Algoritmos para Compressão de Microcódigo

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Edson Borin e aprovada pela Banca Examinadora.

Campinas, 4 de abril de 2007.

Prof. Dr. Guido Costa Souza de Araújo
(Orientador)

Tese apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Borin, Edson

B644a Algoritmos para compressão de microcódigo / Edson Borin -- Campinas, [S.P. :s.n.], 2007.

Orientador : Guido Costa Souza de Araújo

Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computadores. 2. Compressão de dados (Computação). I. Araújo, Guido Costa Souza de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Microcode compression algorithms.

Palavras-chave em inglês (Keywords): Computer architecture. 2. Data compression (Computer science)

Área de concentração: Arquitetura de Computadores

Titulação: Doutor em Ciência da Computação

Banca examinadora: Prof. Dr. Edil Severiano Tavares Fernandes (UFRJ)
Dr. Mauricio Breternitz Jr. (INTEL-USA)
Prof. Dr. Paulo Cesar Centoducate (IC-UNICAMP)
Prof. Dr. Rodolfo Jardim de Azevedo (IC-UNICAMP)
Prof. Dr. Luiz Cláudio Villar dos Santos (UFSC)
Prof. Dr. Sandro Rigo (IC-UNICAMP)

Data da defesa: 04/04/2007

Programa de Pós-Graduação: Doutorado em Ciência da Computação

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 04 de abril de 2007, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Edil Severiano Tavares Fernandez
COPPE - UFRJ

Prof. Dr. Mauricio Breternitz Junior
INTEL

Prof. Dr. Paulo Cesar Centoducatte
IC - UNICAMP

Prof. Dr. Rodolfo Jardim de Azevedo
IC - UNICAMP

Prof. Dr. Guido Costa Souza de Araújo
IC - UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Algoritmos para Compressão de Microcódigo

Edson Borin[1]

Abril de 2007

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo (Orientador)
- Prof. Dr. Edil Severiano Tavares Fernandes
  COPPE – UFRJ
- Dr. Mauricio Breternitz Jr.
  Programming Systems Lab / MTL – Intel Corporation (EUA)
- Prof. Dr. Paulo Cesar Centoducatte
  Instituto de Computação – Unicamp
- Prof. Dr. Rodolfo Jardim de Azevedo
  Instituto de Computação – Unicamp
- Prof. Dr. Luiz Cláudio Villar dos Santos (Suplente)
  Departamento de Informática e de Estatística – UFSC
- Prof. Dr. Sandro Rigo (Suplente)
  Instituto de Computação – Unicamp

[1] **Financiado pelo CNPq e pela Fapesp.**

Resumo

Microprogramação é uma técnica comum no projeto de unidades de controle em processadores. Além de facilitar a implementação da unidade de controle, o microcódigo pode ser modificado para adicionar novas funcionalidades ou aplicar correções a projetos já existentes. À medida que novas funcionalidades são adicionadas à CPU, a área e o consumo de energia associados ao microcódigo também aumentam. Em um projeto recente de um processador da Intel, direcionado a baixo consumo de energia e área reduzida, estimou-se que a área e o consumo de energia associados ao microcódigo corresponderiam a 20% do total do *chip*.

Neste trabalho, investigamos a utilização de técnicas de compressão para reduzir o tamanho do microcódigo. A partir das restrições impostas no projeto de processadores de alto desempenho, fizemos uma análise qualitativa das técnicas de compressão de código e microcódigo e mostramos que a compressão de microcódigo em dois níveis é a técnica mais adequada para se comprimir o microcódigo nesses processadores. Na compressão de microcódigo em dois níveis, as microinstruções são substituídas por apontadores para dicionários que armazenam os padrões de *bits* extraídos do microcódigo. Os apontadores são armazenados em uma ROM denominada "vetor de apontadores" e os padrões de *bits* residem em ROMs distintas, denominadas "dicionários". A técnica também permite que as colunas do microcódigo sejam agrupadas em conjuntos de forma a reduzir o número de padrões de *bits* nos dicionários. O agrupamento de colunas similares é fundamental para minimizar o número de padrões de *bits* nos dicionários e, conseqüentemente, maximizar a redução do tamanho do microcódigo.

A principal contribuição desta tese é um conjunto de algoritmos para agrupar as colunas do microcódigo e maximizar a compressão. Resultados experimentais, com microcódigos extraídos de processadores em produção e em estágios avançados de desenvolvimento, mostram que os algoritmos propostos melhoram de 6% a 20% os resultados obtidos com os outros algoritmos encontrados na literatura e comprimem o microcódigo em até 50% do seu tamanho original.

Ainda neste trabalho, identificamos a necessidade de se comprimir o microcódigo com restrições no número de dicionários e na quantidade de colunas por dicionário. Também provamos que, com essas restrições, o agrupamento de colunas do microcódigo é um problema $\mathcal{NP}$-Completo. Por fim, propomos um algoritmo para agrupar colunas sob estas restrições. Os resultados experimentais mostram que o algoritmo proposto é capaz de produzir bons resultados de compressão.

Abstract

Microprogramming is a widely known technique used to implement processor control units. Microcode makes the control unit design process easier, as it can be modified to enhance functionality and to apply patches to an existing design. As more features get added to a CPU core, the area and power costs associated with the microcode increase. In a recent Intel internal design, targeted to low power and small footprint, the area and the power consumption costs associated with the microcode approached 20% of the total die.

In this work, we investigate the use of compression techniques to reduce the microcode size. Based on the constraints imposed by high performance processor design, we analyze the existing microcode and code compression techniques and show that the two level microcode compression technique is the most appropriate to compress the microcode on high performance processor. This technique replaces the original microinstructions by pointers to dictionaries that hold bit patterns extracted from the microcode. The "pointer arrays" and the "dictionaries" are ROMs that store the pointers and the bit patterns, respectively. The technique allows the microcode columns to be grouped into clusters, so that the number of bit patterns inside the dictionaries is reduced. In order to maximize the microcode compression, similar columns must be grouped together.

The main contribution of this thesis is a set of algorithms to group similar microcode columns into clusters, so as to maximize the microcode size reduction. Experimental results, using microcodes from production processors and processors in advanced development stages, show that the proposed algorithms improve from 6% to 20% the compression results found by previous works and compress the microcode to 50% of its original size.

We show the importance of compressing microcode under design constraints such as the number of dictionaries and the number of columns per dictionary. We also prove that, under these constraints, the problem of grouping similar columns is $\mathcal{NP}$-Complete. Finally, we propose an algorithm to group similar columns under such constraints. The experimental results show that the proposed algorithm provides good compression results.

À minha mãe Ivanete e ao meu pai Edson.

Agradecimentos

Esta tese é resultado de um doutorado realizado no Instituto de Computação da Unicamp. Reservo este espaço para agradecer a todos que contribuíram direta, ou indiretamente, para a realização deste trabalho e para a minha formação como pesquisador.

Guido, suas observações, críticas, conselhos e, principalmente, incentivos foram fundamentais para a minha formação como pesquisador. Muito obrigado por ter aceito me orientar e pela sua excelência no papel de orientador.

Pai, mãe, Daniel e Carol, o doutorado foi uma etapa da minha vida cheia de altos e baixos, com dúvidas, incertezas e conquistas. A nossa família é o porto seguro que me tranqüilizou em todos os momentos dessa etapa. Obrigado pelo apoio e pelo carinho de vocês.

Juliana, conhecer você foi o melhor acontecimento do meu doutorado. Você completou a minha vida e a tornou muito mais agradável. Além disso, você foi uma excelente revisora de textos e uma ótima consultora da língua portuguesa. Obrigado pela sua ajuda e pela sua paciência.

Mauricio, as nossas discussões técnicas sobre compressão de microcódigo contribuíram significativamente para a realização desta tese. O seu incentivo e entusiasmo contagiante tornaram os experimentos mais interessantes e fizeram com que eu gostasse cada vez mais da pesquisa. Obrigado por toda a sua ajuda e incentivo.

Youfeng and Jesse, thank you very much for the internship at PSL. I learned a lot and I had a great time at California.

Ducatte, suas críticas, sugestões e revisões contribuíram muito para a minha formação e foram de suma importância para elevar a qualidade do meu trabalho. Obrigado.

Professores Guido, Anido, Côrtes, Ducatte, Edmundo e Pannain, obrigado pelas disciplinas que cursei no Instituto de Computação. O conhecimento que adquiri foi fundamental para a minha formação e contribuiu significativamente para elevar a qualidade deste trabalho.

Cid e Célia, as provas de $\mathcal{NP}$-Completude no apêndice desta tese foram possíveis graças ao aprendizado que tive nas disciplinas de complexidade de algoritmos e de grafos. Vocês são professores exemplares. Obrigado.

Agradeço aos coordenadores da pós graduação do Instituto de Computação. Em especial, agradeço aos professores Pedro Rezende, Neucimar Leite, Flávio Miyazawa e Rodolfo Azevedo

pela ajuda nos momentos que recorri à CPG.

Além da infraestrutura, indispensável para a realização deste trabalho, o Instituto de Computação, seus funcionários e os professores me proporcionaram um ambiente acolhedor e enriquecedor. Em especial, agradeço os professores e alunos integrantes do Laboratório de Sistemas de Computação. A todos vocês: Muito obrigado.

Também aproveito este espaço para agradecer a todos os amigos que tornaram o meu doutorado e a minha vida em Campinas mais agradável. Em especial, agradeço àqueles que estiveram presentes e me apoiaram nas horas mais difíceis. Sandro, Bazinho, Dani, Pará, Zeh, Bartho, Gregório. Obrigado.

Por fim, mas não menos importante, agradeço ao CNPq, à Fapesp e à Intel Corporation pelo suporte financeiro através de bolsas de estudo, reserva técnica e estágio.

