Revista Ibero-Americana de Humanidades, Ciências e Educação — REASE

doi.org/10.51891/rease.v12i1.23857

# STAR: UMA LINGUAGEM ASSEMBLY E MÁQUINA VIRTUAL EXTENSÍVEL DE 16 BITS

**João Gabriel Freitas Cavalcante**[1]
**Ivan Saraiva Silva**[2]
**Maryane Francisca Araujo de Freitas Cavalcante**[3]

RESUMO: O presente artigo tem como objetivo apresentar a linguagem assembly Star e a Star Virtual Machine, desenvolvidas por João G. F. Cavalcante, como um ecossistema educacional destinado ao ensino de arquitetura de computadores e programação de baixo nível. Os métodos adotados envolvem a análise comparativa de arquiteturas clássicas, como MIPS, RISC-V e 6502, a definição de uma ISA compacta de 16 bits e a implementação de um compilador e máquina virtual em Rust. Técnicas como divisão alto/baixo, extensão de opcode e uso de pseudo-instruções foram incorporadas para ampliar a expressividade dentro das limitações do formato. A arquitetura Harvard e o modelo Big-Endian foram escolhidos para facilitar a visualização da organização interna da memória e das instruções. Como resultados, o projeto oferece uma linguagem simples, flexível e portátil, associada a um ambiente interativo que permite executar e depurar programas em tempo real, exibindo registradores, memória e fluxo de instruções. Conclui-se que o ecossistema Star constitui uma ferramenta eficaz para ensino, simulação e experimentação prática, contribuindo para a compreensão aprofundada de princípios fundamentais de arquitetura, compiladores e execução em nível de máquina.

**Palavras-chave:** Máquinas Virtuais. Linguagem Assembly. Arquitetura de Computadores. Rust. ISA de 16 Bits.



## INTRODUÇÃO

A linguagem Star e Star Virtual Machine constituem um ecossistema didático integrado destinado ao estudo e à experimentação de conceitos fundamentais de arquitetura de computadores e programação de baixo nível. A linguagem Star define uma ISA de 16 bits, simples e acessível, enquanto a Star Virtual Machine fornece o ambiente de execução, depuração e desenvolvimento responsável por interpretar o código gerado. Essa integração permite que o estudante explore, em um único ambiente, desde os princípios elementares de organização de instruções até o funcionamento detalhado de uma máquina virtual.

O desenvolvimento da linguagem Star e de sua máquina virtual decorre da necessidade de um ambiente educacional que combina simplicidade conceitual e profundidade técnica. A

[1]Graduando Bacharelado em Ciências da Computação – Universidade Federal do Piauí (UFPI)
[2]Doutor em Informática – Universidade Federal do Piauí (UFPI).
[3]Mestranda em Propriedade Intelectual – Instituto Federal do Piauí (IFPI).

proposta atende especialmente à formação de estudantes e entusiastas que desejam compreender, de forma prática, elementos como registradores, códigos de operação, extensão de opcode, modelos de pilha, manipulação de memória e controle de fluxo. Esses elementos foram planejados previamente e gradualmente incorporados ao núcleo da Star Virtual Machine, possibilitando uma aprendizagem ativa, interativa e incremental.

A linguagem Star foi projetada para ser acessível, com sintaxe direta e intuitiva, reduzindo barreiras de entrada para quem inicia no estudo de linguagens de montagem. Essa simplicidade não compromete a expressividade: ao contrário, orienta o aprendiz a focar profundamente nos princípios da arquitetura computacional, evitando que aspectos sintáticos complexos desviem a atenção dos mecanismos fundamentais da execução em baixo nível.

A Star Virtual Machine é implementada em Rust, uma linguagem moderna que oferece segurança, desempenho e controle preciso do uso de memória. A adoção de Rust não apenas reforça a robustez do ambiente Star, como também introduz boas práticas de engenharia de software, essenciais quando se trabalha com estruturas sensíveis como registradores, buffers e modelos de execução determinística. Assim, Rust contribui diretamente para a confiabilidade e a estabilidade da máquina virtual.



O presente trabalho tem como objetivo apresentar a linguagem assembly Star, detalhando sua arquitetura interna, suas instruções e seu processo de construção. Além disso, discute-se o funcionamento completo da Star Virtual Machine, incluindo o compilador, seus módulos de análise e geração de código, e a máquina virtual propriamente dita. São tratados conceitos de construção de linguagens, compiladores e VMs, destacando decisões de projeto, técnicas de implementação e aspectos de eficiência e segurança.

Outro objetivo deste estudo é demonstrar como uma arquitetura de 16 bits pode ser implementada de forma prática e funcional. A Star Virtual Machine simula fielmente essa arquitetura e permite a exportação de código binário em formato textual, possibilitando sua utilização em ambientes externos, como simuladores de hardware (VHDL ou Verilog). Isso amplia o potencial educacional da linguagem, tornando-a útil tanto para cursos introdutórios quanto para disciplinas avançadas de hardware digital e sistemas embarcados.

Por fim, discute-se também os desafios enfrentados durante o desenvolvimento do ecossistema Star, especialmente no que se refere à escolha da linguagem Rust e ao equilíbrio entre simplicidade pedagógica e fidelidade técnica. As soluções encontradas demonstram que é possível criar um ambiente acessível, seguro e eficiente para o ensino de compiladores,

arquitetura e programação de baixo nível, contribuindo significativamente para a formação de profissionais mais preparados e conscientes dos fundamentos que sustentam os sistemas computacionais modernos.

