade de processamento e da memória da arquitetura RISC-V, que por sua vez interagem entre si para assegurar o comportamento do ciclo de *fetch-decode-execute*. Além disso, o *Loader* consiste na entidade responsável por ler o programa formatado em ELF, decodificá-lo e escrevê-lo na memória, tal como previsto pela especificação desse formato. (TOOL INTERFACE

STANDARDS COMMITTEE, 1995).

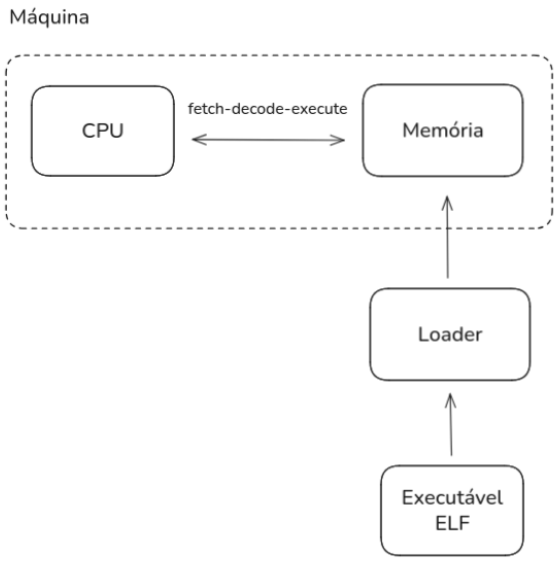

Figura 6 – Panorama geral da arquitetura do emulador

Uma vez carregado em memória, a estrutura do programa pode ser simplificada, para fins de especificação, como um conjunto de duas regiões reservadas em memória, que armazenam o código, os dados e a pilha. A Figura 7 exibe o modelo em questão, onde a seção de código consiste em uma região de leitura e que contém as instruções a serem lidas e executadas sequencialmente. A seção de dados se trata de uma região de leitura e escrita, que armazena variáveis de diferentes tipos e a pilha, que consiste numa região de memória com espaço pré-alocado e que é reservada para operações de *push* e *pop*.

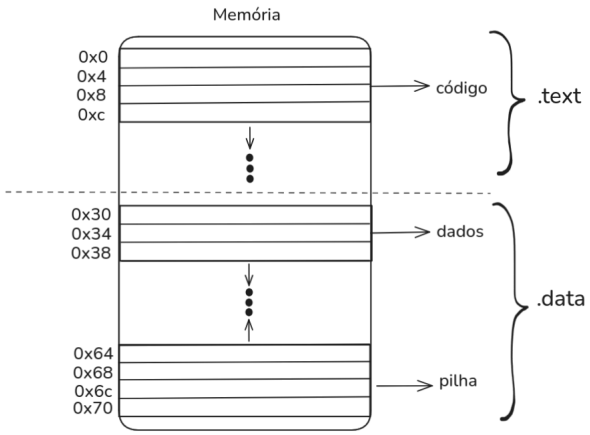

Figura 7 – Modelo de memória utilizada pelo emulador

A partir da configuração do PC para o início da região de memória que contém o código do programa, é possível iniciar o ciclo de instrução caracterizado pela leitura sequencial das instruções, seguida da decodificação do tipo de instrução e a execução

dessas no ambiente. Nesse contexto, um possível modelo que representa o funcionamento desse processo pode ser visto na Figura 8, que esboça um esquema um pouco mais sofisticado para o processo de execução, o qual inclui a antecipação do valor do PC com base na instrução decodificada, mas sem execução paralela ou suporte a pipeline. Essa escolha arquitetural do emulador visa facilitar a interface entre a entidade *Máquina* e o *stub*, uma entidade pertencente ao âmbito da depuração e que se beneficia desse recurso para a definição de *breakpoints*. Para além desse recurso adicional, é importante ressaltar que o ciclo de *fetch-decode-execute* pode incluir *features* que proporcionam performance aprimorada, tais aqueles que preveem suporte ao procedimento de *pipelining*, no entanto essa e outras funcionalidades não foram consideradas a fim de simplificar posteriormente a implementação do emulador.

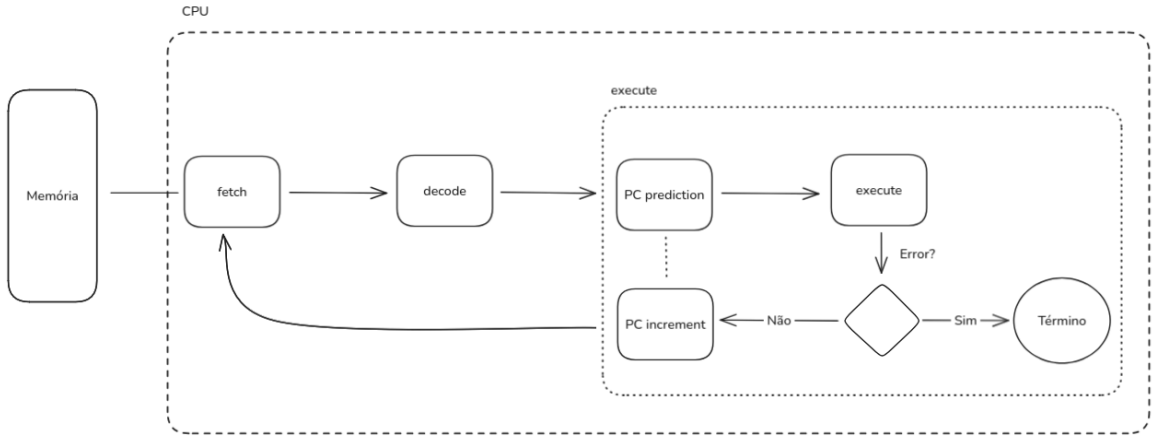

Figura 8 – Modelo do ciclo *fetch-decode-execute* feito pelo emulador

Com a arquitetura de emulação esboçada, espera-se que seja possível emular o funcionamento da arquitetura RISC-V de forma fidedigna, ainda que simplificada. No entanto, é necessário também o planejamento de uma etapa que permita a interação e inspeção do estado interno do ambiente de emulação, de modo a oferecer controle e transparência acerca do funcionamento dos programas em execução. Nesse sentido, a próxima seção busca se debruçar sobre a arquitetura de depuração, que se beneficia do protocolo remoto GDB para sustentar esse objetivo.

3.3 Depuração

O sistema de depuração busca assegurar o controle externo da execução do programa. Assim como explicado na subseção 2.6, a arquitetura do sistema de depuração é baseada no Protocolo Remoto Serial GDB (FREE SOFTWARE FOUNDATION, 2023)[p.311-315], de forma a utilizar um design cliente-servidor no seu planejamento para integrar os seus 3 elementos constituintes: a máquina hospedeira, o *stub* e a máquina alvo.

A Figura 9 exibe a interface entrada-saída entre os 3 elementos em questão, e também exibe a decisão tomada no projeto de integrar o *stub* e o *emulador* no mesmo ambiente de execução, a fim de simplificar a prototipação da solução com a presença de apenas 2 ambientes, o da máquina hospedeira (que executa a sessão de depuração via GDB) e da máquina alvo (que oferece o ambiente de emulação para a arquitetura RISC-V).

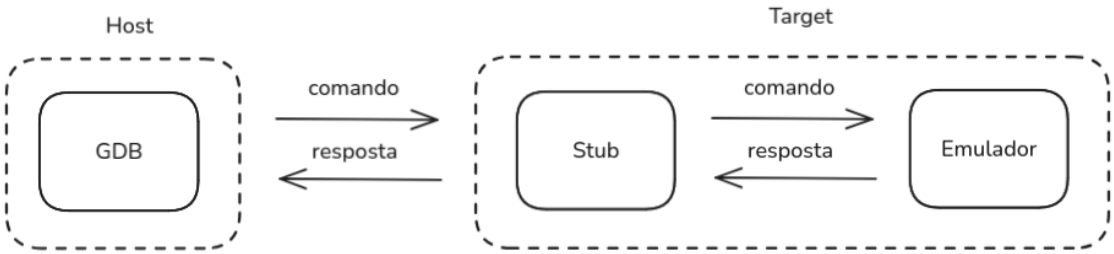

Figura 9 – Panorama geral da arquitetura do depurador

De forma complementar, vale ressaltar o papel da entidade *stub* nesse processo e fundamentar a existência de um intermediário no processo de comunicação serial feita entre a máquina hospedeira e a máquina alvo. Nesse contexto, caberá ao *stub* a tarefa de centralizar o gerenciamento da comunicação em série, a fim de encaminhar comandos completos e significativos entre as partes e poupa-las desse encargo. Caso isso não fosse feito por uma unidade separada, o controle da comunicação entre as máquinas hospedeira e alvo teria de ser feito por elas mesmas, o que implicaria em duplicação de lógica e aumento na complexidade de implementação de ambas as partes. Na prática, a presença de uma unidade específica para o trato da comunicação serial e gerenciamento dos detalhes do protocolo GDB proporciona uma simplificação significativa nas pontas do modelo e possibilita que essas foquem em desempenhar suas funções específicas com excelência.

Em suma, a Figura 10 exibe a arquitetura básica de funcionamento do *stub*, que prevê: a construção e encaminhamento de comandos completos entre as partes a partir do envio sequencial de bytes, gerenciamento do estado interno e repasse da resposta do ambiente de emulação ao ambiente de depuração.

Com a arquitetura de depuração esboçada, espera-se que o projeto em voga ofereça um ambiente controlável e observável, que seja adequado para a escrita de código em linguagem de montagem para RISC-V e análise desse em um ambiente análogo e que emule

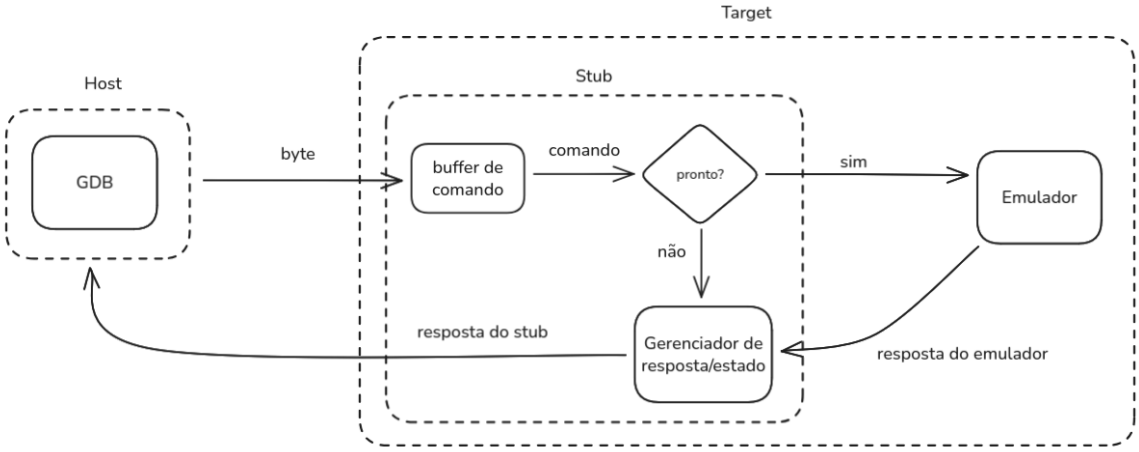

Figura 10 – Modelo de funcionamento do *stub*

a arquitetura. As definições estabelecidas neste capítulo servem, portanto, como base para a implementação descrita nos capítulos seguintes, onde os componentes especificados são concretizados e avaliados experimentalmente.

4 Projeto Arquitetural e Implementação

A implementação do montador, emulador e depurador busca refletir de maneira direta os modelos conceituais vistos na seção 3, de modo a estabelecer uma relação clara entre a especificação e o código. Nesse sentido, o código seguiu um modelo de desenvolvimento modular e orientado a contratos e testes, a fim de permitir que as entidades discutidas fossem implementadas e testadas individualmente, de modo a respeitar a interface entrada-saída discutida. Para tanto, a linguagem Rust se mostrou como uma ferramenta valiosa na modularização e no cumprimento das interfaces planejadas, à medida que oferece um paradigma de programação direcionado a contratos, que permite estabelecer a *API* que as unidades no código deverão implementar e respeitar. Concomitantemente, isso permite que outras entidades possam depender dos contratos em si e não dos agentes que as implementam, o que facilita a integração de implementações diferentes que sigam os mesmos contratos.

A exposição das diferentes ferramentas construídas nesse projeto é feita por meio de uma interface de linha de comando única, que permite interagir com uma das três frentes em questão por vez.

Por fim, todo o desenvolvimento do projeto está disponível no Github (VIEIRA, 2025), que contempla o código fonte, scripts, recursos de documentação e imagens utilizadas para a escrita desse documento.

4.1 Montador

4.1.1 Lexer

A entidade Lexer corresponde a etapa inicial de transformação da linguagem de montagem em formato binário. Nesse contexto, a implementação em código dessa entidade deve receber como entrada o código fonte e devolver como saída as palavras que formam um conjunto do alfabeto da linguagem de montagem RISC-V. Uma vez identificadas as unidades básicas do código, essas serão classificadas de acordo com sua função sintática pela próxima entidade do Montador, o *Tokenizer*.

A listagem 4.1 exibe o contrato em código a ser implementado para a entidade do Lexer. Além de atender ao requisito da interface de entrada-saída discutido anteriormente, esse contrato inclui para cada palavra retornada a sua posição inicial no código fonte. Essa decisão foi tomada a fim de enriquecer posteriormente as sessões de depuração com a informação da linha e coluna das palavras em código.

Listagem 4.1 – Contrato da entidade *Lexer*

```
pub trait Lexer {
    fn get_tokens(&mut self, buffer: &str) -> Vec<(String, Position)> ;
}
```

A implementação do contrato exibido na listagem 4.1 envolveu a escolha de um alfabeto restrito de palavras a serem reconhecidas e a utilização de um mecanismo de scanner incremental, orientado a lookahead mínimo, que percorre o código fonte caractere por caractere, delegando o reconhecimento de palavras a classificadores especializados, preservando posição (linha e coluna) e adiando validações sintáticas para etapas posteriores.

Listagem 4.2 – Implementação do contrato da entidade *Lexer*

```
impl<T: CommonClassifier> Lexer for T {
    fn get_tokens(&mut self, buffer: &str) -> Vec<(String, Position)>
    {
        let mut it = CharStreamReader::new(buffer.chars(), '\n');

        let mut tokens = Vec::new();

        while it.current_token().is_some() {
            let pos = it.current_position().unwrap();

            match self.handle_token(&mut it) {
                Ok(Some(token)) => tokens.push((token, pos)),
                Ok(None) => { }
                Err(err) => panic!("{}", err),
            }
        }

        tokens
    }
}
```

A rotina responsável pelo comportamento descrito no parágrafo anterior está disponível na listagem 4.2. Dessa forma, a linguagem Rust garante que o contrato do Lexer seja implementado para toda e qualquer unidade do código que implemente o contrato *CommonClassifier*, que corresponde ao contrato que define o método *handle_token*, responsável por reconhecer todo o conjunto restrito de palavras pertencentes ao alfabeto da linguagem de montagem especificado para esse projeto. É também possível observar que a implementação faz uso do tipo *CharStreamReader*, que corresponde a implementação de

um *lookahead-1* e que se responsabiliza pelo controle do cursor da entrada. Nesse sentido, a rotina contida no loop consiste em:

1. Verificar se há um caracter não processado na entrada. Se sim, continuar. Caso contrário, terminar o loop.
2. Obter a posição atual do cursor na entrada.
3. Reconhecer a palavra sob o cursor por meio do método *handle_token*.
4. Caso seja reconhecida uma classe de palavra que deve ser salva, então armazena-la junto da sua posição no retorno. Caso a palavra seja ignorável, então seguir para a próxima iteração do loop. Caso algum erro tenha ocorrido no mecanismo de controle da entrada, então interromper o programa com uma exceção.

