CENTRO DE TECNOLOGIA - UNIVERSIDADE FEDERAL DE SANTA MARIA

CIÊNCIA DA COMPUTAÇÃO

LUÍS GUSTAVO WERLE TOZEVICH
JAIME ANTÔNIO DANIEL FILHO

DESMONTADOR PARA ARQUITETURA MIPS: IMPLEMENTAÇÃO E ANÁLISE DE INSTRUÇÕES

SANTA MARIA/RS
2023

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

MIPS - Microprocessor without Interlocked Pipeline Stages.

IDE - Integrated Development Environment.

PET-SI - Programa de Educação Tutorial - Sistemas de Informação.

CPU - Central Processing Unit.

RISC - Reduced Instruction Set Computer.

CISC - Complex Instruction Set Computer.

VLSI - Very Large Scale Integration.

SUMÁRIO

INTRODUÇÃO

No âmbito da computação, a análise e a compreensão de programas escritos em linguagem de máquina desempenham um papel fundamental em diversas situações. Com frequência, é necessário o entendimento do funcionamento interno de diversos programas, seja para fins de engenharia reversa, resolução de problemas ou simplesmente para depuração de código. Um dos desafios intrínsecos a esse processo reside na interpretação das instruções codificadas em linguagem de máquina, representadas por códigos binários de complexa interpretação por parte dos seres humanos.

A arquitetura de processador MIPS destaca-se como uma das arquiteturas RISC amplamente adotadas no desenvolvimento de sistemas embarcados, sistemas operacionais e em outros domínios da computação. Entretanto, para analisar programas codificados em linguagem de máquina MIPS, é imperativo traduzir essas instruções binárias para uma representação textual em linguagem assembly, o que, frequentemente, constitui uma tarefa complexa e demorada.

Nesse contexto, apresenta-se um desmontador direcionado à arquitetura de processador MIPS. A criação desse desmontador simplifica o processo de engenharia reversa, depuração de código, desenvolvimento de software e pesquisas acadêmicas relacionadas à arquitetura MIPS. Fomentando avanços na compreensão e análise de programas desenvolvidos para essa arquitetura.

1. OBJETIVOS

Nesta seção, descrevem-se os objetivos que orientam a elaboração deste projeto.

1.1 OBJETIVO GERAL

O objetivo deste estudo consiste na concepção e no desenvolvimento de um desmontador voltado para a arquitetura de processador MIPS, com a finalidade de habilitá-lo à leitura de um arquivo contendo código em linguagem de máquina. Este desmontador terá a responsabilidade de efetuar a desmontagem das instruções nele contidas e, por conseguinte, elaborar um arquivo de saída em formato textual que compreenderá as seguintes informações: endereço da instrução desmontada, a instrução original em linguagem de máquina e a representação correspondente em linguagem assembly.

1.2 OBJETIVOS ESPECÍFICOS

a) Elaborar um programa em linguagem assembly específica para a arquitetura MIPS, com ênfase na viabilização da etapa de leitura de um arquivo de código em linguagem de máquina, visando a fácil expansão do código.
b) Implementar de maneira sistemática o procedimento de desmontagem de instruções, mapeando os códigos de operação (opcodes) pertinentes, identificando os campos relativos aos registradores e aos valores imediatos presentes nas instruções, e assegurando que o código seja facilmente extensível para novas instruções.
c) Confeccionar um arquivo de saída de natureza textual que compreenderá os elementos obrigatórios, como o endereço da instrução desmontada, a representação original da instrução em linguagem de máquina e a expressão equivalente em linguagem assembly, com a devida atenção à facilidade de expansão.
d) Realizar uma avaliação criteriosa, do desempenho do desmontador, especialmente no que se refere à sua habilidade de reconhecimento de instruções e a capacidade de expansão do código.

2. REVISÃO BIBLIOGRÁFICA

Nesta seção, realizaremos uma revisão da literatura relacionada, busca-se estabelecer uma sólida base teórica, que permitirá o entendimento completo dos conceitos, das aplicações e dos desafios inerentes à elaboração de um desmontador utilizando Assembly MIPS.

2.1 ARQUITETURA MIPS

Pode-se classificar um processador de acordo com a quantidade de instruções que são suportadas por ele, resultando na distinção entre processadores RISC e CISC. De acordo com Masood (2011), a arquitetura RISC se fundamenta em um conjunto otimizado de instruções reduzidas. Por sua vez, a arquitetura CISC, conforme observado por Jamil (1995), tem como objetivo simplificar a programação, permitindo a realização de tarefas complexas com um código mais enxuto, devido à capacidade do processador de executar várias operações em uma única instrução.