## A Linguagem Assembly Star

A linguagem assembly Star é uma linguagem de baixo nível projetada para ser simples, acessível e fácil de aprender, permitindo que os usuários se concentrem nos conceitos fundamentais de programação e arquitetura de computadores. Inspirada em linguagens assembly clássicas, como MIPS, RISC-V e 6502, a linguagem Star apresenta uma sintaxe simplificada e intuitiva, ideal para iniciantes e entusiastas de computação de baixo nível.

A linguagem é baseada em uma arquitetura de conjunto de instruções de 16 bits, que inclui um conjunto abrangente de instruções para manipulação de controle de fluxo, operações aritméticas e lógicas, manipulação de memória e chamadas de sistema. Essa abordagem permite que os desenvolvedores tenham controle direto sobre os recursos do sistema, possibilitando a criação de programas eficientes e otimizados.

**Figura 1.** Exemplo de código em Star Assembly.



```
.data
string: .stringz "Hello, World!"
.instr
start:
la $g, string
li $a, 0
loop:
llb $a, $g
beqa $a, $zero, end

li $aux1, 7
move $aux2, $a
mcall

inc $g
ja loop
end:    nope
```

**Fonte:** Elaborada pelos autores (2026).

Perceba que o código acima é um exemplo simples e direto de como a linguagem assembly Star pode ser usada para imprimir uma string na tela. O programa carrega o endereço da string "Hello, World!" no registrador $g e, em seguida, entra em um loop onde lê cada byte

da string, verifica se é o byte nulo (indicando o fim da string) e chama uma rotina de sistema para imprimir o byte na tela. O loop continua até que todos os bytes da string sejam processados.

Na Interface de Aplicação Binária (ABI) da Star, o registrador $aux1 é utilizado para indicar o tipo de chamada de sistema, enquanto os registradores $aux2 e $aux3 são usados para passar argumentos adicionais. Neste exemplo, o valor 7 é carregado em $aux1 para indicar que a chamada de sistema é uma operação de escrita, e o byte lido da string é passado em $aux2. Embora exista uma chamada de sistema nativa para imprimir uma string na tela, o objetivo do exemplo acima é demonstrar como a linguagem assembly Star pode ser utilizada para manipular strings e realizar operações de entrada/saída de forma direta.

A linguagem assembly Star combina simplicidade, portabilidade e flexibilidade, tornando-se adequada tanto para iniciantes quanto para quem deseja compreender a lógica interna de arquiteturas computacionais. Sua sintaxe direta facilita o aprendizado, enquanto a compatibilidade entre diferentes implementações da Star Virtual Machine garante uma execução consistente dos programas. Além disso, sua capacidade de operar desde algoritmos básicos até simulações de hardware complexas, aliada à possibilidade de depuração interativa, torna o processo de aprendizagem mais dinâmico e eficaz.



A Star se mostra versátil em aplicações práticas, permitindo simular o comportamento de dispositivos de hardware de forma controlada. Isso é particularmente valioso em cenários educacionais, onde o estudante pode experimentar conceitos de arquitetura digital e compreender como instruções, sinais e operações são processados internamente. Essa capacidade de simulação fornece uma ponte entre o modelo teórico e sua aplicação concreta, criando um ambiente seguro para explorar ciclos de execução, manipulação de registradores e fluxo de instruções.

Além das simulações, a linguagem é adequada para o desenvolvimento de jogos e aplicações interativas, especialmente em contextos nos quais é necessário controlar diretamente registradores, estados e operações em tempo real. O controle granular oferecido pela linguagem permite que comportamentos lógicos sejam implementados de forma transparente, reforçando a compreensão sobre como máquinas reais tratam eventos, cálculos e mudanças de estado. Isso ajuda o aprendiz a desenvolver uma intuição sólida sobre o funcionamento interno de sistemas de computação.

A linguagem Star também se destaca no ensino de algoritmos e estruturas de dados. Ao implementar rotinas como ordenação, busca, pilhas ou filas em assembly, o estudante visualiza

de modo explícito cada operação e ajuste de memória necessário para executar tais algoritmos. Esse tipo de prática revela a mecânica interna do processamento, muitas vezes ocultada em linguagens de alto nível, tornando mais evidente o custo computacional real de cada instrução e reforçando o entendimento sobre eficiência e organização de dados.

Por fim, o fato de a Star Virtual Machine permitir a execução tanto do código-fonte quanto de binários amplia as possibilidades de aprendizagem e investigação. Essa dualidade permite que o usuário compreenda diferentes etapas de transformação do código, desde sua escrita até a representação binária final. Com isso, esse recurso possibilita observar como registradores, instruções e pseudo-instruções interagem dentro da máquina virtual, oferecendo uma visão clara dos principais componentes da linguagem Star e estabelecendo bases sólidas para estudos mais avançados.

## O Planejamento e Registradores da linguagem Assembly Star

O planejamento da linguagem assembly Star envolveu a definição cuidadosa de uma arquitetura de 16 bits composta por instruções além de registradores e pseudo-instruções que ampliam sua expressividade. Esse processo foi fundamentado na análise de arquiteturas consagradas, como MIPS, RISC-V e 6502, escolhidas por sua clareza estrutural e relevância histórica. A adoção de uma ISA de 16 bits buscou privilegiar simplicidade e acessibilidade, facilitando a visualização do fluxo de dados e instruções e tornando a linguagem especialmente adequada ao ensino e à compreensão de fundamentos de arquitetura computacional.