A implementação feita para a entidade Lexer se comprometeu a reconhecer um conjunto restrito da linguagem de montagem RISC-V, cujas palavras se enquadram em uma de sete classes: *Comment*, *Number*, *String*, *Identifier*, *Unit*, *Ignore* e *Ambiguous*.

Tabela 3 – Classes léxicas e respectivas expressões regulares utilizadas pelo Lexer

Classe	**Descrição**	**Expressão Regular**
Comment	Comentários de código	`//.*'\n'`
Number	Números decimais e hexadecimais	`('+'|'-')?'0x'([0-9a-fA-F])+ | ('+'|'-')?[1-9]([0-9])*`
String	Literais de string	`"([^"] | \")*"`
Identifier	Nome de identificadores quaisquer (variáveis, seções, rótulos)	`('.' | ':' | '_' | [a-zA-Z0-9])*`
Unit	Pontuação	`',' | '(' | ')'`
Ignore	Caracteres a serem ignorados	`'\n' | '\t' | '\b'`
Ambiguous	Caracteres dependentes de contexto	`'+' | '-'`

A tabela 3 exibe as classes de palavras reconhecidas pela implementação feita para o Lexer. As classes em questão foram escolhidas pelo fato de serem frequentes na escrita de programas e por poder serem descritas por meio de expressões regulares.

Como o papel do Lexer consiste em aceitar palavras pertencentes à linguagem sem a necessidade de classificação elaborada e específica (etapa feita pela entidade Tokenizer), escolheu-se simplificar algumas das expressões regulares, a fim de facilitar a implementação do mecanismo de reconhecimento. Esse é o caso da expressão regular associada a

identificadores, que aceitaria palavras como “:::”, “_____” e “0aaa”, as quais ainda que não sejam nomes válidos de identificadores, podem ser invalidadas pela próxima entidade (Tokenizer). Outra escolha de implementação corresponde à classe *Ambiguous*, que foi criada para permitir que o implementador da entidade do Lexer possa consumir a entrada de forma personalizada e que não esteja presa às demais classificações. Esse uso serve bem para a identificação de caracteres com mais de um sentido, tal como os caracteres ‘+’ e ‘-’, que podem ser palavras únicas (como operadores matemáticos) ou parte de um número decimal ou hexadecimal.

O reconhecimento das 7 classes descritas anteriormente é formalizado na forma do contrato *CommonClassifier*, descrito na listagem 1. Esse contrato oferece uma implementação padrão para o método *handle_token*, a qual classifica o caracter atual em uma das 7 categorias e consome a entrada enquanto a expressão regular associada a classe escolhida for válida. Além disso, permite que o código implementador sobreescreva os seus métodos, de modo a controlar a identificação e o consumo da entrada, caso necessário. A implementação completa dessa interface encontra-se disponível no repositório do projeto Vieira (2025).

Do ponto de vista teórico, a implementação do Lexer em questão se assemelha a de um autômato finito determinístico, no qual cada classe léxica representa um estado de reconhecimento e as transições são definidas pelos caracteres consumidos da entrada.

Dessa forma, a entidade Lexer cumpre seu papel de segmentar o código fonte em unidades básicas, de forma a fornecer uma entrada estruturada e enriquecida com informação posicional para a próxima etapa, o *Tokenizer*.

4.1.2 Tokenizer

A entidade Tokenizer é responsável por classificar as palavras provenientes da etapa passada e devolve-las na forma de *tokens* como saída. Para tanto, é necessário inicialmente a definição do que exatamente corresponde um *Token* nesse projeto, para então esclarecer como a implementação busca mapear as palavras lidas pelo Lexer em instâncias desse tipo.

4.1.2.1 O tipo *Token*

A listagem 4.3 exibe a implementação do tipo *Token*, que nesse projeto assume a forma de um *enum* de 15 variantes. Cada variante pode estar associada a valores semânticos, que podem ser tipos ordinários (ex: números, strings) ou contratos, como *Extension*, *Pseudo* e *Directive*. Nesse último caso, que corresponde às variantes *Token::Op*, *Token::Pseudo* e *Token::Directive* (respectivamente), seus tokens podem estar associados a qualquer tipo de dado que implemente o contrato em questão. Isso lança garantias de API e de comportamento para qualquer valor associado a esses tokens, o que permite

que novos tipos sejam adicionados, utilizados e suportados futuramente pelo projeto. A capacidade do código de permitir a extensibilidade dos valores ligados a opcodes, pseudo-instruções e diretivas vai ao encontro do objetivo de permitir o suporte a novas extensões e funcionalidades da arquitetura RISC-V e da linguagem de montagem, que deve acompanhar tais mudanças.

Listagem 4.3 – definição do tipo *Token* utilizado pelo *Tokenizer*

```
pub enum Token {
    Op(Box<dyn Extension>, Position),
    Pseudo(Box<dyn Pseudo>, Position),
    AssemblyDirective(Box<dyn Directive>, Position),
    LinkerDirective(String, Position),
    Reg(Register),
    Name(String, i32),
    Str(String),
    Label(String, Position),
    Number(i32),
    Section(String, Position),
    Plus,
    Minus,
    Lpar,
    Rpar,
    Comma,
}
```

A exemplo da variante *Token::Op*, essa pode estar associada a qualquer tipo de dado que implemente o contrato *Extension*, que corresponde a interface a ser implementada por toda e qualquer extensão da ISA do RISC-V a ser suportada nesse projeto. Essa interface preconiza que todo opcode deve:

1. Declarar qual formato de instrução está associado.
2. Declarar sua sintaxe, ou seja, quantos e quais argumentos espera receber.

A exemplo da extensão RV32I, essa consiste num conjunto de 40 instruções, cuja implementação segue os moldes demonstrados na listagem 4.4.

Listagem 4.4 – Implementação da extensão RV32I

```
pub enum RV32I {
    LUI, ADD, LW, SW, JAL, ECALL, ...
}
```

```
4
5  impl Extension for RV32I {
6      fn get_instruction_format(&self, rs1: u32, rs2: u32, rd: u32, imm:
       i32) -> InstructionFormat  {
7          match self {
8              RV32I::ADD   => InstructionFormat::R { ... },
9              RV32I::LUI   => InstructionFormat::U { ... },
10             RV32I::JAL   => InstructionFormat::J { ... },
11             RV32I::ECALL => InstructionFormat::I { ... },
12             RV32I::LW    => InstructionFormat::I { ... },
13             RV32I::SW    => InstructionFormat::S { ... },
14         }
15     }
16
17     fn get_calling_syntax(&self) -> ArgSyntax {
18         match self {
19             RV32I::ADD   => ArgSyntax::N3(ArgName::RD, ArgName::RS1,
       ArgName::RS2),
20             RV32I::LUI   => ArgSyntax::N2(ArgName::RD, ArgName::IMM),
21             RV32I::JAL   => ArgSyntax::N2(ArgName::RD, ArgName::OFF),
22             RV32I::ECALL => ArgSyntax::N0,
23             RV32I::LW    => ArgSyntax::N3(ArgName::RD, ArgName::OFF,
       ArgName::RS1),
24             RV32I::SW    => ArgSyntax::N3(ArgName::RS2, ArgName::OFF,
       ArgName::RS1),
25         }
26     }
27 }
```

Essa estratégia também permite portar conjuntos parciais de instruções relacionadas a extensões do RISC-V, bem como também suportar extensões personalizadas ou experimentais.

As pseudo-instruções seguem uma linha semelhante a dos opcodes, no sentido que todo tipo associado a seus valores semânticos deve implementar o contrato *Pseudo*, que preconiza que toda pseudo-instrução deve:

1. Ser capaz de ser reescrita enquanto um conjunto de instruções/opcodes e de argumentos.

A exemplo do conjunto de pseudo-instruções implementados atualmente, é possível

observar como isso é feito para a instrução RET na listagem 4.5.

Listagem 4.5 – Implementação da pseudo-instrução RET

```
pub enum PseudoInstruction {
    LI, RET, MV, LA, NOP,
}

impl Pseudo for PseudoInstruction {
    fn translate(&self, args: Vec<ArgValue>) -> Vec<OpcodeLine>  {
        match self {
            ...
            PseudoInstruction::RET => {
                let jalr_line = OpcodeLine {
                    keyword: Box::new(RV32I::JALR),
                    args: vec![
                        ArgValue::Register(Register::ZERO),
                        ArgValue::Register(Register::RA),
                        ArgValue::Number(0),
                    ],
                };
                return vec![jalr_line];
            },
        }

        Vec::new()
    }
}
```

Finalmente, todo tipo a ser associado a uma diretiva deve implementar o contrato *Directive*, que preconiza que toda diretiva deve:

1. Ser capaz de ser reescrita enquanto uma sequência de bytes.

O contrato acima é adequado para diretivas de montagem tais como aquelas que adicionam dados a seção data ou bss, e foi pensada dessa maneira para viabilizar o suporte a tais seções. Outras diretivas tais como aquelas associadas a ligação e/ou a compilação não necessariamente adicionam dados e portanto apenas retornam uma sequência vazia de bytes.

4.1.2.2 Mapeamento palavra-token

A listagem 4.6 exibe o contrato de implementação da entidade Tokenizer. O tipo *Token* deve implementar a interface *ToGenericToken*, que garante o mapeamento de qualquer token para o tipo *GenericToken*. Na prática, o tipo *GenericToken* prevê que todo *token* deve ser identificado ou como um token-chave (*KeyToken*) ou como um token-argumento (*ArgToken*). Um token-chave corresponde a um token que marca o início de uma nova instrução em linguagem de montagem, como a declaração de uma nova seção do programa, a declaração de um label, uma nova instrução *assembly*, uma diretiva de montagem ou ligação e entre outros. De forma complementar, tokens-argumento sucedem um token-chave e são utilizados como operandos, sendo geralmente números, literais, identificadores e outros.

Listagem 4.6 – Contrato da entidade *Tokenizer*

```
pub enum GenericToken {
    KeyToken(KeyValue, Position),
    ArgToken(ArgValue),
}

pub trait ToGenericToken {
    fn to_generic_token(self) -> Option<GenericToken>;
}

pub trait Tokenizer {
    type Token: ToGenericToken;
    fn parse(&self, tokens: Vec<(String, Position)>) -> Vec<<Self as
    Tokenizer>::Token> ;
}

impl ToGenericToken for Token {
    fn to_generic_token(self) -> Option<GenericToken> {
        match self {
            ...
            Token::Op(extension, pos)   => Some(GenericToken::KeyToken(
    KeyValue::Op(extension), pos)),
            Token::Pseudo(pseudo, pos)  => Some(GenericToken::KeyToken(
    KeyValue::Pseudo(pseudo), pos)),
            Token::Label(label, pos)    => Some(GenericToken::KeyToken(
    KeyValue::Label(label), pos)),
```

```
22              Token::Name(name, off)      => Some(GenericToken::ArgToken(
    ArgValue::Use(name, off))),
23              Token::Number(n)            => Some(GenericToken::ArgToken(
    ArgValue::Number(n))),
24              ...
25          }
26      }
27  }
```

A partir da definição dos tipos de token da subseção 4.1.2.1, o processamento da entrada envolve o mapeamento sequencial de cada palavra a um desses tipos, o que é feito por meio do código disponível na listagem 4.7. Nesse contexto, a linguagem Rust garante que o contrato do Tokenizer será implementado para toda e qualquer unidade do código que implemente a interface *TokenClassifier*, que por sua vez executa o mapeamento por meio do método *handle_token*. O código da listagem 4.7 também conta com a utilização do tipo *PositionedStringStreamReader*, o qual implementa um mecanismo *lookahead-1* e garante ao Tokenizer controle sobre a entrada, o que o permite inspeciona-la e percorre-la sequencialmente. Nesse método, o fluxo em loop do processamento de tokens consiste em:

1. Verificar se há uma palavra não processado na entrada. Se sim, continuar. Caso contrário, terminar o loop.
2. Classificar a palavra sob o cursor por meio do método *handle_token*.
3. Caso a palavra seja mapeável para algum tipo de token, então armazena-la no retorno. Caso contrário, então seguir para a próxima palavra.

Listagem 4.7 – Implementação do contrato da entidade *Tokenizer*

```
1  impl<T: ToGenericToken, C: TokenClassifier<Token = T>> Tokenizer for C
       {
2      type Token = T;
3
4      fn parse(&self, tokens: Vec<(String, Position)>) -> Vec<<Self as
       Tokenizer>::Token>  {
5          let mut it = PositionedStringStreamReader::new(tokens.into_iter
       (), (String::from("\n"), Position::new(0, 0, 0)));
6
7          let mut tokens = Vec::new();
8
9          while let Some(_) = it.current_token() {
```

```
            if let Some(lex) = self.handle_token(&mut it) {
                tokens.push(lex);
            }
            it.advance();
        }

        tokens
    }
}
```

Um ponto crucial dessa rotina corresponde a implementação do contrato *Token-Classifier*, que permite personalizar quais palavras serão mapeadas a um tipo de token. A listagem 2 exibe a API definida por esse elemento. Para a implementação vigente, a personalização dos métodos acima permite que o Tokenizer classifique as palavras da entrada como registradores, opcodes, pseudo-instruções e/ou outros recursos da linguagem de montagem. Para além disso, o Tokenizer também é capaz de associar valores semânticos a esses tokens, como aqueles discutidos na subseção 4.1.2.1.

Sob essa luz, pode-se entender o papel do Tokenizer como o de relacionar termos puramente simbólicos (palavras pertencentes a linguagem de montagem) a funcionalidades implementadas em código (como as extensões e pseudo-instruções).

Dessa forma, o Tokenizer atua como uma camada intermediária entre a análise léxica e as etapas sintáticas do montador, convertendo palavras em tokens semanticamente enriquecidos, porém ainda independentes da estrutura global do programa. Assim, o *Parser* será responsável por condensar a sequência de tokens em uma saída mais organizada e já pré-processada para a geração do código a ser feita pelo *Núcleo do Montador*.

4.1.3 Parser

A listagem 4.8 exibe o contrato de implementação da entidade Parser. A interface conta como entrada uma sequência de tokens, que serão processados e devolvidos na forma de alguma saída que seja coerente com o que a entidade do *Núcleo do Montador* espera. Nesse sentido, uma vantagem da utilização de um contrato genérico é a de permitir que diferentes estratégias de parsing possam ser implementadas sem acoplamento direto ao tipo concreto de token ou a saída, o que garante flexibilidade e adaptabilidade a unidade do código que a define.

Listagem 4.8 – Contrato da entidade *Parser*

```
pub trait Parser {
    type Token;
    type Output;
```

```
4
5       fn parse(&self, token: Vec<Self::Token>) -> Self::Output ;
6   }
```

A implementação do contrato do Parser segue descrita na listagem 4.9, a qual define como saída um vetor do tipo *GenericBlock*. A definição do tipo retornado pode ser vista na listagem 4.10, que é utilizada para estruturar um grupo de tokens enquanto uma sequência de linhas/comandos associadas a uma seção de código (texto, dados, bss ou metadados).

Listagem 4.9 – Contrato da entidade *Parser*

```
1  impl parser::Parser for Parser {
2      type Token = Token;
3      type Output = Vec<GenericBlock>;
4
5      fn parse(&self, tokens: Vec<Self::Token>) -> Self::Output {
6          let lines  = parser::tokens_to_lines(tokens);
7          let blocks = parser::lines_to_blocks(lines);
8          blocks
9      }
10 }
```

Listagem 4.10 – Contrato da entidade *Parser*

```
1  pub struct GenericBlock {
2      pub(crate) name: SectionName,
3      pub(crate) lines: Vec<GenericLine>
4  }
5
6  pub enum SectionName {
7      Metadata,
8      Text,
9      Data,
10     Bss,
11     Custom(String)
12 }
13
14 pub struct GenericLine {
15     pub(crate) id: usize,
16     pub(crate) file_pos: Position,
17     pub(crate) keyword: KeyValue,
```

```
18     pub(crate) args: Vec<ArgValue>
19 }
```

Acerca da implementação exibida na listagem 4.9, é possível perceber que o método *parse* consiste em um procedimento de 2 etapas:

1. agrupar tokens em linhas de código (através da função *parser::token_to_lines*).
2. agrupar linhas em blocos de código (através da função *parser::lines_to_blocks*).