De acordo com Henessy et al. (1982), a arquitetura MIPS é uma arquitetura de microprocessador concebida para ser implementada em um único chip VLSI, visando a obtenção de alto desempenho na execução de código compilado. A diferenciação entre as arquiteturas RISC e CISC é fundamental para a compreensão da abordagem adotada pela MIPS. Enquanto as arquiteturas CISC suportam um amplo conjunto de instruções, muitas das quais são complexas e altamente especializadas, as arquiteturas RISC, como a empregada pela MIPS, optam por reconhecer um número limitado de instruções, porém mais simples e, portanto, mais rápidas de executar. A abordagem RISC se destaca pela eficiência na execução de código compilado, tornando-a uma opção viável para aplicações que exigem alto desempenho.

A arquitetura MIPS, como uma implementação RISC, possui características distintas que a tornam eficiente. Segundo Patterson e Hennessy (2013), a arquitetura MIPS adota uma abordagem de pipeline, onde várias instruções são executadas simultaneamente em diferentes estágios do pipeline. Isso permite que a CPU execute mais instruções em um determinado período de tempo, aumentando assim o desempenho geral. Além disso, a arquitetura MIPS utiliza um conjunto de instruções de tamanho fixo. Cada instrução é codificada em uma palavra de 32 bits, o que simplifica o design do hardware e permite um decodificador de instruções mais rápido (Patterson e Hennessy, 2013). Contrastando com as arquiteturas CISC, que podem ter instruções de diversos tamanhos.

Outra característica importante da arquitetura MIPS é o uso de um grande número de registradores. A arquitetura MIPS possui 32 registradores de propósito geral, permitindo que muitos

dados sejam armazenados diretamente na CPU para acesso rápido (Patterson e Hennessy, 2013). Reduzindo a necessidade de acessos à memória, que são mais lentos. Desta forma, a arquitetura MIPS representa uma solução eficaz para atender às demandas de processamento de dados de forma rápida e eficiente.

2.2 IDE MARS

Segundo Vollmar e Sanderson (2006) o software MARS representa um ambiente de desenvolvimento integrado, com ênfase na programação em linguagem assembly MIPS. Seu principal propósito reside na esfera educacional (PET-SI, 2023). Conforme Vollmar e Sanderson (2006) o programa foi desenvolvido em Java, suportando aproximadamente 155 instruções fundamentais da arquitetura MIPS, cerca de 370 pseudo-instruções e chamadas de sistema. Esta IDE, destaca-se ao proporcionar uma interface gráfica amigável ao usuário (VOLLMAR; SANDERSON, 2006).

O MARS possui uma a depuração interativa. De forma que, o programador pode definir ou remover pontos de interrupção da execução, além de avançar ou retroceder na execução (com opção de desfazer), enquanto simultaneamente visualiza e edita diretamente o conteúdo armazenado em registradores e na memória (VOLLMAR; SANDERSON, 2006). É amplamente empregado em disciplinas relacionadas à Organização de Computadores e Arquitetura de Computadores (PET-SI, 2023). Possibilitando ao usuário o entendimento do funcionamento de um processador (PET-SI, 2023).

2.3 O QUE É ASSEMBLY MIPS

Assembly MIPS é uma linguagem de programação de baixo nível que utiliza uma representação textual de instruções binárias que são diretamente executadas pelo processador. Diferentemente das linguagens de alto nível, o Assembly MIPS é altamente específico para a arquitetura MIPS e depende fortemente da estrutura do processador.

2.4 DESMONTADOR (DISASSEMBLER)

Segundo Popa (2012), um desmontador é um programa que traduz linguagem de máquina para a linguagem assembly equivalente, a operação exatamente inversa de um montador. Durante esse processo, o conjunto de instruções é formatado para ser facilmente compreendido por humanos

e, nem sempre, é apropriado para ser montado novamente. Por conseguinte, a principal aplicação dos desmontadores é a engenharia reversa de programas para o entendimento do seu funcionamento interno ou para a busca de falhas ou vulnerabilidades, notavelmente em softwares que o código fonte não está prontamente disponível, seja por motivos de proteção intelectual, seja por situações em que o código original foi perdido.

A desmontagem também é fundamental para a compreensão das linguagens de programação em níveis mais profundos. Isso porque, assim como explicitado por De Brujin (2022), as distintas maneiras que compiladores implementam estruturas de controle e operações são apenas visualizáveis diretamente na linguagem de máquina, visto que variam para as diferentes arquiteturas de processadores. Desse modo, um desmontador auxilia desenvolvedores a verificarem os impactos das abstrações de alto nível no desempenho de sua aplicação.

No entanto, a desmontagem de programas por vezes não é possível, pois, conforme afirma Andriesse et al. (2016), o código em binário pode conter construções complexas, como dados embutidos em regiões executáveis, códigos automodificáveis ou sobrepostos. Isso é especialmente notável em binários ofuscados, presentes comumente em programas mal-intencionados, que tornam a desmontagem um processo extremamente desafiador e complexo. Além do mais, alguns desmontadores procuram por determinadas propriedades emitidas por compiladores para diferenciarem regiões executáveis de regiões de dados as quais, muitas vezes, são retiradas no binário de distribuição.

