Compressor LZ78 em Assembly MIPS

Dep. de Ciencia da Computação – Universidade de Brasilia
Organização e Arquitetura de Computadores – Turma B
Brasília, 06 de maio de 2018

Ian Moura Alexandre
15/0129661
ianzeba@gmail.com

Igor R. O. Beduin
14/0143882
beduinigor@gmail.com

Abstract

Development of a LZ78 compressor and decoder using Assembly MIPS, comparing the compression ratio for each file and using the MARS's utility Instruction Statistics, verifying the calling procedures after running the program.

1. Introdução

Lidando-se com computação, muitas vezes depara-se com um nível muito grande de dados. Muitas vezes isso se torna um problema para que dados sejam transmitidos e mandados para outras pessoas, demandado às vezes de muito tempo para mandar ou muita memória disponível. Considerando este problema, foi-se necessário procurar uma maneira de tornar os arquivos menores, para se tornar possível esta transmissão de maneira prática e rápida, mantendo o máximo possível a sua integridade. Com o intuito de se resolver tal problema que foram criados os algoritmos de compressão.

A compressão de dados é uma maneira de se menos espaço físico para se armazenar uma quantidade maior de informações, mantendo ao máximo sua integridade. Na realização desta atividade, possuem vários algoritmos diferentes para tal função, sendo divididos em dois tipos diferentes: sem perdas (loless) e com perdas. Em muitos casos, arquivos não precisam se manter totalmente íntegros durante a operação, podendo ter perdas de detalhes durante o processo de compactação, como vídeos, imagens e áudios, mantendo-se apenas o necessário para que possa ser entendível a informação. Em outros casos, entretanto, é necessário que o arquivo se mantenha íntegro ao ser descompactado, utilizando para isso algoritmos de compactação sem perdas, sendo os principais exemplos destes é o Huffman e o Lempel-Ziv, sendo a sua versão LZ78 a trabalhada neste projeto.

O Lempel-Ziv é um algoritmo de compressão cujo funcionamento se baseia na criação de um dicionário a partir dos dados fornecidos pelo arquivo de entrada. À partir de um certo padrão de repetições no texto, ele vai registrando isto, e imprimindo como saída o índice das repetições que ocorrem no programa. Existem duas versões deste algoritmo, sendo última o LZ78. Nesta versão a saída é composta pelo índice do elemento, podendo ser um string ou um caractere, e do último caractere coletado no programa, possibilitando com que o dicionário não se mantenha da compressão para a descompressão, economizando assim memória. Entretanto, este algoritmo possui algumas desvantagens, como por exemplo a demanda por memória durante a execução do programa, para a formação do dicionário, já que a medida que o arquivo aumenta, o dicionário vai aumentando gradativamente. O LZ78 tem como característica negativa também o crescimento lento do dicionário, fazendo com que seja realmente viável somente com um número muito alto de informação lida.

Tal algoritmo pode ser implementado de diversas formas. Para a implementação mostrada no projeto será usada a linguagem Assembly MIPS, sendo uma linguagem de baixo nível utilizada pela arquitetura de processadores MIPS. Na programação feita em tal linguagem, cada instrução realizada equivale a uma operação do computador, considerando-se as instruções nativas. Ela é usada como uma forma legível de os humanos entenderem as operações e as ações que acontecem dentros dos registradores de memória. Na arquitetura MIPS, possuem 32 registradores, cada um possuindo 32 bits. Eles são utilizados para diferentes aplicações diferentes, de acordo com a tabela 1 A partir dos códigos em binário registrado neles, pode-se identificar uma certa operação, ou endereço de dado, ou dado em si.

Name	Register Number	Usage
$0	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	procedure return values
$a0-$a3	4-7	procedure arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved variables
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	OS temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	procedure return address