Entretanto, a limitação de espaço inerente às instruções de 16 bits exigiu soluções de design que maximizam o aproveitamento do formato binário restrito. A limitação de espaço para representar opcode, registradores e imediatos em apenas 16 bits levou à adoção de técnicas como divisão das instruções em partes alta e baixa, extensão de opcode e uso de pseudo-instruções expandidas pelo compilador. Essas soluções ampliaram a expressividade da linguagem sem perder simplicidade, permitindo criar programas complexos mesmo em uma arquitetura compacta.

A Star Virtual Machine complementa essa arquitetura ao adotar o formato Big-Endian, no qual o byte mais significativo é armazenado no menor endereço de memória. Essa decisão foi motivada tanto por razões pedagógicas quanto pela intenção de alinhar a Star a arquiteturas históricas amplamente documentadas. O uso de Big-Endian torna mais intuitiva a leitura de representações numéricas em diferentes bases, especialmente hexadecimal e binária,

favorecendo a compreensão visual da organização de dados na memória e reforçando o caráter didático e transparente do ecossistema Star.

A origem dos registradores da Star Virtual Machine foi inspirada em arquiteturas clássicas de computadores, como MIPS e RISC-V, que utilizam um conjunto de registradores de uso geral para armazenar dados temporários e resultados de operações. A escolha de um conjunto de 16 registradores de 16 bits se deve à limitação de espaço da arquitetura de 16 bits, permitindo que os usuários tenham acesso a um número suficiente de registradores para realizar operações complexas sem sobrecarregar a memória.

A numeração e nome dos registradores foram projetadas para serem intuitivas e fáceis de lembrar, facilitando o aprendizado e a utilização da linguagem assembly Star.

Cada registrador tem uma função específica, como armazenar resultados de operações aritméticas, valores temporários ou endereços de memória, permitindo que os usuários compreendam rapidamente o propósito de cada registrador.

## Registradores de uso geral e ocultos

A Star possui um conjunto de 16 registradores de uso geral, cada um com a capacidade de armazenar valores de 16 bits. Esses registradores são usados para armazenar dados temporários e estáticos durante a execução do programa e são acessados diretamente pelas instruções da linguagem assembly. Os registradores podem ser numerados de 0 a 15, porém cada um possui um nome associado.



A seguir estão a numeração, nomes e descrição dos registradores de uso geral:

**0 - Zero** - Este registrador é sempre zero e não pode ser modificado. Ele é usado para operações que requerem um valor constante de zero.

**1 - A** - Este registrador é usado para armazenar o resultado de operações aritméticas e lógicas. Ele é frequentemente usado como o registrador de destino para instruções que produzem um resultado.

**2 - B** - Este registrador é usado para armazenar valores temporários durante a execução do programa. Ele é frequentemente usado como o registrador de origem para instruções que requerem um valor de entrada.

**3 - C** - Este registrador é usado para armazenar valores temporários durante a execução do programa. Assim como o registrador B, ele é frequentemente utilizado como o registrador de origem para instruções que requerem um valor de entrada.

**4 - D** - Este registrador é usado para armazenar o resultado de operações aritméticas e lógicas. Assim como o registrador A, ele é frequentemente utilizado como o registrador de destino para instruções que produzem um resultado.

**5 - E** - Este registrador é usado para armazenar valores temporários durante a execução do programa. Ele é frequentemente utilizado como o registrador de origem para instruções que requerem um valor de entrada.

**6 - F** - Este registrador é usado para armazenar o resultado de operações aritméticas e lógicas. Assim como os registradores A e D, ele é frequentemente utilizado como o registrador de destino para instruções que produzem um resultado.

**7 - G** - Este registrador é usado para armazenar valores temporários durante a execução do programa. Ele é frequentemente utilizado como o registrador de origem para instruções que requerem um valor de entrada.

**8 - Aux1** - Este registrador é usado para armazenar valores utilizados por pseudo-instruções. Ele também é utilizado como interface primária em instruções de chamadas de sistema.

**9 - Aux2** - Este registrador é usado para armazenar valores utilizados por pseudo-instruções. Ele também é utilizado como argumento primário em instruções de chamadas de sistema.



**10 - Aux3** - Este registrador é usado para armazenar valores utilizados por pseudo-instruções. Ele também é utilizado como argumento secundário em instruções de chamadas de sistema.

**11 - Carry** - Este registrador é usado para armazenar o valor do carry (ou transporte) durante operações aritméticas. Ele é utilizado para indicar se ocorreu um carry durante a execução de uma operação aritmética, como adição ou subtração.

**12 - Low** - Este registrador é usado para armazenar o resultado baixo de operações aritméticas que resultam em valores maiores que 16 bits. Ele é utilizado para armazenar a parte inferior do resultado de uma operação aritmética.

**13 - High** - Este registrador é usado para armazenar o resultado alto de operações aritméticas que resultam em valores maiores que 16 bits. Ele é utilizado para armazenar a parte superior do resultado de uma operação aritmética.

**14 - Return Address** - Este registrador é utilizado pela máquina virtual para armazenar o endereço de retorno obtido a partir de instruções de branching ou jumping. Ele é utilizado para retornar ao ponto correto do programa após a execução de uma sub-rotina.

**15 - Stack Pointer** - Este registrador é utilizado pela máquina virtual para armazenar o endereço do topo da pilha. Ele é utilizado para gerenciar a pilha durante a execução do programa, permitindo que valores sejam empilhados e desempilhados conforme necessário. Ele sempre começa apontando para o endereço do topo da pilha, que é o endereço mais alto da memória alocada para a pilha.