2.5 INSTRUÇÕES MIPS

O conjunto de instruções da arquitetura MIPS compreende instruções e pseudoinstruções. Todavia, esse processador é compatível apenas com as instruções básicas, de forma que, as pseudoinstruções existem para facilitar o processo de programação. Conforme dito por Dandamudi (2003) as pseudoinstruções são, na verdade, traduzidas pelo montador em uma sequência de uma ou mais instruções do processador. As instruções básicas do MIPS empregam estruturas definidas de comprimento fixo, devido à sua natureza como um processador RISC. Dessa forma, utiliza-se três formatos distintos de instruções, que serão apresentadas a seguir. A tabela 1 apresenta informações essenciais sobre a codificação de instruções na arquitetura.

Tabela 1: Informações sobre as instruções da arquitetura MIPS

Campo	Descrição
opcode	Código primário da operação (6 bits).
function	Código secundário da operação (6 bits).
rd	Número para o registrador destino (5 bits).
rs	Número para o registrador fonte (5 bits).
rt	Número para o registrador alvo (5 bits).
sa	Quantidade de deslocamento (5 bits).
immediate	Constante armazenada no formato da instrução (16 bits).
offset	Imediato usado para identificar o endereço de memória relativo ao endereço da instrução seguinte (16 bits).
instr_index	Índice de 26 bits deslocado 2 bits à esquerda para suprir os 28 bits menos significativos do endereço alvo. Os 4 bits mais significativos são dados pelo endereço da instrução seguinte.

Fonte: Autor (2023)

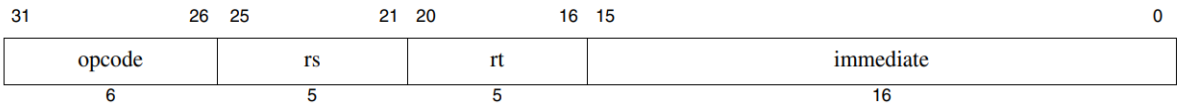
2.5.1 Instrução tipo I

Segundo Dandamudi (2003):

> "All load and store instructions use this instruction format. The immediate value is a signed 16-bit integer. In addition, arithmetic and logical instructions that use an immediate value also use this format. Branch instructions use a 16-bit signed offset relative to the program counter and are encoded in the I-type format."(Dandamudi, 2003, p. 620)[1]

O formato dessa instrução pode ser visualizado na figura 1.

Figura 1- Representação das intruções do tipo I



Fonte: Dandamudi (2003)

[1] "Todas as instruções de carga e armazenamento utilizam esse formato de instrução. O valor imediato é um inteiro de 16 bits com sinal. Além disso, as instruções de aritmética e lógica que utilizam um valor imediato também utilizam esse formato. As instruções de ramificação utilizam um deslocamento de 16 bits com sinal em relação ao contador de programa e são codificadas no formato I-type. " (Dandamudi, 2003, p. 620, tradução nossa)

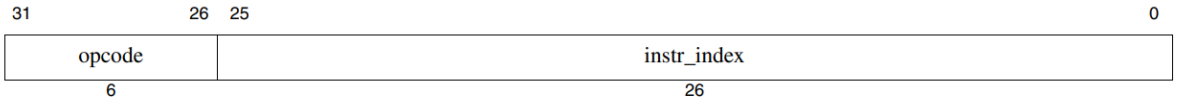
2.5.2 Instrução tipo J

Segundo Dandamudi (2003):

> "Jump instructions that specify a 26-bit target address use this instruction format. These 26 bits are combined with the higher-order bits of the program counter to get the absolute address." (Dandamudi, 2003, p. 620) [2]

O formato dessa instrução pode ser visualizado na figura 2.

Figura 2- Representação das intruções do tipo J



Fonte: Dandamudi (2003)

2.5.3 Instrução tipo R

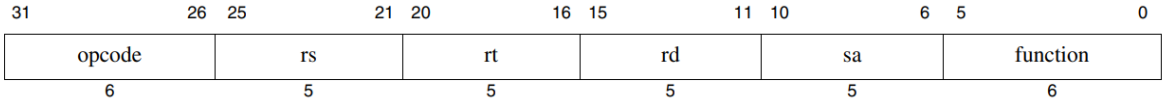
Segundo Dandamudi (2003):

> "Arithmetic and logical instructions use this instruction format. In addition, the jump instruction in which the target address is specified indirectly via a register also uses this instruction format." (Dandamudi, 2003, p. 620) [3]

O formato dessa instrução pode ser visualizado na figura 3.

