Universidade Federal do Espírito Santo
Colegiado do Curso de Engenharia de Computação
Trabalho de Conclusão de Curso I
Opção 2: Desenvolvimento de um Protótipo

RELATÓRIO DE RESULTADOS PARCIAIS

Estudante: Carlos Daniel Albertino Vieira
Orientador: Rodolfo da Silva Villaca

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. Atualmente, a RISC-V International corresponde a organização internacional que reúne as especificações da arquitetura a toda comunidade, o que impulsiona sua difusão e adoção.

A especificação da arquitetura inclui seu *Instruction Architecture Set (ISA)*, que consiste no conjunto de instruções suportadas e como essas instruções estão organizadas a nível binário na memória. Atualmente, a RISC-V International reúne as especificações técnicas da ISA e de extensões em seu website.

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 capaz de simular e emular a arquitetura do RISC-V 32 bits, com o objetivo de prototipar uma solução que permita inspecionar o estado de execução do programa em diferentes estágios e de gerar programas funcionais em formatos de representação em memória e também no formato ELF, sendo capaz de rodar em máquinas virtuais de RISC-V 32 bits.

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.

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. 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 simulador e emulador de RISC-V 32 bits e que, diferente dos demais projetos, consiga produzir arquivos no formato ELF que sejam funcionais em máquinas virtuais de RISC-V 32 bits.

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

1.2. OBJETIVO

O presente projeto tem como objetivo principal construir um programa que seja capaz de simular e emular programas escritos em um arquivo assembly para a arquitetura RISC-V 32 bits, a fim de representar de forma fidedigna as mudanças de estado ligadas a registradores e memória relacionados à execução do programa e ser capaz de inspecionar tais estágios. O projeto também traz como objetivo fornecer uma visualização gráfica do processo acima descrito na forma de uma interface gráfica e de uma interface em linha de comando. Por fim, outro objetivo a ser alcançado é o de permitir a tradução e exportação de um código assembly RISC-V para o formato ELF, de modo que possa ser executado em um ambiente virtual real e também possa ser utilizado como entrada para o emulador.

2. DESCRIÇÃO DO PROTÓTIPO

2.1. RESUMO DA PROPOSTA

A proposta consiste na implementação de um programa capaz de: traduzir um arquivo assembly para formatos intermediários em memória e no formato ELF, decodificar arquivos no formato ELF e executar as instruções lidas diretamente do binário em um ambiente virtual que conta com registradores e um modelo de memória simples (sem virtualização, paginamento e alinhamento).

A tradução do código assembly inclui a extensão básica da linguagem (RVI32), suporte às seções de texto, data e bss, declaração de variáveis e chamada de funções. Inicialmente, espera-se que seja suportado o uso de operações tais como:

1. Operações aritméticas e lógicas com inteiros
2. Saltos condicionais
3. Leitura e escrita de memória
4. Carregamento de valores imediatos
5. Pseudo instruções comuns (como RET, LA, LI e outras)
6. Declaração de variáveis (WORD, BYTE, ASCII)
7. Declaração de labels

2.2. IMPLEMENTAÇÃO

A implementação desse projeto está disponível no seguinte repositório do Github: https://github.com/cdavieira/rustv.

O projeto está sendo desenvolvido em duas frentes:

1. Conversão de código assembly em binário
2. Leitura e execução do formato ELF em um ambiente virtual

A conversão do código assembly em binário (formato ELF) visa obedecer a especificação da ISA do RISC-V 32 bits e oferecer suporte a implementação de diferentes extensões da arquitetura futuramente. De imediato, traz-se suporte a extensão core/básica da arquitetura (RVI32), que inclui operações básicas relacionadas à inteiros, saltos condicionais e outras primitivas.

A leitura e execução do formato ELF inclui o parse do binário e execução desse diretamente em um ambiente virtual simples, que conta com registradores, memória e um modo de execução fetch-decode típico de CPUs. Esse consiste na parte de emulação/simulação do projeto.