Sobre os registradores auxiliares, os registradores Aux1, Aux2 e Aux3 são registradores de uso geral que podem ser utilizados para armazenar valores temporários durante a execução do programa. Eles também são frequentemente utilizados em pseudo-instruções e chamadas de sistema, onde são usados para passar argumentos e armazenar resultados intermediários.

Segue um exemplo de uma pseudo-instrução que utiliza os registradores auxiliares:

**Figura 2**

```
addi $a, $a, 1
```

Fonte: Elaborada pelos autores (2026).



Será traduzida para:

**Figura 3**

```
lli $aux1, 0x01
lai $aux1, 0x00
add $a, $a, $aux1
```

**Fonte:** Elaborada pelos autores (2026).

A Star Virtual Machine também possui registradores ocultos, que são utilizados internamente pela máquina virtual para gerenciar o estado da execução do programa. Esses registradores não são acessíveis diretamente pelo usuário, mas desempenham um papel crucial na operação da máquina virtual.

**Program Counter** - Este registrador é utilizado para armazenar o índice, também podendo ser chamado de endereço fictício da próxima instrução a ser executada. Ele é atualizado automaticamente pela máquina virtual após cada instrução executada, permitindo que o fluxo de controle do programa seja mantido. Na implementação da Star, o Program Counter é um registrador de 16 bits que armazena o índice da próxima instrução a ser executada.

**Instruction Pointer Register** - Este registrador é utilizado para armazenar o endereço real da instrução apontada virtualmente pelo Program Counter. Ele é necessário por conta do modelo de memória de instruções da Star, que divide as instruções em duas partes: alta e baixa.

O endereço da parte alta da instrução apontada pelo Instruction Pointer Register é sempre obtido pela equação:

$$InstructionPointerRegister\_alta = ProgramCounter \times 2$$

Já a parte baixa da instrução é obtida por:

$$InstructionPointerRegister\_baixa = ProgramCounter \times 2 + 1$$

Em uma implementação de hardware real, o Instruction Pointer Register seria implementado como um registrador auxiliar de 17 bits.

**Instruction Register** - Este registrador é utilizado para armazenar a instrução atualmente sendo executada. Ele é atualizado pela máquina virtual antes da execução de cada instrução, permitindo que a máquina virtual decodifique e execute a instrução corretamente. Na implementação da Star, o Instruction Register é um registrador de 16 bits que armazena a instrução em execução.

## INSTRUÇÕES



As instruções da Star foram previamente planejadas para cobrir uma ampla gama de operações, desde aritmética básica até controle de fluxo e manipulação de memória. A escolha de instruções foi baseada na limitação de espaço e técnica de extensão de código de operação, permitindo que a linguagem suporte uma variedade de operações sem exceder o limite de 16 bits por instrução. A linguagem inclui instruções para operações aritméticas, lógicas, de controle de fluxo, manipulação de memória e chamadas de sistema, proporcionando uma base sólida para o desenvolvimento de programas complexos. Além disso, a Star Virtual Machine suporta pseudo-instruções, que são instruções de alto nível que são traduzidas em uma ou mais instruções de baixo nível durante o processo de compilação, permitindo que os usuários escrevam código fonte mais legível e escalável.

Segue abaixo uma lista com uma breve descrição de cada instrução da Star Virtual Machine:

**add** - Adição - *Soma os valores em dois registradores e armazena o resultado em um terceiro.*

**sub** - Subtração - *Subtrai o valor de um registrador de outro e armazena o resultado.*

**and** - Operação Lógica AND - *Realiza o AND bit a bit entre dois registradores.*

**or** - Operação Lógica OR - *Realiza o OR bit a bit entre dois registradores.*

**xor** - Operação Lógica XOR - *Realiza o XOR bit a bit entre dois registradores.*

**shl** - Operação de Deslocamento Lógico à Esquerda - *Desloca o valor de um registrador para a esquerda.*

**shr** - Operação de Deslocamento Lógico à Direita - *Desloca o valor de um registrador para a direita.*

**lai** - Carregar Imediato Alto - *Carrega os 8 bits inferiores do valor imediato em um registrador.*

**lli** - Carregar Imediato Baixo - *Carrega os 8 bits superiores do valor imediato em um registrador.*

**beqr** - Salto Condicional Igual Relativo - *Salta se dois registradores forem iguais.*

**bneqr** - Salto Condicional Diferente Relativo - *Salta se dois registradores forem diferentes.*

**bgtr** - Salto Condicional Maior Que Relativo (com sinal) - *Salta se um registrador for maior que outro (com sinal).*

**bltr** - Salto Condicional Menor Que Relativo (com sinal) - *Salta se um registrador for menor que outro (com sinal).*

**bgtur** - Salta Condicional Maior Que Relativo (sem sinal) - *Salta se um registrador for maior que outro (sem sinal).*



**bltur** - Salto Condicional Menor Que Relativo (sem sinal) - *Salta se um registrador for menor que outro (sem sinal).*

**mulhl** - Multiplicação de Registradores (com sinal) - *Multiplica dois registradores (com sinal); resultado dividido entre low e high.*

**divhl** - Divisão de Registradores (com sinal) - *Divide dois registradores (com sinal); quociente em low, resto em high.*

**muluhl** - Multiplicação de Registradores (sem sinal) - *Multiplica dois registradores (sem sinal); resultado dividido entre low e high.*

**divuhl** - Divisão de Registradores (sem sinal) - *Divide dois registradores (sem sinal); quociente em low, resto em high.*