Tradução:

Figura 3- Representação das intruções do tipo R



Fonte: Dandamudi (2003)

[2] "Instruções de salto que especificam um endereço de destino de 26 bits utilizam esse formato de instrução. Esses 26 bits são combinados com os bits de ordem superior do contador de programa para obter o endereço absoluto." (Dandamudi, 2003, p. 620, tradução nossa)

[3] "Instruções aritméticas e lógicas utilizam esse formato de instrução. Além disso, a instrução de salto na qual o endereço de destino é especificado indiretamente por meio de um registrador também utiliza esse formato de instrução." (Dandamudi, 2003, p. 620, tradução nossa)

3. METODOLOGIA

Nesta seção, detalharemos o processo de implementação do nosso desmontador para a arquitetura MIPS. Esta seção tem como objetivo fornecer uma compreensão clara e completa do processo de trabalho e dos métodos utilizados.

3.1 ESTRUTURAÇÃO DO PROJETO

A estruturação do projeto desempenha um papel fundamental na organização e no desenvolvimento eficiente do desmontador. Esta subseção descreverá a organização do código-fonte e a arquitetura geral do projeto, a divisão dos arquivos de código pode ser visualizada a seguir:

a) constants.asm: contém definições de constantes e valores importantes utilizados ao longo do projeto.
b) main.asm: o arquivo de entrada do desmontador, coordenando o fluxo de execução, a leitura do arquivo binário, a formatação das instruções e a geração do arquivo de saída com o código desmontado.
c) decoder.asm: contém a lógica necessária para buscar o tipo da instrução, separar os campos e interpretar os operandos.
d) instdefs.asm: contém as definições das instruções MIPS, incluindo os campos que compõem cada instrução, os formatos de instruções e outras informações essenciais para a correta decodificação.
e) itostr.asm: conversão de valores numéricos em representações de texto hexadecimal e decimal.
f) regdefs.asm: contém as definições dos registradores da arquitetura MIPS, mapeando números de registradores para seus nomes simbólicos, como "$zero," "$t0," entre outros.
g) signextend.asm: lida com a extensão de sinal de valores imediatos usados nas instruções MIPS, sendo essencial para garantir a interpretação correta desses valores.
h) strcpy.asm: é responsável por implementar a função de cópia de strings, útil em situações em que é necessário copiar texto de um local para outro.
i) strfmt.asm: lida com a formatação de strings para exibição no arquivo de saída, contribuindo para a apresentação clara e organizada do código desmontado.

A divisão dos arquivos proporciona a vantagem de abordar cada componente do projeto de maneira independente, de modo a facilitar, assim, a colaboração entre os membros da equipe. Além

disso, essa prática promove a legibilidade, a manutenibilidade e a modularidade do código, permitindo a reutilização de componentes em projetos subsequentes. Dessa forma, a organização do projeto, com a utilização desses arquivos distintos, contribuiu de forma significativa para a eficiência e a clareza do processo de desmontagem. Pois, cada arquivo desempenha uma função específica no desmontador, simplificando, dessa forma, a compreensão e a manutenção do código-fonte

3.2 REPRESENTAÇÃO DAS INSTRUÇÕES

As instruções da arquitetura do MIPS são representadas pelas estruturas de "Tipo de Instrução" e "Definição de Instrução" que desempenham um papel fundamental no processo de decodificação. Elas servem como uma espécie de "dicionário" que traduz os códigos de máquina brutos em instruções legíveis por humanos, permitindo que o desmontador compreenda a estrutura e os detalhes de cada instrução MIPS. Essas estruturas são projetadas para capturar as nuances dos diferentes formatos de instrução, tornando possível a extração e a exibição correta do mnemônico e dos operandos contidos nas instruções MIPS.

3.2.1 Estrutura “Tipo de Instrução”

A estrutura de "Tipo de Instrução" é um conjunto de campos que definem os diversos formatos de instruções suportadas pela arquitetura MIPS. Ela permite a identificação e a interpretação correta das instruções, distinguindo entre os diferentes tipos de formatos (R, I e J no caso do MIPS).

a) Tipo R: Usado para operações entre registradores, onde o código da operação, registradores fonte e de destino, deslocamento de bits e função específica são os campos relevantes.
b) Tipo I: Instruções com um valor imediato (constante) que é carregado ou operado com um registrador específico. Geralmente, essas instruções incluem uma constante de 16 bits e um registrador de destino.
c) Tipo J: Utilizado principalmente para instruções de salto incondicional, onde um endereço de destino é especificado.

Cada formato de instrução é representado por um vetor de campos, contendo informações como máscaras (bit a bit) e deslocamentos que permitem extrair os diferentes campos da instrução (endereços de memória, registradores, constantes etc.) durante o processo de decodificação.

3.2.2 Estrutura “Definição de Instrução”

A estrutura de "Definição de Instrução" associa uma instrução específica aos tipos de operandos respectivos a cada campo do tipo da instrução, como registradores, constantes imediatas, deslocamentos de memória, entre outros. Essa estrutura permite identificar o padrão de uma instrução e a maneira como os operandos devem ser interpretados no texto.

3.3 TEXTOS DE FORMATO

Durante a implementação do *disassembler*, incorporou-se o uso de macros para definir os formatos de instruções e determinar os tipos de operandos aceitos por cada instrução. Essas macros estabelecem uma associação entre uma instrução específica e uma string de formato, o que proporciona uma representação clara da estrutura da instrução.