A definição de ambas etapas está visível na Figura 4.11, e podem ser entendidas como pipelines de transformação que refinam suas entradas até o resultado final. Mais especificamente, a etapa inicial corresponde ao processo de agrupamento de tokens em linhas, o qual é feito por meio de 4 passos:

1. transformação dos tokens na sua representação genérica
2. formação de grupos por meio do agrupamento de cada token-chave e os tokens-argumento próximos
3. reescrita das pseudo-instruções em instruções básicas (opcodes)
4. expansão das diretivas de montagem (inserção de dados)

Listagem 4.11 – Implementação do contrato da entidade *Tokenizer*

```
1  pub fn tokens_to_lines<T: ToGenericToken>(tokens: Vec<T>) -> Vec<
       GenericLine> {
2      let tokens = generalize_tokens(tokens);
3      let groups = group_tokens(tokens);
4      let lines  = expand_pseudos(groups);
5      let lines  = expand_assembly_directives(lines);
6      lines
7  }
8
9  pub fn lines_to_blocks(lines: Vec<GenericLine>) -> Vec<GenericBlock> {
10     let blocks = group_lines(lines);
11     let blocks = merge_blocks(blocks);
12     blocks
13 }
```

A transformação dos tokens em sua representação genérica é possível graças à restrição imposta na implementação da entidade Tokenizer quanto ao tipo *Token*, o qual

deve implementar o contrato *ToGenericToken*. Essa conversão de tipos é fundamental, pois permite que quaisquer detalhes provenientes da etapa anterior sejam filtrados e abstraídos, de modo a simplificar as transformações subsequentes feitas no Parser.

A etapa seguinte, o agrupamento de tokens, é baseada na identificação de tokens-chave, que representam instruções, pseudo-instruções, diretivas, declarações de label e/ou de seção. A partir de um token-chave identificado, todos demais tokens até o próximo token-chave são identificados como argumentos da mesma linha, o que permite agrupar comandos/linhas do programa.

Uma vez que o programa passa a estar estruturado em comandos, a próxima etapa consiste na expansão das pseudo-instruções, que servem de abstrações fornecidas ao programador para operações que não possuem equivalência direta no código de montagem. Novamente, o procedimento da expansão de pseudo-instruções é assegurado pelo contrato *Pseudo*, que deve ser implementado a todos os valores que estão atrelados a símbolos referentes a pseudo-instruções no código.

A última etapa é caraterizada pela expansão das diretivas de montagem, que respondem por aquelas que inserem bytes arbitrários nas seções de dados e de bss. Na prática, essa funcionalidade permite que variáveis sejam criadas e populem regiões válidas do programa, de modo a possibilitar a escrita de dados de forma alinhada em múltiplos de 4 bytes, em conformidade com o modelo de memória da arquitetura RISC-V.

Ao final do método *parser::tokens_to_lines*, os tokens estão agrupados em comandos/linhas e expandidos. Portanto, a próxima etapa consiste no processo de agrupamento das linhas em blocos de código, o qual é feito por meio 2 passos:

1. agrupamento de todas as linhas que sigam a definição de uma seção de código
2. união de seções que possuam o mesmo nome

De modo a refletir a estrutura típica do formato ELF, que é composto por seções de código, a primeira subtarefa feita consiste no agrupamento das linhas do programa de acordo com a seção do programa que foram declaradas. Caso as linhas precedam a declaração de qualquer seção do programa (texto, data ou bss), então essas são colocadas por default em uma seção de texto. Essa decisão, por sua vez, pode gerar comportamentos inesperados caso o programador inadvertidamente inicie a escrita do código com a declaração de variáveis fora da seção de dados.

Ao final do passo anterior, as linhas encontram-se agrupadas em seções de código. Nesse ponto, seções idênticas porém declaradas em diferentes partes do programa são mescladas em uma mesma seção, o que elimina a multiplicidade de seções ao longo do código e garante uma estrutura final consistente. Uma limitação dessa etapa é que apenas 4 tipos de seção são atualmente suportadas, sendo essas as seções de texto, dados, bss e

de metadados. Essa característica não é mandatória para o propósito da montagem, mas possibilitou simplificar a estrutura final dos executáveis ELF gerados, que são escritos sempre com uma mesma ordem dessas seções, que é formada pela seção de texto, seguida pela de dados e pela de bss ao final.

A abordagem baseada em múltiplas passagens e representações intermediárias contribui para a clareza do código, facilita a depuração e permite a extensão incremental do montador sem a necessidade de reescrever etapas já consolidadas.

4.1.4 Núcleo do Montador

O Núcleo do Montador consiste na etapa final da fase de montagem, antecedendo processos como ligação e carregamento. A entrada fornecida a essa entidade corresponde ao resultado entregue pelo Parser, que nesse projeto corresponde a instruções típicas de linguagem de montagem, como declaração de labels e sequência de comandos, já estruturadas e organizadas em seções. Nesse contexto, a informação ali presente encontra-se parcialmente processada, uma vez que pseudo instruções e diretivas de montagem já foram expandidas. No entanto, os dados de entrada carecem de detalhes importantes para a execução do programa, como o endereçamento das seções e instruções em memória, resolução de símbolos, codificação das instruções e entre outros. Nesse sentido, cabe ao Núcleo do Montador resolver tais pendências de mais baixo nível, de modo a aproximar os dados do formato final destinado.

A listagem 4.12 contém o contrato a ser implementado por unidades no código que se propõem a cumprir com a função esperada para a entidade do Núcleo do Montador. O contrato em questão considera uma entrada genérica, em vista da flexibilidade suportada pelo contrato da entidade do Parser de poder fornecer diferentes tipos como saída. Paralelamente, o resultado desse contrato consiste no tipo *AssemblerTools*, que se trata de um tipo cuja estrutura reflete a organização do formato ELF, com a presença de campos para o armazenamento de tabelas variadas, metadados e dados em binário. A definição do tipo *AssemblerTools* se encontra na listagem 4.13. Ainda que a saída dessa etapa conforme com a estrutura típicamente utilizada para a escrita de objetos no formato ELF, o tipo *AssemblerTools* oferece campos que podem ser aproveitados de outras maneiras, como para o carregamento de programas diretamente na memória do emulador e até a exportação para outros formatos de arquivo que sejam compatíveis.

Listagem 4.12 – Contrato da entidade *Assembler*

```
1 pub trait Assembler {
2     type Input;
3     fn assemble(&self, instructions: Self::Input) -> AssemblerTools ;
4 }
```

Listagem 4.13 – Definição da saída gerada pelo Núcleo do Montador

```rust
pub struct AssemblerTools {
    pub(crate) metadata: Option<GenericBlock>,
    pub(crate) sections: HashMap<String, Section>,
    pub(crate) symbols:  HashMap<String, Symbol>,
    pub(crate) strings:  Vec<String>,
    pub(crate) relocations:  HashMap<String, Vec<RelocationEntry>>,
    pub(crate) blocks: Vec<PositionedEncodedBlock>,
}
```

Em vista da definição da saída exibida na listagem 4.13, a implementação vigente do Núcleo do Montador busca:

1. Extrair a seção de metadados para que possa ser posteriormente aproveitada por ferramentas pós montagem (como ligadores)
2. Endereçar as seções e instruções do código
3. Resolver símbolos como labels e nomes de variável para formatos numéricos
4. Codificar instruções em binário, tal como previsto pela ISA

Para satisfazer os pontos acima citados, a implementação segue a estrutura de código definida na listagem 4.14, que envolve uma sequência de procedimentos para resolver as questões levantadas. Nesse sentido, a estratégia dessa rotina segue uma cadência em pipeline tal como aquela elaborada para a entidade Parser, o que beneficia a modularização dessa seção do código e facilita a identificação de problemas. A seguir, o processo de cada uma das etapas enumeradas anteriormente serão descritas com mais detalhes e na ordem como ocorrem na implementação em questão.

Listagem 4.14 – Implementação do contrato do Núcleo do Montador

```rust
impl assembler::Assembler for Assembler {
    type Input = Vec<GenericBlock>;

    fn assemble(&self, instruction: Self::Input) -> AssemblerTools {
        assembler::assemble(instruction)
    }
}

pub fn assemble(mut blocks: Vec<GenericBlock>) -> AssemblerTools {
    let metadata = extract_metadata(&mut blocks);
```

```
11
12      let blocks = cast_generic_to_positioned_blocks(blocks);
13      let blocks = gen_section_address(blocks, 0, 4);
14      let blocks = gen_line_address(blocks);
15      let blocks = gen_root_line_address(blocks);
16
17      let sections = gen_section_table(&blocks);
18      let symbols  = gen_symbol_table(&blocks);
19      let strings  = gen_string_table(&blocks);
20      let relocations = gen_relocation_table(&blocks, &symbols, &sections
        );
21
22      let blocks = resolve_symbols(blocks, &symbols, &sections);
23      let blocks = args_to_numbers(blocks);
24      let blocks = encode_blocks(blocks);
25
26      AssemblerTools {
27          metadata,
28          sections,
29          symbols,
30          strings,
31          relocations,
32          blocks,
33      }
34  }
```

4.1.4.1 Extração de metadados

A extração de metadados consiste primariamente na extração da seção de metadados que é propositalmente preservada pela entidade Parser. Essa seção permite que o programador declare diretivas que controlem as ferramentas envoltas na geração do executável, o que inclui o próprio montador como ferramentas que sucedem o processo de montagem, como ligadores. Nesse sentido, em outras ferramentas de compilação, seções do programa como essa encontram aplicações como a definição do ponto de entrada do programa, controle da visibilidade de símbolos e outros.

No contexto do montador feito nesse projeto, a seção de metadados foi planejada primariamente para possibilitar a definição do ponto de entrada do programa utilizando labels arbitrários, no entanto essa funcionalidade foi despriorizada ao longo do desenvolvimento. Em vez da flexibilidade de labels arbitrários, a definição do ponto de entrada

do programa é definida pelo posicionamento do label *_start*, que deve ser explicitamente declarado no programa como garantia para que o programa comece a execução no ponto definido e esperado pelo programador.

Por fim, essa seção é isolada das demais durante as etapas iniciais do pipeline e armazenada separadamente na estrutura de saída, não participando diretamente do endereçamento nem da codificação das instruções. Essa separação permite que as informações declaradas possam ser posteriormente reutilizadas ou estendidas sem impactar as demais etapas do processo de montagem.

4.1.4.2 Endereçamento de seções e instruções

O procedimento de endereçamento inicia-se com a conversão do tipo *GenericBlock* (proveniente da entidade Parser) para o tipo *PositionedGenericBlock*, o qual permite o endereçamento das seções e instruções do programa. A partir desse preparativo, a atribuição de endereços é feita inicialmente com as seções e depois das suas respectivas instruções.

O endereçamento das seções é baseado na ordem já pré-definida pela entidade Parser das seções constituintes do programa, que correspondem às seções de texto, de dados e bss (respectivamente). O posicionamento fixo das seções facilita o entendimento do arranjo do programa em memória e garante previsibilidade sobre o endereçamento final das seções. Além disso, essa etapa busca assegurar que o endereço das seções esteja alinhado em 4 bytes, a fim de simplificar a leitura de regiões de memória com instruções, e também visa separar as seções com um offset fixo de no mínimo 4 bytes. Seções que possuem tamanhos não divisíveis por 4 são complementados com 0's até o próximo múltiplo, o que assegura o modelo de memória desejado. Nesse sentido, o endereço 0 é garantidamente atribuído a seção de texto, seguido de endereços múltiplos de 4 para as demais seções do código.

A definição do endereços das regiões do programa é sucedida pela atribuição de endereço a suas instruções, cujo valor é relativo ao início da seção a que pertencem. Desse modo, a primeira instrução de toda seção possui o endereço zero, enquanto as demais possuem endereços múltiplos de 4. Essa restrição de alinhamento se aplica aos dados de qualquer seção do código, seja texto, dados ou bss.

Desse modo, o endereço base de uma próxima instrução é gerado a partir do endereço anterior somado ao tamanho da instrução anterior e aos bytes de preenchimento (caso necessário para garantir alinhamento em um múltiplo de 4). Além do endereço base gerado, todas as linhas também são associadas a um segundo endereço, denominado endereço raiz, que é o mesmo para instruções que foram geradas a partir de outras instruções. O endereço raiz é particularmente importante posteriormente na criação das relocações, no contexto da criação de variáveis na seção de dados e referenciamento dessas na seção de texto.

Dessa forma, ao final dessa etapa, todas as seções e instruções do programa encontram-se associadas a endereços bem definidos e alinhados, permitindo que as etapas subsequentes operem sobre uma representação consistente do *layout* de memória do programa.

4.1.4.3 Preenchimento de tabelas do programa

O formato ELF é caracterizado pela presença de tabelas que resumem aspectos diversos de um programa de baixo nível, como as seções, símbolos e strings que o compõem. A informação armazenada nesses tabelas pode então ser processada por outras ferramentas para auxiliar no seu entendimento, como programas de inspeção de binário, ligadores, carregadores e outros. Nesse projeto, as tabelas montadas correspondem a de seções, símbolos e relocações.

A tabela de seções mapeia o nome simbólico de cada seção que compoe o programa a uma estrutura que descreve informações variadas. Como o projeto conta apenas com as seções de texto, dados e bss, essas correspondem às seções descritas pela tabela, que disponibiliza o endereço de início de cada uma. Concomitantemente, a listagem 4.15 exibe o tipo implementado que armazena os dados de uma seção do código. A partir dessa informação, o formato ELF do executável final poderá ser inspecionado por utilitários de terminal como *objdump* e *readelf* para averiguar o endereço final em memória assumido por essas seções.

Listagem 4.15 – Tipo utilizado pela tabela de seções para armazenar os dados das seções do código

```
1 pub struct Section {
2     pub(crate) address: usize,
3     pub(crate) name: SectionName,
4 }
```

A tabela de símbolos mapeia o nome simbólico de todos os labels declarados no código fonte, de modo a armazenar o nome da seção e o endereço de declaração, o tamanho em bytes do bloco que referencia e o escopo de sua visibilidade. A listagem 4.16 exibe o tipo implementado para armazenar os dados referentes a um símbolo no código. Nesse contexto, cabe esclarecer que labels declarados na seção de dados correspondem a variáveis que podem ser utilizadas na seção de texto, e por esse motivo a tabela de símbolos armazena o tamanho em bytes ligado a esses labels. Além disso, a visibilidade dos labels é também mantida, para que seja utilizada posteriormente pelo ligador.

Listagem 4.16 – Tipo utilizado pela tabela de símbolos para armazenar os dados das variáveis e labels utilizadas no código

```rust
pub struct Symbol {
    pub(crate) section: SectionName,
    pub(crate) relative_address: usize,
    pub(crate) length: usize,
    pub(crate) scope: String,
}
```

A tabela de relocações corresponde a estrutura responsável por permitir que a ferramenta de ligação atue no reendereçamento das variáveis e labels utilizados ao longo da seção de texto. O procedimento de relocagem é importante, pois permite que o executável final (pós procedimento de ligação) possa ser carregado em ambientes que reservem diferentes regiões de memória para a execução do programa. Nesse sentido, esse processo garante que as referências às labels e variáveis feitas no código se mantenham válidas, mesmo após eventuais modificações feitas pelo ligador no endereçamento final das seções de código e suas instruções. A listagem 4.17 exibe o tipo implementado para armazenar as informações referentes a relocação de algum item no código. Esse tipo conta com um id incremental, gerado automaticamente pelo programa, para diferenciar as relocações, o que é um requisito para a escrita do executável ELF final. Além disso, o tipo em questão conta com o endereço de uso da variável ou label em questão, bem como o campo *addend*, que permite a utilização de *offsets* juntamente das referências em questão.