Sumário

Resumo **ix**

Abstract **xi**

Agradecimentos **xv**

1 Introdução **1**
1.1 Microprogramação 1
1.2 Restrições de Projeto em Processadores de Alto Desempenho 4
1.3 Contribuições 8
1.4 Organização da Tese 8

2 Trabalhos Relacionados **9**
2.1 Compressão de Dados 9
2.2 Compressão de Código 10
2.2.1 Descompressão em *Software* 11
2.2.2 Descompressão com Auxílio de *Hardware* 12
2.2.3 Outras Abordagens 17
2.3 Compressão de Microcódigo 18
2.3.1 Redução do Número de Microinstruções 18
2.3.2 Redução da Largura do Microcódigo 19
2.3.3 Outras Abordagens 30
2.3.4 Quadro Comparativo 31
2.4 Conclusões 34

3 Compressão de Microcódigo em Dois Níveis **35**
3.1 Esquema de Compressão Básico 35
3.2 Esquema de Compressão com Particionamento 37
3.3 Esquema de Compressão com Agrupamento 39
3.3.1 Colunas Não-Comprimidas 41

3.3.2 Conversão de Endereços 41
3.4 Análise de Desempenho 42
3.5 Redução de *Bits* versus Redução da Área 44
3.6 Conclusões 45

4 Algoritmos para Agrupamento de Colunas **47**
4.1 Terminologia 48
4.2 Agrupamento de Colunas sem Restrições 49
4.2.1 Colunas Seqüenciais 49
4.2.2 Ordenação Linear e Colunas Seqüenciais 53
4.2.3 Ordenação Circular e Colunas Seqüenciais 54
4.2.4 AKL - Agrupamento Baseado em Kernighan e Lin 56
4.2.5 Outras Heurísticas 59
4.3 Agrupamento de Colunas com Restrições 62
4.3.1 AKL com Restrições 62
4.4 Conclusões 64

5 Resultados Experimentais **65**
5.1 Compressão sem Restrições 66
5.1.1 Compressão do Microcódigo versus Número de Colunas 68
5.1.2 Tempo de Execução dos Algoritmos 71
5.1.3 Estabilidade da Razão de Compressão 74
5.2 Compressão com Restrições 80
5.2.1 Compressão do Microcódigo versus Número de Dicionários 80
5.2.2 Regularidade no Projeto do Descompressor 84
5.2.3 Compressão de Microcódigo Padronizando Dicionários 85
5.3 Conclusões 89

6 Conclusões **91**
6.1 Trabalhos Futuros 92
6.1.1 Compressão de Microcódigo em Dois Níveis 93
6.1.2 Agrupamento de Colunas 97
6.1.3 Outros Trabalhos 99

Bibliografia **101**

A Provas de $\mathcal{NP}$-Completude **115**
A.1 Compressão com um único dicionário de tamanho fixo 115
A.2 Compressão com j dicionários de tamanhos fixos 118

Lista de Tabelas

2.1 Avaliação das técnicas de compressão de microcódigo. 32

3.1 Redução de *bits* versus redução da área. 45

4.1 Categorias do problema de agrupamento de colunas. 47

5.1 Descrição dos microcódigos. 65
5.2 Tempo de execução das heurísticas com execução única. 72
5.3 Tempo de execução da heurística de ordenação linear. 73
5.4 Tempo de execução da heurística AKL. 74
5.5 Melhores razões de compressão comum e as respectivas razões de compressão padronizada para a compressão sem restrições dos microcódigos A, B, C e D. . 86
5.6 Configuração dos dicionários para a compressão do microcódigo D sem padronização dos dicionários. 87
5.7 Melhores razões de compressão padronizada e as respectivas razões de compressão comum após a compressão com padronização dos dicionários. 88
5.8 Custo da padronização dos dicionários. 88

Lista de Figuras

1.1 μROM posicionada entre registradores de *pipeline*. 6
1.2 Circuito de decodificação posicionado (a) entre a μROM e o registrador de *pipeline* e (b) após o registrador de *pipeline*. 6
1.3 Circuito de decodificação posicionado em um novo estágio de *pipeline*. 7

2.1 Organização do *Compressed Code RISC Processor* (CCRP). 12
2.2 Exemplo de codificação máxima. (a) Microcódigo original. (b) Microcódigo codificado. 20
2.3 Exemplo de microcódigo horizontal. 21
2.4 Exemplo do modelo de codificação proposto por Schwartz [116]. 22
2.5 Exemplo de inclusão de *codewords* no decodificador do campo. 24
2.6 Exemplos de codificação de microinstruções. (a) Sem codificação. (b) Codificação direta. (c) Codificação indireta. 25
2.7 Geração de colunas a partir do conjunto gerador $S^{\star}$. As colunas b, e, g, h, i, j e k são geradas a partir das colunas a, c, d, f. 26
2.8 Esquema de compressão em dois níveis proposto por Grasselli [53]. 27
2.9 Organização do microcódigo em dois níveis: μROM e *nano*ROM. 28
2.10 Microcódigo em dois níveis com mais de uma *nano*ROM. 29

3.1 Idéia básica da compressão de microcódigo. (a) μROM original. (b) μROM comprimida. 36
3.2 Compressão do microcódigo particionado. (a) μROM original. (b) μROM comprimida. 37
3.3 Método de partição simples. O primeiro conjunto contém as colunas de 1 a 3 e o segundo conjunto, as colunas de 4 a 6. 39
3.4 Método de agrupamento de colunas. O primeiro conjunto contém as colunas 1, 3 e 5 e o segundo conjunto, as colunas 2, 4 e 6. 40
3.5 Compressão baseada no agrupamento de colunas. 40
3.6 Esquema de compressão com parte das colunas não-comprimidas. 41
3.7 Esquema de compressão com *pipeline*. 42

3.8 Organização de uma ROM. 44

4.1 Grafo G (b) construído a partir de conjuntos de colunas (a). Cada vértice v_i representa um conjunto C_i. Os conjuntos C_i com colunas (c_j) em comum possuem arestas entre seus respectivos vértices (v_i) no grafo G. 50
4.2 (a) Conjuntos de colunas seqüenciais (b) representados como intervalos. Cada conjunto C_i é representado por um intervalo I_i. 51
4.3 Razões de compressão para o microcódigo com as colunas reorganizadas. O gráfico mostra a razão de compressão final quando a LC é inicializada com cada uma das colunas do microcódigo. 54
4.4 Movimento da lista sobre o vetor de ordenação para $W = 3$. (a) A lista crescendo ($|LC| < W$). (b) A lista atinge o tamanho máximo ($|LC| = W$). (c) A lista de colunas se deslocando sobre o vetor. (d) Quando a lista chega ao final do vetor, ela continua se deslocando para o início do vetor. 55
4.5 Trocando os vértices v_1 e v_2. 57

5.1 Razões de compressão produzidas pelas técnicas de agrupamento de colunas sem restrições. 67
5.2 Resultado da compressão dos microcódigos em partes. 69
5.3 Melhores resultados após a divisão do microcódigo em partes. 71
5.4 Razões de compressão mínima, média e máxima do microcódigo A em seus diversos estágios de desenvolvimento virtual. 76
5.5 Razões de compressão mínima, média e máxima do microcódigo B em seus diversos estágios de desenvolvimento virtual. 77
5.6 Razões de compressão mínima, média e máxima do microcódigo C em seus diversos estágios de desenvolvimento virtual. 78
5.7 Razões de compressão mínima, média e máxima do microcódigo D em seus diversos estágios de desenvolvimento virtual. 79
5.8 Tamanho em *bits* dos dicionários e dos vetores de apontadores em função do número de dicionários (K). 81
5.9 Razão de compressão versus quantidade de conjuntos (K). 83
5.10 Dicionários implementados com o mesmo bloco de *hardware*. 85

6.1 Descompressor com espaço extra reservado para a adição de novos padrões. . . 94
6.2 Reorganização da μROM para aumentar o número de colunas. (a) μROM original. (b) μROM reorganizada. 98

A.1 Exemplo de uma matriz de *bits* construída a partir de um grafo. 117
A.2 Geração de um grafo G' a partir de G para $j = 4$, $L_1 = k$, $L_2 = 2$, $L_3 = 5$ e $L_4 = 4$. 119

Lista de Algoritmos

4.1 Ordenação Linear 53
4.2 Ordenação Circular 56
4.3 Heurística de Kernighan e Lin 58
4.4 AKL - Agrupamento de colunas baseado em Kernighan e Lin 59
4.5 Particionamento das Instruções VLIW em Campos [67] 60
4.6 Particionamento de Microcódigo [136] 61

Lista de Acrônimos

AMD	Advanced Micro Devices
ARM	Advanced RISC Machines
ASPP	Application Specific Programmable Processor
CAD	Computer-Aided Design
CCRP	Compressed Code RISC Processor
CISC	Complex Instruction Set Computer
CLB	Cache Line Address Lookaside Buffer
CM	Control Memory
CP	Control Part
CPU	Central Processing Unit
EDSAC	Electronic Delay Storage Automatic Calculator
FPGA	Field-Programmable Gate Array
IBC	Instruction-Based Compression
IBM	International Business Machines
IP	Instruction Pointer
ISA	Instruction Set Architecture
LAT	Line Address Table
MIPS	Microprocessor without Interlocked Pipeline Stages
PBC	Pattern-Based Compression
PFM	Pathfinder Memory
PLA	Programmable Logic Array
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
SADC	Semiadaptive Dictionary Compression
SAMC	Semiadaptive Markov Compression
SoC	System-on-a-Chip
TBC	Tree-Based Compression
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration

Capítulo 1

Introdução

1.1 Microprogramação

A unidade central de processamento (CPU) de um microprocessador é dividida em duas partes: a via de dados e a unidade de controle. A via de dados é composta por registradores, unidades funcionais, barramentos internos, dentre outros dispositivos e é encarregada da manipulação dos dados, como a transferência e a modificação de *bits* através de operações nas unidades funcionais. A unidade de controle, por sua vez, coordena as ações tomadas pela via de dados para executar as instruções fornecidas à CPU. Em suma, a unidade de controle interpreta, ou decodifica, a instrução fornecida e envia sinais de controle aos diversos dispositivos da via de dados para que os dados sejam manipulados de acordo com a semântica da instrução fornecida.

O método de implementação, em *hardware*, da unidade de controle depende da complexidade do conjunto de instruções e da via de dados a ser controlada. Por exemplo, em um microprocessador onde o conjunto de instruções possui uma lógica de execução simples, com execução em um único ciclo do relógio, a unidade de controle pode ser implementada diretamente com um circuito combinacional [108]. Por outro lado, se o conjunto de instruções possui instruções com lógica de execução complexa, que tomam mais do que um ciclo do relógio, um circuito seqüencial faz-se necessário.

As unidades de controle baseadas em circuitos seqüenciais são implementadas através de uma máquina de estados onde um bloco lógico toma como entrada o estado atual e o campo de operação da instrução e produz como saída os sinais de controle para a via de dados e o valor do próximo estado. A máquina de estados em questão pode ser projetada através de um diagrama de estados finitos ou através de um microprograma, onde cada microinstrução representa um estado.

Inicialmente, a unidade de controle dos computadores era implementada com lógica aleatória [108]. O processo de implementação se iniciava com a criação do diagrama de estados finitos que eram transformados em um conjunto de equações lógicas e por fim implementado

com lógica aleatória ou com um *programmable logic array* (PLA) [108]. A unidade de controle baseada em lógica aleatória permite que o projetista modele o circuito livremente, o que a torna mais vantajosa em termos de desempenho. Entretanto, se o conjunto de instruções for grande ou complexo, esse processo de implementação torna-se complexo e suscetível a erros.