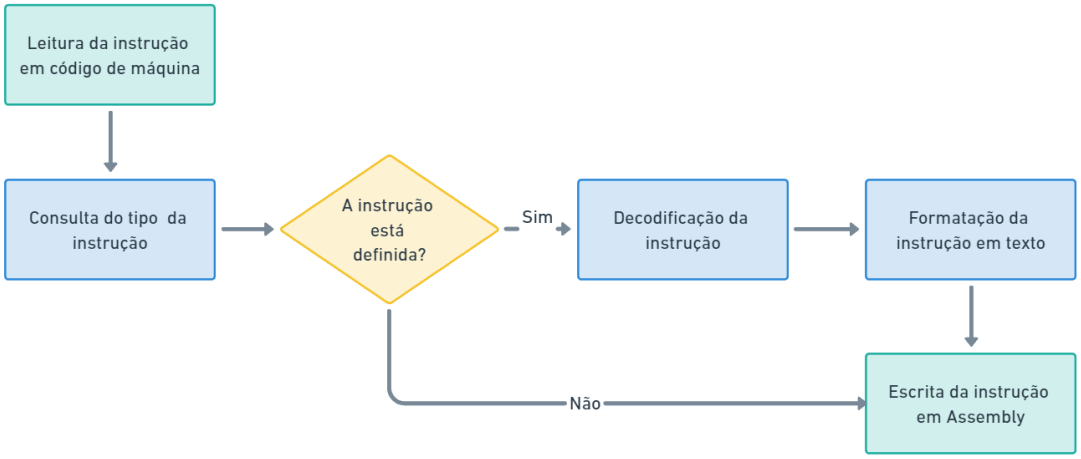
As strings de formato exercem um papel crítico, uma vez que descrevem a aparência das instruções MIPS, delineando a ordem e os tipos de operandos que cada instrução requer. Por exemplo, podemos utilizar um formato como "addi #1, #0, #2," para caracterizar uma instrução "addi" que contém três operandos distintos.

O processo de interpretação e formatação das strings de formato é realizado por meio da função "strfmt." Esta função tem a responsabilidade de analisar a string de formato, identificar as diretivas de operandos (sinalizadas por '#' seguido de um número) e, com base nas informações disponíveis no vetor de operandos, realizar a formatação dos operandos apropriados. À medida que os operandos são processados, eles são inseridos na saída do código desmontado, resultando em uma representação legível e de fácil compreensão das instruções MIPS. Tal abordagem é essencial para permitir a compreensão do código desmontado.

3.4 IMPLEMENTAÇÃO

A figura 4 representa as principais etapas do processo do desmontador.

Figura 4- Esquema de implementação do desmontador



Fonte: Autor (2023)

3.5 LEITURA DO ARQUIVO

O primeiro passo é a abertura de um arquivo de entrada e um arquivo de saída por meio das funções openfile, que realiza a abertura do arquivo de leitura “object.bin” e do arquivo de saída “assembly.txt”. Os descritores dos arquivos abertos são armazenados nas variáveis “input_descriptor” e “output_descriptor”. Após isso, o programa lê blocos de bytes do arquivo de entrada, armazenando-os no buffer read_buffer. A quantidade máxima de bytes que pode ser lida de uma só vez é definida pela constante READ_BUFFER_SIZE.

3.6 DECODIFICAÇÃO

Para realizar a decodificação das instruções, primeiramente, a função "decodeinst" exerce a função de supervisão no processo de decodificação. Em primeiro momento, essa função é parametrizada com o endereço da estrutura "Instrução Decodificada," destinada a armazenar informações referentes à instrução sob análise. Ademais, são fornecidos à função o endereço da posição da instrução (ou seja, o endereço de memória onde a instrução está alojada) e o código de máquina da instrução.

Primeiramente, ocorre a chamada da função "lookupinst," esta associa o código de máquina da instrução a uma definição específica, a qual inclui detalhes sobre os campos da instrução e os tipos de operandos aceitáveis. Através do opcode, a pesquisa é realizada em uma tabela de

definições específica para aquele opcode, essa função identifica a definição correspondente ao código de máquina fornecido, simplificando o processo de decodificação. Essa abordagem permite que o desmontador entenda imediatamente a estrutura da instrução e os tipos de operandos envolvidos, agilizando o processo de interpretação e garantindo a precisão da decodificação.

Após obter a definição da instrução, a função "decodeinst" procede com a validação da definição. Esta etapa possui grande importância, uma vez que, na ausência de uma definição válida (ou seja, quando o endereço da definição é nulo), a instrução torna-se intraduzível. Caso este cenário ocorra, a função "decodeinst" encerra o processo de imediato, dada a impossibilidade de decodificação.

