Desenvolvimento de Aplicações em Assembly MIPS

Departamento de Ciência da Computação - Universidade de Brasília
Disciplina: Organização e Arquitetura de Computadores
Professor: Flávio de Barros Vidal
Turma: B

Pedro Saman Cesarino - 15/0144890
Faculdade de Tecnologia - UnB
pedrosaman96@gmail.com

Rafael Augusto Torres - 15/0145365
Faculdade de Tecnologia - UnB
rafael.a.t97@gmail.com

Brasília
5 de março de 2020

Abstract

Developing applications using assembly is interesting when it's necessary to have a precise execution time management since the algorithm doesn't need to be translated and/or compiled in advance of being assembled. That can reduce the difference between the written code and the machine-code instructions. In this project we used the language Assembly-MIPS to implement the Lempel-Ziv algorithm of coding and decoding files. During the project's execution we developed means of critical thinking towards how to use the many tools available of this language, we also noted the limitations of the proposed algorithm when used in certain cases.

Resumo

Desenvolver aplicações em assembly é interessante quando se deseja ter uma boa gestão do tempo de execução de determinada aplicação, uma vez que o algoritmo não precisa ser traduzido e/ou compilado, reduzindo então a diferença entre o código escrito e as intruções em código de máquina. Nesse projeto buscamos utilizar a linguagem Assembly-MIPS para implementar o algoritmo Lempel-Ziv de codificação e decodificação de arquivos. Ao longo do projeto desenvolvemos um pensamento crítico em relação aos usos de diferentes ferramentas dessa linguagem, notando também as limitações do algoritmo proposto quando aplicado em determinados casos.

1. Objetivos

O principal objetivo desse projeto foi propiciar ao aluno um método de familiarização com a linguagem *Assembly MIPS* e suas metodologias, buscando estudar suas aplicações de modo eficiente e otimizando, assim como as convenções utilizadas ao se programar nessa linguagem. A realização do projeto busca complementar os estudos teóricos adquiridos ao longo da disciplina 'Organização e Arquitetura de Computadores'.

Como objetivo secundário desse projeto, buscou-se implementar uma aplicação que realiza a codificação e decodificação de arquivos de acordo com o algoritmo Lempel-Ziv, versão LZ78, analisando seu funcionamento e avaliando o seu desempenho em comprimir arquivos, comparando o tamanho inicial do arquivo original com o tamanho final do arquivo comprimido gerado.

2. Introdução

Para compreender os resultados e implementações realizadas nesse projeto, é necessário ter conhecimento sobre a linguagem de programação *Assembly MIPS*, assim como sobre o método Lempel-Ziv de codificação de arquivos, portanto, uma leve introdução sobre esses assuntos será fornecida logo a seguir. Como os processadores com arquitetura MIPS estão sendo utilizados em aplicações específicas e os estudantes não possuem acesso direto a um processador desse tipo, foi necessário utilizar um ambiente que simulasse o funcionamento desse processador.

2.1. Método Lempel-Ziv

Lempel-Ziv é um método de compressão e descompressão de arquivos que se baseia no uso de um dicionário

que codifica strings. A variante LZ78 dessa implementação mantém um dicionário com as frases já encontradas indexadas por uma chave. Esse algoritmo costuma ser utilizado em compressão de arquivos muito repetitivos ou em imagens em que não se pode perder a resolução original [2].

A compressão consiste em vários pares formados por um número e um caractere que indicam uma chave do dicionário e um caractere que quebrou aquela sequência. A cada passo do algoritmo lê-se caracteres do arquivo original até se encontrar algum conjunto destes que ainda não existe no dicionário, quando isso ocorrer é impresso no arquivo comprimido o índice do conjunto que compõe a frase até ali mais o caractere que quebrou a sequência e, então, é adicionado esse novo conjunto ao dicionário [2].

Agora para realizar a descompressão são necessários dois valores localizados no arquivo compactado a cada iteração: o número do índice e o caractere que quebrou a sequência. Com esse número é impressa a sequência associada a ele no dicionário e depois o caractere pego antes. Dessa forma é reescrito o arquivo original [2].