Figura 1. Mapa de representação dos registradores

Nesta representação, é possível ver a atuação de cada um dos registradores. Os registradores $0 possui seu valor fixo em zero, permitindo com que se tenha sempre no programa uma constante zero para poder se fazer operações com ele. Os $v0 e $v1 são registradores utilizados para o retorno de valores de procedimentos e para as chamadas de sistemas, também chamadas de *syscall*. Quando se tem uma operação onde é necessário a atuação do sistema operacional, como impressão ou leitura de dados, eles vão retornar a saída da operação. Como argumentos para esses procedimentos, utiliza-se os $a0-$a3. Durante a execução do programa, é necessário a manipulação de dados intermediários e dos dados de entrada para se obter o resultado desejado, e para isso utiliza-se os registradores de dados temporários ($t0 - $t9) e os registradores de dados salvos ($s0 - $s7). O registrador $gp, conhecido como *global pointer*, armazena o endereço do meio de um bloco de memória de 64K. Neste código, um dos registradores mais importantes utilizado nele é o $sp, ou stack pointer. Ele aponta para o topo da pilha e sua manipulação permite com que se faça uso de um espaço maior do que o disponível nos registradores

1.1. Objetivos

O projeto realizado tem como intuito o uso dos conhecimentos ensinados na disciplina Organização e Arquitetura de Computadores para a implementação de um codificador e um decodificador *Lempel-Ziv* (LZ78) na linguagem Assembly MIPS, a verificação da taxa de compactação do código e análise da chamada dos procedimentos utilizados nele a partir da ferramenta *Instruction Statistics* presente no montador MARS.

1.2. Materiais

Para a realização deste experimento foi-se utilizado apenas o software *MARS 4.5: Mips Assembler and Runtime Simulator* e os quatro arquivos de extensão .txt utilizados como entrada para o código implementado.

2. Implementação

A implementação dos códigos codificação e decodificação foram separadas nas seguintes etapas: leitura e manipulação dos arquivos, manipulação dos caracteres e implementação da lógica Lempel-Ziv para codificação e decodificação, cálculo da taxa de compactação e análise das chamadas de procedimentos na execução do código

2.1. Manipulação de arquivos

Para poder se executar o algoritmo realizado, é necessário que ele tenha a capacidade de fazer manipulações com arquivos de maneira eficiente. Para isso então será usado quatro funções: a abertura, a leitura, a escrita e o fechamento. Na realização de todas estas funções, faz-se uso das chamadas de sistemas, descritas como *syscalls*, onde o sistema operacional da máquina permite que programa em baixo nível possa realizar as atividades de entrada e saída e, principalmente para este caso, ter acesso a memória para conseguir manipular com arquivos.

A abertura é o começo para o início da manipulação de arquivos. Por meio dela permite-se o acesso a arquivos já existentes na memória do computador, utilizado para abrir o arquivo .txt que está na memória, e a criação de um novo arquivo, utilizado para a impressão da saída do programa e do dicionário. Lembrando-se que na descompressão, a entrada utilizada é a saída do programa de compressão. Para a execução destas operações, define-se o registrador $v0 com o valor inteiro de 13, o registrador $a0 como o endereço com o nome do arquivo a ser aberto, o $a1 para a flag da operação a ser realizada sobre o arquivo (0 para leitura e 1 para escrita) e o $a2 como o modo de operação do arquivo. Depois de realizado a operação, o $v0 retorna o ponteiro que aponta para o primeiro termo do arquivo.

A leitura é feita com o intuito de se coletar os dados no formato de string para poder trabalhar com ele. No caso do código, ele é feito para que possa se analizar os caracteres e à partir deles implementar o dicionário e fazer a compactação do arquivo. Nesta operação, atribui-se o valor de $v0 igual a 14, e os argumentos $a0, $a1 e $a2 recebem respectivamente o ponteiro que aponta para primeiro termo do arquivo (file descriptor), o endereço do buffer que receberá o texto, e a quantidade máxima de caracteres a serem lidos.

A escrita é feita quando, aberto um arquivo com a flag de escrita, escrever dados de tipo string dentro de seu arquivo .txt. Desta forma, depois de realizadas as operações de saída e formação do dicionário, pode-se realizar a criação destes dois arquivos separados. Para a escrita, o $v0 recebe o valor 15 e os argumentos $a0, $a1 e $a2 recebem o file descriptor, o endereço de *buffer* que contém o dado a ser impresso no arquivo, e o número de caracteres a serem impressos. Nesta finalidade, cria-se um arquivo novo, colocando-se na abertura de arquivo o nome de um arquivo

que não existe no diretório, e neste novo arquivo em branco se adiciona os dados.
Por último, tem-se a função de fechamento de arquivo, utilizado depois de se realizar todas as operações necessárias com os arquivos. Diferentemente da linguagem C, não é obrigatório o fechamento do arquivo com a abertura dele, apesar de que é uma boa prática a ser realizada. Deixar o arquivo aberto pode fazer com que o dado presente no arquivo não se mantenha íntegro, podendo afetar futuras execuções do programa.
Dessa maneira, com os conhecimentos de Organização de Computadores e as estratégias de implementação de um codificador LZ78, implementou-se a sua implementação em Assembly e estudou-se seu comportamento, de maneira a analizar seu desempenho e garantir o estudo em baixo nível.