Listagem 4.17 – Tipo utilizado pela tabela de relocações para armazenar os dados das relocações do código

```rust
pub struct RelocationEntry {
    pub(crate) id: usize,
    pub(crate) address: usize,
    pub(crate) addend: i32,
}
```

4.1.4.4 Resolução de símbolos

A resolução de símbolos corresponde a etapa que transforma referências simbólicas no código, feitas a variáveis e ou labels, em deslocamentos, ou *offsets*. Nesse projeto, os deslocamentos correspondem a diferença entre o endereço absoluto da instrução que referencia o símbolo e o endereço absoluto onde aquele símbolo está declarado no programa. O endereço absoluto de um label/variável é calculado nesse projeto por meio da fórmula 4.1:

$$A_{\text{abs}} = A_{\text{sec}} + A_{\text{inst}} + \text{addend} \tag{4.1}$$

Onde:

- A_{abs} é o endereço absoluto da instrução (pré-ligação);
- A_{sec} é o endereço base da seção;
- A_{inst} é o endereço relativo da instrução dentro da seção;
- *addend* é um deslocamento opcional associado à instrução.

Para tanto, o valor dos endereços utilizados na fórmula são obtidos das tabelas de seções e de símbolos, que foram construídas no passo anterior.

4.1.4.5 Codificação das instruções em binário

A codificação das instruções em binário é crucial na consolidação do programa enquanto um binário válido e de acordo com a especificação da ISA do RISC-V. Os dados de entrada dessa etapa consistem nas seções e instruções do programa já endereçadas, sem pseudo instruções e sem referências simbólicas a labels e variáveis, as quais serão submetidas a um procedimento composto por duas etapas: a normalização de tipos para inteiros e a codificação das instruções segundo a ISA.

A primeira etapa é responsável por garantir que todos os argumentos associados a instruções sejam do tipo inteiro *i32*, que em Rust armazena inteiros com sinal de 32 bits. A etapa em questão é feita pela rotina *args_to_numbers* e assegura que todo e qualquer argumento (como registradores, literais, endereços e outros) estejam contidos em um tipo único de inteiro, para a aplicação da etapa seguinte.

A segunda etapa, feita pela rotina *encode_blocks*, é responsável pela codificação em si das instruções. Assim como visto na seção 4.1.2.1, toda instrução possui um token-chave associado, que por sua vez está associado a um valor que implementa o contrato *Extension*. Nesse sentido, o método *instruction_to_binary* permite transformar qualquer instrução cujo token-chave implemente o contrato *Extension* em um inteiro de 32 bits. O método supracitado pode ser visto na listagem 4.18, cujo funcionamento consiste em:

- Obter o número e tipo de parâmetros esperados pela instrução
- Mapear os argumentos passados de acordo com os parâmetros que são esperados pela instrução
- Obter o formato de instrução da instrução
- Codificar os argumentos passados de acordo com a regra que rege o formato de instrução em jogo

Listagem 4.18 – Função responsável por converter uma instrução associada a seus argumentos em um inteiro de 32 bits

```
pub fn instruction_to_binary(inst: &Box<dyn Extension>, args: &Vec<i32>) -> u32 {
    let fields = match inst.get_calling_syntax() {
        ArgSyntax::N0 => vec![],
        ArgSyntax::N1(f0) => vec![f0],
        ArgSyntax::N2(f0, f1) => vec![f0, f1],
        ArgSyntax::N3(f0, f1, f2) => vec![f0, f1, f2],
        ArgSyntax::N4(f0, f1, f2, f3) => vec![f0, f1, f2, f3],
    };
    let (rs1, rs2, rd, imm) = get_args(fields, args);
    let iformat = inst.get_instruction_format(rs1, rs2, rd, imm);
    iformat.encode()
}

fn get_args(
    fields: Vec<ArgName>,
    args: &Vec<i32>
) -> (u32, u32, u32, i32)
{
    if fields.len() != args.len() {
        println!("{:?}", args);
        println!("{:?}", fields);
        panic!("Insuficient number of arguments: {} != {}", fields.len(), args.len());
    }

    let mut rs1: u32 = 0;
    let mut rs2: u32 = 0;
    let mut rd:  u32 = 0;
    let mut imm: i32 = 0;
    for (field, arg) in fields.iter().zip(args.iter()) {
        match field {
            ArgName::RS1 => rs1 = (*arg) as u32,
            ArgName::RS2 => rs2 = (*arg) as u32,
            ArgName::RD =>  rd  = (*arg) as u32,
            ArgName::IMM | ArgName::OFF => imm = *arg,
        }
```

```
36      }
37      (rs1, rs2, rd, imm)
38  }
```

Desse modo, em um processo iterativo para cada instrução que compõe a seção de texto, todas as instruções são convertidas em palavras de 32 bits, o que efetivamente conclui a codificação das instruções tal como previsto pela ISA do RISC-V. Com isso, a implementação em questão gera a saída final da entidade do *Núcleo do montador*, que engloba as tabelas previamente definidas, bem como as seções codificadas em formato binário que compõem o programa. A partir disso, a saída dessa entidade pode ser encaminhada para diferentes usos, como por exemplo a escrita de um arquivo em formato ELF.

4.1.4.6 Escrita do arquivo final no formato ELF

Ao final da sua rotina, a entidade do *Núcleo do montador* foi responsável pela codificação binária do código fonte e por estruturar uma saída capaz de descrever diferentes aspectos que caracterizam o programa vigente, como suas seções e símbolos, bem como também definir o endereçamento temporário (pré-ligação) das seções e instruções que o formam. Nesse aspecto, a escrita do arquivo no formato ELF é facilitada pelo fato da própria estrutura da saída condizer com a lógica utilizada por esse formato na representação de um programa.

A rotina completa responsável por conduzir o processo de montagem do código fonte e a escrita no formato ELF pode ser vista na listagem 4.19. Nesse contexto, o projeto utiliza a *crate*/biblioteca *object* para a escrita do arquivo objeto em formato ELF, a qual fornece uma API amigável para a escrita de arquivos em diferentes formatos para diversas arquiteturas, o que é feito na função *write_from_tools*. O código contido na rotina *write_from_tools* realiza sequencialmente o seguinte conjunto de etapas:

- Definição do ponto de entrada do programa para o endereço referente ao label *_start*
- Escrita de todos os símbolos identificados na tabela de símbolos
- Escrita das 3 seções suportadas (de texto, de dados e bss)
- Escrita das relações segundo a tabela de relocações

Por fim, as informações de depuração são adicionadas ao arquivo antes da sua escrita final por meio do uso da biblioteca *gimli*, que oferece uma API que abstrai o formato DWARF aplicado sobre o formato ELF. Uma vez terminada a etapa de montagem, o arquivo final pode então ser encaminhado a etapa de ligação, para que o executável seja disponibilizado com êxito.

Listagem 4.19 – Rotina completa de montagem do código de montagem à escrita do arquivo no formato ELF

```
pub fn encode_to_elf_with_debug(
    code: &str,
    input_file: &str,
    output_file: &str,
) -> elfwriter::Result<()> {
    let mut lexer = syntax::gas::Lexer;
    let tokenizer = syntax::gas::Tokenizer;
    let parser = syntax::gas::Parser;
    let assembler = syntax::gas::Assembler;

    let lexemes = lexer.get_tokens(code);
    let tokens = tokenizer.parse(lexemes);
    let blocks = parser.parse(tokens);
    let tools = assembler.assemble(blocks);
    let (mut writer, tools) = write_from_tools(output);
    add_debug_information(&mut writer, tools, input_file.as_bytes());

    writer.save(output_file)
}
```

4.1.5 Ligador

A ligação corresponde a última etapa na escrita do arquivo final, sendo responsável por transformar o único arquivo objeto gerado pelo montador em um executável que possa ser carregado em uma região válida de memória da máquina/montador. A implementação de um ligador próprio está fora do escopo deste trabalho, portanto a etapa de ligação foi delegada ao utilitário de linha de comando *ld*, que corresponde a uma ferramenta vastamente utilizada em ambientes *linux* e que foi compilada manualmente nesse trabalho para que gerasse executáveis válidos para a arquitetura RISC-V 32 bits. A decisão pelo uso de uma ferramenta terceira se deu para agilizar o desenvolvimento do montador e para cumprir com compromisso de suportar o uso de programas além daqueles que foram desenvolvidos especificamente para esse projeto.

4.1.6 Compilação do programa *hello-world.s*

A rotina completa de geração de um executável RISC-V 32 bits nesse projeto pode ser vista na Figura 11, a qual demonstra a compilação de um programa que exibe a

mensagem *Hello world!*. O código fonte desse programa pode ser visto na listagem 4.20.

Listagem 4.20 – Programa simples de um *Hello world* escrito em linguagem de montagem para a arquitetura RISC-V

```
1   .globl _start
2
3   .section .data
4 msg:   .ascii "Hello world!\n" // 13 bytes including newline
5
6   .section .text
7 _start:
8   // write(stdout=1, msg, len)
9   li a0, 1              // fd = 1 (stdout)
10  la a1, msg            // buffer address
11  li a2, 13             // length
12  li a7, 64             // syscall: write
13  ecall
14 exit:
15  // exit(0)
16  li a0, 0              // status
17  li a7, 93             // syscall: exit
18  ecall
```

```
[16:19:10] carlos@carlosarch /home/carlos/repos/rustv
> ls
alt  doc  examples  src  target  auto.sh  Cargo.lock  Cargo.toml  init.gdb  README.md
[16:19:11] carlos@carlosarch /home/carlos/repos/rustv
> ./target/release/rustv --assemble ./examples/hello-world.s
[16:19:15] carlos@carlosarch /home/carlos/repos/rustv
> ls
alt  doc  examples  src  target  auto.sh  Cargo.lock  Cargo.toml  init.gdb  main.o  README.md
[16:19:16] carlos@carlosarch /home/carlos/repos/rustv
> riscv32-unknown-linux-gnu-ld main.o -o main
[16:19:20] carlos@carlosarch /home/carlos/repos/rustv
> ls
alt  doc  examples  src  target  auto.sh  Cargo.lock  Cargo.toml  init.gdb  main  main.o  README.md
[16:19:22] carlos@carlosarch /home/carlos/repos/rustv
>
```

Figura 11 – Compilação do programa 'hello-world.s' utilizando o modo montador feito para esse projeto e o ligador *ld*

O executável final *main* pode ser inspecionado por meio de ferramentas de linha de comando compatíveis com a arquitetura RISC-V 32 bits, a fim de averiguar se a codificação das instruções está correta com o código original. Para tanto, o projeto contou com a compilação manual das ferramentas *objdump* e *readelf* para a arquitetura RISC-V 32 bits, por meio das quais é possível inspecionar o arquivo ELF. A Figura 12 contém a saída associada a execução utilitário *readelf*, que permite a inspeção das seções, símbolos e

```
> riscv32-unknown-linux-gnu-readelf -s -h -S main
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x10074
  Start of program headers:          52 (bytes into file)
  Start of section headers:          512 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         6
  Section header string table index: 5

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00010074 000074 000024 00  AX  0   0  4
  [ 2] .data             PROGBITS        00011098 000098 000010 00  WA  0   0  1
  [ 3] .symtab           SYMTAB          00000000 0000a8 0000d0 10      4   3  4
  [ 4] .strtab           STRTAB          00000000 000178 00005f 00      0   0  1
  [ 5] .shstrtab         STRTAB          00000000 0001d7 000027 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), p (processor specific)

Symbol table '.symtab' contains 13 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00010074     0 SECTION LOCAL  DEFAULT    1 .text
     2: 00011098     0 SECTION LOCAL  DEFAULT    2 .data
     3: 00011898     0 NOTYPE  GLOBAL DEFAULT  ABS __global_pointer$
     4: 000110a8     0 NOTYPE  GLOBAL DEFAULT    2 __SDATA_BEGIN__
     5: 00011098     0 OBJECT  GLOBAL HIDDEN     2 msg
     6: 00010074     0 FUNC    GLOBAL DEFAULT    1 _start
     7: 000110a8     0 NOTYPE  GLOBAL DEFAULT    2 __BSS_END__
     8: 000110a8     0 NOTYPE  GLOBAL DEFAULT    2 __bss_start
     9: 00011098     0 NOTYPE  GLOBAL DEFAULT    2 __DATA_BEGIN__
    10: 000110a8     0 NOTYPE  GLOBAL DEFAULT    2 _edata
    11: 000110a8     0 NOTYPE  GLOBAL DEFAULT    2 _end
    12: 0001008c     0 NOTYPE  GLOBAL HIDDEN     1 exit
[16:44:21] carlos@carlosarch /home/carlos/repos/rustv
```

Figura 12 – Inspeção do executável 'main' por meio da ferramenta *readelf*, que exibe os metadados, seções e símbolos contidos no arquivo.

outras informações acerca do programa, enquanto a Figura 13 exibe a saída da ferramenta *objdump*, que é capaz de exibir as instruções de montagem originais que foram codificadas e gravadas no executável.

Por fim, o executável final *main* pode ser ainda testado em uma ferramenta de emulação da arquitetura RISC-V 32 bits, para conferir se a mensagem de *Hello world!* de

```
[16:46:01] carlos@carlosarch /home/carlos/repos/rustv
> riscv32-unknown-linux-gnu-objdump -d main

main:     file format elf32-littleriscv


Disassembly of section .text:

00010074 <_start>:
   10074:       00100513                li      a0,1
   10078:       00001597                auipc   a1,0x1
   1007c:       02058593                addi    a1,a1,32 # 11098 <msg>
   10080:       00d00613                li      a2,13
   10084:       04000893                li      a7,64
   10088:       00000073                ecall

0001008c <exit>:
   1008c:       00000513                li      a0,0
   10090:       05d00893                li      a7,93
   10094:       00000073                ecall
[16:46:05] carlos@carlosarch /home/carlos/repos/rustv
>
```

Figura 13 – Inspeção do executável 'main' por meio da ferramenta *objdump*, que exibe o código original que gerou a seção de texto do programa.

fato é emitida assim como esperado. Para tanto, nesse projeto foi utilizado o programa *QEMU* para construir um ambiente de emulação RISC-V 32 bits que fosse testável para os programas escritos.

Assim, a listagem 3 exibe a execução do arquivo *main* na máquina descrita, que escreve a palavra designada no terminal. Esse experimento valida a corretude do processo de montagem e estabelece a base para a etapa de execução e depuração apresentada no capítulo seguinte.

4.2 Emulador

A implementação do emulador busca refletir diretamente o modelo de emulação tal como arquitetado e exibido na seção 3.2. Nesse sentido, o ambiente de emulação é constituído por um único núcleo de processamento e que conta com uma unidade linear de memória. A partir desses componentes, o emulador é capaz de executar um programa principal, que contempla todos os recursos suportados pelo montador feito para esse projeto, o que inclui definição de variáveis de tipos inteiros e strings, chamada de funções, saltos condicionais e incondicionais e instruções do conjunto base (RV32I).

Para tanto, as rotinas de utilização do emulador incluem ao todo 5 etapas:

- Instanciação da Máquina a executar os programas

- Obtenção dos dados do programa a ser carregado
- Carregamento/Escrita dos dados do programa na memória da Máquina
- Definição do ponto de entrada do programa
- Controle de execução da Máquina por meio da API oferecida pelo contrato da entidade *Machine*

Nas próximas seções, explica-se sobre a implementação da entidade *Machine* e das suas partes internas, que oferecem os meios básicos para a interação com o emulador e as rotinas de funcionamento que viabilizam o ciclo de execução de instruções e outros. Em seguida, explica-se a forma como o carregamento dos dados de um programa (proveniente de diferentes fontes) é feito, seguido do detalhamento da implementação do ciclo de *fetch-execute-decode*, que é o núcleo da emulação.