2.2. Assembly MIPS

A linguagem de programação *Assembly MIPS* é uma linguagem voltada para o controle baixo nível de processadores que utilizam a arquitetura MIPS. Por ser uma linguagem baixo nível, aplicações criadas utilizando essa linguagem não necessitam de serem compiladas, podendo ser prontamente submetidas ao processo de montagem, ou seja, o código fonte não precisa ser traduzido, mas há a necessidade única de se eliminar pseudo-instruções transformando-as em instruções nativas, já convertendo diretamente todas as instruções para código de máquina para posteriormente serem executadas [1].

Por ser considerada uma arquitetura do tipo RISC, *Reduced Instruction Set Computer*, a arquitetura MIPS busca reduzir o número de instruções de sua ISA, *Instruction Set Architecture*, desse modo, o número de instruções disponíveis para se escrever o código fonte de programas que utilizam a linguagem *Assembly MIPS* é limitado. Por este fato, a arquitetura MIPS é bastante atrativa para iniciantes, pois o uso repetitivo de mnemônicos mais simples permite ao programador entender mais facilmente como realmente funciona um processador no baixo nível, entendo também o uso dos 32 registradores presentes na arquitetura MIPS.

2.3. MARS

Para simular a atuação do processador com arquitetura MIPS, utilizou-se durante a realização do projeto o ambiente MARS. Esse ambiente, criado com base na linguagem de programação java, permite ter controle dos registradores do processador e seus valores, assim como os valores na memória, durante a simulação de um programa, facilitando a correção de problemas mais facilmente pelos programadores.

Esse ambiente de simulação possui também algumas ferramentas interessantes, algumas possibilitam mensurar o desempenho do código escrito, como por exemplo realizar a contagem de instruções de cada tipo, já outras ferramentas fornecem outros meios de visualizar o fluxo do código, mas o tipo de ferramenta mais utilizada ao longo do trabalho foram as chamadas de sistema, ou *syscall*, que permitem ao simulador MARS utilizar serviços de sistema, como por exemplo abrir arquivos e utilizar métodos especiais de entrada e saída do programa.

3. Materiais

- Ambiente de simulação MARS;
- Computadores.

4. Métodos

Para melhor desenvolvimento do algoritmo pedido a dupla se organizou para discutir qual abordagem seria melhor e foi decidido que iriamos, após devido entendimento do problema, desenvolver um pseudocódigo escrito a mão para cada aspecto do trabalho, que foram três ao total: montagem do dicionário na memória, comprimir um arquivo de extensão “.txt” e efetivamente escrever o arquivo compactado em um arquivo com extensão “.lzw” e, em um programa separado, descompactar esse arquivo “lzw” de volta para um arquivo “.txt”.

4.1. Compactação

Para realizar a compactação, foram utilizados 3 *buffers* para guardar strings e/ou montar o dicionário que será escrito. Ao iniciar o programa, uma janela é criada que pergunta ao usuário o nome do arquivo a ser compactado, o usuário deve então digitar o nome de um arquivo que contenha no máximo 25 caracteres e que esteja na mesma pasta onde se localiza o arquivo “.jar” do simulador MARS, caso o programa não encontre o arquivo solicitado ou o nome do arquivo contenha algum problema, uma mensagem de erro será escrita na tela e o programa será finalizado, sem gerar um arquivo compactado ou um dicionário.

A lógica utilizada para criar o arquivo compactado foi a seguinte: Foi escolhido no .data um endereço de memória que começava alinhado com uma .word para ser o início do dicionário, de tal modo que o label que guarda este início será sempre constante. Foi definido ao longo do programa um registrador responsável por armazenar quantos pares do tipo <.word, .ascii>, ou <tamanho, string>, o dicionário contém, sendo esse iniciado como 0. Para realizar a leitura do arquivo a ser compactado foi decidido ler um caractere por vez, esse caractere lido era concatenado a um buffer e esse buffer era então comparado com as strings no dicionário. Para tentar otimizar um pouco essa comparação,