2.2. Codificação Lempel-Ziv

Com a implementação de manipulação de arquivos realizada, precisa-se pegar os caractéres presente nos arquivos data e elaborar a estratégia para a sua compactação, em seguida salvando o resultado do processo em um arquivo .lzw. Como dito na Introdução, o Lempel-Ziv é um algoritmo de compactação com base em dicionário. O algoritmo para a sua criação e para criação da saída pode ser visualizado na figura 5.

```
cadeia := '' # inicializa com a cadeia de tamanho 0
D.insere(cadeia)
enquanto c := leia_novo_caractere()
   se D contém cadeia concatenada com c
      cadeia := cadeia concatenada com c
   senão
      imprime_na_saida([D.código(cadeia), c])
      D.insere(cadeia concatenada com c)
      cadeia = ''
   fim se
 fim enquanto
 imprime_na_saida([D.código(cadeia), ''])
```

Figura 2. Algoritmo da compressão LZ78 em pseudocódigo

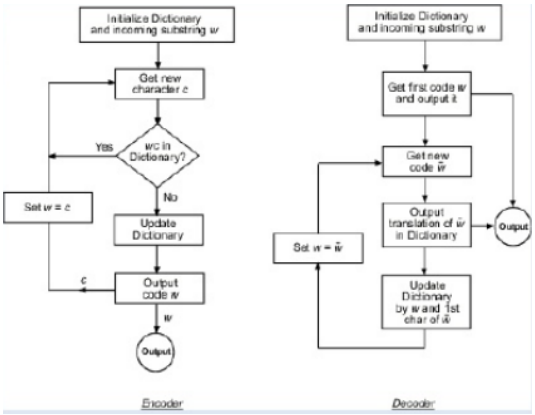

Figura 3. Fluxograma compressão e descompressão Lempel-Ziv

Como pode-se ver no algoritmo tirado do ***Wikipedia***, pode-se ver que começa-se com um dicionário vazio. Quando se pega o próximo caractére do dado de entrada, ele é armazenado em espaço C e se faz a verificação com ele para ver se este caractére somado com o caractére anterior, armazenado em um espaço P, já estam presente no dicionário ou não. Caso já esteja, o espaço P passa a receber o caractere já presente nele junto com o caractere presente em C (P = P + C), caso não, joga na saída o código que representa o caractere ou string presente em P, adiciona a string P+C ao dicionário e agora o espaço P passa a receber o caractere presente em C (P = C).
Tendo tal esquemático realizado, realiza-se a estratégia para a sua implementação em baixo nível. Na entrada, como o arquivo é grande, inviabilizando a sua importação para a sua manipulação na memória RAM, reservou-se um buffer na RAM de 4 bytes e o arquivo era lido de word em word. O dicionário criado é armazenado na pilha, representando o seu fim o $sp. Com este espaço, o próximo caractere é armazenado no registrador $t3, representado o que seria o espaço C falado anteriormente. Para verificar a sua presença ou não no dicionário, chama-se a função *searchDict*, que percorre a pilha e verifica se o caractere está presente ou não. Caso esteja, ele adiciona o caractere no espaço descrito P e passa para o próximo caractere, voltando para a função *readFile*. Caso não esteja, ele é adiocionado ao dicionário através do *pushbackDict*.
Na figura 4 mostra-se um exemplo da montagem do dicionário e da saída utilizando o algoritmo LZ78, retirado do site ***(Youtube)***.