De modo geral, a ideia do projeto seria de implementar um modelo tal como:

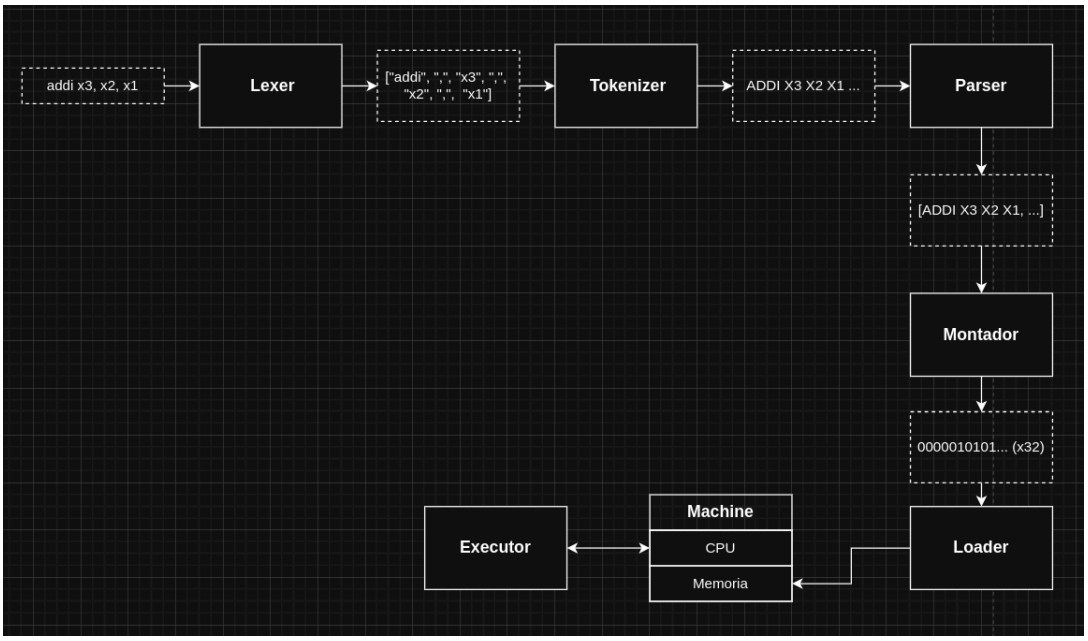


Imagem 1 - Modelo de entidades a serem implementadas para construção do simulador e emulador

No modelo acima, as instruções assembly são processadas inicialmente pelo Lexer, que aceita o alfabeto básico da linguagem assembly e devolve uma stream de palavras. A entidade Tokenizer é então responsável por classificar as palavras e transformá-las em uma stream de tokens como opcodes, registradores e labels. A entidade Parser é então responsável por agrupar os tokens e organizá-los de modo que possam ser recepcionados pelo Montador, para serem transformados em binário e opcionalmente gravados em um formato como o ELF. A partir disso, a entidade Loader se encarrega de decodificar o arquivo ELF e carregar as instruções no ambiente virtual de execução (Machine), que é então controlada opcionalmente por um executor, tal como um debugger ou uma entidade que possa inspecionar o estado da máquina em execução.

2.2.1 Tecnologias usadas

Estão sendo utilizadas as seguintes tecnologias:

- Rust
- QEMU
- RISC-V Toolchain

O projeto está sendo desenvolvido na linguagem Rust, que é uma linguagem de programação moderna, com bom suporte a testes unitários e que permite um design de implementação por meio de contratos/interfaces a serem implementados. Nesse sentido, o projeto opta por um design orientado a teste, a fim de permitir que as diferentes etapas associadas a simulação e emulação possam ser testadas e validadas separadamente. A linguagem também oferece suporte a construção de arquivos ELF por meio da biblioteca 'object', que oferece uma interface amigável para produção de arquivos ELF que atendem às especificações do formato, de forma a obedecer ao layout de seções, símbolos, diretivas e outros pontos relevantes a esse tipo de arquivo.

A validação dos arquivos ELF produzidos foi feita por meio de uma máquina virtual RISC-V 32 bits, que utiliza o programa QEMU para emulação do hardware de uma máquina RISC-V. A máquina virtual executa a versão mais recente do kernel do Linux (compilada para a arquitetura do RISC-V 32 bits) e possui todo o utilitário de linha de comando disponível no busybox (também compilado para a arquitetura RISC-V 32 bits). Por meio do ambiente em questão, é possível rodar os executáveis ELF criados no projeto e de testar se realmente funcionam/podem ser executados em um ambiente emulado real.