Em 1951, enquanto estava envolvido com a construção do computador EDSAC 1 (*Electronic Delay Storage Automatic Calculator*) na universidade de Cambridge, Maurice Wilkes percebeu que os sinais gerados pela unidade de controle se assemelhavam à execução de instruções de um programa. A partir dessa analogia, Wilkes criou o conceito de microprogramação [128]. O EDSAC 2 foi o primeiro computador equipado com uma unidade de controle microprogramada [129]. A técnica de microprogramação permite a abstração, através do microcódigo, do projeto da unidade de controle em módulos e procedimentos. A modularidade torna os projetos grandes menos suscetíveis a erros e a programabilidade permite que correções possam ser aplicadas a projetos já existentes [59, 108].

Em 1964, a *International Business Machines* (IBM) introduziu no mercado a linha de computadores IBM System/360 [124]. Essa linha era formada por diversos modelos de computadores que, apesar de possuírem desempenho e preços distintos, implementavam o mesmo conjunto de instruções e eram capazes de executar o mesmo *software*. David Patterson [107] aponta que a compatibilidade de *software* entre os modelos da linha IBM System/360 deu origem à distinção entre a arquitetura de um processador e a sua implementação em *hardware*. A técnica de microprogramação foi fundamental para que os modelos menores da linha IBM System/360 provessem o mesmo conjunto de instruções que os modelos maiores. Tucker [124] afirma que o custo da unidade de controle baseada em lógica aleatória, comum em processadores simples e/ou rápidos, era proporcional ao tamanho do conjunto de instruções e seria economicamente inviável implementar o conjunto de instruções completo nos modelos menores da linha IBM System/360. Por outro lado, na unidade de controle microprogramada, há um custo inicial com o dispositivo de armazenamento e a lógica de seqüenciamento do microcódigo e, a partir daí, o custo da adição de novas instruções, através da microprogramação, é relativamente pequeno. Dessa forma, à medida que o tamanho e a complexidade do conjunto de instruções cresce, a técnica de microprogramação se torna mais atrativa.

Na década de 60, as aplicações eram geralmente escritas em linguagem de montagem e era comum a inserção, ao conjunto de instruções dos processadores, de instruções de alto nível para auxiliar os programadores [108]. Adicionalmente, o acesso à memória de instruções era muito lento e a utilização de instruções de alto nível reduzia o número de acessos à memória, pois estas instruções permitiam a execução de diversas operações a partir de uma única instrução. Esse modelo, também conhecido como *Complex Instruction Set Computer* (CISC), simplificava o processo manual de geração de código e reduzia o número de acessos à memória de instruções. Entretanto, essa prática tornava o conjunto de instruções do processador grande e a lógica de execução de algumas instruções complexas, o que dificultava a implementação da unidade

de controle com lógica aleatória. Assim sendo, as vantagens proporcionadas pela microprogramação, como programabilidade e modularização da unidade de controle, e o sucesso da linha de computadores IBM System/360 consolidaram a técnica de microprogramação e, com exceção dos processadores mais rápidos ou mais simples, essa técnica passou a ser utilizada na grande maioria dos computadores [80, 123, 124] e microprocessadores CISC [6, 96, 119].

A introdução de memórias *cache*, na década de 80, minimizou o problema de desempenho no carregamento das instruções. Esse fato, aliado ao desenvolvimento de compiladores mais eficientes, tornou interessante a utilização de microprocessadores com conjuntos de instruções reduzidos, também conhecidos como *Reduced Instruction Set Computers* (RISC). A unidade de controle desses processadores é simples e, na maioria das vezes, é implementada com lógica aleatória [108], ao invés de microprogramação.

A introdução de novas técnicas para aumentar o desempenho dos processadores, como *pipelining*, execução super-escalar e execução fora de ordem, tornaram a microarquitetura dos processadores ainda mais complexa. A complexidade dessas técnicas aliada à execução multiciclo das instruções de alto nível tornaram o projeto de microprocessadores CISC um desafio a parte. Assim sendo, o modelo RISC se popularizou nas décadas de 80 e 90. De fato, a maioria dos processadores CISC foi descontinuada e grande parte do mercado atual de processadores utiliza processadores baseados em tecnologia RISC. Por outro lado, a quantidade de aplicações existentes para processadores x86 (CISC) foi determinante para a sobrevivência dessa arquitetura. A capacidade das empresas fabricantes de processadores compatíveis com a arquitetura x86 de melhorar o desempenho mantendo a compatibilidade com código legado fez com que os processadores compatíveis com a arquitetura x86 dominassem o mercado de computadores pessoais.

Para introduzir técnicas como execução super-escalar e fora de ordem, sem incrementar demasiadamente a complexidade do projeto do processador, os fabricantes de processadores x86, Intel e AMD, optaram por traduzir dinamicamente as instruções complexas (x86) para instruções simples de forma que o núcleo do processador seja uma arquitetura RISC. Basicamente, para cada instrução carregada da memória, os processadores da linha Pentium [62, 105], fabricados pela Intel, e das linhas K5 [33], K6 [117], K7 [52], K8 [77], pertencentes à AMD, convertem a instrução x86 em uma ou mais instruções simples, chamadas de uOps e ROps nos processadores da Intel e da AMD, respectivamente. Por exemplo, o decodificador de instruções da microarquitetura P6 (Pentium Pro, II e III) é implementado com lógica aleatória e converte uma instrução x86 em até quatro uOps. Quando a instrução x86 exige mais de quatro uOps, uma seqüência de uOps é carregada de uma ROM interna ao processador. As uOps armazenadas nesta ROM formam o microcódigo do processador.

Além de converter as instruções x86 para uOps, o microcódigo da microarquitetura P6 possui rotinas para implementar funcionalidades como tratamento de interrupções e exceções e testes de estruturas internas do microprocessador [29]. Avanços recentes têm migrado diver-

sas funcionalidades para o microcódigo dos processadores. Proteção, virtualização e funções de gerenciamento são exemplos dessas funcionalidades [1, 12]. Como conseqüência da adição de novas funcionalidades, o microcódigo aumenta de tamanho e a μROM[1], utilizada para armazená-lo, ocupa uma área maior. A Intel estima que cerca de 30% dos transistores utilizados no primeiro Pentium eram dedicados à compatibilidade com a arquitetura x86 e boa parte desses eram utilizados na μROM [30].

O tamanho da μROM é particularmente crítico em processadores para aplicações que demandam área reduzida e pequeno consumo de energia, como processadores embarcados ou processadores com múltiplos núcleos (*many-cores*).

Processadores com múltiplos núcleos são desenvolvidos de forma compacta e, muitas vezes, não apresentam aparatos como predição de saltos, *trace caches* e até mesmo execução de instruções fora de ordem. Apesar da ausência ou simplificação dessas estruturas, o microcódigo não é reduzido. Isso acontece porque a compatibilidade com código legado e as funcionalidades mencionadas anteriormente são importantes para o processador. Em um projeto recente de um processador da Intel, direcionado à redução de área e de consumo de energia, estimou-se que a área e o consumo de energia associados ao microcódigo corresponderiam a 20% do total do *chip* [25].

Desde a criação do modelo de controle baseado em microprogramação, diversas abordagens foram propostas para reduzir o tamanho do microcódigo. Técnicas de escalonamento de microoperações e codificação de microinstruções foram propostas para reduzir o número de microinstruções e a largura destas, respectivamente. Outra solução para o problema do tamanho do microcódigo seria utilizar técnicas de compressão de código [11, 27, 130] para reduzir o tamanho do microcódigo. O objetivo é armazenar o microcódigo de maneira comprimida e descomprimí-lo durante a execução. Desta forma, o tamanho da μROM poderia ser reduzido.

A descompressão do microcódigo em tempo de execução exige a inclusão de um *hardware* especializado. Este, por sua vez, afeta o projeto do processador e deve satisfazer restrições de desempenho e tamanho. Antes de analisarmos as técnicas propostas para reduzir o microcódigo, devemos identificar as principais restrições impostas a estas pelo projeto de processadores de alto desempenho.

1.2 Restrições de Projeto em Processadores de Alto Desempenho

Apesar do objetivo ser a redução da área da μROM, o potencial aumento da complexidade do *hardware*, o atraso da propagação do sinal ou mesmo a falta de flexibilidade na modificação do microcódigo podem inviabilizar a utilização de determinadas técnicas de redução do mi-

[1] Utilizaremos o termo μROM para denotar a memória onde o microcódigo reside.

crocódigo. Nesta seção, levantaremos algumas das restrições de projeto e implementação em processadores de alto desempenho.

Flexibilidade

No início do projeto de um microprocessador, o arquiteto principal particiona a área do *chip* em estruturas tais como *cache*, unidade de controle, μROM, unidade de ponto flutuante, etc. Esse particionamento tem como principal objetivo definir a interface e atribuir limites ao tamanho de cada estrutura. Dessa forma, diferentes equipes podem trabalhar em estruturas diversas com um certo grau de independência.

O particionamento do *chip* pode sofrer modificações durante o desenvolvimento do projeto de acordo com as necessidades de cada estrutura. Por exemplo, a equipe encarregada pela estrutura de renomeação de registradores pode requisitar mais área durante a fase de implementação da estrutura. Para atribuir mais área a uma estrutura, o arquiteto deve reduzir a área previamente alocada para outras estruturas ou aumentar o tamanho total do *chip*. Dependendo do aumento da área em questão, essas modificações podem prejudicar ou até mesmo inviabilizar o projeto do processador. Portanto, a previsão e manutenção do tamanho de cada estrutura deve ser a mais precisa possível.

O desenvolvimento do microcódigo ocorre de forma paralela ao desenvolvimento do processador. A quantidade de *bits* e o número de microinstruções são cuidadosamente estimados com base no microcódigo de processadores da mesma família ou semelhantes. Assim sendo, a área a ser ocupada pela μROM pode ser estimada com certo grau de precisão.

As técnicas para reduzir o microcódigo devem ser utilizadas de modo que o tamanho do microcódigo no final do projeto seja previsível. Uma técnica de redução de microcódigo é flexível quando a organização e o tamanho do *hardware* não são alterados de forma significativa por modificações no conteúdo do microcódigo. Assim sendo, a flexibilidade da técnica é um fator fundamental na estimativa da área final do microcódigo.

Atraso da Propagação do Sinal

Processadores de alto desempenho utilizam a técnica de *pipelining* [108] para particionar a via de dados em estágios e aumentar a vazão de instruções. Nesse modelo, a duração do ciclo do relógio é determinado pelo estágio de *pipeline* mais lento, ou seja, com maior latência.

A latência da μROM, que é o tempo gasto para se carregar uma microinstrução da μROM, é geralmente menor do que o ciclo do relógio do processador de forma que a μROM pode ser enquadrada no *pipeline* sem a necessidade de se dividir o ciclo do relógio. A Figura 1.1 mostra a μROM posicionada entre dois registradores de *pipeline*.