a .word que é inserida junto com as strings no dicionário contém o tamanho dessa string, e não o seu índice, pois como percorremos o dicionário de modo linear, o valor do índice daquela string já estará salvo em um registrador, logo, evitamos redundâncias e ganhamos um pouco de tempo para realizar os testes necessários com os códigos criados. Inicialmente então, o tamanho do buffer é comparado com o tamanho da string do par atual no dicionário, caso os tamanhos sejam diferentes, o programa já avança para o próximo par, caso sejam iguais, será realizada então uma comparação caractere a caractere até encontrar uma correspondência ou não entre as strings. Para avançar a leitura do dicionário para o próximo ponto alinhado com uma .word que indica o tamanho da string do próximo par a ser comparado, utilizamos uma simples fórmula matemática para avançar ao próximo espaço de memória livre e múltiplo de 4.

Ao percorrer todo o dicionário e não encontrar uma correspondência com nenhuma de suas strings, um novo par será acrescentado ao dicionário com a nova string e seu tamanho, incrementando também o registrador que representa o tamanho do dicionário. Quando uma nova string é inserida no dicionário, significa que algum caractere provocou a presença de uma nova string, onde o buffer antes de ter esse caractere concatenado possuia alguma correspondência com alguma string no dicionário, logo, esse caractere foi o responsável por "quebrar" essa correspondência, por isso, ao ser inserida uma nova string no dicionários, inserimos também no arquivo compactado qual era o índice do par no dicionário cuja string era idêntica àquela no buffer e inserimos também o caractere responsável por terminar com essa correspondência, seguindo o algoritmo de Lempel-Ziv. Desse modo, o nosso arquivo criado compactado é continuamente modificado até que se leiam todas as strings do arquivo de entrada, quando ele será realmente completo.

Quando a codificação atinge seu fim, o arquivo de saída estará completo e terá o mesmo nome do arquivo de entrada, com o detalhe de trocar a sua extensão de ".txt" para ".lzw", onde serão visíveis as relações de índices de pares no dicionário e caracteres de quebra no seguinte modo: índice-caractere.

O código fonte para realizar esse processo de codificação foi enviado como "compact.asm".

4.2. Montagem do Dicionário

Como já dito na seção de compressão do arquivo acima, o dicionário é criado de tal modo que o seu começo é guardado por um label criado na área .data do programa de codificação. A string vazia foi omitida do dicionário por ser redundante, do mesmo modo que os índices das strings foram omitidos ao longo da criação do dicionário enquanto se codificava o arquivo de saída. Quando o arquivo de entrada for totalmente codificado, o dicionário já estará pronto para ser escrito em um arquivo do tipo ".txt", para que seja verificado o correto funcionamento do algoritmo de codificação.

Os dados foram armazenados no dicionário de saída, definido como "dictionary.txt", no seguinte modo: "índice∼string∼". O uso do caractere ∼ será explicado na próxima seção deste documento.

4.3. Descompactação

A descompactação do arquivo serve como um bom meio de se verificar se o algoritmo utilizado para a compactação está funcionando adequadamente já que é possível fazer uma comparação do arquivo descompactado com o original e verificar se eles são realmente idênticos. A realização dessa operação foi mais simples do que as anteriores, pois já tínhamos o dicionário montado em um arquivo texto, então, era necessário apenas procurar cada chave do arquivo compactado no arquivo do dicionário.

A implementação foi feita da seguinte forma: leia do arquivo .lzw desejado um caractere por vez e o concatene em um buffer até achar um "-", então temos concatenado o inteiro que indica qual chave do dicionário forma o par com a string desejada, atentando à conversão de char para inteiro, então leia mais um caractere que indica o que quebrou aquela sequência. Com esses dados, procuramos então no arquivo *dictonary.txt* pela chave que temos guardada, em seguida, pegamos o valor associado àquela chave e escrevemos no *uncompressed_file.txt* tal valor e então escrevermos também o caractere que tínhamos lido após a chave do dicionário. Repetiu-se esse procedimento até que o *syscall* de leitura acusasse *EOF*.

Toda essa rotina de descodificar os arquivos foi enviada como "decompress.asm".