**not** - Operação Lógica NOT - *Realiza o NOT bit a bit em um registrador.*

**xlb** - Extensão de Byte Baixo - *Estende o sinal do byte inferior de um registrador para 16 bits.*

**lab** - Carrega o Byte Alto - *Carrega o byte alto da memória no endereço de um registrador.*

**llb** - Carrega o Byte Baixo - *Carrega o byte baixo da memória no endereço de um registrador.*

**sab** - Armazena o Byte Alto - *Armazena o byte alto de um registrador na memória.*

**slb** - Armazena o Byte Baixo - *Armazena o byte baixo de um registrador na memória.*

**j** - Salto Incondicional - *Salta para o endereço contido em um registrador.*

**mcall** - Chamada de Sistema - *Chama uma rotina de sistema definida pelos registradores auxiliares.*

## Pseudo-Instruções

As pseudo-instruções da Star Virtual Machine foram projetadas para simplificar o processo de escrita de código assembly, permitindo que os usuários escrevam instruções de alto nível que são traduzidas em uma ou mais instruções de baixo nível durante a compilação. Essas pseudo-instruções são especialmente úteis para operações comuns, como manipulação de memória, operações aritméticas e lógicas, e controle de fluxo, permitindo que os usuários escrevam código mais legível e fácil de entender. As pseudo-instruções são uma extensão da linguagem assembly Star e não são executadas diretamente pela Star Virtual Machine, mas sim traduzidas em instruções de baixo nível durante o processo de compilação. Isso permite que os usuários escrevam código mais expressivo e fácil de manter, sem sacrificar a eficiência e o controle sobre os recursos do sistema.



Abaixo estão algumas das principais pseudo-instruções suportadas pela Star Virtual Machine:

**nope** - *Não tem nenhum impacto.*

**move $rd, $rs** - *Copia o valor de um registrador para outro.*

**neg $rd, $rs** - *Nega (complemento de dois) o valor de um registrador.*

**jr $rs** - *Salta para o endereço em $rs.*

**ret** - *Retorna para o endereço em $ra.*

**li $rd, imm** - *Carrega um imediato de 16 bits em um registrador.*

**la $rd, label** - *Carrega o endereço de um rótulo em um registrador.*

**lb $r1, $r5[offset]** - *Carrega um byte da memória (sign-extend).*

**sb $r1, $r5[offset]** - *Armazena o byte menos significativo na memória.*

**lw $r1, $r5[offset]** - *Carrega uma word (16 bits) da memória.*

**sw $r1, $r5[offset]** - *Armazena uma word (16 bits) na memória.*

**mul $rd, $rs, $rt** - *Multiplica dois registradores.*

**div $rd, $rs, $rt** - *Divide dois registradores (quociente).*

**mod $rd, $rs, $rt** - *Divide dois registradores (resto).*

**swap $r1, $r2** - *Troca os valores de dois registradores.*

**addi $rd, $rs, imm** - *Soma imediato a um registrador.*

**subi $rd, $rs, imm** - *Subtrai imediato de um registrador.*

**andi $rd, $rs, imm** - *AND entre registrador e imediato.*

**ori $rd, $rs, imm** - *OR entre registrador e imediato.*

**xori $rd, $rs, imm** - *XOR entre registrador e imediato.*

**shli $rd, $rs, imm** - *Desloca à esquerda por imediato.*

**shri $rd, $rs, imm** - *Desloca à direita por imediato.*

**inc $r** - *Incrementa um registrador.*

**dec $r** - *Decrementa um registrador.*

**muli $rd, $rs, imm** - *Multiplica por imediato.*

**divi $rd, $rs, imm** - *Divide por imediato (quociente).*

**modi $rd, $rs, imm** - *Divide por imediato (resto).*

**beqa $rs, $rt, label** - *Salta se $rs = $rt (com sinal).*

**bneqa $rs, $rt, label** - *Salta se $rs ≠ $rt (com sinal).*

**bgta $rs, $rt, label** - *Salta se $rs > $rt (com sinal).*

**blta $rs, $rt, label** - *Salta se $rs < $rt (com sinal).*

**bgtua $rs, $rt, label** - *Salta se $rs > $rt (sem sinal).*

**bltua $rs, $rt, label** - *Salta se $rs < $rt (sem sinal).*

**j label** - *Salto incondicional para um rótulo.*



## Processadores

Os processadores na linguagem Star são mecanismos de pré-processamento que ampliam a flexibilidade e a modularidade do código fonte. Inspirados em sistemas de pré-processamento de linguagens como C e Rust, os processadores permitem a inclusão de arquivos, definição de macros, proteção contra múltiplas inclusões e outras operações que ocorrem antes da análise sintática do compilador.

Na Star, processadores são identificados por comandos iniciados pelo caractere @, como @include, @define, @once. Eles são processados pelo scanner, que reconhece essas instruções especiais e executa as ações correspondentes antes de passar o código para as próximas etapas do compilador.

## O Processador Include

O processador @include permite que o conteúdo de outro arquivo Star seja inserido no ponto em que o processador é encontrado. Isso facilita a reutilização de código, permitindo que funções, constantes e definições sejam compartilhadas entre diferentes arquivos. Por exemplo, ao incluir um arquivo com funções utilitárias, você pode usar essas funções em qualquer parte do seu código sem precisar reescrevê-las.

O caminho do arquivo incluído é inserido entre aspas após o comando @include, como em @include "arquivo.star". O scanner irá localizar o arquivo especificado e inserir seu conteúdo no local apropriado do código fonte.