Algumas das técnicas apresentadas nesta tese codificam o microcódigo de tal forma que a inclusão de um circuito de decodificação na saída da μROM faz-se necessária. Seja $\Delta_{Dec.}$ o

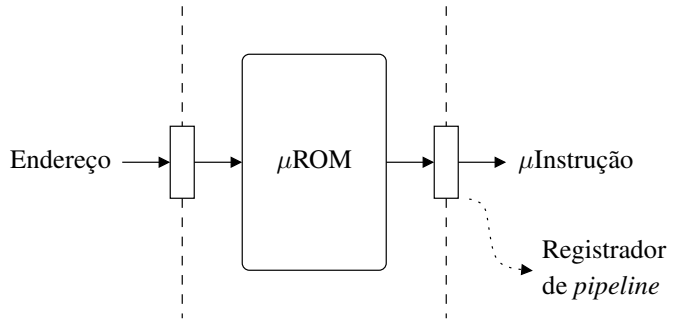


Figura 1.1: μROM posicionada entre registradores de *pipeline*.

atraso proporcionado pelo circuito de decodificação, T o tempo do ciclo do relógio do processador e $\Delta_{\mu ROM}$ a latência da μROM codificada, podemos fazer as seguintes observações:

- Se $\Delta_{\mu ROM} + \Delta_{Dec.} \leq T$, então o circuito de decodificação pode ser adicionado entre a μROM e o registrador de *pipeline*, de modo que a decodificação ocorra no mesmo estágio de *pipeline*. A Figura 1.2 (a) mostra essa organização.
- Se $\Delta_{\mu ROM} + \Delta_{Dec.} > T$, então o circuito de decodificação deve ser posicionado após o registrador de *pipeline* (Figura 1.2 (b)). Nesse caso, o registrador de *pipeline* armazena a microinstrução codificada.

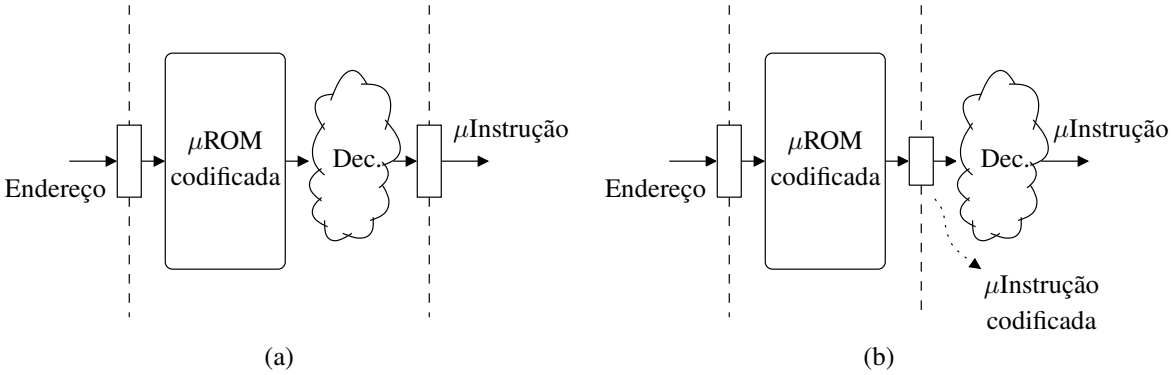


Figura 1.2: Circuito de decodificação posicionado (a) entre a μROM e o registrador de *pipeline* e (b) após o registrador de *pipeline*.

O posicionamento da lógica de decodificação no estágio seguinte do *pipeline* transfere o atraso $\Delta_{Dec.}$ para o próximo estágio. Nesse caso, uma nova análise é realizada para verificar se o atraso proporcionado pelo circuito de decodificação torna o tempo de propagação do sinal no estágio seguinte maior do que o tempo do ciclo do relógio (T). Caso o estágio seguinte comporte a lógica de decodificação sem violar o tempo do ciclo do relógio, a lógica de decodificação é

colocada no início do próximo estágio, como mostrado na Figura 1.2 (b). Por outro lado, se o atraso adicionado pelo circuito de decodificação viola o tempo do ciclo do relógio, um novo estágio de *pipeline* é criado para alojar a lógica de decodificação. A Figura 1.3 mostra a lógica de decodificação posicionada em um novo estágio de *pipeline*.

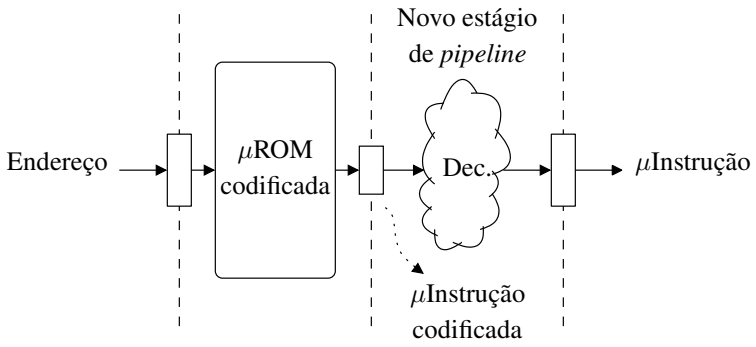


Figura 1.3: Circuito de decodificação posicionado em um novo estágio de *pipeline*.

O novo estágio aumenta em um ciclo o tempo necessário para se preencher o *pipeline* e pode prejudicar o desempenho do processador. Entretanto, processadores modernos, como o Intel Pentium 4 [23, 62] e o AMD Athlon 64 [2], utilizam microcódigo para implementar apenas as instruções e funcionalidades mais complexas. Além disso, as instruções executadas a partir do microcódigo necessitam vários ciclos para serem completadas [2, 66], portanto, um ciclo extra, adicionado pelo novo estágio de *pipeline*, não causaria um grande impacto no desempenho dessas instruções. Adicionalmente, o fato dessas instruções serem pouco executadas pelas aplicações em geral, torna o impacto no desempenho total do sistema ainda menor.

Complexidade do *Hardware*

A utilização de estruturas complexas aumenta o risco de fracasso do projeto do processador. Assim sendo, estruturas complexas só são adotadas quando o benefício proporcionado, seja em área ou desempenho, compensa o risco associado.

Os trabalhos apresentados nesta tese utilizam estruturas de *hardware* que vão desde simples decodificadores combinacionais até máquinas de estados complexas para decodificação em vários ciclos. Classificaremos como complexas as técnicas que utilizam circuitos seqüenciais, incluindo máquinas de estados, ou lógicas aleatórias para auxiliar na decodificação do microcódigo.

1.3 Contribuições

Neste trabalho investigamos a compressão do microcódigo em processadores de alto desempenho. Além de uma boa capacidade de compressão, as restrições de projeto em processadores de alto desempenho exigem que a técnica de compressão do microcódigo tenha características como: flexibilidade na modificação do microcódigo, pouco atraso de propagação do sinal e um circuito de *hardware* simples. A partir dessas restrições fizemos uma análise qualitativa das técnicas existentes de compressão de código e microcódigo e concluímos que a técnica de organização do microcódigo em dois níveis é a mais adequada para se comprimir o microcódigo nesses processadores.

Na organização do microcódigo em dois níveis, as microinstruções são substituídas por apontadores para dicionários que armazenam os padrões de *bits* extraídos do microcódigo. A capacidade de compressão da técnica é sensível ao número de dicionários e à quantidade de colunas e padrões de *bits* em cada dicionário. A técnica também permite que as colunas do microcódigo sejam agrupadas em conjuntos de forma a reduzir o número de padrões de *bits* nos dicionários. O agrupamento de colunas similares é fundamental para minimizar o número de padrões de *bits* nos dicionários e, conseqüentemente, minimizar o tamanho da μROM.

Intuitivamente, o agrupamento de colunas similares é um problema $\mathcal{NP}$-Difícil. De fato, no Apêndice A provamos que, quando o número de colunas por dicionário e a quantidade de dicionários são fixos, o problema é $\mathcal{NP}$-Completo. Esse resultado indica a necessidade de heurísticas para realizarmos o agrupamento das colunas do microcódigo de forma eficiente. Assim sendo, propomos quatro heurísticas para agrupar as colunas do microcódigo sem restrições no número e no tamanho dos dicionários. As heurísticas propostas nesta tese foram comparadas com outras heurísticas encontradas na literatura [67, 136] e os experimentos realizados indicam que as nossas heurísticas provêem resultados superiores.

Ainda neste trabalho, identificamos a necessidade de se comprimir o microcódigo com o número de dicionários e a quantidade de colunas por dicionário fixos. Por fim, propomos o algoritmo AKL, baseado na heurística de Kernighan e Lin [71], para agrupar colunas sob estas restrições. Os resultados experimentais mostram que, mesmo sob restrições, o algoritmo AKL é capaz de produzir bons resultados de compressão.

1.4 Organização da Tese

Esta tese está dividida em seis capítulos. O Capítulo 2 apresenta os trabalhos relacionados à compressão de código e microcódigo e uma análise qualitativa das técnicas para compressão do microcódigo. O Capítulo 3 detalha o esquema adotado para compressão do microcódigo. O Capítulo 4 descreve os algoritmos para agrupar colunas similares do microcódigo e o Capítulo 5 mostra os resultados experimentais. Por fim, o Capítulo 6 apresenta as conclusões e aponta possíveis trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Neste capítulo descrevemos técnicas de compressão de código e microcódigo. A Seção 2.1 faz uma breve introdução à compressão de dados e aponta as características que tornam a compressão de código diferente da compressão de dados. A Seção 2.2 apresenta métodos propostos para comprimir código e os problemas associados à utilização desses métodos para compressão de microcódigo. Por fim, a Seção 2.3 aponta os principais métodos conhecidos para compressão de microcódigo.

2.1 Compressão de Dados

A compressão de dados é uma área vastamente explorada. O avanço dos meios de comunicação de dados, como a Internet, tornaram-na ainda mais importante, tendo em vista que o consumo de banda dos canais de comunicação é função do tamanho dos dados transmitidos.

Em 1949, Shannon e Fano (citado por [89]) desenvolveram um método sistemático para atribuir *codewords* a blocos de informações baseado em probabilidade. Os blocos, ou subtrechos, de informações mais freqüentes eram substituídos por *codewords* menores. Ao final, a informação era representada por um conjunto de *codewords* pequenas, que poderiam, então, ser substituídas novamente pelos blocos associados previamente para recuperar o conteúdo original. Huffman [64] propôs um método ótimo para escolha dessas *codewords*. Na década de 70, Lempel e Ziv [90, 137] propuseram algoritmos que substituíam ocorrências repetidas de blocos de dados por referências compactas à primeira ocorrência. Desde então, muitas técnicas para compressão de diferentes tipos de dados (texto, vídeo, voz e outros) foram propostas.