Um detalhe importante que tem que ser levado em conta é que, graças à forma de implementação, a descompactação trava se houver o caractere "∼"presente no texto, por isso esse caractere não deve aparecer no texto original. Isso ocorre porque na descompactação o algoritmo procura o caractere "∼" no dicionário para saber onde começa e onde termina chave e conteúdo. Foi escolhido esse caractere como divisão no arquivo pois ele não é um caractere que aparece normalmente em textos. Poderia ter sido escolhido algum caractere especial de controle presente na tabela ascii para evitar esse problema, porém a visualização humana do dicionário ficaria prejudicada já que esse tipo de caractere não tem representação visual simples em ascii e como o trabalho tem caráter de aprendizado optou-se por deixar o caractere "∼" fazendo a divisão.

4.4. Rotina de Testes

Para testar o desempenho das aplicações de codificação e decodificação criadas, foram utilizados determinados benchmarks, que serão enviados junto à este relatório. Como os

benchamarks fornecidos pelo professor eram relativamente grandes e o algoritmo utilizado para codificar e decodificar os arquivos se mostrou relativamente lento, além do simulador em java utilizar grande parte do processamento de CPU dos computadores enquanto realizavam os testes, optou-se por utilizar benchmarks menores mas que também fornecem resultados que valessem a pena ser analisados.

Para verificar qual o tipo de instruções mais utilizado ao longo da codificação e decodificação, tentamos utilizar uma ferramenta do MARS que nos forneceria esse tipo de informação, mas, infelizmente, os computadores utilizados para rodar os testes muitas vezes não suportavam o processamento exigido pelo MARS quando a contagem de instruções atingia valores muito altos e a aplicação do MARS travava. Porém, em alguns casos, foi possível obter alguma informação da contagem de instruções antes da aplicação cessar o seu funcionamento correto e houve até um caso onde a aplicação finalizou sem demais problemas.

5. Resultados

Inicialmente, é necessário descrever os arquivos de teste utilizados:

- data1reduzido: Esse arquivo foi criado utilizando-se somente as 500 primeiras linhas do benchmark data1.txt fornecido pelo professor. Esse arquivo se trata de um arquivo texto bastante repetitivo, se assemelhando a um banco de dados com entradas semelhantes.
- data2reduzido: Assim como o primeiro arquivo teste, foi necessário reduzir para 1539 linhas o tamanho do benchmark data2.txt fornecido pelo professor.
- data3: Esse arquivo trata-se do benchmark data3.txt em sua íntegra, que é um texto pouco repetitivo.
- textoingles: Esse arquivo está escrito em inglês e foi pego aleatoriamente da *Wikipedia* para usar como comparação entre as línguas.
- textoportugues: Buscando analisar a diferença de repetições entre o inglês e o português, foi realizada uma colagem de diferentes artigos em português da wikipédia, levando-se em conta apenas caracteres da tabela ascii normal, para servir como benchmark.
- teste1: Esse pequeno arquivo é o usado como exemplo na referência dada pelo professor. Foi usado apenas nos estágios iniciais do projeto para fins de verificação da resolução e identificação de possíveis erros
- binario: Buscando analisar a aplicação do algoritmo de Lempel-Ziv para arquivos binários, foi gerado um arquivo desse tipo aleatoriamente, o arquivo apresenta ao todo 1000 linhas e cada linha possui 32 bits como dado.

Tabela 1. Tamanho original dos arquivos

Nome do arquivo	Tamanho original
data1reduzido	60,2 kB
data2reduzido	58,9 kB
data3	269,1 kB
textoingles	3,6 kB
textoportugues	3,6 KB
teste1	14 B
binario	33 kB

Tabela 2. Tamanho dos arquivos e dicionários gerados

Nome	Arq. comprimido	Dicionário
data1reduzido	38,9 kB	101,2 kB
data2reduzido	53,4 kB	128,3 kB
data3	259,5 kB	547,2 kB
textoingles	5,3 kB	9,6 kB
textoportugues	5,1 kB	9,4 kB
teste1	27 B	41 B
binario	19,5 kB	53,7 kB

Na tabela 3 podemos ver percentualmente o quanto cada texto diminuiu após a compactação. Os valores negativos indicam que os arquivos aumentaram.