## O Processador Once

O processador @once é utilizado para garantir que o conteúdo do arquivo atual seja incluído apenas uma vez durante o processo de compilação. Isso evita problemas de múltiplas inclusões acidentais, que podem levar a erros de redefinição ou conflitos de símbolos. Quando o scanner encontra um @once, ele verifica se o conteúdo escrito após o processador já foi incluído anteriormente. Se sim, ele ignora o conteúdo; caso contrário, ele processa o conteúdo normalmente.



## O Processador Define

O processador @define permite a definição de macros, que são substituições de texto que ocorrem durante o pré-processamento do código. Quando um macro é definido com @define NOME valor, todas as ocorrências de NOME no código fonte são substituídas pelo *valor* especificado. Isso é útil para definir constantes, endereços ou até mesmo sequências de instruções que podem ser reutilizadas em várias partes do código.

Na linguagem Star, esse processador também suporta o que chamamos de Macros com Argumentos, onde você pode definir macros com parâmetros. Por exemplo, @define MACRO (%x) addi $aux1, $aux2, %x define um macro que recebe um argumento %x e o utiliza na instrução addi. Quando o macro é chamado, o valor passado substitui %x na instrução.

O símbolo **%** no início da expressão é utilizado para indicar que a expressão como um todo é um argumento de macro. Isso garante a segurança e a clareza na substituição de macros, evitando ambiguidades com outros símbolos ou palavras-chave da linguagem.

## Extensibilidade e Chamadas de Sistema

A característica de extensibilidade da Star refere-se à sua capacidade de permitir que o desenvolvedor defina e personalize como a máquina virtual interage com um sistema hospedeiro. Em vez de ter um conjunto fixo e imutável de operações chamada de sistema, a Star utiliza um sistema de interfacing que delega a execução de chamadas mcall para uma implementação fornecida pelo usuário.

A operação mcall é o ponto central da interação com o sistema. Ele funciona como uma ponte: sempre que o código assembly executado na VM faz uma operação de sistema, o controle é passado para a função externa definida por uma interface conectada a Star Virtual Machine.

Essa arquitetura torna a Star uma ferramenta poderosa para fins educacionais, pois permite simular diferentes sistemas apenas mudando a implementação da interface, sem a necessidade de modificar o compilador ou o núcleo da máquina virtual.

## Memória e Modelos de Arquitetura

O núcleo da Star Virtual Machine adota o modelo de arquitetura de Harvard, no qual a memória é fisicamente separada em dois espaços distintos: um dedicado ao armazenamento de instruções (código executável) e outro reservado para dados (variáveis, buffers, pilha, etc).



Essa separação estrutural é fundamental para garantir maior segurança, previsibilidade e eficiência na execução dos programas. O núcleo da Star Virtual Machine organiza sua memória em segmentos bem definidos:

**Memória de Instruções:** Armazena o código de máquina Star. É carregada durante a inicialização do programa e permanece inalterada durante a execução.

**Memória de Dados:** usada para armazenar variáveis globais, dados estáticos, buffers e a pilha de execução.

**Memória de Posições:** Armazena as posições (arquivo, linha e coluna) de definição de todas as instruções presentes na memória de instruções. Sua existência tem como objetivo principal aprimorar o processo de depuração, permitindo que, durante a execução ou análise de um programa, seja possível rastrear exatamente onde cada instrução foi definida no código fonte. Isso facilita a identificação de erros, a análise do fluxo do programa e a geração de mensagens de depuração mais precisas.

Cada segmento possui políticas de acesso e proteção específicas, reforçando a robustez e a previsibilidade do ambiente de execução.

Cabe ressaltar que a memória de posições serve apenas para auxiliar o processo de depuração. Em uma arquitetura real, sua implementação poderia ser descartada, uma vez que não é necessária para a execução do programa. No entanto, sua presença na Star Virtual Machine é fundamental para fornecer informações detalhadas sobre o código fonte durante a depuração, melhorando a experiência do desenvolvedor.

## Formato de Instruções

A Star Virtual Machine utiliza cinco formatos de instruções diferentes, cada um com uma estrutura específica para acomodar diferentes tipos de operações. Esses formatos são projetados para aproveitar ao máximo a arquitetura de 16 bits da máquina virtual, permitindo uma codificação eficiente e compacta das instruções.

**Trinity:** add $rd, $r1, $r2

Três registradores: destino e duas fontes. Usado para operações aritméticas e lógicas.

**Hime:** lai $rd, imm8

Um registrador e um valor imediato de 8 bits. Utilizado para carregar imediatos em registradores.

**Pair:** mulhl $r1, $r2

Dois registradores. Usado para multiplicação, divisão e manipulação de bytes.

**Clover:** j $rs

Um único registrador. Utilizado para instruções de salto.

**Ark:** mcall

Sem operandos explícitos; utiliza registradores auxiliares para chamadas de sistema.