O repositório Toolchain RISCV oferece uma forma de compilar um conjunto de utilitários de linha de comando (como compiladores, assemblers, debuggers, linkers e outros) que podem ser utilizados em computadores de arquiteturas diversas (x86-64, amd64 e outras) para cross compilar programas a serem executados em um ambiente alvo RISC-V(virtual ou não). O toolchain em questão permite validar a corretude do projeto na conversão das instruções assembly em binário, de modo a observar se o resultado final é compatível àquele de utilitários padrão GNU, como o *as*. O toolchain também conta com o linker *ld* (utilitário GNU) que foi utilizado para linkar os arquivos objetos finais produzidos pelo programa em um formato compatível para ser rodado na máquina virtual RISCV (tal como o formato a.out). O toolchain também conta com o debugger gdb, que pode ser usado para depurar programas que tenham sido compilados com suporte a debug.

2.2.2 Conversão de assembly em binário

A conversão do assembly em binário segue o modelo proposto na imagem 1. Atualmente, as entidades que representam o Lexer, Tokenizer, Parser e Montador se encontram implementadas, funcionais e passíveis de serem estendidas para suportar novas extensões da arquitetura RISC-V, pseudo instruções e diretivas de assembly/linkagem.

2.2.2.1 Implementação do Lexer

A entidade Lexer foi implementada no formato de um autômato finito para reconhecer uma linguagem que pode ser descrita por meio de expressões regulares. Nesse contexto, a entrada do Lexer corresponde ao código assembly, que é lido caracter por caracter. Uma vez lido um caractere, esse é classificado em uma de 7 classes. A depender da classe escolhida, o programa consome a entrada até que se reconheça a expressão regular associada àquela classe. Ao final dessa etapa, o código assembly é convertido em um stream de palavras/strings, que correspondem ao alfabeto da linguagem assembly.

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, labels)	(‘.’ | ‘:’ | ‘_’ | [a-zA-Z0-9])*
UNIT	Pontuação	‘,’ | ‘(‘ | ‘)’
IGNORE	Caracteres a serem ignorados (espaço em branco)	\n | \t | \b
AMBIGUOUS	Caracteres que dependem dos próximos a serem lidos para aceite	‘+’ | ‘-’

Tabela 1 - Classes reconhecidas pelo Lexer e suas expressões regulares equivalentes

Como o papel do Lexer consiste simplesmente em aceitar palavras pertencentes à linguagem sem a necessidade de regras e classificações elaboradas (etapa feita pela entidade Tokenizer), escolheu-se simplificar algumas das expressões regulares, a fim de facilitar a implementação do autômato. 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.

2.2.2.2 Implementação do Tokenizer

A entidade do Tokenizer tem como responsabilidade classificar os termos identificados pelo Lexer de acordo com um tipo de Token (a ser definido também pelo implementador) e devolver como saída uma stream desses Tokens. Para tratar da linguagem reconhecida pelo Lexer, a implementação vigente do Tokenizer busca categorizar seus termos em 1 de 14 classes:

CLASSE	DESCRIÇÃO
OP	Instruções/opcodes de alguma extensão RISC-V
PSEUDO	Pseudo instruções (instruções compostas de outras instruções)
DIRECTIVE	Diretivas de compilação/ligação/assembly
REG	Nomes de registradores
NAME	Nomes de identificadores
STR	Strings literais
LABEL	Declarações de label
NUMBER	Números
SECTION	Nomes de seções
PLUS	Operador de soma
MINUS	Operador de subtração
LPAR	‘(‘
RPAR	‘)‘
COMMA	‘,‘

Tabela 2 - Classes usadas pela implementação atual de Tokenizer

Além de classificar os termos da linguagem de acordo com uma dessas categorias e também de possivelmente apontar erros, o Tokenizer também pode opcionalmente associar um valor semântico a esses tokens. A exemplo de números, a string reconhecida enquanto número (tanto decimal como hexadecimal) pode ser convertida em um valor decimal a ser armazenado no token. O mesmo processo ocorre para outros tipos, tal como strings e registradores.

O valor semântico associado não precisa necessariamente ser de um tipo trivial tal como inteiros ou strings, mas também pode ser associado a tipos compostos de dados.