Programas de computadores pertencem a uma classe particular de dados que merece atenção especial quando se trata de compressão. Muitas vezes não é vantajoso, ou é inviável, descomprimir o programa inteiro antes da sua execução. Isso é geralmente verdade em sistemas embarcados com restrições críticas de memória. Dessa forma, o código deve ser descomprimido por demanda, ou seja, apenas aqueles trechos de código que serão executados devem ser

descomprimidos.

Outra característica importante é o padrão de acesso aleatório às instruções do programa. O fluxo de controle da aplicação, ou seja, a seqüência de acesso às instruções, pode ser aleatório, pois depende de como o fluxo de execução do programa reage aos dados e eventos de entrada do programa. Assim sendo, o algoritmo de descompressão deve ser capaz de descomprimir seqüências de instruções aleatórias de forma eficiente. Essas características inviabilizam ou dificultam a utilização das técnicas de compressão de dados para comprimir código. Na próxima seção, apontamos os trabalhos voltados à compressão de código.

2.2 Compressão de Código

A popularização de SoCs (*System-on-a-Chip*) e o aumento significativo do tamanho do código executado por esses sistemas tornou a compressão de código uma área muito estudada nos últimos anos [13, 21, 31, 101]. A descompressão de trechos do programa sob demanda, e o padrão de acesso aleatório a esses trechos, tornam as técnicas tradicionais de compressão de dados inapropriadas para a compressão de código. Devido a esse fato, diversas técnicas de compressão de código foram propostas nos últimos anos.

Antes de discutirmos compressão de código, é importante lembrar que existem diversas técnicas de otimização voltadas para redução do tamanho do código. *Strength reduction*, remoção de código morto, *tail merging* e remoção de sub-expressões comuns são exemplos de otimizações tradicionais em compiladores [5]. Abstração de procedimentos [35, 37, 44] e abstração de procedimentos parametrizada [135] também permitem reduções significativas no tamanho do código. Beszédes *et alii* [18] definem essas técnicas como "compactação de código" e as distinguem de técnicas de compressão de código por não implicarem na descompressão do código durante a execução.

No caso do microcódigo, como este é desenvolvido manualmente, podemos assumir que o melhor esforço para redução do tamanho por meio de otimizações em *software* já foi realizado. Assim sendo, voltamos nossa atenção aos métodos que descomprimem o código durante a execução.

A idéia básica da compressão de código é armazenar o código de forma comprimida e descomprimí-lo sob demanda durante a execução. Esse processo pode ser feito exclusivamente via *software* ou ter auxílio de um *hardware* especializado. A próxima seção apresenta as técnicas que realizam a descompressão via *software*.

2.2.1 Descompressão em *Software*

Em 1995, Fraser e Proebsting [45] utilizaram um compilador para gerar uma representação compacta do programa e um interpretador para essa representação. A razão de compressão[1] média é de 50%, no entanto, a perda de desempenho[2] pode tornar o tempo de execução 20 vezes maior.

Em outro trabalho, Ernst *et alii* [40] melhoraram a compressão utilizando técnicas que identificam padrões no código, mas a perda de desempenho continua sendo grande. Os autores também utilizaram os padrões encontrados no código para comprimir os programas com razões de compressão que chegam a 20%, mas nesse caso, o programa deve ser descomprimido por inteiro antes da execução.

Kirovski *et alii* [72] propuseram um modelo de compressão baseado em procedimentos. Cada procedimento é comprimido separadamente e a descompressão é realizada sob demanda durante a execução do programa. A ferramenta de ligação (*linker*) substitui toda chamada de procedimento por uma requisição através de um identificador único. Se o procedimento associado ao identificador já estiver na *cache* de procedimentos (*pcache*), o fluxo de controle é redirecionado para o procedimento alvo e a execução segue, caso contrário, o descompressor (em *software*) aloca espaço e descomprime o procedimento na *pcache*. A razão de compressão e a perda de desempenho médias (para *caches* com 64K *bytes*) são 60% e 160%, respectivamente.

Lefurgy *et alii* [83] propuseram um esquema de compressão baseado em dicionários com descompressão por *software*. O programa comprimido é armazenado na memória e a *cache* de instruções armazena as instruções descomprimidas. Sempre que há uma falta (*miss*) na *cache*, o processador gera uma exceção e essa, por sua vez, é tratada por uma rotina que descomprime as instruções da memória e as coloca na *cache*. A técnica requer pequenas modificações no processador para permitir o gerenciamento da *cache* via *software*. Os resultados mostram razões de compressão entre 65% e 83% e perda de desempenho de até 5 vezes para *caches* de 1K *byte* e de até 80% para *caches* de 64K *bytes*. Embora empregue modificações no *hardware* para auxiliar na descompressão, a técnica ainda realiza a maior parte da descompressão via *software*.

Clausen *et alii* [34] estenderam a máquina virtual Java para interpretar *bytecode* comprimido. Padrões de código são transformados em *macros* e, durante a execução, a máquina virtual Java utiliza informações agregadas ao programa para interpretar as *macros*.

Diversas técnicas de compressão de código com descompressão baseada em *software* foram propostas [34, 40, 45, 72, 83]. Entretanto, a degradação de desempenho causada pela descompressão torna essa abordagem inviável para compressão do microcódigo. A próxima seção descreve as técnicas que utilizam auxílio de *hardware* para reduzir a perda de desempenho

[1] Razão de compressão é a razão entre o tamanho do programa comprimido e o tamanho do programa original. Portanto, quanto menor o valor, melhor é a compressão.

[2] A perda de desempenho proporcionada pela técnica de compressão é definida como sendo a razão entre o tempo de execução da aplicação sem compressão e com compressão.

durante a descompressão do código.

2.2.2 Descompressão com Auxílio de *Hardware*

Em 1992, Wolfe e Chanin [130] propuseram o *Compressed Code RISC Processor* (CCRP). Nesse esquema, o código é armazenado de forma comprimida na memória principal e o descompressor é colocado entre a memória principal e a *cache* de instruções. A descompressão de instruções é realizada sempre que há uma falta na *cache*. As instruções no código comprimido possuem endereços diferentes das instruções no código original, por esse motivo, o CCRP utiliza uma tabela de tradução de endereços (LAT-*"Line Address Table"*) para converter os endereços das linhas de *cache* do programa original para o programa comprimido. O CCRP ainda utiliza a *Cache Line Address Lookaside Buffer* (CLB) para armazenar as entradas mais freqüentes da tabela de tradução de endereços. A Figura 2.1 apresenta a organização do CCRP.

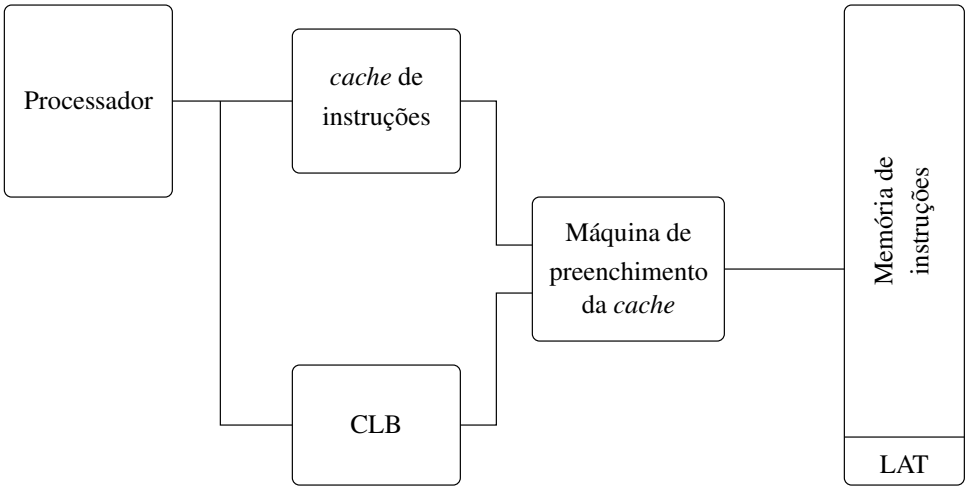


Figura 2.1: Organização do *Compressed Code RISC Processor* (CCRP).

O CCRP é baseado em codificação de Huffman e alcançou razões de compressão em torno de 70% em programas com instruções do MIPS R2000. Nos trabalhos seguintes, Wolfe *et alii* sugerem diversas melhorias na abordagem básica do CCRP. Kozuch e Wolfe [76] investigaram a entropia do código comprimido e o desempenho do CCRP e Benĕs *et alii* [15, 16] discutiram detalhes de implementação e propuseram um modelo de *hardware* do descompressor baseado em Huffman capaz de descomprimir até 32 *bits* a cada $25ns$.

Breternitz e Smith [27] melhoram o CCRP atualizando o código com o endereço das instruções comprimidas, ou seja, os endereços alvos das instruções de desvio foram substituídos pelo endereço das instruções comprimidas. Essa modificação elimina a necessidade de uma tabela de tradução de endereços, pois sempre que um salto causa uma falha na *cache*, o endereço do

alvo corresponde ao endereço da instrução comprimida, o que simplifica o projeto do descompressor. Infelizmente, faltas na *cache* geradas por instruções que não são alvos de desvios não têm o endereço previamente traduzido. Isso acontece porque o endereço dessas instruções é gerado pela lógica de seqüenciamento de instruções do processador, geralmente representada pelo *instruction pointer* (IP), e não pela execução de uma instrução de desvio de fluxo. Os autores propõem modificações no código e no *hardware* de descompressão para tratar esses casos, mas não fica claro qual o impacto da modificação em *software* na razão de compressão e quão complexa é a modificação do *hardware*.

Araújo *et alii* [11] observaram que a entropia de alguns campos das instruções é menor do que de outros, ou seja, o programa é mais regular em determinados campos da instrução. Os autores investigaram as árvores de expressões no código binário e observaram que grande parte dessas árvores diferiam apenas nos operandos. Assim sendo, os autores propuseram a técnica *Pattern-Based Compression* (PBC), que fatorava as expressões em padrões de árvores e padrões de operandos para reduzir a quantidade de árvores distintas no código. Os autores também estudaram diferentes métodos para representar os padrões no código comprimido e obtiveram razões de compressão média de 43% para codificação baseada em Huffman e 48% para codificação de tamanho variável.