No caso da identificação de uma definição válida, a função "decodeinst" prossegue com a decodificação dos operandos, recorrendo à função "extractops." Esta função é responsável por isolar e decodificar os operandos da instrução com base na definição da instrução e nos tipos de operandos aceitos. Este procedimento engloba a utilização de máscaras e deslocamentos para extrair os valores apropriados dos campos do código de máquina. Os valores extraídos são, em seguida, interpretados de acordo com o seu tipo dado pela definição da instrução e, por fim, arquivados na estrutura "Instrução Decodificada."

3.7 ARQUIVO DE SAÍDA

Após a formatação das instruções (sejam elas conhecidas ou desconhecidas), é necessário escrevê-las no buffer de saída. O buffer de saída é uma área de memória destinada a armazenar as instruções formatadas em formato de texto. Cada instrução é escrita sequencialmente do buffer de de saída, garantindo a ordem correta.

Para determinar a quantidade de bytes a serem escritos no arquivo, o código calcula o tamanho da string formatada. A escrita efetiva no arquivo de saída é realizada por meio da chamada da função "writetofile". Essa função é responsável por escrever os dados do buffer de saída no arquivo de saída “object.bin”. Ela utiliza o descritor do arquivo para direcionar a saída adequadamente.

3.8 VERIFICAÇÕES DE ERROS

Durante a execução do programa, realiza-se diversas verificações de erros, principalmente relacionadas à abertura, leitura e escrita de arquivos. Busca-se assegurar a confiabilidade do programa e lidar com situações inesperadas de maneira apropriada.

Quando o programa abre os arquivos de entrada e saída, verifica-se o estado da operação, ou seja, se a operação foi bem-sucedida ao utilizar o valor de retorno da função openfile. Se a abertura de um arquivo falhar (ou seja, se o descritor for menor que zero), o programa imprime uma mensagem de erro. Após a impressão da mensagem de erro, o programa é encerrado com um código de retorno 1 (indicando erro) usando a função exit.

Durante a leitura do arquivo de entrada, o programa verifica o valor de retorno da função readfile. Se esse valor for negativo, indica um erro na leitura do arquivo. Nesse caso, o programa imprime outra mensagem de erro específica e é encerrado usando a função exit.

A verificação de erros também se aplica à escrita das instruções decodificadas no arquivo de saída, onde o valor de retorno da função writetofile é examinado. Se esse valor for menor que zero, sinaliza um erro na escrita do arquivo. Nessa situação, o programa imprime uma mensagem de erro específica e é encerrado usando a função exit.

Essas verificações de erros são fundamentais para manter a integridade das operações de abertura, leitura e escrita de arquivos. Garantindo que o programa possa lidar adequadamente com falhas imprevistas durante a execução.

4. EXPERIMENTOS

Esta seção busca realizar uma grande quantidade de testes para demonstrar o comportamento e os resultados do desmontador. Os experimentos abordam uma variedade de cenários, desde exemplos simples até casos mais complexos, com o objetivo de avaliar a precisão e a eficácia do desmontador em converter as instruções da arquitetura MIPS.

4.1 INSTRUÇÕES BÁSICAS

As instruções da arquitetura MIPS são projetadas para serem executadas de forma independente e direta, sem a necessidade de avaliação em conjunto com outras instruções. Desse modo, o processo de tradução é simplificado, pois as instruções possuem sempre um único comportamento, indiferente do seu entorno. A tabela 2 explora esse paradigma e exibe o desmontador em casos isolados para cada um dos principais tipos de instruções.

Tabela 2: Conversão de instruções

Instrução original	Linguagem de máquina	Instrução traduzida
add $t0, $t1, $t2	0x012A4020	add $t0, $t1, $t2
slti $s0, $s1, 255	0x2A3000FF	slti $s0, $s1, 255
j 0x0040000C	0x08100003	j 0x0040000C
nop	0x00000000	sll $zero, $zero, $zero

Fonte: Autor (2023)

4.2 PROGRAMA DE TESTE

Por fim, exportamos as instruções em código de máquina de um programa completo para examinar o desmontador quando submetido a uma grande quantidade de instruções. A figura 5 ilustra o arquivo de entrada utilizado.

Figura 5- Arquivo de entrada

```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  F8 FF BD 27 05 00 08 24 04 00 A8 AF 20 20 08 00
00000010  0F 00 10 0C 04 00 A8 8F 21 48 02 00 00 00 A2 AF
00000020  21 20 08 00 21 28 09 00 1C 00 10 0C 08 00 BD 27
00000030  11 00 02 24 00 00 04 24 0C 00 00 00 F8 FF BD 27
00000040  04 00 BF AF 00 00 A4 AF 02 00 04 14 01 00 02 20
00000050  1A 00 10 08 FF FF 84 20 0F 00 10 0C 00 00 A4 8F
00000060  02 10 82 70 04 00 BF 8F 08 00 BD 23 08 00 E0 03
00000070  FC FF BD 27 00 00 A4 AF 04 00 02 24 01 10 01 3C
00000080  00 00 24 34 0C 00 00 00 00 00 A4 8F 01 00 02 24
00000090  0C 00 00 00 01 10 01 3C 0F 00 24 34 04 00 02 24
000000A0  0C 00 00 00 21 20 05 00 01 00 02 24 0C 00 00 00
000000B0  0A 00 04 24 0B 00 02 24 0C 00 00 00 04 00 BD 27
000000C0  08 00 E0 03
```