Esse é o caso dos valores pertencentes às classes de opcodes, pseudo instruções e diretivas, que podem ser associados a qualquer tipo de dado, desde que esse implemente um contrato específico relacionado a essas classes. Tais contratos garantem que qualquer tipo associado a opcodes, pseudo instruções e diretivas apresentarão sempre uma mesma interface e comportamento, o que permite que novos termos desses tipos sejam adicionados e suportados futuramente pelo projeto (como possíveis novas extensões da arquitetura RISC-V). Sob essa luz, pode-se também entender que um dos papéis do Tokenizer seria o de relacionar termos puramente simbólicos (palavras pertencentes a linguagem assembly) a funcionalidades suportadas, implementadas no código e que apresentam um interface bem definida de como serem usadas.

No caso do tipo associado a opcodes, esse deve implementar o contrato de *Extension*, o qual preconiza que todo opcode deve:

1) Declarar qual formato de instrução está associado (assim como previsto na especificação do RISC-V, ex: formatos R, B, I, J e outros)
2) Declarar sua sintaxe, ou seja, quantos e quais argumentos espera receber.

A exemplo da extensão RV32I, essa consiste num conjunto de 40 instruções, as quais foram implementadas como segue:

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

impl Extension for RV32I {
    fn get_instruction_format(&self, rs1: u32, rs2: u32, rd: u32, imm:
i32) -> InstructionFormat  {
        match self {
            RV32I::ADD   => InstructionFormat::R { ... },
            RV32I::LUI   => InstructionFormat::U { ... },
            RV32I::AUIPC => InstructionFormat::U { ... },
            RV32I::JAL   => InstructionFormat::J { ... },
            RV32I::JALR  => InstructionFormat::I { ... },
            RV32I::ECALL => InstructionFormat::I { ... },
            RV32I::LW    => InstructionFormat::I { ... },
            RV32I::SW    => InstructionFormat::S { ... },
        }
    }

    fn get_calling_syntax(&self) -> ArgSyntax {
        match self {
            RV32I::ADD   => ArgSyntax::N3(ArgName::RD, ArgName::RS1,
ArgName::RS2),
            RV32I::LUI   => ArgSyntax::N2(ArgName::RD, ArgName::IMM),
            RV32I::AUIPC => ArgSyntax::N2(ArgName::RD, ArgName::IMM),
```

```
            RV32I::JAL   => ArgSyntax::N2(ArgName::RD, ArgName::OFF),
            RV32I::JALR  => ArgSyntax::N3(ArgName::RD, ArgName::RS1,
ArgName::OFF),
            RV32I::ECALL => ArgSyntax::N0,
            RV32I::LW    => ArgSyntax::N3(ArgName::RD, ArgName::OFF,
ArgName::RS1),
            RV32I::SW    => ArgSyntax::N3(ArgName::RS2, ArgName::OFF,
ArgName::RS1),
        }
    }
}
```

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 das instruções, 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:

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

impl Pseudo for PseudoInstruction {
    fn translate(&self, args: Vec<ArgValue>) -> Vec<(Box<dyn Extension>,
Vec<ArgValue>)>  {
        ...
        match self {
            //ret, becomes:
            //jalr x0 x1 0
            PseudoInstruction::RET => {
                let mut args = Vec::new();
                args.push(ArgValue::REGISTER(Register::X0));
                args.push(ArgValue::REGISTER(Register::X1));
                args.push(ArgValue::NUMBER(0));
                response.push((Box::new(RV32I::JALR), args));
            },
        }
        ...
```

```
    }
}
```

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 assembly 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, portanto apenas retornam uma sequência vazia de bytes.

Por fim, uma vez implementados os contratos vistos acima, instruções reais/pseudo instruções e diretivas podem ser associadas como valores semânticos a tokens, ao serem reconhecidos pelo Tokenizer. Essa completa separação entre implementação e uso de tais tipos desonera o Tokenizer de ter que implementá-los. Na prática, o suporte a novas extensões, pseudo instruções e diretivas pode ser completamente compartimentado por outras unidades do código, de forma a evitar que essa seja uma preocupação do Tokenizer.