4.2.1 Máquina

Os recursos computacionais de processamento e de memória estão envoltos da entidade *Máquina*, que consiste em uma interface que abstrai a interação entre seus elementos internos, o que inclui o mecanismo de *fetch-decode-execute*, leitura e escrita de memória, leitura e escrita de registradores, inspeção do estado da máquina. Nesse viés, a entidade Machine atua como elemento central de orquestração entre CPU e memória, expondo uma API unificada para controle da execução.

A listagem 4.21 exibe o contrato a ser implementado para a entidade *Machine* e a API esperada a ser suportada. A partir dessa interface, é possível controlar programaticamente as ações tomadas pela máquina, o que é fundamental para o propósito da emulação em si.

Listagem 4.21 – Contrato da entidade *Máquina*

```
1 pub trait Machine {
2     // Init
3     fn from_bytes_size(byte_count: usize, machine_endian:
    DataEndianness) -> Self ;
4     fn from_words_size(word_count: usize, machine_endian:
    DataEndianness) -> Self ;
5     fn from_bytes(data: &Vec<u8>, machine_endian: DataEndianness) ->
    Self ;
6     fn from_words(data: &Vec<u32>, machine_endian: DataEndianness) ->
    Self ;
7
```

```
8
9       // Core
10      fn load(&mut self, start_addr: usize, instrs: &Vec<u32>) -> () ;
11      fn fetch(&self) -> u32 ;
12      fn jump(&mut self, off: usize) -> () ;
13      fn set_pc(&mut self, new_pc: usize) -> () ;
14      fn decode(&mut self) -> Result<MachineState, MachineError> ;
15      fn endianness(&self) -> DataEndianness ;
16
17
18      // CPU
19      fn read_registers(&self) -> Vec<u32> ;
20      fn write_registers(&mut self, gprs: Vec<u32>, pc: usize) -> () ;
21      fn read_pc(&self) -> u32;
22
23
24      // Memory
25      fn bytes_count(&self) -> usize ;
26      fn words_count(&self) -> usize ;
27
28      fn bytes(&self) -> Vec<u8> ;
29      fn words(&self) -> Vec<u32> ;
30
31      fn read_memory_byte(&self, addr: usize) -> u8;
32      fn write_memory_byte(&mut self, addr: usize, value: u8) -> () ;
33
34      fn read_memory_bytes(&self, addr: usize, count: usize, alignment:
        usize) -> Vec<u8> ;
35      fn write_memory_bytes(&mut self, addr: usize, values: &[u8]) -> ()
        ;
36
37      fn read_memory_word(&self, addr: usize) -> u32 ;
38      fn write_memory_word(&mut self, addr: usize, value: u32) -> () ;
39
40      fn read_memory_words(&self, addr: usize, count: usize) -> Vec<u32>
        ;
41      fn write_memory_words(&mut self, addr: usize, values: &[u32]) -> ()
         ;
42
```

```
43
44      // Debug
45      fn assert_reg(&self, reg: u32, val: u32) -> bool ;
46      fn assert_pc(&self, val: u32) -> bool ;
47      fn assert_memory_words(&self, addr: usize, word_count: usize,
        values: &[u32]) -> bool ;
48      fn assert_memory_bytes(&self, addr: usize, byte_count: usize,
        values: &[u8], alignment: usize) -> bool ;
49      fn predict_next_pc(&self) -> usize ;
50  }
```

Para esse projeto, a implementação do contrato em questão é feito pelo tipo *SimpleMachine*, que pode ser visto na listagem 4.22. Por sua vez, esse tipo conta com a implementação da unidade de processamento (*SimpleCPU*) e de memória (*SimpleMemory*), que implementam seus respectivos contratos a serem detalhados nas próximas subseções. Além disso, a definição conta ainda com a escolha do *endianness* da máquina, que ditará qual orientação será escolhida para as operações de leitura e escrita de dados na memória.

Listagem 4.22 – Contrato da entidade *SimpleMachine*

```
1  pub struct SimpleMachine {
2      cpu: SimpleCPU,
3      mem: SimpleMemory,
4      endian: DataEndianness,
5  }
```

4.2.1.1 CPU

O contrato definido para a entidade *CPU* encontra-se na listagem 4.23, o qual define a API de escrita e leitura dos registradores que o compõem. Essa unidade é primordial para a definição do ponto de entrada do programa a ser carregado na máquina, bem como pelo armazenamento do seu estado de execução ao longo do processamento.

Nesse projeto, a definição desse contrato se dá por meio da entidade *SimpleCPU*, que conta com um conjunto de registradores de propósito geral e o registrador PC, os quais compõem o estado da máquina.

O estado inicial da máquina é definido de acordo com o ambiente de execução a ser utilizado, seja esse a sessão de depuração ou o ambiente hospedeiro que executa o emulador. Nesse cenário, o papel da Máquina consiste em dirigir o mecanismo responsável por modificar seu estado em decorrência da execução de um programa. Para garantir isso, o tipo *SimpleMachine* se beneficia da API exposta pelo contrato *Machine* para controlar programaticamente a unidade de processamento durante a execução do programa, o que

inclui o gerenciamento desse componente durante a execução do ciclo de *fetch-decode-execute*, feito pelo método *decode*.

Nesse contexto, caberá a instância da Máquina ler e escrever os registradores providenciados pela CPU para a execução das instruções RISC-V, de forma a garantir seus efeitos esperados.

Listagem 4.23 – Contrato da entidade *SimpleCPU*

```
pub trait CPU {
    fn write(&mut self, reg: usize, v: u32) ;
    fn read(&self, reg: usize) -> u32 ;

    fn write_pc(&mut self, v: usize) ;
    fn read_pc(&self) -> usize ;

    fn read_all(&self) -> Vec<u32>;
    fn write_all(&mut self, gps: Vec<u32>, pc: usize) -> () ;
}

pub struct SimpleCPU {
    registers: Vec<u32>,
    pc: usize,
}
```

4.2.1.2 Memória

A memória permite que a máquina armazene as instruções e dados referentes ao programa, para que sejam posteriormente lidos sequencialmente e alimentem o ciclo de *fetch-decode-execute* enquanto estiver em atuação. O modo de operação desse elemento se consolida por meio do contrato *Memory* disponível na listagem 4.24, o qual define diferentes métodos para a escrita e leitura das regiões de memória da máquina.

No contexto desse projeto, a definição do contrato supracitado se materializa por meio da entidade *SimpleMemory*, que permite a definição de um espaço de memória de tamanho arbitrário e a escolha do *endianness* a ser utilizado internamente.

Listagem 4.24 – Contrato da entidade *SimpleMemory*

```
pub trait Memory {
    fn endianness(&self) -> DataEndianness;

    fn bytes_count(&self) -> usize;
```

```
5       fn words_count(&self) -> usize;
6
7       fn reserve_bytes(&mut self, sz: usize) ;
8       fn reserve_words(&mut self, sz: usize) ;
9       fn clear(&mut self) ;
10
11      fn bytes(&self) -> Vec<u8>;
12      fn words(&self) -> Vec<u32>;
13
14      fn write_file(&self, filename: &str) -> io::Result<()> ;
15      fn read_file(&mut self, filename: &str) -> io::Result<()> ;
16
17      fn read_byte(&self, idx: usize) -> u8 ;
18      fn write_byte(&mut self, idx: usize, v: u8) -> () ;
19
20      fn read_word(&self, idx: usize) -> u32 ;
21      fn write_word(&mut self, idx: usize, val: u32) -> () ;
22
23      fn read_bytes(&self, start_addr: usize, count: usize,
24          res_endian: DataEndianness, alignment: usize
25      ) -> Vec<u8> ;
26      fn write_bytes(&mut self, start_addr: usize, data: &[u8],
        src_endian: DataEndianness) -> () ;
27
28      fn read_words(&self, start_addr: usize, count: usize,
29          res_endian: DataEndianness) -> Vec<u32> ;
30      fn write_words(&mut self, start_addr: usize, data: &[u32]) -> () {
31          for (idx, i) in data.iter().enumerate() {
32              self.write_word(start_addr + idx, *i);
33          }
34      }
35  }
36
37  pub struct SimpleMemory {
38      data: Vec<u8>,
39      endianness: DataEndianness,
40  }
```

O espaço de memória é linear e não conta com virtualização. Nesse aspecto, a escrita e leitura de diferentes regiões da memória consistem em acessos diretos aos endereços reais

e em uso pela máquina, o que simplifica o modelo em operação e o entendimento de como o mapeamento de seções e instruções é feito. Com efeito, a implementação de uma unidade de gerenciamento de memória (MMU) também seria possível futuramente, graças ao fato do componente ser modular e independente dos demais, o que permite a adição desse e outros recursos mais complexos à operação caso desejado.

Além disso, a decisão em definir o *endianness* do componente se deve pelo fato da Máquina poder ser utilizada por diferentes atores (ex: carregadores, emuladores e depuradores), os quais podem esperar diferentes formatos de ordenação de memória a depender de como funcionam. Desse modo, o suporte a escolha de algum *endianness* da memória permite que os dados gravados e/ou lidos sejam automaticamente gerenciados pela unidade de acordo com a necessidade do código chamador.

4.2.2 Loader

A entidade Loader faz parte da etapa de preparo do ambiente de execução do qual faz parte a Máquina. Nesse sentido, o Loader tem como papel efetuar o carregamento da memória, a fim de fornecer os dados do programa a serem lidos e executados, os quais podem ser provenientes de fontes de diferentes formatos. Em outras palavras, independentemente da origem dos dados, o papel do Loader é garantir que, ao final do processo, a memória e o PC estejam configurados de forma consistente para o início da execução. Para tanto, o mecanismo envolto dessa dinâmica deve fazer uso da interface *Machine*, que provê as primitivas de escrita da memória, bem como também a de escrita do registrador PC, que definirá o endereço na memória que servirá como ponto de entrada para a execução do programa principal.

No que tange a implementação, esse elemento constitui uma exceção ao desenvolvimento orientado a contratos que rege boa parte do projeto. Na prática, não há um contrato ou API única que abstrai o processo de carregamento de dados, mas sim rotinas distintas, que se diferem de acordo com a fonte de obtenção da informação. Essa decisão se justifica pelo fato de o processo de carregamento depender fortemente da fonte dos dados, tornando a definição de um contrato único menos expressiva do ponto de vista arquitetural. Assim, neste projeto existem quatro formas básicas para o carregamento dos dados em memória, utilizadas a depender se a entrada consiste em:

- Um executável no formato ELF
- Códigos montados em memória e carregados diretamente no emulador
- Dados provenientes do depurador
- Um arquivo *dump* de memória

Para efeito da entrada caracterizada como um executável ELF, o processo de carregamento se da pela rotina *new_machine_from_elf*, cuja implementação pode ser vista pela listagem 4.25.

Listagem 4.25 – Rotina responsável pela instanciação de uma Máquina a partir de um executável ELF.

```
pub fn new_machine_from_elf(filename: &str) -> SimpleMachine {
    let data = std::fs::read(filename).expect("Failed␣reading␣elf␣file"
    );

    let reader = elfreader::ElfReader::new(&data, DataEndianness::Be)
        .expect("Failed␣instantiating␣elf␣file␣reader");

    let textsec = reader.section(".text").unwrap();
    let datasec = reader.section(".data");

    let textdata = &textsec.data;
    let text_start = textsec.address as usize;

    let has_data_section = datasec.is_some();

    let (memsize, data_start, datadata) = {
        if !has_data_section {
            let memsize = text_start + textdata.len();
            (memsize, None, None)
        } else {
            let datadata = &datasec.unwrap().data;
            let data_start = datasec.unwrap().address as usize;
            let minsize = textdata.len() + datadata.len();
            let max_start = if text_start > data_start {
                text_start
            } else {
                data_start
            };
            let memsize = if max_start > minsize {
                max_start + minsize
            } else {
                max_start + (minsize - max_start)
            };
```

```
33              (memsize, Some(data_start), Some(datadata))
34          }
35      };
36
37      let pc = reader.pc();
38
39      let mut m = SimpleMachine::from_bytes_size(memsize, DataEndianness
        ::Be);
40
41      m.write_memory_bytes(text_start, textdata);
42      if has_data_section {
43          m.write_memory_bytes(data_start.unwrap(), datadata.unwrap());
44      }
45      m.jump(pc);
46
47      m
48  }
```

O funcionamento do código em questão se resume aos seguintes pontos:

- Leitura do arquivo ELF
- Obtenção dos dados referentes as seções de texto, dados e do registrador PC
- Instanciação de uma máquina com memória suficiente para comportar todas as seções
- Escrita das regiões de memória que contém as instruções e os dados
- Configuração do ponto de entrada do programa

De modo semelhante, as demais rotinas, que foram feitas para tratar diferentes fontes de programas, também apresentam a mesma lógica daquela apresentada para o executável no formato ELF, ainda que possam incluir etapas intermediárias adicionais a depender de detalhes e ou especificidades que lhes convenham. De todo modo, o modelo apresentado permite que os dados estejam disponíveis para uso pela máquina, que poderá iniciar a execução do programa por meio do método *decode*.

4.2.3 Ciclo *fetch-decode-execute*

O ciclo de instrução é o mecanismo central que permite que um programa previamente carregado em memória seja efetivamente executado pela Máquina. Nesse projeto, tal

ciclo é implementado de forma iterativa e controlada pela entidade *Machine*, que coordena a leitura da memória, a decodificação das instruções e a aplicação de seus efeitos sobre o estado da CPU.

De forma geral, o funcionamento do ciclo consiste na leitura de uma palavra de memória a partir do endereço apontado pelo registrador PC, seguida da interpretação dessa palavra como uma instrução válida da arquitetura RISC-V. Uma vez decodificada, a instrução é executada, produzindo efeitos sobre os registradores, a memória e o próprio PC, que é então atualizado para apontar para a próxima instrução a ser processada.

No contexto da implementação proposta, a execução de uma iteração completa do ciclo *fetch-decode-execute* é realizada por meio do método *decode*, exposto pela entidade *Machine*. Cada chamada a esse método corresponde à execução de exatamente uma instrução, permitindo que a Máquina avance de forma controlada pelo fluxo do programa.

Na etapa de fetch, a leitura da memória é feita por meio da leitura do registrador PC, que corresponde ao endereço na memória que inicia a palavra a ser processada. Logo em seguida, é feita a leitura do endereço de memória onde está a instrução, cujo tamanho é de 4 bytes para a arquitetura RISC-V 32 bits. É importante ressaltar que a leitura de memória pressupõe que o PC aponta para um endereço de memória válido, isto é, um endereço múltiplo de 4 e que indica que o alinhamento das instruções está em conformidade com o que a ISA determina.

A etapa seguinte corresponde ao decode da instrução, na qual é feita a identificação da palavra de 32 bits obtida na etapa anterior enquanto um formato de instrução conhecido pelo emulador. Para isso, são extraídos os 7 bits menos significativos da palavra para a determinação de qual formato de instrução está em operação, o qual é denominado *opcode*. A utilização de 7 bits para identificar o formato de instrução ergue uma restrição inicial de 127 possíveis formatos reconhecíveis. Neste projeto, os formatos de instrução implementados consistem naqueles previstos pela extensão RV32I, os quais são identificados pelas letras R, S, U, I, B e J. Quaisquer outros formatos fazem com que a máquina lance um erro de instrução não conhecida, o que finaliza o ambiente de execução. Uma vez identificado o formato primário da instrução, é feita uma extração personalizada para cada formato de acordo com a ISA da extensão base RV32I. Ao final desse procedimento, é retornado um tipo que contém uma estrutura interna capaz de prover facilmente os campos referentes a registradores de origem, registrador de destino, imediato e outras informações necessárias para a execução do código.