Fonte: Autor (2023)

O programa é composto por um total de 49 instruções, sendo 13 distintas, que incluem operações aritméticas e lógicas, pulos condicionais e incondicionais, chamadas de sistema, entre outras. Assim, garantimos o processamento de cada um dos formatos de instrução e de cada um dos tipos de operandos múltiplas vezes em uma diversidade de situações.

5. RESULTADOS E DISCUSSÃO

Primeiramente, conduziu-se experimentos com instruções básicas da arquitetura MIPS. Como mencionado no tópico 4.1. Os resultados desses testes indicam que o desmontador é capaz de converter as instruções da linguagem de máquina para sua representação legível com precisão. Exemplos de instruções, seus equivalentes em linguagem de máquina e as instruções traduzidas foram apresentados na tabela 2.

Os resultados desses testes demonstram que o desmontador consegue identificar e converter corretamente as instruções de adição (tipo r), comparação imediata (tipo i), desvio incondicional (tipo j), e a instrução nop, que não possui efeito em um contexto de execução. O desmontador também manteve a formatação das instruções traduzidas de acordo com a notação MIPS. No entanto, a instrução nop foi interpretada como uma instrução do tipo sll. Embora essa correspondência possa parecer menos intuitiva, é, na realidade, uma representação precisa, pois a instrução nop é tradicionalmente implementada em processadores MIPS como um deslocamento de zero bits à esquerda.

Em seguida, avaliou-se o desempenho do desmontador ao lidar com um programa completo. O arquivo de teste consistia em 49 instruções, incluindo uma variedade de operações aritméticas, lógicas, desvios condicionais e incondicionais, chamadas de sistema e outras instruções distintas. O objetivo era verificar se o desmontador seria capaz de lidar com uma grande quantidade de instruções e diferentes tipos de operandos de forma eficaz.

Figura 6- Resultado do desmontador

```
1   0x00400000 0x27BDFFF8 addiu $sp, $sp, -8
2   0x00400004 0x24080005 addiu $t0, $zero, 5
3   0x00400008 0xAFA80004 sw $t0, 4($sp)
4   0x0040000C 0x00082020 add $a0, $zero, $t0
5   0x00400010 0x0C10000F jal 0x0040003C
6   0x00400014 0x8FA80004 lw $t0, 4($sp)
7   0x00400018 0x00024821 addu $t1, $zero, $v0
8   0x0040001C 0xAFA20000 sw $v0, 0($sp)
9   0x00400020 0x00082021 addu $a0, $zero, $t0
10  0x00400024 0x00092821 addu $a1, $zero, $t1
11  0x00400028 0x0C10001C jal 0x00400070
12  0x0040002C 0x27BD0008 addiu $sp, $sp, 8
13  0x00400030 0x24020011 addiu $v0, $zero, 17
14  0x00400034 0x24040000 addiu $a0, $zero, 0
15  0x00400038 0x0000000C syscall
16  0x0040003C 0x27BDFFF8 addiu $sp, $sp, -8
17  0x00400040 0xAFBF0004 sw $ra, 4($sp)
18  0x00400044 0xAFA40000 sw $a0, 0($sp)
19  0x00400048 0x14040002 bne $zero, $a0, 0x00400054
20  0x0040004C 0x20020001 addi $v0, $zero, 1
21  0x00400050 0x0810001A j 0x00400068
22  0x00400054 0x2084FFFF addi $a0, $a0, -1
23  0x00400058 0x0C10000F jal 0x0040003C
24  0x0040005C 0x8FA40000 lw $a0, 0($sp)
25  0x00400060 0x70821002 mul $v0, $a0, $v0
26  0x00400064 0x8FBF0004 lw $ra, 4($sp)
27  0x00400068 0x23BD0008 addi $sp, $sp, 8
28  0x0040006C 0x03E00008 jr $ra
29  0x00400070 0x27BDFFFC addiu $sp, $sp, -4
30  0x00400074 0xAFA40000 sw $a0, 0($sp)
31  0x00400078 0x24020004 addiu $v0, $zero, 4
32  0x0040007C 0x3C011001 lui $at, 4097
33  0x00400080 0x34240000 ori $a0, $at, 0
34  0x00400084 0x0000000C syscall
35  0x00400088 0x8FA40000 lw $a0, 0($sp)
36  0x0040008C 0x24020001 addiu $v0, $zero, 1
37  0x00400090 0x0000000C syscall
38  0x00400094 0x3C011001 lui $at, 4097
39  0x00400098 0x3424000F ori $a0, $at, 15
40  0x0040009C 0x24020004 addiu $v0, $zero, 4
41  0x004000A0 0x0000000C syscall
42  0x004000A4 0x00052021 addu $a0, $zero, $a1
43  0x004000A8 0x24020001 addiu $v0, $zero, 1
44  0x004000AC 0x0000000C syscall
45  0x004000B0 0x2404000A addiu $a0, $zero, 10
46  0x004000B4 0x2402000B addiu $v0, $zero, 11
47  0x004000B8 0x0000000C syscall
48  0x004000BC 0x27BD0004 addiu $sp, $sp, 4
49  0x004000C0 0x03E00008 jr $ra
50
```