2.2.2.3 Implementação do Parser

A entidade Parser tem como objetivo agrupar e organizar a sequência de Tokens provenientes do Tokenizer na forma de uma saída que possa ser processada pela próxima entidade (Assembler). A implementação atual do Parser busca desempenhar esse papel através de uma sequência de etapas bem definidas:

1. Agrupar os tokens em grupos de instrução
2. Para cada grupo de instrução, agrupar todos os argumentos ali presentes em um subgrupo (argumentos da instrução)
3. Reescrever as pseudo instruções enquanto um conjunto de instruções reais e outros argumentos
4. Tratar as diretivas de código, de modo a inserir bytes crus ou não a depender do tipo de diretiva
    a. Remover no processo as diretivas específicas de ligação e compilação e guardá-las para serem usadas pelo Assembler
5. Agrupar os grupos de instrução em seções
6. Fundir seções de mesmo tipo em seções únicas
7. Gerar o endereço do início de cada seção (respeitando alinhamento)
8. Gerar o endereço relativo de cada instrução com relação a sua seção base
9. Gerar a tabela de seções
10. Gerar a tabela de símbolos
11. Substituir as referências dos símbolos pelos seus respectivos endereços
12. Converter todos os argumentos das seções em números

Ao final desse processo, Obtém-se 4 subprodutos:

1. Conjunto de seções endereçados, com suas respectivas instruções (também endereçadas) e todos os símbolos já resolvidos
2. A tabela de seções
3. A tabela de símbolos
4. Metadados (diretivas de compilação + ligação)

Os subprodutos obtidos da entidade do Parser podem ser então empregados pelo Assembler para diferentes propósitos, como execução em memória ou exportação em algum formato de objeto (como ELF ou a.out).

Atualmente, algumas das etapas da entidade Parser estão sob observação e podem ser modificadas. Uma dessas etapas é a etapa de resolução de símbolos (Passo 11), que possivelmente será movida para ser feita pela entidade do Assembler, a fim de postergar ao máximo o processo de definição dos endereços finais a serem usados. Essa escolha se dá pelo fato de que a forma de endereçamento a ser usada no corpo das instruções pode ser diferente, a depender do propósito final a ser decidido pelo Assembler. Caso o objetivo seja rodar as instruções em memória, então é de interesse do Assembler resolver os símbolos pendentes e substituir as devidas referências feitas, para que as instruções sejam executadas com os endereços resolvidos/numéricos. No entanto, isso pode não ser necessário caso o objetivo seja exportar o programa em algum formato tal como ELF.

Outro ponto ainda a ser resolvido consiste no referenciamento de labels ou símbolos situados em seções externas à seção onde foram utilizados. Esse é um problema clássico para permitir que variáveis situadas na seção de data possam ser referenciadas na seção de texto do código. Atualmente, esse problema está parcialmente resolvido, no entanto é necessário generalizar a solução, para que consiga lidar com seções que estejam muito afastadas entre si.

2.2.2.4 Implementação do Assembler

O Assembler corresponde à entidade responsável por processar a saída do Parser. A implementação atual da entidade do Assembler busca salvar a saída em um arquivo objeto do formato ELF, que pode então ser ligado por meio do linker *ld* para ser executado em uma máquina virtual RISC-V emulada.

A construção do arquivo no formato ELF para RISC-V está sendo feita por meio da crate/biblioteca *object*, que oferece uma API familiar para manipular seções e símbolos a serem usados na escrita do objeto final. Além disso, diretivas de compilação e ligação, tais como a visibilidade a ser associada a certos labels, podem ser resolvidas também por meio desta biblioteca.

O processo de escrita do binário se inicia com a definição do ponto de entrada do programa, que por padrão está associado ao label *_start*. Feito isso, os demais simbolos presentes na tabela de símbolos são adicionados, seguido dos bytes da seção de dados. As instruções da seção de texto são então codificadas para o formato previsto pelo seu formato de instrução e são adicionadas uma a uma no arquivo ELF. A conversão das instruções para sua forma binária é feita por meio da seguinte função utilitária, que faz uso

do contrato *Extension* definido para todos opcodes, o que garante o funcionamento da próxima etapa para todos os opcodes suportados no projeto:

```
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);
    inst.get_instruction_format(rs1, rs2, rd, imm).encode()
}
```

A função funciona em 3 etapas:

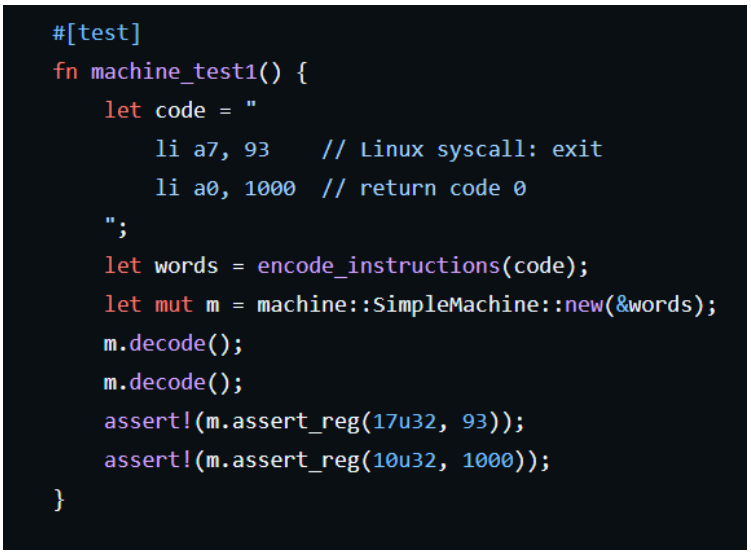
1. Identificação dos valores a serem passados como argumento para a instrução de acordo com sua sintaxe (função *get_args()*)
2. Obtenção do formato da instrução (método *get_instruction_format()*)
3. Codificação em binário (método *encode()*, definido para todos diferentes formatos de instrução)

Ao final desse processo, um único arquivo final ELF é escrito, contendo toda seção de código, dados, tabela de símbolos e outras informações necessárias para ligação e posteriormente execução.

Atualmente, a entidade Assembler não é capaz de processar as informações ainda em memória provenientes do Parser, de modo que possam ser executadas no contexto virtual de máquina feito, que conta com registradores e uma memória *flat* simples. No entanto, assim como discutido na seção de implementação do Parser, esse processo futuramente poderá acontecer, o que também necessitará da implementação de uma unidade Executora responsável por dirigir o estado da execução da Máquina.

2.2.3 Leitura e execução do formato ELF em um ambiente virtual

O ambiente virtual para execução do programa conta com 32 registradores de uso geral, 1 registrador PC e um modelo de memória simples (sem virtualização, paginamento e alinhamento), no qual as instruções são lidas uma a uma da memória utilizando o endereço armazenado no registrador PC e são executadas, num modelo de execução fetch-decode típico de CPUs. Esse processo pode ser observado em um dos testes feitos no projeto para assegurar o funcionamento da execução na máquina virtual:



```rust
#[test]
fn machine_test1() {
    let code = "
        li a7, 93    // Linux syscall: exit
        li a0, 1000  // return code 0
    ";
    let words = encode_instructions(code);
    let mut m = machine::SimpleMachine::new(&words);
    m.decode();
    m.decode();
    assert!(m.assert_reg(17u32, 93));
    assert!(m.assert_reg(10u32, 1000));
}
```

Imagem 2 - Código de teste feito para inspecionar a escrita dos registradores da máquina

A imagem 2 traz um exemplo de código, no qual duas pseudo instruções são convertidas em palavras (cada palavra contendo 4 bytes), que então são carregadas na memória da máquina virtual. Cada chamada ao método *decode()* faz o ambiente virtual executar uma instrução por vez, de modo a executar as instruções sequencialmente nesse processo. Esse modelo de funcionamento permitirá que implementações da unidade Executora tenham controle da execução da máquina virtual, sendo ao mesmo tempo capazes de examinar seu estado durante a execução. Ao final do teste, verifica-se se os valores contidos nos registradores em questão batem com aqueles esperados.

A forma como o método *decode()* funciona é o seguinte:

1. A máquina utiliza o valor armazenado no registrador PC como offset e lê a palavra (equivalente a 4 bytes) contida neste endereço em memória.
2. A máquina avança o registrador PC em 1 instrução
3. A palavra é decodificada em um dos formatos de instrução conhecidos (R, I, B, J, S e U)
4. Com o formato de instrução, analisa-se os campos que permitem identificar qual instrução deve ser executada (Ex: ADD, SUB, BEQ, LW, SW, …) e quais argumentos utilizar
5. É executada a ação correspondente daquela instrução no ambiente de execução