Lekatsas e Wolfe [87] também utilizaram a informação de entropia para comprimir os campos das instruções separadamente. Os autores apresentam dois modelos de compressão: o *Semiadaptive Markov Compression* (SAMC) e o *Semiadaptive Dictionary Compression* (SADC). A estrutura geral do descompressor, em ambos os casos, é a mesma do CCRP, incluindo a tabela de tradução de endereços. A diferença está no método de compressão e descompressão: o SAMC é baseado em codificação aritmética, enquanto o SADC é baseado em dicionários. As razões de compressão apresentadas indicam que o SAMC é melhor do que a codificação de Huffman [130] e ficam em torno de 50% e 65% para instruções do MIPS e do i386, respectivamente. Nesse trabalho, os autores não mediram o impacto do descompressor no desempenho do processador.

Lefurgy [81] utiliza um dicionário para armazenar seqüências de instruções repetidas no programa. As seqüências de instruções presentes no código original são substituídas por *codewords*, que representam índices para as seqüências no dicionário. Experimentos com programas para arquiteturas PowerPC, ARM e i386 apresentaram razões de compressão entre 60% e 70%. O impacto do descompressor no desempenho do processador não foi medido.

A tecnologia de compressão *CodePack* [49, 83] foi desenvolvida pela IBM e implementada em alguns processadores da linha PowerPC. As instruções do código original (32 *bits*) são divididas ao meio e dois dicionários são utilizados para armazenar os padrões encontrados em cada metade (16 *bits*). O código comprimido consiste em pares de índices para os dois dicionários e os endereços são convertidos através de uma tabela de compressão de índices. A descompressão acontece em blocos de 16 instruções para preencher uma linha da *cache* de instruções. A razão

de compressão final varia de 60% a 65% e o desempenho de -10% a 10%.

Em outro trabalho, Araújo *et alii* [10] compararam os métodos *Tree-Based Compression* (TBC), *Pattern-Based Compression* (PBC) e *Instruction-Based Compression* (IBC). Os resultados mostram que o método TBC possui a melhor capacidade de compressão do código, atingindo razão de compressão média de 27,2%, contra 39,8% e 31,5% das técnicas PBC e IBC. Entretanto, o *hardware* do descompressor da TBC é mais complexo do que o da IBC e quando o tamanho do descompressor é incluído no cálculo da razão de compressão, a técnica IBC se mostra mais vantajosa, atingindo razão de compressão média de 53,6%, contra 60,7% e 61,3% das técnicas TBC e PBC.

Lekatsas *et alii* [85, 86, 88] observaram que a compressão de código também trazia benefícios no consumo de energia. Eles reposicionaram o descompressor entre o processador e a memória *cache* de forma a minimizar o tráfego de informações nos barramentos que conectam a memória principal, a *cache* e o processador. O desempenho do descompressor e do sistema como um todo não foi apresentado.

Netto *et alii* [20, 102] estenderam a técnica IBC [10] considerando informação de contagem dinâmica de instruções, baseada em *profiling*, para associar as *codewords* menores às instruções mais executadas. O descompressor foi posicionado entre o processador e a memória *cache* do processador SPARC V8 LEON [48] e implementado em uma FPGA. As razões de compressão variaram de 72% a 88% (incluindo o tamanho do descompressor). Os autores reportaram ganhos de até 45% no desempenho e de até 35% na economia de energia.

Nam *et alii* [99] utilizaram a fatoração de operandos [11] para comprimir código VLIW. Os autores dividiram o fluxo de instruções VLIW em *opcodes* e operandos que são mapeados em dois dicionários. O código comprimido contém instruções originais (descomprimidas) e *codewords*. Essa abordagem atingiu razões de compressão de 63% a 71% para arquiteturas VLIW com 4 a 12 operações por instrução. O *hardware* de descompressão é simples, no entanto, a tradução de endereços não é explicada.

Xie *et alii* [132] propuseram uma técnica de compressão de código VLIW baseada em codificação aritmética capaz de comprimir formatos flexíveis de instruções. A técnica permitiu razões de compressão na faixa de 70% a 80%. Em outro artigo, Xie *et alii* [133] utilizaram codificação variável-para-fixa, baseada em Markov, comprimindo código IA-64 e TMS320C6X para 56% e 70% do tamanho original, respectivamente.

Apesar da semelhança com código VLIW, por apresentar instruções longas com diversos campos e operações, o microcódigo possui restrições de projeto que limitam a área e o desempenho do descompressor. Além disso, a natureza de codificação manual torna o microcódigo mais compacto e difícil de se comprimir com as técnicas de compressão de código apresentadas.

Área do Descompressor

Muitos dos trabalhos de compressão de código definem a razão de compressão como sendo a razão entre os tamanhos do programa comprimido e do original, mas omitem do cálculo a área utilizada pelo *hardware* do descompressor de código. Essa omissão pode trazer a falsa sensação de que uma técnica é melhor do que outra. Os resultados apresentados por Araújo *et alii* [10], por exemplo, apontam razões de compressão de 60,7%/53,6% para os algoritmos TBC/IBC quando considerado o tamanho do *hardware* de descompressão e razões de 27,2%/31,5% quando essa variável não é considerada. Observe que se não considerarmos a área ocupada pelo descompressor, é possível concluir equivocadamente que a técnica TBC é melhor do que a técnica IBC. Com o propósito de medir a compressão de microcódigo, utilizaremos a seguinte definição para razão de compressão:

$$Razão\ de\ compressão = \frac{tamanho\ comprimido + tamanho\ do\ descompressor}{tamanho\ original} \quad (2.1)$$

A Equação 2.1 tem duas componentes que variam de acordo com a técnica de compressão: o tamanho do programa comprimido e o tamanho do descompressor[3]. O tamanho do programa comprimido depende exclusivamente da capacidade de compressão da técnica, enquanto que o tamanho do descompressor é associado à complexidade do *hardware* de descompressão.

O fator que determina a importância de cada componente é o tamanho do programa original. Em um cenário onde o tamanho do programa original é muito grande quando comparado ao *hardware* de descompressão, a capacidade de compressão da técnica torna-se mais importante do que o tamanho do descompressor no cálculo da razão de compressão. Assim sendo, técnicas com maior poder de compressão, mesmo sob a penalidade de um descompressor maior, passam a ser mais vantajosas.

O cenário descrito é muito comum na compressão de código, dado que, geralmente, o processador é utilizado para executar programas grandes e complexos, ou mesmo um conjunto de aplicações. Por outro lado, o microcódigo de um processador costuma ter um tamanho limitado e a área do descompressor é mais importante nesse caso.

Desempenho do Descompressor

Outro problema da compressão de código é o tempo gasto para descomprimí-lo, ou seja, a latência do descompressor.

Em muitos trabalhos o descompressor é posicionado entre a memória principal e a *cache* de instruções. O objetivo dessa organização é minimizar o impacto da latência do descompressor no desempenho do sistema, visto que o descompressor só é executado quando ocorre falta

[3]O tamanho do descompressor é a quantidade de memória, em *bits*, que seria acrescentada ao *chip* se o *hardware* do descompressor fosse substituído por um circuito de memória.

de instruções na *cache*. Um fato interessante é que o código comprimido permite que mais instruções sejam trazidas da memória com uma quantidade menor de acessos à mesma, e alguns resultados mostram ganho de desempenho e até mesmo economia de energia quando se utiliza compressão de código [17, 21, 84, 102].

Lefurgy *et alii* [82] estudaram o impacto que o tamanho da *cache* de instruções, a largura de banda e a latência da memória principal têm sobre o desempenho de um processador super-escalar equipado com um esquema de compressão de código baseado no *CodePack*. A partir dos resultados [82], podemos fazer as seguintes observações:

- Largura de banda da memória: quanto menor a largura de banda, mais acessos à memória são necessários para preencher uma linha de *cache*. Um processador equipado com compressão de código tira vantagem dessa situação, pois o número de acessos à memória para preencher a *cache* é geralmente reduzido.

- Latência da memória: a latência da memória ajuda a ocultar a latência do descompressor. O descompressor, conectado à memória, funciona como um *pipeline*. Em outras palavras, enquanto a memória lê um determinado dado, o descompressor processa o dado lido anteriormente da memória. O descompressor adiciona um estágio ao *pipeline* e, conseqüentemente, o tempo necessário para preenchê-lo. Entretanto, a partir do momento em que o *pipeline* está cheio, sua vazão é determinada pela latência do elemento mais lento e, nesse caso, uma memória lenta oculta a latência do descompressor.

- Tamanho da *cache* de instruções: quanto menor a *cache*, maior o número de acessos à memória e ao descompressor. Se o descompressor tira proveito da largura de banda da memória e melhora o desempenho do sistema, uma *cache* menor faz com que essa melhora seja mais significativa, pois o descompressor é acessado mais vezes. Esse mesmo efeito ocorre quando o descompressor prejudica o desempenho do processador devido a latência da memória, nesse caso, a piora do desempenho se torna maior.

As observações sobre o desempenho do descompressor são feitas com base em um sistema com o descompressor posicionado entre a memória principal e a memória *cache*. É válido lembrar que, na ausência da *cache* de instruções, o descompressor é acessado sempre que uma instrução é carregada da memória e, nesse caso, seu desempenho tem maior impacto no sistema.

Note que o número de combinações para se posicionar o descompressor e a quantidade de organizações e hierarquias de memória pode ser muito grande. Assim sendo, restringimos nossa análise apenas àqueles casos que se enquadram no cenário do microcódigo, ou seja, arquiteturas sem *cache* ou com uma única *cache* entre o microcódigo e a via de dados (que é o caso da fila do escalonador, presente em processadores super-escalares com execução de microinstruções fora de ordem).

A memória que armazena o microcódigo (μROM) é implementada dentro do *chip* do processador, portanto, possui uma latência muito pequena (normalmente de um ciclo) e a largura de banda pode ser ajustada de acordo com as necessidades do projeto do *chip*. Desse modo, o descompressor não pode tirar vantagem da largura de banda da memória e sua latência deve ser tão pequena quanto ou menor do que a latência da μROM. Essas duas características tornam o projeto do descompressor de microcódigo um desafio.

Qualidade do Microcódigo

O desempenho do microcódigo é de extrema importância e por isso ele é geralmente codificado de forma manual. O melhor esforço para remover qualquer tipo de redundância em software é realizado e otimizações clássicas de compiladores como *strength reduction*, remoção de código morto, *tail merging*, remoção de sub-expressões comuns [37] e até mesmo técnicas mais elaboradas como abstração de procedimentos [44, 92], são aplicadas ao microcódigo. Técnicas de compressão que tiram proveito de seqüências repetidas no código têm menores chances de comprimir com eficiência o microcódigo. Outra característica que reduz a chance de seqüências repetidas é o formato longo das microinstruções com diversos campos.

Os fatores descritos acima apontam que, independente do algoritmo de compressão, o descompressor deve ter área e latência pequenas. Aliado a essas restrições, a natureza de codificação manual torna o microcódigo mais compacto e difícil de se comprimir, tornando as técnicas de compressão de código, apresentadas nesta seção, inadequadas para compressão do microcódigo.