Fonte: Autor (2023)

Durante o processo de teste, observou-se que o desmontador foi capaz de desmontar todas as instruções corretamente, mantendo a estrutura e formatação esperadas de acordo com a arquitetura

MIPS. Incluindo a interpretação adequada de rótulos, identificação de operadores e operandos. A conversão para código assembly pode ser visualizado na figura 6.

Assim, percebe-se que os experimentos realizados forneceram resultados positivos, indicando que o desmontador é capaz de converter com precisão e eficácia diversas instruções da arquitetura em questão. Mostrou-se adequado para lidar com uma ampla gama de cenários, desde instruções simples até programas complexos, garantindo a fidelidade na tradução e a manutenção da estrutura da linguagem proposta. Evidenciando resultados promissores para aplicações que dependem da análise e compreensão de programas que estejam em linguagem de máquina.

CONCLUSÃO E PERSPECTIVAS

Neste trabalho, desenvolveu-se um desmontador para a arquitetura de processador MIPS, com o objetivo de transformar códigos em linguagem de máquina em linguagem assembly. O desmontador foi projetado para atender a uma variedade de instruções, incluindo operações aritméticas, lógicas, de desvio, chamadas de sistema e outras. Além disso, elaborou-se para ser facilmente extensível e acomodar novas instruções, visto que as instruções de ponto flutuante não foram implementadas neste projeto.

Durante os experimentos, verificou-se que o desmontador foi capaz de converter com precisão todas as instruções testadas, mantendo a estrutura e a formatação esperadas. Demonstrou consistência na interpretação de rótulos, identificação de operadores e operandos, e na geração de código assembly.

O projeto se beneficiou da estruturação do código em vários arquivos independentes, o que promoveu a modularidade e a manutenibilidade do desmontador. A utilização de estruturas de "Tipo de Instrução" e "Definição de Instrução" permitiu uma decodificação eficaz das instruções, garantindo a interpretação correta dos operandos. O desmontador também apresentou bom desempenho ao lidar com um programa completo. Demonstrando sua capacidade de processar uma grande quantidade de instruções em diferentes cenários.

Perspectivas futuras para o desenvolvimento deste desmontador incluem a expansão do suporte a instruções adicionais da arquitetura MIPS, aprimoramento na detecção de rótulos em programas complexos, criação de uma interface gráfica, documentação abrangente para facilitar o uso, suporte a múltiplas arquiteturas, otimizações de desempenho para aumentar a eficiência de processamento e, em geral, o contínuo aperfeiçoamento deste desmontador. Dessa forma, pode-se concluir que, este projeto alcançou seus objetivos previstos, proporcionando uma ferramenta valiosa para a análise e compreensão de programas escritos para a arquitetura MIPS.

REFERÊNCIAS

ANDRIESSE, Dennis, et al. **An In-Depth Analysis of Disassembly** on Full-Scale x86/x64 Binaries, 2016.

DANDAMUDI, Sivarama P. **MIPS Assembly Language.** Fundamentals of Computer Organization and Design, p. 615-661, 2003.

DE BRUIJIN, M. **Switch Statement Disassembly**, 2022.

HENNESSY, John et al. **MIPS: A microprocessor architecture.** ACM SIGMICRO Newsletter, v. 13, n. 4, p. 17-22, 1982.

JAMIL, Tariq. **Risc versus cisc**. Ieee Potentials, v. 14, n. 3, p. 13-16, 1995.

MASOOD, Farhat. Risc and cisc. arXiv preprint arXiv:1101.5364, 2011.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design: The Hardware/Software Interface**. 5th ed. Morgan Kaufmann, 2014.

POPA, Marius. **Binary code disassembly for reverse engineering.** Journal of Mobile, Embedded and Distributed Systems, v. 4, n. 4, p. 233-248, 2012.

PET-SI, 2023. **MARS: IDE para programação em Assembly.** Disponível em: [https://www.ufsm.br/pet/sistemas-de-informacao/2018/09/01/mars-ide-para-programacao-em-assembl]. Acesso em: 27 out. 2023.

VOLLMAR, Kenneth; SANDERSON, Pete. **MARS: an education-oriented MIPS assembly language simulator.** In: Proceedings of the 37th SIGCSE technical symposium on Computer science education. p. 239-243. 2006