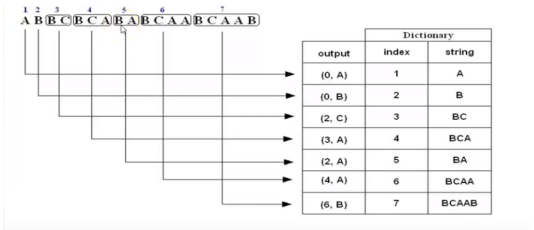

Figura 4. Exemplo de compressão LZ78

Pode-se visualizar neste exemplo o funcionamento do código com um pequeno escopo de dados de entrada. Para os primeiros dois caracteres presentes na entrada 'A' e 'B', eles são impressos na saída e adicioandos ao dicionário, junto com o indíce zero, que não faz referência ao elemento nulo do dicionário. Quando se chega ao 3 caractere de entrada, o segundo B, como ele já está presente no dicionário, ele é apenas mandado para a string P. pegando o próximo caractere 'C', verifica-se a presença de P+C no dicionário. Como ele não está presente, adiciona ele ao dicionário, e imprime na saída o índice de P mais o caractere C. Com

essa lógica, segue-se até se concontrar o fim da string de entrada.

2.3. Decodificação Lempel-Ziv

A decodificação segue um caminho semelhante da codificação. Tendo o arquivo .lzw comprimido pelo primeiro algoritmo, deve-se que achar a estratégia para se obter o arquivo original sem perdas. Pelo algoritmo mostrado abaixo, consegue-se visualizar a lógica implementada para a decodificação.

```
1.No inicio o dicionário contém todas as raizes possiveis;
2.cW <= primeira palavra código na sequência codificada (sempre é uma
raiz);
3.Coloque a string(cW) na sequência de saida;
4.pW <= cW;
5.cW <= próxima palavra código da sequência codificada;
6.A string(cW) existe no dicionário ?
  a.se sim,
    i.coloque a string(cW) na sequência de saida;
    ii.P <= string(pW);
    iii.C <= primeiro caracter da string(cW);
    iv.adicione a string P+C ao dicionário;
  b.se não,
    i.P <= string(pW);
    ii.C <= primeiro caracter da string(pW);
    iii.coloque a string P+C na sequência de saida e adicione-a ao
dicionário;
7.Existem mais palavras código na sequência codificada ?
  a.se sim,
    i.volte ao passo 4;
  b.se não,
  i.FIM.
```

Figura 5. Algoritmo da compressão LZ78 em pseudocódigo

Como pode-se ver, no primeiro passo o primeiro caractere é colocado na saída e adicionado no dicionário. No algoritmo acima, começa-se com um dicionário contendo todas os possíveis caracteres individuais, mas preferiu-se criar um novo dicionário a cada execução do código. No LZ78, todo os componentes são colocados no formato (índice da string P, caractere C). Sendo assim, na saída do código sempre será impresso a string representada pelo índice P mais o caractere C. No dicionário, a palavra adicionada será a mesma da impressa como saída. Na figura 7, mostra-se o exemplo da decodificação do código mostrado como exemplo na seção Codificação Lempel-Ziv.

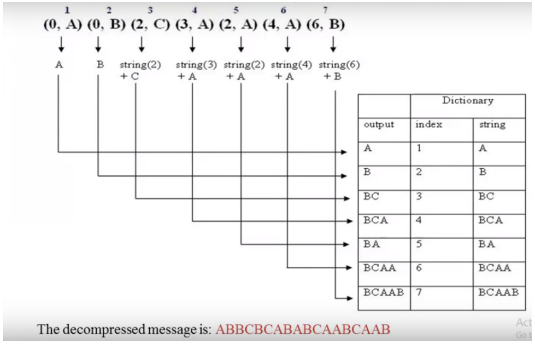

Figura 6. Algoritmo da compressão LZ78 em pseudocódigo

Para o código de comprimido na primeira parte, pode-se visualizar o caminho inverso para se obter o arquivo original. O processo de decodificação mostra-se mais fácil do que o processo de codificação, onde todo elemento P+C é adicionado ao dicionário e impresso na saída.

2.4. Cálculo da taxa de compactação e chamadas de procedimentos

O cálculo da taxa de compactação é feita de maneira simples. Depois de feito a compactação de cada arquivo, fez-se um algoritmo que contava o número de caracteres presentes na entrada do programa e na saída. Em seguida, tendo o número de bytes de cada um deles, fez-se a razão entre o número de caracteres presentes no compactado pelo número de caracteres presentes no arquivo original. Os resultados obtidos estão dispostos na seção de resultados.
Para mostrar as chamadas de procedimentos, utilizou-se a ferramenta *Instruction Statistics*, presente no MARS, e analizou-se a sua atuação no algoritmo de compactação.