Tabela 3. Redução do tamanho na compressão

Nome	Porcentagem de redução
data1reduzido	35,38%
data2reduzido	9,34%
data3	3,57%
textoingles	-47,22%
textoportugues	-41,67%
teste1	-92,86%
binario	40,91%

Como dito anteriormente, buscou-se analisar também qual tipo de instruções foi mais utilizado ao longo dos testes, mas, devido ao grande processamento pedido pelo MARS, esta medida não pôde ser concretizada de maneira correta, no entanto, alguns resultados ainda puderam ser obtidos, dado a maneira interativa de como é feita a contagem de instruções.

Visto que a contagem de instruções é feita em tempo real enquanto o MARS executa o programa, é possível tirar conclusões e verificar qual seria a porcentagem de cada tipo de instrução ao longo da decodificação e codificação. Diversos testes foram realizados com diversos benchmarks e a porcentagem de cada tipo de instruções aparentava sempre convergir para um valor constante, de acordo com a tabela 4.

Tabela 4. Contagem de instruções na compactação

Tipo de instrução	Porcentagem
ALU	48%
Jump	14%
Branch	15%
Memory	8%
Other	15%

Essa distribuição de instruções foi percebida quando já haviam sido realizadas aproximadamente 2 milhões de instruções ao total, e, assim como foi dito anteriormente, essa mesma distribuição foi observada para diversos casos testes, mesmo quando um caso teste finalizou sem problemas enquando se realizava a contagem de instruções, totalizando 25 milhões de instruções, logo, admite-se que essa é a distribuição correta para a rotina de codificação.

Já no processo de descompactação a distribuição das porcentagens pode ser vista na tabela 5. Essa distribuição foi constatada durante a descompactação do arquivo textoingles e teve um total de 24 milhões de instruções. Foram executados a decodificação para os arquivos textoportugues e binario e chegou-se à mesma distribuição indicada na tabela 5.

Tabela 5. Contagem de instruções na descompactação

Tipo de instrução	Porcentagem
ALU	56%
Jump	5%
Branch	10%
Memory	8%
Other	21%

6. Discussão e Conclusões

Analisando os resultados propostos nas tabelas 1 e 2, vemos que o algoritmo de Lempel-Ziv deve ser considerado mais como método de codificação do que método de compressão. Para se garantir que o arquivo codificado seja decodificado corretamente, é necessário que o dicionário seja enviado junto com o arquivo gerado, e como o dicionário é sempre maior em tamanho do que o arquivo original, esse método se prova pouco eficiente na compressão de arquivos. Ignorando agora o tamanho do dicionário gerado, vemos que para textos repetitivos e/ou grandes, o método prova ser mais eficiente, enquanto que quando utilizado em arquivos pequenos, o arquivo codificado pode ser até maior do que o arquivo original.

Vemos também que o método apresenta melhores resultados quando aplicado em arquivos com dados repetitivos, como pode ser visto ao se analisar a diferença entre o original e o arquivo codificado para os benchmarks dos arquivos "data1reduzido" e "binario", mesmo que o arquivo "binario" seja considerado relativamente pequeno.

O maior arquivo codificado foi o benchmark "data3", que apresentou uma leve taxa de compressão, sendo que o arquivo comprimido apresenta 96,43% do tamanho original, um resultado pouco agradável, mas esperado tendo em mente que esse benchmark se tratava de um arquivo pouco repetitivo. Esse arquivo, que era o menor dentre aqueles fornecidos para teste pelo professor, demorou aproximadamente 7 horas para ser totalmente codificado, o que indica uma certa incapacidade de se rodarem os outros benchmarks em sua totalidade, pois isso exigiria um gasto tremendo de processamento do computador, resultando em gasto de energia e tempo onde o computador ficaria parcialmente inviabilizado de realizar outras tarefas.

Foi realizada também uma simples comparação entre as línguas português e inglês, buscando verificar qual delas apresenta uma maior repetição em suas palavras. Sabendo que ambos os testes "textoingles" e "textoportugues" foram obtidos de maneira aleatoria e possuem aproximadamente o mesmo tamanho, podemos realizar uma leve comparação dos resultados obtidos nesses dois casos. Os tamanhos finais, ambos do arquivo comprimido e do dicionário foram bem semelhantes para os dois casos, com o texto em portugues apresentando um desempenho um pouco melhor, no entanto, como foram realizados apenas 1 teste de cada linguagem, pouco se pode concluir desse simples resultado, seria necessário um estudo voltado para esse campo para realmente verificar ou recusar a hipótese de que arquivos textos escritos em portugues apresentam melhor taxa de compressão do que textos em inglês, seguindo-se o algoritmo de Lempel-Ziv.