## Visualização dos Quartetos de Cada Formato

Figura 3

| Formato | Quarteto 4 | Quarteto 3 | Quarteto 2 | Quarteto 1 | Visualização |
|---|---|---|---|---|---|
| **Trinity** | zzzz | yyyy | xxxx | oooo | \| zzzz \| yyyy \| xxxx \| oooo \| |
| **Hime** | iiii | iiii | xxxx | oooo | \| iiii-iiii \| xxxx \| oooo \| |
| **Pair** | yyyy | xxxx | oooo | iiii | \| yyyy \| xxxx \| oooo \| iiii \| |
| **Clover** | xxxx | oooo | iiii | iiii | \| xxxx \| oooo \| iiii \| iiii \| |
| **Ark** | oooo | iiii | iiii | iiii | \| oooo \| iiii \| iiii \| iiii \| |

**Fonte:** Elaborada pelos autores (2026).

**oooo** - Código de operação (opcode) de 4 bits, que identifica a instrução a ser executada.

**xxxx** - Registrador, que geralmente é o registrador de destino.

**yyyy** - Registrador fonte, que é o registrador de origem para a operação.

**zzzz** - Registrador fonte adicional, usado em instruções que requerem mais de um registrador fonte.

**iiii-iiii** - Valor imediato de 8 bits, que é um valor constante usado na instrução.

**1111** - Padrão reservado utilizado para indicar a extensão de código de operação.

## Extensão de Código de Operação (Opcode Extension)

A extensão de código de operação é uma técnica empregada para ampliar o conjunto de instruções disponíveis sem alterar o tamanho fixo das instruções da arquitetura. Isso é possível ao reservar padrões específicos de bits dentro do opcode para indicar instruções especiais ou formatos diferenciados, permitindo assim a implementação de operações mais complexas e variadas.

Por exemplo, quando o código de operação de um determinado formato de instrução é igual a 15 (1111 em binário), isso indica que a instrução utilizará um formato estendido. Nesse caso, o opcode é seguido por um segundo quarteto que especifica a operação adicional, podendo também estender o formato para um próximo.



Essa abordagem é o coração da flexibilidade da Star Virtual Machine, permitindo que novas instruções sejam adicionadas sem a necessidade de redefinir toda a arquitetura. A extensão de código de operação é uma técnica comum em arquiteturas de conjunto de instruções compactas, como a Star, e é essencial para maximizar o uso do espaço limitado disponível para cada instrução.

Embora a extensão de código de operação ofereça flexibilidade e capacidade de expansão, ela também introduz complexidade no processo de decodificação das instruções. A máquina virtual precisa ser capaz de identificar quando uma instrução utiliza um formato estendido e, em seguida, processar os bits adicionais corretamente. Isso pode aumentar o tempo de execução da decodificação e exigir lógica adicional para lidar com os diferentes formatos de instrução.

Como também há uma troca de espaço por novas instruções. A cada novo quarteto adicionado, há uma redução no número de bits disponíveis para a definição de registradores e valores imediatos. Isso pode limitar a complexidade das operações que podem ser realizadas em

uma única instrução, exigindo que algumas operações mais complexas sejam divididas em várias instruções.

Durante o planejamento da linguagem assembly Star, foi necessário considerar essas desvantagens e encontrar um equilíbrio entre a flexibilidade do conjunto de instruções e a simplicidade da arquitetura. Instruções previamente planejadas foram cuidadosamente selecionadas para garantir que a linguagem fosse expressiva o suficiente para atender às necessidades dos usuários, enquanto ainda permanecia dentro dos limites de espaço e complexidade da arquitetura de 16 bits.

Ao utilizar esses padrões, a Star Virtual Machine consegue expandir significativamente o número de instruções suportadas, mesmo com o espaço restrito de 16 bits por instrução. Essa abordagem garante flexibilidade e expressividade à linguagem, sem comprometer a simplicidade da arquitetura.

## Arquitetura do Compilador Star

O compilador da Star Virtual Machine foi projetado para ser modular e eficiente, traduzindo o código fonte Star Assembly para código de máquina Star. Os principais componentes são:



**Scanner:** Lê o código fonte e converte em tokens.

**Parser:** Analisa os tokens e constrói a AST.

**Resolver:** Resolve pseudo-instruções e constrói a tabela de símbolos.

**Generator:** Gera o código de máquina Star e organiza a memória.

**Debugger:** Auxilia na depuração, notificando erros e permitindo inspeção do estado do programa.

**Binary Scanner:** Lê código binário pré-compilado para execução na VM.

## A Linguagem de Programação Rust

A escolha da linguagem de programação Rust para a implementação da Star Virtual Machine foi motivada por diversas características que a tornam especialmente adequada ao desenvolvimento de sistemas e aplicações de baixo nível, como compiladores e máquinas virtuais.

Rust é uma linguagem moderna que combina segurança, desempenho e concorrência, oferecendo recursos avançados de gerenciamento de memória sem a necessidade de coletor de

lixo. Isso é fundamental para a Star Virtual Machine, que precisa operar de forma eficiente em um ambiente de 16 bits, onde o controle preciso dos recursos é crucial.

Além disso, Rust possui um sistema de tipos forte e expressivo, que ajuda a evitar erros comuns de programação, como estouros de buffer e condições de corrida. Esse aspecto é essencial para garantir a robustez e a segurança da máquina virtual, especialmente ao lidar com código de máquina e manipulação direta de memória.

O sistema de correspondência de padrões (pattern matching) do Rust facilita a implementação de lógica complexa, como decodificação de instruções e a execução de operações, tornando o código mais legível e fácil de manter. A linguagem oferece abstrações poderosas, como traits e enums, que permitem modelar comportamentos e estados de forma clara e concisa.

No Rust, o pattern matching exige que todos os casos possíveis sejam tratados. Isso significa que, ao usar match em um enum, por exemplo, o compilador verifica se você cobriu todas as variantes. Se faltar algum caso, o compilador gera um erro, evitando bugs causados por casos não tratados.

Isso garante que, ao adicionar novas variantes de um tipo de enum, o desenvolvedor seja obrigado a atualizar o match, evitando que casos sejam esquecidos e garantindo que o código esteja sempre completo e correto.



The power of match comes from the expressiveness of the patterns and the fact that the compiler confirms that all possible cases are handled.