Por fim, a etapa de execução é caracterizada pela modificação do estado da máquina por via da instrução atual. Esse processo é iniciado pelo mecanismo de predição de PC, que calcula o próximo valor a ser assumido por esse registrador, sem modificar o estado atual da máquina. A lógica desse procedimento consiste em efetuar o cálculo do próximo endereço caso a instrução seja a de um salto condicional ou incondicional. Caso contrário,

o método simplesmente prevê que o próximo endereço válido corresponde aquele que está 1 palavra a frente do valor atual do PC. Uma vez obtido o endereço da próxima instrução a ser executada, a instrução atual passa a ser desestruturada nos campos que a constituem, como registradores, imediato e opcode. Em sequência, a máquina mapeia a operação que deve ser executada para aquela instrução (como uma soma ou subtração), efetua a leitura dos registradores de origem, executa a operação e escreve o resultado no registrador de destino. Instruções que não são mapeadas em operações conhecidas fazem com que a máquina lance o erro de instrução não tratada, que então finaliza o ambiente de execução. Por outro lado, instruções reconhecidas e que são executadas corretamente fazem o fluxo continuar com a escrita do PC com o valor previsto anteriormente, o que prepara o emulador para que todo o ciclo possa ser repetido, até que o programa termine. O término do programa se dá ou com o lançamento de um erro ou com a chamada de sistema *exit*, que segue a convenção da ABI do Linux para ser efetuada.

4.3 Depuração

Neste projeto, a depuração busca permitir que um usuário seja capaz de acompanhar e interagir com o estado da máquina durante a execução do programa, de modo a controlar o ritmo com que as instruções são processadas. Isso permite que um programa possa ser entendido com mais profundidade e detalhes, o que potencializa o aprendizado de computação e auxilia na detecção de problemas. Para atender a esse requisito, o sistema foi projetado de modo a permitir a interrupção, inspeção e retomada da execução de forma controlada, explorando o fato de que a Máquina executa programas por meio do ciclo *fetch-decode-execute*. Essa característica possibilita que o avanço da execução seja desacoplado do andamento ininterrupto e usual do programa, viabilizando mecanismos como execução passo a passo, definição de pontos de parada e leitura e escrita de registradores e memória.

O suporte à depuração se apoia na utilização do protocolo GDB, que permite conectar a interface de um programa de depuração amplamente utilizado com o emulador implementado. Dessa forma, é possível que o usuário se comunique com o Emulador remotamente e envie comandos que permitam extrair informações sobre o estado da máquina e também exercer controle sobre o ritmo com que o programa é executado. Paralelamente, os programas produzidos pelo Montador são enriquecidos com informações no formato DWARF, o que acontece na etapa de escrita do arquivo objeto, que é exibida na listagem 4.19.

Assim como tratado na seção 2.6, a implementação do esquema de depuração envolve a presença de três elementos: a máquina hospedeira, o *stub* e a máquina alvo. Nesse projeto, a máquina hospedeira se refere ao dispositivo que executa o GDB, que provê uma interface de usuário por linha de comando, a partir da qual é possível se conectar ao

ambiente de emulação. Concomitantemente, o *stub* consiste em um intermediador entre a máquina hospedeiro e o emulador, sendo o ator responsável por efetivamente integrar os comandos enviados pelo usuário com o emulador. Nesse aspecto, é importante que o emulador ofereça meios para que essa ferramenta externa seja capaz de obter e controlar o estado de execução do programa, o que é satisfeito primariamente pela API lançada pelo contrato *Machine*, que foi desbravado na seção 4.2.1. Por fim, o elemento presente na outra ponta corresponde ao emulador, que é responsável por emular uma máquina sobre influência de algum agente que seja capaz de controla-lá.

Para tanto, a implementação faz uso da biblioteca *gdbstub*, que oferece uma API que abstrai boa parte do protocolo remoto de comunicação GDB. Nesse sentido, cabe ao programador o desenvolvimento do *stub*, que deve lidar com a comunicação serial e chamar os métodos adequados para o controle do ambiente de emulação, quando oportuno.

As próximas seções descrevem com mais detalhes a execução do programa a partir de uma instância do GDB, e como é definida e controlada a definição de *breakpoints* e execução de comandos básicos de uma sessão de depuração como *step*, *continue* e inspeção de memória. Em sequência, será tratado sobre as limitações e extensões futuras desse processo.

4.3.1 Modelo de controle da execução

A sessão de depuração inicia-se com a disponibilização do emulador por via de um endereço de rede e uma porta. Para o escopo desse projeto, o emulador é disponibilizado no endereço *localhost* e na porta 9999 por padrão, no entanto também seria possível disponibilizar o ambiente de emulação em um servidor remoto. A Figura 14 exibe o comando feito no terminal para iniciar o servidor de depuração da arquitetura RISC-V 32 bits.

```
[00:00:15] carlos@carlosarch /home/carlos/repos/rustv
> ./target/release/rustv --debugger 9999
Waiting for GDB to connect to target at localhost:9999
Enter gdb and type:
  gdb> target remote :9999
  gdb> load
  gdb> x/1xw 0x10074
```

Figura 14 – Inicialização do emulador com suporte à depuração remota via GDB

Uma vez em funcionamento, é necessário que seja feita a conexão entre o servidor que executa o emulador e o programa cliente que roda o GDB. Esse procedimento pode ser visto na Figura 16.

```
[23:14:46] carlos@carlosarch /home/carlos/repos/rustv
> gdb main --quiet
Reading symbols from main...
(gdb)
```

Figura 15 – Instanciação de uma sessão de depuração pelo terminal utilizando o GDB

Esse procedimento é marcado pelo trâmite dos recursos suportados por ambas as partes, bem como pela definição das regiões de memória da máquina que são passíveis de leitura, escrita, execução e outros. Além disso, é feita a definição do estado inicial da máquina, que conta com a leitura do estado corrente dos registradores e de toda a memória acessível a um programa. Essas definições são tratadas pelo *stub*, que mapeia a leitura de todos os registradores ao método *read_registers* e a leitura da memória ao métodos *bytes* do contrato *Máquina*.

Após estabelecida a conexão entre o cliente GDB e o emulador, procede-se com o carregamento da memória do emulador com o executável a ser depurado. A Figura 16 exibe os comandos feitos no lado do cliente GDB para a leitura do executável *main*, disponível no lado da máquina hospedeira. Nesse caso, o arquivo em questão contém o código do *hello-world.s*, o mesmo que foi utilizado na seção 4.2. O mapeamento desse comando corresponde a chamada do método *write_memory_bytes*, que permite a escrita de bytes em porções arbitrárias da memória do emulador. Nesse aspecto, a tentativa da escrita em porções inválidas da memória (fora do limite estabelecido) geram um erro fatal do emulador e fazem com que esse seja finalizado, o que interrompe a conexão feita com o cliente GDB.

```
(gdb) target remote :9999
Remote debugging using :9999
0x00000000 in ?? ()
(gdb) load
Loading section .text, size 0x24 lma 0x10074
Loading section .data, size 0x10 lma 0x11098
Start address 0x00010074, load size 52
Transfer rate: 50 KB/sec, 26 bytes/write.
(gdb)
```

Figura 16 – Carregamento do executável a ser depurado na memória do emulador

Uma vez que o programa tenha sido carregado em memória, é possível iniciar a utilização das rotinas de depuração usuais. Entre essas, está a definição *breakpoints*, que nesse projeto podem ser definidos para quaisquer labels existentes no programa. A exemplo

disso, a Figura 17 mostra a definição de um *breakpoint* na label *_start*, a qual marca o início do programa.

```
(gdb) b
-function                 -probe                    -qualified                _start
-label                    -probe-dtrace             -source                   exit
-line                     -probe-stap               ././examples/hello-world.s
(gdb) b _start
Breakpoint 1 at 0x10078: file ././examples/hello-world.s, line 10.
(gdb)
```

Figura 17 – Definição de um breakpoint no programa

A definição de um *breakpoint* faz com que o *stub* execute o método *predict_next_pc* da API do emulador e armazene o próximo endereço a ser inspecionado. Nesse sentido, o método de predição do próximo PC é responsável por lidar com cenários simples, desde a definição do próximo endereço válido disponível, como a definição de um endereço distante, relacionado a saltos ou retorno de funções. A partir de uma lista dos *breakpoints* definidos, o *stub* é capaz decidir em qual momento deve parar a execução do método *decode* do emulador e retornar o controle da sessão para o cliente GDB.

O programa pode, então, ser reproduzido sequencialmente pelo usuário, que conta com dois comandos para controlar o fluxo do programa: *step* e *continue*. Ambos os comandos instruem o *stub* a chamar o método *decode*, responsável pelo ciclo *fetch-decode-execute*, até que o próximo ponto de parada seja encontrado. A Figura 18 exibe essa dinâmica, na qual é utilizado a versão abreviada do comando step para executar a primeira instrução do código de montagem, que consiste na instrução *li a0, 1*.

```
(gdb) b _start
Breakpoint 1 at 0x10078: file ././examples/hello-world.s, line 10.
(gdb) si

Breakpoint 1, _start () at ././examples/hello-world.s:10
10              la a1, msg              // buffer address
(gdb) list
5
6               .section .text
7       _start:
8               // write(stdout=1, msg, len)
9               li a0, 1                // fd = 1 (stdout)
10              la a1, msg              // buffer address
11              li a2, 13               // length
12              li a7, 64               // syscall: write
13              ecall
14      exit:
```

Figura 18 – Execução do comando *step* para executar um unico ciclo *fetch-decode-execute* e retornar o controle ao cliente GDB

Após o início da execução do programa, espera-se que o estado da máquina seja alterado de acordo com as instruções processadas. Para a instrução previamente executada,

esse cenário requer a inspeção do estado dos registradores da máquina. A Figura 19 exibe o comando que permite visualizar o estado do programa, que nesse caso apresenta o valor 1 carregado no registrador a0, assim como esperado.

```
(gdb) info registers
ra             0x0        0x0
sp             0x0        0x0
gp             0x0        0x0
tp             0x0        0x0
t0             0x0        0
t1             0x0        0
t2             0x0        0
fp             0x0        0x0
s1             0x0        0
a0             0x1        1
```

Figura 19 – Inspeção do estado de todos os registradores da máquina

Adicionalmente, é importante a habilidade de inspecionar o conteúdo carregado em endereços arbitrários da memória, o que permite a inspeção da seção de texto, dados e bss. A Figura 20 demonstra esse procedimento, onde é possível visualizar a representação binária da próxima instrução a ser executada.

```
t3             0x0        0
t4             0x0        0
t5             0x0        0
t6             0x0        0
pc             0x10078    0x10078 <_start+4>
(gdb) x/1xw 0x10078
0x10078 <_start+4>:     0x00001597
(gdb)
```

Figura 20 – Inspecção do endereço armazenado em PC

Por fim, a finalização de um programa nesse projeto deve contar com a chamada de sistema *exit*, que pode ser feita por meio da convenção da ABI do linux. No momento, o suporte a chamada de sistemas inclui as funções *exit* e *write* do sistema linux, o que permite que o emulador seja terminado e comunique seu status ao ambiente de execução, bem como também que escreva mensagens no console do terminal, o que permite que exiba a mensagem desejada no lado da máquina alvo. A Figura 21 exibe a execução do programa até a chamada de sistema *exit*, enquanto a Figura 22 exibe o resultado feito a chamada de sistema *write* para escrita da mensagem 'Hello World!' no lado da máquina alvo.

```
(gdb) c
Continuing.
[Inferior 1 (process 1) exited normally]
(gdb)
```

Figura 21 – Execução do comando *continue* seguido da finalização do processo existente no emulador

```
  gdb> x/1xw 0x10074
Hello world!
[00:00:57] carlos@carlosarch /home/carlos/repos/rustv
>
```

Figura 22 – Texto "Hello World!"exibido no lado do emulador

4.3.2 Limitações e extensões futuras

A biblioteca utilizada para abstração do protocolo GDB (*gdbstub*) possui suporte a maior parte das funcionalidades exibidas por esse protocolo, o que inclui o reconhecimento de diversos comandos que podem ser enviados pelo cliente GDB. Nesse aspecto, o *stub* e *emulador* implementados podem ser enriquecidos para que suportem a uma gama maior de comandos provenientes do usuário, tais como *reverse step*, suporte a *watchpoints* e outros.

4.4 Suite de Testes

A elaboração de testes corresponde a uma prática importante no preparo de um projeto que ofereça robustez, corretude e confiança. Com efeito, a capacidade de verificar o funcionamento dos componentes de um sistema confere a possibilidade de acompanhar a evolução do projeto, consolidar comportamentos esperados e identificar possíveis problemas emanentes. Em particular, projetos modularizáveis se beneficiam da possibilidade de serem validados, a medida que suas partes são testadas e aprovadas.

O planejamento de uma arquitetura modular para o montador e emulador apontou para um desenvolvimento orientado a testes, que reforçaria a segurança ao longo da implementação deste trabalho. Nesse sentido, a incorporação de testes guiou a escolha de uma linguagem que oferecesse suporte a testes unitários, que permitisse o teste dos elementos que compõem as entidades arquitetadas para o montador e emulador.

Os testes elaborados abrangem tanto testes unitários, voltados à validação de componentes isolados, quanto testes de integração, que verificam o comportamento conjunto do montador e do emulador durante a execução de programas completos. Os testes feitos se baseiam nos testes realizados em outros projetos de emuladores, como rars (BERG; CONTRIBUTORS, 2024), ripes (MORTENSEN; CONTRIBUTORS, 2024), bem como

também na bateria de testes oficial da organização riscv RISC-V Software Source and contributors (2024).

Ao todo foram implementados 69 testes, os quais estão disponíveis no arquivo *src/lib.rs* do repositório desse projeto (VIEIRA, 2025). Uma relação da quantidade de testes feitos para os aspectos do projeto que foram validados está disponível na tabela 4.

A Figura 23 exibe o status de execução de todos os testes elaborados.

```
test tests::gas::program_multiply_vector ... ok
test tests::gas::program_rec_fcall ... ok
test tests::gas::program_selection_sort ... ok

test result: ok. 69 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

     Running unittests src/main.rs (target/debug/deps/rustv-fdb8422b757ff8e1)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

   Doc-tests rustv

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

[21:23:42] carlos@carlosarch /home/carlos/repos/rustv
>
```

Figura 23 – Resultado de todos os testes preparados

Como forma adicional de validação externa, foi empregado um ambiente de emulação independente, no qual o projeto contou com a criação de um ambiente de emulação alternativo para testar os executáveis gerados a serem processados na arquitetura RISC-V 32 bits. Esse ambiente contou com a compilação do kernel linux e do programa *busybox* para a arquitetura RISC-V 32 bits, juntamente da utilização do emulador *QEMU* para o carregamento do kernel e do programa *busybox*. A partir disso, foi possível validar se os programas feitos pelo montador de fato seriam capazes de rodar em outros emuladores. A exemplo disso, a listagem 3 demonstra a execução do programa *hello-world.s*, no qual é possível averiguar que a mensagem "Hello world!"de fato é exibida no terminal do ambiente de emulação em questão.