2.2.3 Outras Abordagens

Em 1999, Liao *et alii* [91] propuseram a utilização da instrução *CALD* (*call-dictionary*) para comprimir código. Essa instrução possui como parâmetro um deslocamento e a quantidade de instruções a serem executadas. Quando uma instrução *CALD* é encontrada, o processador utiliza esses parâmetros para executar uma seqüência de instruções armazenada em um dicionário. Fraser [43] propôs um esquema semelhante ao de Liao [91] para comprimir *bytecode* Java. Da mesma forma que a instrução *CALD*, a instrução *echo* possui como parâmetro um deslocamento e a quantidade de instruções a serem executadas, no entanto, ao invés de referenciar um dicionário, a instrução *echo* desvia o fluxo de controle para uma seqüência de instruções do próprio código. As instruções de *echo* reduziram o *bytecode* Java em até 30% (razão de compressão de 70%). Em outro trabalho, Wu *et alii* [131] utilizaram a instrução *echo* para comprimir código i386. Os resultados mostram razões de compressão que variam de 80% a 83% para programas dos *benchmarks* EEMBC [39] e SPEC CPU2000 [61].

Como mencionamos acima, técnicas de compressão que tiram proveito de seqüências repetidas no código têm menores chances de comprimir com eficiência o microcódigo. Portanto,

essas técnicas não são adequadas para compressão do microcódigo.

Outra abordagem para a redução do tamanho do programa é a recodificação do conjunto de instruções do processador. Bunda *et alii* [28] propuseram o D16, um modelo compacto do conjunto de instruções de 32 *bits* do processador DLXe [60]. O formato reduzido das instruções do D16, com apenas 16 *bits*, limitou o tamanho dos imediatos, a quantidade de operandos e o número de registradores acessados pelas instruções. Os autores ainda mostraram que essas limitações exigem mais instruções do D16 para expressar o mesmo poder de computação do DLXe. Dessa forma, mesmo com instruções com metade do tamanho, o código gerado para D16 é cerca de 65% do tamanho do código para DLXe.

A tecnologia *Thumb* [118, 125] é outro exemplo desse tipo de abordagem. O conjunto de instruções do *Thumb* é uma versão compacta (16 *bits*) das instruções do processador ARM [69] (32 *bits*). Para se reduzir o tamanho do código, algumas instruções e modos de endereçamento foram removidos e o acesso a alguns recursos, como registradores, também foi limitado. Apesar das instruções do *Thumb* possuírem a metade do tamanho das instruções do ARM, a razão de compressão do código não chega a 50%. Da mesma forma que o D16 [28], o código gerado para o *Thumb* necessita mais instruções do que o código para ARM. A razão de compressão varia de 60% a 70% e o desempenho é degradado em cerca de 20%. Assim como o *Thumb*, o MIPS16 [73] é um modo de execução de instruções compactas para o processador MIPS-III.

A recodificação do conjunto de instruções reduz o desempenho do microprocessador e requer modificações grandes no *hardware*. Portanto, não é adequada para a redução do tamanho do microcódigo. Na próxima seção, descrevemos as técnicas para compressão de microcódigo.

2.3 Compressão de Microcódigo

Desde a criação do modelo de controle baseado em microprogramação em 1951 [128], diversas abordagens foram propostas para reduzir o tamanho do microcódigo.

O microcódigo pode ser visto como uma matriz de $L \times N$ *bits*, onde L é a largura, ou o número de *bits* por microinstrução, e N é a altura, ou o número de microinstruções no microcódigo. As próximas seções descrevem técnicas de otimização utilizadas para reduzir a altura e a largura do microcódigo.

2.3.1 Redução do Número de Microinstruções

O microcódigo pode ser caracterizado como "horizontal" ou "vertical". No microcódigo vertical, as microinstruções geralmente executam uma única micro-operação[4] e se assemelham às instruções de máquinas comuns, com um operador e dois ou mais operandos. Por outro lado, no microcódigo horizontal, as microinstruções executam diversas micro-operações em paralelo [4].

[4]Micro-operações são operações executadas pelas microinstruções.

A execução em paralelo das micro-operações tornaram as máquinas horizontais[5] mais rápidas e a utilização desse modelo de microprogramação se difundiu na década de 70 [79]. Apesar do alto desempenho, a complexidade envolvida na programação era grande e a qualidade do microcódigo dependia muito da experiência do programador, por isso, diversas técnicas para compilação de microcódigo horizontal surgiram no final da década de 70 e início da década de 80 [42, 68, 78, 93, 120].

Essas técnicas, conhecidas como compactação de microcódigo, consistiam em agrupar micro-operações distintas em uma mesma microinstrução. Para serem agrupadas em uma mesma microinstrução, duas micro-operações não podem ter dependência de dados entre si nem competir por recursos. Além de reduzir o tempo de execução, a compactação também diminui o tamanho do microcódigo.

Inicialmente, o agrupamento de micro-operações era feito somente dentro de blocos básicos [79] e, apesar do problema ser $\mathcal{NP}$-Difícil [38, 113], as heurísticas propostas geralmente alcançavam bons resultados. Após o escalonamento dentro dos blocos básicos, os programadores utilizavam o "método do menu" [42] para mover, de forma manual, as micro-operações entre os blocos básicos.

Fisher [42] foi o primeiro a propor um método sistemático para compactação de microcódigo de forma global, ou seja, que permitia a movimentação de micro-operações entre blocos básicos sem o auxílio do programador. Após Fisher [42], outros algoritmos foram propostos [68, 78, 93, 120] e essas técnicas evoluíram para o que hoje conhecemos como algoritmos para geração de código para arquiteturas VLIW.

A compactação do microcódigo é utilizada para reduzir o tamanho do microprograma durante a fase de compilação. Depois de compilado, o microprograma pode ser reduzido ainda mais através das técnicas de redução da largura do microcódigo.

2.3.2 Redução da Largura do Microcódigo

Codificação Máxima

A forma mais eficiente de se reduzir a largura do microcódigo é através da codificação máxima [36]. Nessa técnica, cada microinstrução é substituída por uma *codeword*, e uma lógica de decodificação é adicionada à saída da μROM para transformar a *codeword* na microinstrução original.

Seja M o número de microinstruções únicas no microcódigo, a quantidade de *bits* necessários para codificar cada *codeword* é $\lceil log_2 M \rceil$. A Figura 2.2 (a) mostra um microcódigo com $L = 4$, $N = 6$ e $M = 4$. A codificação máxima para o mesmo microcódigo é exibida na Figura 2.2 (b).

[5] Máquinas horizontais são arquiteturas que possuem microcódigo horizontal.

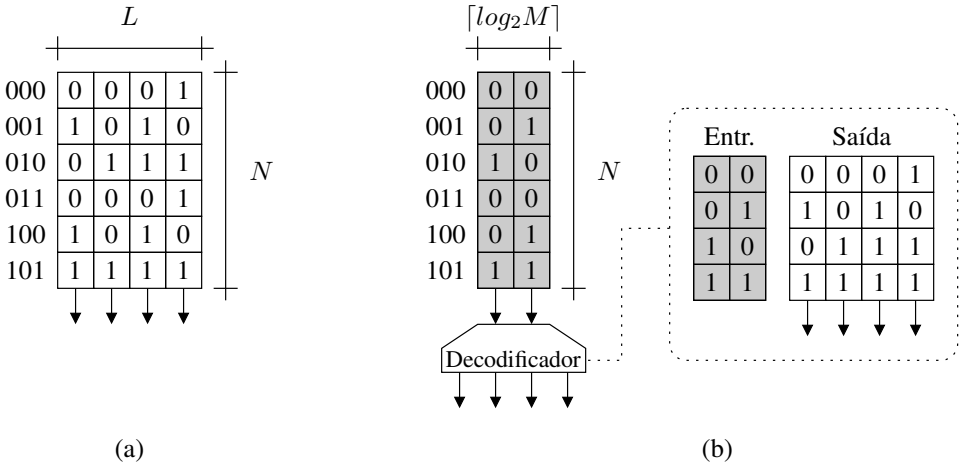


Figura 2.2: Exemplo de codificação máxima. (a) Microcódigo original. (b) Microcódigo codificado.

Apesar de proporcionar a redução do número de colunas, a lógica de decodificação adicionada à saída da μROM pode ocasionar diversos problemas. Dentre eles, os mais importantes são:

- Aumento da área: se o tamanho do decodificador for maior do que a área economizada com a redução da largura do microcódigo, então o resultado será o aumento da área total ocupada pelo microcódigo.
- Aumento da complexidade do *hardware*: a lógica para decodificar as *codewords* pode ser extremamente complexa. Apesar dos avanços nas ferramentas CAD para o desenvolvimento de *hardware*, a complexidade do projeto ainda é um fator importante para a implementação em *hardware*.
- Atraso da propagação do sinal: a redução da largura da memória não reduz significativamente o tempo de leitura da ROM, em contrapartida, o decodificador adicionado pode aumentar demasiadamente o tempo de propagação do sinal. O atraso da propagação do sinal pode causar problemas de temporização no projeto do microprocessador.
- Redução da flexibilidade: A alteração ou inclusão de microinstruções no microcódigo pode gerar combinações de *bits* para os quais não existam *codewords* associadas. Nesse caso, o decodificador deve ser modificado para incluir novas *codewords*. A modificação do decodificador nem sempre é trivial e, muitas vezes, pode ocasionar mudanças grandes na área do circuito e no tempo de propagação do sinal.

O aumento da área, a complexidade do decodificador e o atraso do sinal podem ser previstos no início do projeto, mas a falta de flexibilidade da técnica de codificação pode fazer com

que esses ítens sofram modificações significativas durante o desenvolvimento do projeto. Dependendo da intensidade dos problemas mencionados acima, a técnica de codificação máxima torna-se inviável.

Codificação Mínima

Com a popularização da microprogramação horizontal os pesquisadores perceberam que muitos campos do microcódigo não eram utilizados pela maioria das microinstruções. O microcódigo da Figura 2.3 é um exemplo desse tipo de problema. Nesse microcódigo as linhas representam as microinstruções e as colunas os campos do microcódigo. Os campos com valor '1' denotam as micro-operações realizadas pela microinstrução. Observe que a maioria das microinstruções utilizam apenas um ou dois campos do microcódigo.

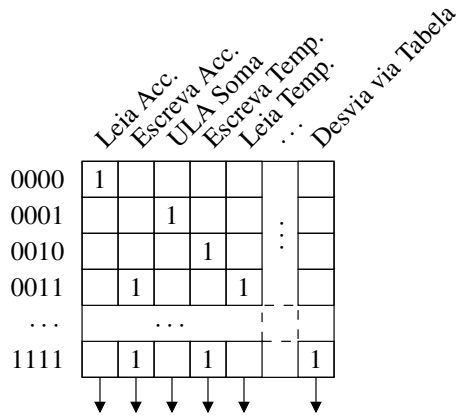


Figura 2.3: Exemplo de microcódigo horizontal.