[Klabnik and Nichols 2018, p. 102]

Um recurso importante que contribuiu para a implementação da Star Virtual Machine foi o sistema de traits do Rust, que permite definir comportamentos comuns entre diferentes tipos de dados e módulos. Isso é especialmente útil na Star, pois possibilita que diferentes componentes da máquina virtual compartilhem funcionalidades, como manipulação de registradores e memória, sem duplicação de código.

O sistema de traits também torna a Star Virtual Machine extensível, facilitando a adição de novos recursos e instruções sem a necessidade de reescrever grandes partes do código. Isso é importante para a evolução da linguagem Star e para a inclusão de novas funcionalidades na máquina virtual.

Outro ponto positivo é o sistema de gerenciamento de dependências robusto, com o Cargo, que facilita a inclusão de bibliotecas e ferramentas externas no projeto.

Além disso, Rust conta com uma comunidade ativa e um ecossistema em constante crescimento, que oferecem bibliotecas e ferramentas úteis para o desenvolvimento de sistemas. A linguagem também possui uma curva de aprendizado relativamente suave, facilitando a adoção por novos desenvolvedores.

Essas características tornam Rust uma escolha ideal para a implementação da Star Virtual Machine, permitindo que o projeto seja desenvolvido de forma eficiente, segura e com alto desempenho.

## CONCLUSÃO

A linguagem assembly Star e a Star Virtual Machine constituem um ecossistema educacional integrado, projetado para tornar acessível o estudo de arquitetura de computadores e programação de baixo nível. Ao priorizar simplicidade e clareza conceitual, o projeto cria uma relação direta entre código fonte, instruções binárias e comportamento interno de execução. Essa estrutura permite que aprendizes compreendam, de forma progressiva, os mecanismos fundamentais que regem sistemas computacionais.

Com isso, a estrutura da Star, demonstra que mesmo arquiteturas compactas podem atingir flexibilidade e expressividade. A incorporação de extensão de opcode, divisão alto/baixo e pseudo-instruções amplia significativamente o alcance da linguagem, superando as limitações inerentes ao formato sem comprometer sua simplicidade. Essa combinação equilibrada de restrição e capacidade torna o ecossistema adequado ao desenvolvimento gradual de competências, desde tarefas elementares até construções de lógica mais sofisticadas.



Portanto, a implementação da Star Virtual Machine em Rust contribui para a robustez do projeto, garantindo segurança na manipulação de memória e reduzindo vulnerabilidades típicas de ambientes de baixo nível. Com a adoção do modelo Harvard e da abordagem Big-Endian ele oferece maior previsibilidade e clareza na organização interna do sistema, aspectos que facilitam a visualização de dados e instruções durante a execução. O debugger integrado complementa esses recursos, permitindo uma análise minuciosa do estado dos registradores, da memória e do fluxo de controle.

Além de cumprir sua função pedagógica, o ecossistema demonstra potencial para aplicações avançadas. Entre elas esta a possibilidade de exportar código binário e integrá-lo a ambientes como VHDL e Verilog ampliando sua utilidade em contextos de simulação de hardware e sistemas embarcados. A modularidade do compilador, com etapas bem definidas

como scanner, parser, resolver e gerador de código, também oferece um terreno fértil para estudos em compiladores, análise de desempenho e experimentação com novas extensões da linguagem.

Dessa forma, a linguagem Star e sua máquina virtual consolidam-se como uma ferramenta completa para o ensino, pesquisa e desenvolvimento em computação de baixo nível. O projeto contribui para a formação de profissionais mais preparados, capazes de compreender e construir sistemas eficientes, seguros e inovadores. O futuro da Star Virtual Machine inclui a expansão do conjunto de instruções, integração com novas ferramentas de análise e depuração, e o fortalecimento da comunidade de usuários e colaboradores, consolidando seu papel como referência no ensino de arquitetura de computadores e compiladores.

## REFERÊNCIAS

AHO, Alfred V.; LAM, Monica S.; SETHI, Ravi; ULLMAN, Jeffrey D. Compilers: principles, techniques, and tools. 2. ed. Boston: Addison-Wesley, 2007.

FERNANDES, S. R.; SILVA, I. S. Relato de experiência interdisciplinar usando MIPS. Revista Internacional de Educação em Arquitetura de Computadores (IJCAE) V.6, N.1, 2017. DOI: https://doi.org/10.5753/ijcae.2017.4866 .

KLABNIK, Steve; NICHOLS, Carol. The Rust Programming Language. San Francisco: No Starch Press, 2018. Disponível em: https://doc.rust-lang.org/book/.

MIQUELINI, R. A. A.; FERRARI, H. O. Logisim: ferramenta para simulação de circuitos combinacionais e sequenciais digitais. Intercursos, Ituiutaba, v. 20, n. 2, 2021. Disponível em: https://revista.uemg.br/intercursosrevistacientifica/article/view/6319/3799 .

NYSTROM, Robert. Crafting Interpreters. 2021. Disponível em: https://craftinginterpreters.com/

PATTERSON, David; HENNESSY, John. Computer organization and design: RISC-V edition. San Francisco: Morgan Kaufmann, 2017.

TANENBAUM, Andrew S.; AUSTIN, Todd. Structured computer organization. 6. ed. Upper Saddle River: Prentice Hall, 2013.

VOLLMAR, Ken; SANDERSON, Pete. MARS: an education-oriented MIPS assembly language simulator. In: SIGCSE'06 – Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education, Houston, 1–5 Mar. 2006. New York: ACM, 2006.