Tabela 4 – Aspectos do projeto validados por meio de testes

Aspecto validado	**Descrição**	**Quantidade**
Extração das palavras do código de montagem	Testes responsáveis por validar a correta separação e identificação dos elementos léxicos do código de montagem RISC-V.	14
Codificação de instruções no formato binário	Validação da conversão das instruções de montagem para sua representação binária conforme o formato definido pela ISA RISC-V.	13
Leitura e escrita dos registradores da CPU	Testes que verificam o correto funcionamento das operações de acesso e modificação dos registradores do processador emulado.	3
Leitura e escrita da memória	Testes destinados a garantir a correta manipulação da memória da máquina emulada, incluindo operações de leitura e escrita.	6
Execução de instruções individuais + Ciclo de execução da máquina	Testes focados na execução correta de instruções isoladas, permitindo verificar seus efeitos no estado da máquina. Esses testes também compreendem o teste do ciclo de *fetch-decode-execute* da máquina.	25
Execução de programas com chamadas de função	Validação da execução de programas que realizam chamadas de função, verificando o correto controle de fluxo.	1
Execução de programas com funções recursivas	Testes voltados à execução correta de funções recursivas, incluindo o empilhamento e desempilhamento de contextos de execução.	1
Utilização da pilha de funções	Validação do uso correto da pilha de execução durante chamadas de função, incluindo passagem de parâmetros e armazenamento de endereços de retorno.	1
Execução de algoritmo de ordenação	Teste da execução de um programa que implementa o algoritmo *selection sort*, validando a execução de programas completos.	1
Execução de programas com vetores	Teste da execução de um programa que utiliza vetores de inteiros.	1
Leitura e escrita de arquivos ELF	Testes destinados a verificar a correta geração, leitura e interpretação de arquivos executáveis no formato ELF.	3

5 Considerações Finais

A arquitetura RISC-V se apresenta como uma ferramenta de estudo moderna e com grande potencial para fomentar o ensino de computação. Em razão de sua especificação ser livre e aberta ao público, o estudo e desenvolvimento da arquitetura RISC-V podem ser também interessantes para atores fora do campo acadêmico, como investidores, engenheiros, entusiastas e fabricantes. Nesse aspecto, a arquitetura RISC-V aparenta se encontrar em um cenário favorável ao seu desenvolvimento, onde é possível vislumbrar o encontro do interesse de diferentes personagens, bem como o amadurecimento da documentação técnica que a sustenta.

Em conclusão, sob a luz do que a arquitetura RISC-V oferece para fins educacionais, é possível defender que o projeto cumpre com o objetivo de desenvolver um ferramental didático que possa ser utilizado por discentes e docentes no ensino da arquitetura, que se inicia na escrita de códigos em linguagem de montagem RISC-V, passa pela codificação das instruções em binário e se estende até o acompanhamento e inspeção das instruções em um ambiente de emulação que suporta depuração. Assim, espera-se que esse trabalho contribua para a adoção de arquiteturas que sejam livres, abertas e acessíveis a todos.

5.1 Trabalhos futuros

O projeto em questão se mostra como um ponto de partida sólido para a construção de um ambiente de desenvolvimento mais realístico, completo e amigável para discentes e docentes. Nesse aspecto, é possível apontar melhorias que focam em aprimorar a experiência do usuário com o sistema, que foi primariamente testado e se sustenta em uma interface baseada em linha de comando, a qual pode ser um ponto limitante para aqueles que não possuem familiaridade com esse formato. Assim, o projeto se beneficiaria com a construção de uma interface gráfica que integre as etapas de escrita de código, montagem, emulação e depuração de programas, de modo a facilitar e agilizar a interação com essas frentes.

Além disso, é possível apontar que tanto a frente do montador, como emulador e depurador, podem ser enriquecidas. O montador pode ser incrementado para que suporte mais extensões, como por exemplo a extensão C (que oferece suporte a instruções menores do que 4 bytes) e a extensão F (que oferece suporte a extensões de ponto flutuante). Paralelamente, o emulador deve ser desenvolvido de modo a suportar quaisquer novas instruções. Por fim, a depuração pode se beneficiar com o suporte de novos comandos provenientes do cliente GDB, como *reverse step*.

Referências

BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In: *Proceedings of the USENIX Annual Technical Conference*. [S.l.: s.n.], 2005. Citado na página 24.

BERG, P.; CONTRIBUTORS. *RARS: RISC-V Assembler and Runtime Simulator*. 2024. Source code repository. Disponível em: <https://github.com/TheThirdOne/rars>. Citado na página 73.

FREE SOFTWARE FOUNDATION. *Debugging with GDB*. Gdb 14.1. [S.l.], 2023. Acesso em: 25 dez. 2025. Disponível em: <https://sourceware.org/gdb/current/onlinedocs/>. Citado 2 vezes nas páginas 22 e 31.

KLABNIK, S.; NICHOLS, C. *The Rust Programming Language*. 2024. Accessed as the official Rust documentation. Disponível em: <https://doc.rust-lang.org/stable/book/>. Citado na página 23.

LU, T. A survey on RISC-V security: Hardware and architecture. *CoRR*, abs/2107.04175, 2021. Disponível em: <https://arxiv.org/abs/2107.04175>. Citado 2 vezes nas páginas 12 e 13.

MEZGER, B. W. et al. A survey of the risc-v architecture software support. *IEEE Access*, v. 10, p. 51394–51411, 2022. Citado na página 16.

MORTENSEN, S.; CONTRIBUTORS. *Ripes: RISC-V Processor Simulator and Assembly Editor*. 2024. Source code repository. Disponível em: <https://github.com/mortbopet/Ripes>. Citado na página 73.

RANTAKARI, T.; TESTA, L. *RISC-V in 2025: Progress, Challenges, and What's Next for Automotive & OpenHardware*. 2025. Industry article. Disponível em: <https://www.design-reuse.com/article/61590-risc-v-in-2025-progress-challenges-and-what-s-next-for-automotive-openhardware/>. Citado na página 13.

RISC-V Collaboration and contributors. *RISC-V GNU Toolchain*. 2024. Source code repository. Disponível em: <https://github.com/riscv-collab/riscv-gnu-toolchain>. Citado na página 24.

RISC-V INTERNATIONAL. *RISC-V Assembly Programmer's Manual*. [S.l.], 2016. Acesso em: 25 dez. 2025. Disponível em: <https://riscv.org>. Citado na página 17.

RISC-V Software Source and contributors. *RISC-V Tests*. 2024. Source code repository. Disponível em: <https://github.com/riscv-software-src/riscv-tests>. Citado na página 74.

TANENBAUM, A. S.; AUSTIN, T. *Structured Computer Organization*. 6. ed. [S.l.]: Pearson, 2013. Citado na página 19.

TOOL INTERFACE STANDARDS COMMITTEE. *Tool Interface Standard (TIS): Executable and Linkable Format (ELF)*. Version 1.1. [S.l.], 1995. Acesso em: 25 dez. 2025.

Disponível em: <https://refspecs.linuxfoundation.org/elf/elf.pdf>. Citado 2 vezes nas páginas 19 e 29.

TORVALDS, L.; CONTRIBUTORS. *Linux Kernel*. 2024. Source code repository. Disponível em: <https://github.com/torvalds/linux>. Citado na página 24.

VIEIRA, C. *rustv: Montador, Emulador e Suporte à Depuração para RISC-V 32 bits*. 2025. <https://github.com/cdavieira/rustv>. Acesso em: janeiro de 2026. Citado 3 vezes nas páginas 33, 36 e 74.

WATERMAN, A. et al. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. Version 20191213. [S.l.], 2019. Acesso em: 25 dez. 2025. Disponível em: <https://riscv.org/technical/specifications/>. Citado 5 vezes nas páginas 12, 15, 16, 17 e 21.

WATERMAN, A. et al. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Version 20190608. [S.l.], 2019. Acesso em: 25 dez. 2025. Disponível em: <https://riscv.org/technical/specifications/>. Citado na página 16.

WORKGROUP, D. *DWARF Debugging Information Format Version 5*. [S.l.], 2017. Acesso em: 25 dez. 2025. Disponível em: <https://dwarfstd.org/doc/DWARF5.pdf>. Citado na página 21.

Apêndices

Listagem 1 – Contrato do tipo *CommonClassifier*

```
pub trait CommonClassifier {
    fn is_ambiguous(&self, ch: char) -> bool;
    fn handle_ambiguous(&self, it: &mut CharStreamReader) -> Option<
    String>;

    fn is_unit(&self, ch: char) -> bool;
    fn handle_unit(&self, it: &mut CharStreamReader) -> Option<String>
    ;

    fn is_comment(&self, ch: char) -> bool;
    fn handle_comment(&self, it: &mut CharStreamReader) -> Option<
    String>;

    fn is_identifier(&self, ch: char) -> bool;
    fn handle_identifier(&self, it: &mut CharStreamReader) -> Option<
    String>;

    fn is_string(&self, ch: char) -> bool ;
    fn handle_string(&self, it: &mut CharStreamReader) -> Option<String
    > ;

    fn is_ignore(&self, ch: char) -> bool ;
    fn handle_ignore(&self, it: &mut CharStreamReader) -> Option<String
    > ;

    fn is_number(&self, ch: char) -> bool ;
    fn handle_number(&self, it: &mut CharStreamReader) -> Option<String
    > ;


    fn is_token(&self, ch: char) -> Option<CommonClass> {
        if self.is_ambiguous(ch) {
            return Some(CommonClass::Ambiguous);
        }
        if self.is_unit(ch) {
            return Some(CommonClass::Unit);
        }
        if self.is_comment(ch) {
```

```
32              return Some(CommonClass::Comment);
33          }
34          if self.is_string(ch) {
35              return Some(CommonClass::String);
36          }
37          if self.is_ignore(ch) {
38              return Some(CommonClass::Ignore);
39          }
40          if self.is_number(ch) {
41              return Some(CommonClass::Number);
42          }
43          if self.is_identifier(ch){
44              return Some(CommonClass::Identifier);
45          }
46          None
47      }
48
49      fn handle_token(
50          &mut self,
51          it: &mut CharStreamReader,
52      ) -> Result<Option<String>>
53      {
54          let Some(ch) = it.current_token() else {
55              return Ok(None);
56          };
57          let class = self.is_token(ch);
58          match class {
59              Some(CommonClass::Ambiguous)  => Ok(self.handle_ambiguous(
    it)),
60              Some(CommonClass::Unit)       => Ok(self.handle_unit(it)),
61              Some(CommonClass::Comment)    => Ok(self.handle_comment(it)
    ),
62              Some(CommonClass::String)     => Ok(self.handle_string(it))
    ,
63              Some(CommonClass::Ignore)     => Ok(self.handle_ignore(it))
    ,
64              Some(CommonClass::Number)     => Ok(self.handle_number(it))
    ,
65              Some(CommonClass::Identifier) => Ok(self.handle_identifier(
```

```
    it)),
66              None => {
67                  let pos = it.current_position().unwrap_or(Position::new
    (0, 0, 0));
68                  Err(LexerError::AutomataException(pos))
69              }
70          }
71      }
72  }
```

Listagem 2 – Contrato do *TokenClassifier*

```
1  pub trait TokenClassifier {
2      type Token;
3
4      fn is_symbol(&self, token: &str) -> bool ;
5      fn is_register(&self, token: &str) -> bool ;
6      fn is_opcode(&self, token: &str) -> bool ;
7      fn is_identifier(&self, token: &str) -> bool ;
8      fn is_section(&self, token: &str) -> bool ;
9      fn is_directive(&self, _: &str) -> bool ;
10     fn is_custom(&self, token: &str) -> bool ;
11     fn is_label(&self, token: &str) -> bool ;
12     fn is_number(&self, token: &str) -> bool ;
13     fn is_string(&self, token: &str) -> bool ;
14
15     fn handle_number(&self, it: &mut PositionedStringStreamReader) ->
    Option<Self::Token> ;
16     fn handle_string(&self, it: &mut PositionedStringStreamReader) ->
    Option<Self::Token> ;
17     fn handle_symbol(&self, it: &mut PositionedStringStreamReader) ->
    Option<Self::Token> ;
18     fn handle_register(&self, it: &mut PositionedStringStreamReader) ->
     Option<Self::Token> ;
19     fn handle_opcode(&self, it: &mut PositionedStringStreamReader) ->
    Option<Self::Token> ;
20     fn handle_identifier(&self, it: &mut PositionedStringStreamReader)
    -> Option<Self::Token> ;
21     fn handle_section(&self, it: &mut PositionedStringStreamReader) ->
    Option<Self::Token> ;
```

```
22      fn handle_directive(&self, it: &mut PositionedStringStreamReader)
    -> Option<Self::Token> ;
23      fn handle_custom(&self, it: &mut PositionedStringStreamReader) ->
    Option<Self::Token> ;
24      fn handle_label(&self, it: &mut PositionedStringStreamReader) ->
    Option<Self::Token> ;
25
26      fn classify(&self, token: &str) -> TokenClass {
27          if self.is_symbol(token) {
28              return TokenClass::Symbol;
29          }
30          if self.is_register(token) {
31              return TokenClass::Register;
32          }
33          if self.is_opcode(token) {
34              return TokenClass::Opcode;
35          }
36          if self.is_directive(token) {
37              return TokenClass::Directive;
38          }
39          if self.is_section(token) {
40              return TokenClass::Section;
41          }
42          if self.is_custom(token) {
43              return TokenClass::Custom;
44          }
45          if self.is_label(token) {
46              return TokenClass::Label;
47          }
48          if self.is_number(token) {
49              return TokenClass::Number;
50          }
51          if self.is_string(token) {
52              return TokenClass::String;
53          }
54          if self.is_identifier(token) {
55              return TokenClass::Identifier;
56          }
57          TokenClass::Ignore
```

```
58      }
59
60      fn handle_token(&self, it: &mut PositionedStringStreamReader) ->
    Option<Self::Token> {
61          let token = it.current_token().expect("Lexer␣failed␣when␣
    retrieving␣token");
62          let class = self.classify(token.0.as_str());
63          // println!("Processing {} as {:?}", token, class);
64          match class {
65              TokenClass::Label      => self.handle_label(it),
66              TokenClass::Number     => self.handle_number(it),
67              TokenClass::Symbol     => self.handle_symbol(it),
68              TokenClass::Section    => self.handle_section(it),
69              TokenClass::Directive  => self.handle_directive(it),
70              TokenClass::Custom     => self.handle_custom(it),
71              TokenClass::Opcode     => self.handle_opcode(it),
72              TokenClass::String     => self.handle_string(it),
73              TokenClass::Identifier => self.handle_identifier(it),
74              TokenClass::Register   => self.handle_register(it),
75              TokenClass::Ignore     => None
76          }
77      }
78  }
```