Analisando o fato de que o tamanho do dicionário foi sempre superior ao do arquivo original, pode-se sugerir uma leve mudança ao modo como o dicionário é escrito no algoritmo de Lempel-Ziv: caso o dicionário fosse escrito de modo semelhante ao como o arquivo codificado é escrito, armazenando um índice e relacionando este com um par de índice-caractere, indicando qual índice desse mesmo dicionário contém uma string igual àquela que será inserida só que sem o último caractere, o seu tamanho poderia acabar por ser reduzido, no entanto, a parte da decodificação que busca os índices no dicionário teria que ser feita de maneira recursiva, o que aumentaria o tempo de execução.

Em relação à porcentagem de cada tipo de instruções ao codificar e decodificar os testes, notamos facilmente que as instruções realizadas na ULA, Unidade Lógico e Aritmética, são predominantes sobre qualquer outro tipo, tanto para codificação quanto para decodificação, esse resultado já era esperado, tendo em mente a quantidade de laços de repetições que são criados ao longo da execução dos programas e outros fatores, como inicialização de variáveis, transformação de ascii para inteiro e afins. Em relação à codificação, vemos que os saltos, tanto condicionais (branch) e não-condicionais (jump), formam 29%

do total de instruções, novamente evidenciado o grande número de laços de repetições utilizados ao longo do programa, mas também demonstrando algumas chamadas de "funções", ou métodos, como são melhor definidos na lingugagem *Assembly MIPS*. Ainda em relação à codificação, a baixa porcentagem de instruções do tipo *memory* nos mostra que embora haja bastante acesso à memória do processador durante a execução do programa, são realizadas muito mais operações de outros tipos, essa baixa porcentagem desse tipo de instruções evidencia também a dificuldade do programa de codificação em checar no dicionário a presença de strings equivalentes antes de perceber que deve inserir uma nova entrada no dicionário, demonstrando quão lento é essa implementação.

A contagem de instruções para decodificação mostra que essa aplicação depende muito mais de instruções realizadas na ULA, os dados deixam claro também que foram utilizados muito mais bifurcações condicionais do que não condicionais.

Concluímos ao longo do projeto que o método de compressão Lempel-Ziv apresenta um baixo desempenho, quando se leva em consideração o método como é gerado o dicionário, mas notamos que esse método é melhor aplicável em textos grandes e bem repetitivos, como por exemplo arquivos binários, sendo que quando aplicado em textos pequenos, o resultado é uma codificação cujo arquivo de saída apresenta um tamanho superior ao arquivo de entrada, sendo totalmente ineficiente nesses casos.

Em relação à contagem de instruções, percebemos como funciona realmente uma linguagem do tipo RISC, pois notamos que grande parte de suas instruções são reutilizações de operações na ULA, como por exemplo a pseudo-instrução "move", que é basicamente um "add".

Ao longo do desenvolvimento do trabalho, várias ideias de diferentes métodos de implementações para resolver esse mesmo problema de codificação e decodificação com o algoritmo de Lempel-Ziv surgiram, como por exemplo: limitar o tamanho do dicionário, utilizar um método de "hash" com as strings já inseridas no dicionário para verificar mais rapidamente a presença ou não, de uma string, e utilização de uma estrutura de dados do tipo árvore para facilitar essa mesma pesquisa, logo, podemos citar como conclusão principal desse projeto a familiarização dos alunos com a linguagem *Assembly MIPS*, assim como o desenvolvimento de um pensamento crítico sobre esta, analisando se seria possível, ou não, a implementação dessas ideias para otimizar os códigos de codificação e decodificação, melhorando o seu desempenho.

Referências

[1] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes, 2013.

[2] K. Sayood. *Introduction to data compression*. Elsevier, 2005.