Um excerto da implementação utilizada para esse método pode ser visto a seguir:

```rust
fn decode(&mut self) -> ()  {
     let word = self.fetch();

     self.jump(1usize);

     match InstructionFormat::decode(word) {
         InstructionFormat::R { funct7, rs2, rs1, funct3, rd, opcode }
```

```
=> {
            match (funct7, funct3) {
                (0b0, 0b0) => { //ADD
                    let v1 = self.cpu.read(rs1 as usize);
                    let v2 = self.cpu.read(rs2 as usize);
                    self.cpu.write(rd as usize, v1 + v2);
                },
                 ...
            }
        },
        InstructionFormat::I { imm, rs1, funct3, rd, opcode } => {
            ...
        },
        ...
    }
}
```

Quanto às limitações atuais dessa frente, vale ressaltar que a leitura e execução do formato ELF funciona apenas para códigos assembly cuja única seção é a de texto e códigos que não fazem uso de símbolos, tal como o seguinte código:

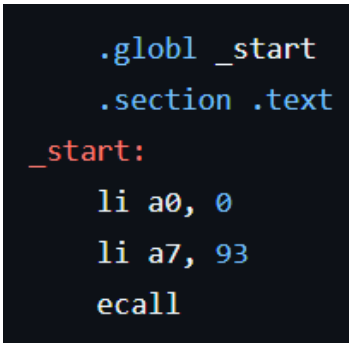


Imagem 3 - Código assembly que executa a chamada de sistema *exit(0)* no Linux

É importante salientar que o código mostrado na imagem 3 não referencia diretamente o label *_start* durante sua execução, pois esse é um label padrão já adicionado à tabela de símbolos do arquivo ELF para definir o começo do programa.

As limitações em questão se devem ao fato da implementação da entidade Loader não ter evoluído o suficiente para lidar com diferentes seções (como as de data e bss) e para fazer uso das tabelas presentes no arquivo ELF (tabela de símbolos, tabela de strings, tabela de seções) durante o carregamento das instruções no modelo de Máquina Virtual proposto.

Além disso, ainda não há uma entidade que implemente formalmente a entidade Executora prevista, que seria responsável por inspecionar e depurar arbitrariamente o estado da máquina durante seu funcionamento.

3. PRÓXIMAS ETAPAS

Atualmente a frente relacionada a conversão do código assembly em formato binário se encontra mais bem desenvolvida do que àquela relacionada a decodificação e execução de instruções no ambiente virtual.

Quanto à frente de decodificação e execução de instruções, essa ainda necessita de uma implementação de Loader que seja capaz de carregar dados referentes a diferentes seções típicas do assembly (tal como .DATA e .BSS) e de lidar possivelmente com a resolução de símbolos extraídos das tabelas presentes no arquivo ELF. Também é necessário implementar o gerenciamento de chamada de funções, que necessitará de um esquema de memória capaz de suportar alocações na stack. Não obstante, a entidade Executora deve ser implementada, possivelmente de modo a permitir que o programa ofereça suporte a debuggers externos, tal como o próprio GDB, por meio do protocolo GDB Remote Debugger, o qual permite servidores compatíveis a controlar o estado do ambiente virtual em análise no GDB.

Quanto à frente de conversão de código assembly em formato binário, é necessário permitir que a entidade do Lexer armazene a posição associada a cada palavra reconhecida no código assembly, a fim de facilitar posteriormente a detecção de possíveis erros a serem reportados, bem como melhorar o suporte a detecção de erros de sintaxe pela entidade do Tokenizer. É importante, também, repensar o momento de execução da etapa de resolução de símbolos sendo feita atualmente no Parser, a fim de que não interfira com o processo de escrita do arquivo ELF pela entidade do Assembler. Por fim, é necessário desenvolver a parte gráfica da aplicação e permitir que um usuário consiga interagir com as diferentes partes do programa de forma facilitada.

4. REFERÊNCIAS

ORGANIZATION, Risc-v. Especificações ISA - RISCV. RISC-V Organization, 2025. Disponível em: https://riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications. Acesso em: 02 ago. 2025

MSYKSPHINZ, Msyksphinz. ISA Specification. Github, 2025. Disponível em: https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html. Acesso em: 01 ago. 2025.

HAOZIWAN. *Interactive RISC-V Simulator*. Disponível em: https://github.com/Haoziwan/Interactive-RISC-V-Simulator. Acesso em: 05 mai. 2025.

MORTBOPET. *Ripes*. Disponível em: https://github.com/mortbopet/Ripes. Acesso em: 24 mai. 2025.

THETHIRDONE. *RARS*. Disponível em: https://github.com/TheThirdOne/rars. Acesso em: 01 jun. 2025.