Listagem 3 – Execução do arquivo *main* em um ambiente RISC-V 32 bits por meio do programa *QEMU*

```
1  [16:58:26] carlos@carlosarch /home/carlos/repos/code/all/fish/projects/
       riscv32
2  > qemu-system-riscv32 \
3        -M virt           \
4        -name riscv32     \
5        -k pt-br          \
6        -m 2G             \
7        -smp 1            \
8        -nographic        \
9        -device virtio-blk-device,drive=riscvim                    \
10       -blockdev driver=file,filename="$HOME/Projects/riscv32/drive.raw"
       ,node-name=riscvim \
11       -kernel "$HOME/Projects/riscv32/vmlinuz"        \
```

```
12        -initrd "$HOME/Projects/riscv32/initramfs.cpio" \
13        -append "console=ttyS0␣root=fe00␣init=/bin/busybox␣sh"
14
15  OpenSBI v1.5.1
16     ____                   _____ ____ _____
17    / __ \                 / ____|  _ \_   _|
18   | |  | |_ __   ___ _ __ | (___ | |_) || |
19   | |  | | '_␣\␣/␣_␣\␣'_ \ \___ \|  _ < | |
20   | |__| | |_) |  __/ | | |____) | |_) || |_
21    \____/| .__/ \___|_| |_|_____/|____/_____|
22          | |
23          |_|
24
25  Platform Name             : riscv-virtio,qemu
26  Platform Features         : medeleg
27  Platform HART Count       : 1
28  Platform IPI Device       : aclint-mswi
29  Platform Timer Device     : aclint-mtimer @ 10000000Hz
30  Platform Console Device   : uart8250
31  Platform HSM Device       : ---
32  Platform PMU Device       : ---
33  Platform Reboot Device    : syscon-reboot
34  Platform Shutdown Device  : syscon-poweroff
35  Platform Suspend Device   : ---
36  Platform CPPC Device      : ---
37  Firmware Base             : 0x80000000
38  Firmware Size             : 319 KB
39  Firmware RW Offset        : 0x40000
40  Firmware RW Size          : 63 KB
41  Firmware Heap Offset      : 0x47000
42  Firmware Heap Size        : 35 KB (total), 2 KB (reserved), 10 KB (used
        ), 22 KB (free)
43  Firmware Scratch Size     : 4096 B (total), 244 B (used), 3852 B (free)
44  Runtime SBI Version       : 2.0
45
46  Domain0 Name              : root
47  Domain0 Boot HART         : 0
48  Domain0 HARTs             : 0*
49  Domain0 Region00          : 0x00100000-0x00100fff M: (I,R,W) S/U: (R,W)
```

```
Domain0 Region01          : 0x10000000-0x10000fff M: (I,R,W) S/U: (R,W)
Domain0 Region02          : 0x02000000-0x0200ffff M: (I,R,W) S/U: ()
Domain0 Region03          : 0x80040000-0x8004ffff M: (R,W) S/U: ()
Domain0 Region04          : 0x80000000-0x8003ffff M: (R,X) S/U: ()
Domain0 Region05          : 0x0c400000-0x0c5fffff M: (I,R,W) S/U: (R,W)
Domain0 Region06          : 0x0c000000-0x0c3fffff M: (I,R,W) S/U: (R,W)
Domain0 Region07          : 0x00000000-0xffffffff M: () S/U: (R,W,X)
Domain0 Next Address      : 0x80400000
Domain0 Next Arg1         : 0xbfe00000
Domain0 Next Mode         : S-mode
Domain0 SysReset          : yes
Domain0 SysSuspend        : yes

Boot HART ID              : 0
Boot HART Domain          : root
Boot HART Priv Version    : v1.12
Boot HART Base ISA        : rv32imafdch
Boot HART ISA Extensions  : sstc,zicntr,zihpm,zicboz,zicbom,sdtrig,
   svadu
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 2 bits
Boot HART PMP Address Bits: 32
Boot HART MHPM Info       : 16 (0x0007fff8)
Boot HART Debug Triggers  : 2 triggers
Boot HART MIDELEG         : 0x00001666
Boot HART MEDELEG         : 0x00f0b509
[    0.000000] Booting Linux on hartid 0
[    0.000000] Linux version 6.18.0-13201-g4a298a43f5e3 (
   carlos@carlosarch) (riscv32-unknown-linux-gnu-gcc () 15.1.0, GNU ld
   (GNU Binutils) 2.45) #1 SMP Sat Dec 13 18:16:14 -03 2025
[    0.000000] random: crng init done
[    0.000000] OF: fdt: Ignoring memory range 0x80000000 - 0x80400000
[    0.000000] Machine model: riscv-virtio,qemu
[    0.000000] SBI specification v2.0 detected
[    0.000000] SBI implementation ID=0x1 Version=0x10005
[    0.000000] SBI TIME extension detected
[    0.000000] SBI IPI extension detected
[    0.000000] SBI RFENCE extension detected
[    0.000000] SBI SRST extension detected
```

```
[    0.000000] SBI DBCN extension detected
[    0.000000] efi: UEFI not found.
[    0.000000] OF: reserved mem: 0x80000000..0x8003ffff (256 KiB) nomap
 non-reusable mmode_resv1@80000000
[    0.000000] OF: reserved mem: 0x80040000..0x8004ffff (64 KiB) nomap
non-reusable mmode_resv0@80040000
[    0.000000] Zone ranges:
[    0.000000]   Normal   [mem 0x0000000080400000-0x00000000c03fffff]
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x0000000080400000-0x00000000c03fffff]
[    0.000000] Initmem setup node 0 [mem 0x0000000080400000-0
x00000000c03fffff]
[    0.000000] On node 0, zone Normal: 1024 pages in unavailable ranges
[    0.000000] On node 0, zone Normal: 31744 pages in unavailable
ranges
[    0.000000] SBI HSM extension detected
[    0.000000] riscv: base ISA extensions acdfhim
[    0.000000] riscv: ELF capabilities acdfim
[    0.000000] Queued spinlock using Ziccrse: enabled
[    0.000000] percpu: Embedded 21 pages/cpu s57036 r8192 d20788 u86016
[    0.000000] Kernel command line: console=ttyS0 root=fe00 init=/bin/
busybox sh
[    0.000000] Unknown kernel command line parameters "sh", will be
passed to user space.
[    0.000000] printk: log buffer data + meta data: 131072 + 409600 =
540672 bytes
[    0.000000] Dentry cache hash table entries: 131072 (order: 7,
524288 bytes, linear)
[    0.000000] Inode-cache hash table entries: 65536 (order: 6, 262144
bytes, linear)
[    0.000000] Built 1 zonelists, mobility grouping on.  Total pages:
262144
[    0.000000] mem auto-init: stack:all(zero), heap alloc:off, heap
free:off
[    0.000000] SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=1, Nodes
=1
[    0.000000] rcu: Hierarchical RCU implementation.
[    0.000000] rcu:   RCU event tracing is enabled.
```

```
113 [    0.000000] rcu:   RCU restricting CPUs from NR_CPUS=64 to
    nr_cpu_ids=1.
114 [    0.000000]  Tracing variant of Tasks RCU enabled.
115 [    0.000000] rcu: RCU calculated value of scheduler-enlistment delay
    is 25 jiffies.
116 [    0.000000] rcu: Adjusting geometry for rcu_fanout_leaf=16,
    nr_cpu_ids=1
117 [    0.000000] RCU Tasks Trace: Setting shift to 0 and lim to 1
    rcu_task_cb_adjust=1 rcu_task_cpu_ids=1.
118 [    0.000000] NR_IRQS: 64, nr_irqs: 64, preallocated irqs: 0
119 [    0.000000] riscv-intc: 32 local interrupts mapped
120 [    0.000000] riscv: providing IPIs using SBI IPI extension
121 [    0.000000] rcu: srcu_init: Setting srcu_struct sizes based on
    contention.
122 [    0.000000] clocksource: riscv_clocksource: mask: 0xffffffffffffffff
     max_cycles: 0x24e6a1710, max_idle_ns: 440795202120 ns
123 [    0.000094] sched_clock: 64 bits at 10MHz, resolution 100ns, wraps
    every 4398046511100ns
124 [    0.000184] riscv-timer: Timer interrupt in S-mode is available via
    sstc extension
125 [    0.004298] Console: colour dummy device 80x25
126 [    0.008026] Calibrating delay loop (skipped), value calculated using
     timer frequency.. 20.00 BogoMIPS (lpj=40000)
127 [    0.008120] pid_max: default: 32768 minimum: 301
128 [    0.011382] Mount-cache hash table entries: 2048 (order: 1, 8192
    bytes, linear)
129 [    0.011411] Mountpoint-cache hash table entries: 2048 (order: 1,
    8192 bytes, linear)
130 [    0.037101] ASID allocator using 9 bits (512 entries)
131 [    0.038320] rcu: Hierarchical SRCU implementation.
132 [    0.038347] rcu:   Max phase no-delay instances is 1000.
133 [    0.040043] EFI services will not be available.
134 [    0.040610] smp: Bringing up secondary CPUs ...
135 [    0.041155] smp: Brought up 1 node, 1 CPU
136 [    0.044836] Memory: 999604K/1048576K available (11170K kernel code,
    9589K rwdata, 4096K rodata, 4333K init, 307K bss, 46648K reserved, 0
    K cma-reserved)
137 [    0.051618] devtmpfs: initialized
138 [    0.059263] clocksource: jiffies: mask: 0xffffffff max_cycles: 0
```

```
    xffffffff, max_idle_ns: 7645041785100000 ns
139 [    0.059368] posixtimers hash table entries: 512 (order: 0, 4096
    bytes, linear)
140 [    0.059530] futex hash table entries: 256 (16384 bytes on 1 NUMA
    nodes, total 16 KiB, linear).
141 [    0.067130] DMI not present or invalid.
142 [    0.068525] NET: Registered PF_NETLINK/PF_ROUTE protocol family
143 [    0.072196] DMA: preallocated 128 KiB GFP_KERNEL pool for atomic
    allocations
144 [    0.072437] audit: initializing netlink subsys (disabled)
145 [    0.074912] thermal_sys: Registered thermal governor 'step_wise'
146 [    0.075512] audit: type=2000 audit(0.064:1): state=initialized
    audit_enabled=0 res=1
147 [    0.075989] cpuidle: using governor menu
148 [    0.101744] cpu0: Ratio of byte access time to unaligned word access
     is 3.73, unaligned accesses are fast
149 [    0.126989] HugeTLB: registered 4.00 MiB page size, pre-allocated 0
    pages
150 [    0.127012] HugeTLB: 0 KiB vmemmap can be freed for a 4.00 MiB page
151 [    0.129328] iommu: Default domain type: Translated
152 [    0.129355] iommu: DMA domain TLB invalidation policy: strict mode
153 [    0.130525] SCSI subsystem initialized
154 [    0.135679] usbcore: registered new interface driver usbfs
155 [    0.135903] usbcore: registered new interface driver hub
156 [    0.135993] usbcore: registered new device driver usb
157 [    0.137341] Advanced Linux Sound Architecture Driver Initialized.
158 [    0.155280] vgaarb: loaded
159 [    0.161286] clocksource: Switched to clocksource riscv_clocksource
160 [    0.186353] NET: Registered PF_INET protocol family
161 [    0.187047] IP idents hash table entries: 16384 (order: 5, 131072
    bytes, linear)
162 [    0.191261] tcp_listen_portaddr_hash hash table entries: 512 (order:
     0, 4096 bytes, linear)
163 [    0.191303] Table-perturb hash table entries: 65536 (order: 6,
    262144 bytes, linear)
164 [    0.191348] TCP established hash table entries: 8192 (order: 3,
    32768 bytes, linear)
165 [    0.191507] TCP bind hash table entries: 8192 (order: 5, 131072
    bytes, linear)
```

```
[    0.191640] TCP: Hash tables configured (established 8192 bind 8192)
[    0.192229] UDP hash table entries: 512 (order: 3, 28672 bytes,
linear)
[    0.192372] UDP-Lite hash table entries: 512 (order: 3, 28672 bytes,
 linear)
[    0.193270] NET: Registered PF_UNIX/PF_LOCAL protocol family
[    0.203421] RPC: Registered named UNIX socket transport module.
[    0.203463] RPC: Registered udp transport module.
[    0.203471] RPC: Registered tcp transport module.
[    0.203477] RPC: Registered tcp-with-tls transport module.
[    0.203482] RPC: Registered tcp NFSv4.1 backchannel transport module
.
[    0.203596] PCI: CLS 0 bytes, default 64
[    0.208799] Unpacking initramfs...
[    0.219108] workingset: timestamp_bits=14 max_order=18 bucket_order
=4
[    0.226435] NFS: Registering the id_resolver key type
[    0.226664] Key type id_resolver registered
[    0.226691] Key type id_legacy registered
[    0.226905] nfs4filelayout_init: NFSv4 File Layout Driver
Registering...
[    0.226972] nfs4flexfilelayout_init: NFSv4 Flexfile Layout Driver
Registering...
[    0.233400] 9p: Installing v9fs 9p2000 file system support
[    0.234685] Block layer SCSI generic (bsg) driver version 0.4 loaded
 (major 246)
[    0.242438] io scheduler mq-deadline registered
[    0.242504] io scheduler kyber registered
[    0.242683] io scheduler bfq registered
[    0.252787] riscv-plic: plic@c000000: mapped 95 interrupts with 1
handlers for 2 contexts.
[    0.254986] pci-host-generic 30000000.pci: host bridge /soc/
pci@30000000 ranges:
[    0.255506] pci-host-generic 30000000.pci:       IO 0x0003000000..0
x000300ffff -> 0x0000000000
[    0.255882] pci-host-generic 30000000.pci:      MEM 0x0040000000..0
x007fffffff -> 0x0040000000
[    0.255940] pci-host-generic 30000000.pci:      MEM 0x0300000000..0
x03ffffffff -> 0x0300000000
```

```
193 [    0.256480] pci-host-generic 30000000.pci: ECAM at [mem 0x30000000-0
    x3fffffff] for [bus 00-ff]
194 [    0.268647] pci-host-generic 30000000.pci: PCI host bridge to bus
    0000:00
195 [    0.268828] pci_bus 0000:00: root bus resource [bus 00-ff]
196 [    0.268883] pci_bus 0000:00: root bus resource [io  0x0000-0xffff]
197 [    0.268918] pci_bus 0000:00: root bus resource [mem 0x40000000-0
    x7fffffff]
198 [    0.269841] pci 0000:00:00.0: [1b36:0008] type 00 class 0x060000
    conventional PCI endpoint
199 [    0.273371] pci_bus 0000:00: resource 4 [io  0x0000-0xffff]
200 [    0.273397] pci_bus 0000:00: resource 5 [mem 0x40000000-0x7fffffff]
201 [    0.280566] Freeing initrd memory: 2600K
202 [    0.382637] Serial: 8250/16550 driver, 4 ports, IRQ sharing disabled
203 [    0.400677] printk: legacy console [ttyS0] disabled
204 [    0.406975] 10000000.serial: ttyS0 at MMIO 0x10000000 (irq = 12,
    base_baud = 230400) is a 16550A
205 [    0.408268] printk: legacy console [ttyS0] enabled
206 [    0.476637] loop: module loaded
207 [    0.477414] virtio_blk virtio0: 1/0/0 default/read/poll queues
208 [    0.482602] virtio_blk virtio0: [vda] 20971520 512-byte logical
    blocks (10.7 GB/10.0 GiB)
209 [    0.504446] e1000e: Intel(R) PRO/1000 Network Driver
210 [    0.504587] e1000e: Copyright(c) 1999 - 2015 Intel Corporation.
211 [    0.514164] usbcore: registered new interface driver uas
212 [    0.514405] usbcore: registered new interface driver usb-storage
213 [    0.515098] mousedev: PS/2 mouse device common for all mice
214 [    0.523836] goldfish_rtc 101000.rtc: registered as rtc0
215 [    0.524512] goldfish_rtc 101000.rtc: setting system clock to
    2026-01-10T20:01:38 UTC (1768075298)
216 [    0.530952] sdhci: Secure Digital Host Controller Interface driver
217 [    0.531097] sdhci: Copyright(c) Pierre Ossman
218 [    0.531226] Synopsys Designware Multimedia Card Interface Driver
219 [    0.531589] sdhci-pltfm: SDHCI platform and OF driver helper
220 [    0.532923] usbcore: registered new interface driver usbhid
221 [    0.533062] usbhid: USB HID core driver
222 [    0.533556] riscv-pmu-sbi: SBI PMU extension is available
223 [    0.534004] riscv-pmu-sbi: 16 firmware and 18 hardware counters
224 [    0.534135] riscv-pmu-sbi: Perf sampling/filtering is not supported
```

```
   as sscof extension is not available
[    0.547347] NET: Registered PF_INET6 protocol family
[    0.558858] Segment Routing with IPv6
[    0.559047] In-situ OAM (IOAM) with IPv6
[    0.559430] sit: IPv6, IPv4 and MPLS over IPv4 tunneling driver
[    0.561518] NET: Registered PF_PACKET protocol family
[    0.562017] 9pnet: Installing 9P2000 support
[    0.562336] Key type dns_resolver registered
[    0.873438] clk: Disabling unused clocks
[    0.873707] PM: genpd: Disabling unused power domains
[    0.873900] ALSA device list:
[    0.873991]   No soundcards found.
[    0.925460] Freeing unused kernel image (initmem) memory: 4332K
[    0.926058] Run /init as init process
/bin/sh: cant access tty; job control turned off
~ # ls
bin      etc      init     mnt      root     sys      usr
dev      home     linuxrc  proc     sbin     tmp
~ # cd home
/home # ls
from-c  main
/home # ./main
Hello world!
/home #
```