3. Resultados

Pegou-se como exemplo um arquivo pequeno para examinar sua atividade. Na entrada do código, utilizou-se um arquivo .txt com o seguinte escrito: ”Rico em nutrientes essenciais para o dia-a-dia!”E analisou-se o seu comportamento para depois analisar para arquivos maiores.

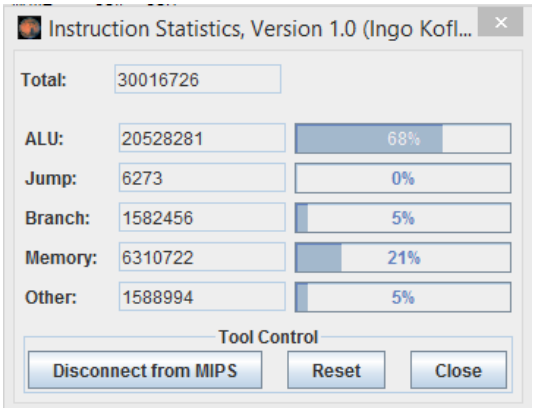

Figura 7. Chamadas de procedimento - data3

Os códigos podem ser visualizados acessando-se o link do repositório do ***Github***

Ao final da execução do programa, o dicionário gerado pode ser visto na parte da memória referente à pilha (*stack*) e o arquivo debuggado com um editor de texto Hex (durante o desevolvimento, foi utilizando o iHex, disponível na App Store do MacOS)

3.1. Discussão

Devido a baixa velocidade de processamento do programa foi impossível comprimir os arquivos data recomendados, apesar disso foi comprovada a funcionalidade do código a partir de alguns arquivos testes. Abaixo estão presentes alguns pontos observados durante os experimentos.

Tabela 1. Tabela comparativa: tam. entrada/tam. saída

Arquivos	Tam. Entrada (kB)	Tam. Saída (kB)
Pequenos c/ alta redundancia	1	0.664
Pequenos c/ baixa redundancia	1	2
Grandes c/ alta redundancia (1)	10	6
Grandes c/ alta redundancia (2)	15	8
Grandes c/ baixa redundancia (1)	10	16
Grandes c/ baixa redundancia (2)	15	22

3.1.1 Tamanho do arquivo de entrada

O programa desenvolvido possui uma limitação de tamanho para arquivos de entrada. A partir de 10kB, o processamento já passa a ser bastante lento, e ficando cada vez mais lento.

Os problemas quanto a tempo de processamento pioram conforme o quão ”não-redundante”ele é e suavizam conforme o quão redundante ele é.

3.2. Conclusões

A partir dos resultados obtidos, não foi possível adquirir os objetivos esperados. As taxas de compactação de cada arquivo mostram os resultados esperados de um código de compactação Lempel-Ziv, sendo pouco eficiente para arquivos pequenos e tendo um melhor desempenho a medida que os arquivos aumentam. Infelizmente, não foi possível se observar o resultado do arquivos cedidos como entrada pelo professor, mas consegue-se ver seu comportamento com pequenos arquivos. Foi possível também se visualizar os problemas descritos na Introdução, como a necessidade de memória para se guardar o dicionário e seu crescimento devagar. O LZ78 tem como vantagem com relação ao LZW a não necessidade de se manter o dicionário para a descompressão.

Observa-se na execução das chamadas de procedimentos, o principal tipo de chamada feita foi relacionado a ALU. Isso se deve devido às várias operações aritméticas exigidas pelo código para manipular os dados na pilha e as suas várias interações. Mas ressalta-se também o mais de um quinto de chamadas de procedimentos da memória, devendo-se ao fato dos inúmeros acessos aos arquivos para leitura dos dados de entrada e escrita dos dados de saída.

O código acaba sendo devagar devido ao fato de que é necessário a realização de várias buscas à memória secundária para a realização das operações. Caso tivesse sido implementado uma forma de levar para a memória RAM todo o conteúdo do arquivo, a sua execução com certeza seria muito mais rápida.

Referências

Slides Organização de computadores
Wikipedia - LZ78
Wikibooks - Compressão
Slides Data Compression Algorithms
Slides Organização e Arquitetura de Computadores - Prof. Flávio Vidal