A baixa utilização dos campos das microinstruções acontecia por diversos fatores:

- O compartilhamento de recursos, como barramentos, limitava a execução em paralelo de algumas micro-operações em campos distintos da microinstrução.
- O modelo de programação, em paralelo, era complexo e os programadores muitas vezes não conseguiam utilizar todo o paralelismo oferecido pela arquitetura. A Seção 2.3.1 mostra que só na década de 70 e 80 surgiram técnicas eficientes para compilação de microcódigo.
- A limitação do paralelismo no microprograma, causada por dependências de dados e de controle, inibia a utilização dos recursos de forma paralela.

Em 1968, Schwartz [116] propôs um método para reduzir a largura do microcódigo que agrupava micro-operações mutuamente exclusivas em campos. O valor do campo era decodificado por uma lógica de decodificação na saída da μROM. A Figura 2.4 mostra um exemplo

onde os campos “ULA Soma”, “Escreva Temp.” e “Leia Temp.”, da Figura 2.3, são codificados em um campo com dois *bits*. O decodificador adicionado à saída da memória converte os *bits* do campo para os valores originais antes da codificação.

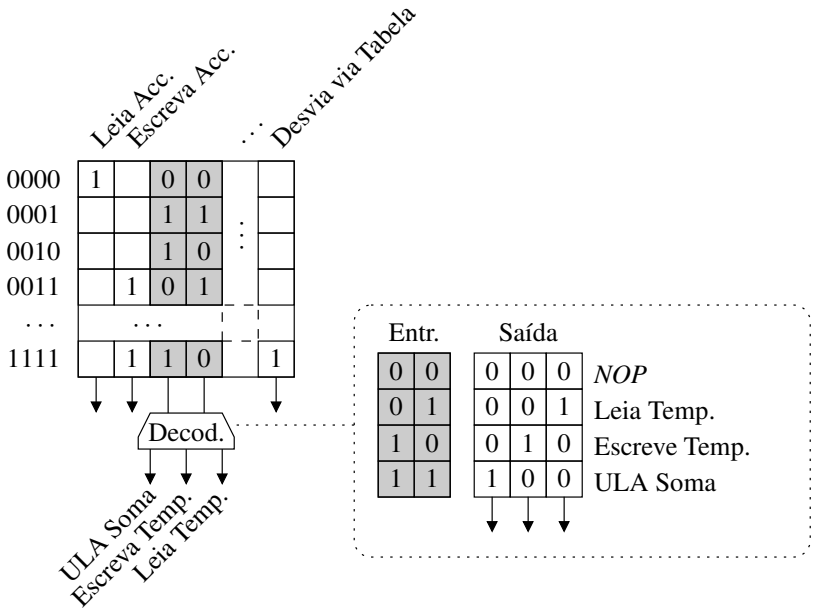


Figura 2.4: Exemplo do modelo de codificação proposto por Schwartz [116].

O decodificador apresentado na Figura 2.4 converte o valor de entrada em uma palavra onde no máximo um dos *bits* tem o valor ‘1’, ou seja, no máximo uma das micro-operações é habilitada. Isso acontece porque somente micro-operações mutuamente exclusivas foram agrupadas nesse campo. Essa característica torna o circuito do decodificador simples [127], fazendo com que o aumento da área e o atraso da propagação do sinal ocasionados pelo decodificador sejam pequenos.

O circuito decodificador da Figura 2.4 possui uma *codeword* que gera a saída *NOP* (*no operation*), ou seja, a saída onde nenhuma micro-operação do campo é ativada. No caso da Figura 2.4, a saída *NOP* é necessária porque a microinstrução no endereço 0000 não executa nenhuma das micro-operações do campo. Entretanto, mesmo que tal microinstrução não exista, a saída *NOP* é sempre incluída no decodificador de forma que, se o microcódigo for modificado e essa combinação for utilizada, o campo e o respectivo decodificador não serão modificados.

Se o projetista optar por agrupar apenas micro-operações que sejam mutuamente exclusivas devido ao compartilhamento de recursos, então possíveis modificações no microcódigo nunca exigirão que duas ou mais micro-operações, pertencentes a um mesmo campo, sejam executadas pela mesma microinstrução. Por exemplo, vamos supor que as micro-operações “Leia Temp.”, “Escreva Temp.” e “ULA Soma”, agrupadas no mesmo campo na Figura 2.4, sejam mutuamente exclusivas por causa de compartilhamento de recursos. Qualquer que seja a modificação

no microcódigo original, não haverá microinstruções que executem essas micro-operações em paralelo, pois elas compartilham recursos do *hardware* e não podem ser executadas ao mesmo tempo. Dessa forma, modificações no microcódigo não produzirão combinações de micro-operações que requerem alterações no decodificador.

Como mencionamos anteriormente, o compartilhamento de recursos não é o único motivo para a execução de micro-operações de forma exclusiva. A deficiência de paralelismo no microcódigo, ocasionada por dependências de dados e controle, e a dificuldade de se gerar microcódigo paralelo de qualidade fazem com que muitas das micro-operações não sejam executadas em paralelo no microprograma em questão. Essas micro-operações também podem ser agrupadas em campos, mas sob a pena da perda de flexibilidade em futuras modificações no microcódigo. Observe que, diferente do exemplo acima, as modificações no microcódigo original podem gerar microinstruções que executam duas ou mais micro-operações agrupadas em um mesmo campo. Entretanto, a perda de flexibilidade não é tão custosa quanto na codificação máxima. Considere o caso em que o microcódigo é modificado e duas micro-operações, inicialmente executadas de forma exclusiva e agrupadas em um mesmo campo, passam a ser executadas por uma mesma microinstrução. Note que, da forma como foi construído, o decodificador do campo não possui uma *codeword* capaz de ativar as duas micro-operações ao mesmo tempo, portanto, o projetista tem as seguintes opções:

- Se o decodificador do campo não estiver completo, ou seja, se ele não possuir 2^q combinações de saídas, onde q é o número de entradas, então o projetista pode optar por adicionar uma nova *codeword* que produza como saída as duas micro-operações. A Figura 2.5 mostra a adição de uma nova *codeword* que ativa as micro-operações μOp_A e μOp_B ao mesmo tempo. O decodificador possui $q = 3$ entradas e antes da adição da nova *codeword* possuía apenas 5 combinações de saídas: *NOP*, μOp_A, μOp_B, μOp_C e μOp_D. A inclusão de novas *codewords* deve ser feita de maneira cuidadosa para que o decodificador não se torne complexo.

- Se as micro-operações puderem ser executadas em tempos distintos, a microinstrução que contém as duas micro-operações pode ser dividida em duas, onde cada micro-operação é executada em uma das duas microinstruções. Essa solução não exige modificações no *hardware*, mas pode resultar em perda de desempenho devido ao aumento do número de microinstruções no microcódigo.

- O projetista pode transferir uma das micro-operações para outro campo compatível, que não tenha micro-operações conflitantes. Nesse caso, os decodificadores dos dois campos e o roteamento dos fios serão modificados.

- Em última instância, as micro-operações podem ser totalmente reagrupadas para respeitar as novas restrições. Essa abordagem pode modificar consideravelmente o *hardware*.

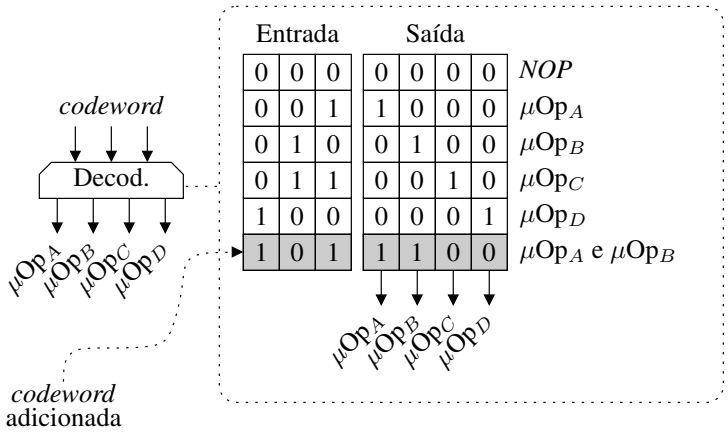


Figura 2.5: Exemplo de inclusão de *codewords* no decodificador do campo.

Para otimizar a redução da largura do microcódigo, Schwartz utilizou uma técnica de busca exaustiva que agrupava as micro-operações no menor número possível de campos [116]. Grasselli e Montanari [54] mostraram que o agrupamento com o menor número de campos não gerava necessariamente a solução com a menor quantidade de *bits* por microinstrução e reformularam o problema utilizando técnicas de minimização de tabelas lógicas [74]. A solução proposta por Grasselli e Montanari era impraticável em microcódigos com um número grande de micro-operações e outras técnicas para agrupamento de micro-operações surgiram. Jayasri e Basu [70] modelaram o problema utilizando programação linear inteira e Baer e Koyama [14] propuseram um algoritmo *Branch-and-Bound* para agrupar as micro-operações. Em 1979, Robertson [113] provou que o problema é $\mathcal{NP}$-Completo e, desde então, diversas heurísticas foram propostas para solucionar o problema [19, 63, 75, 98, 106, 109, 111, 112, 121].

A codificação mínima alia flexibilidade, *hardware* simples e pouco atraso no sinal, sendo possível, muitas vezes, adicionar a lógica de decodificação no mesmo estágio de *pipeline* da μROM. De fato, esse esquema de codificação é vastamente utilizado nas instruções e microinstruções de arquiteturas modernas [60]. As microinstruções dessas arquiteturas são geralmente divididas em campos que, via de regra, codificam operações e registradores acessados de forma exclusiva pela microinstrução. Entretanto, as micro-operações agrupadas em um mesmo campo, como a seleção dos registradores fontes, são geralmente correlacionadas e aparentam ser agrupadas de forma sistemática pelos projetistas do *hardware*, sem o auxílio dos algoritmos de agrupamento descritos acima. Por exemplo, uma prática muito comum é a utilização de campos para seleção de registradores. Observe que o campo agrupa apenas micro-operações do mesmo tipo, como "Leia dado do registrador R_i" ou "Escreva dado no registrador R_i". Essa prática torna o formato da microinstrução mais simples e fácil de ser interpretado, no entanto, impõe restrições no agrupamento das micro-operações e limita a capacidade de compressão do microcódigo. O agrupamento de micro-operações mutuamente exclusivas, para manter a lógica

de decodificação simples, é outro fator que limita a capacidade de compressão do microcódigo.

Codificação Indireta

O modelo proposto por Schwartz também é conhecido como codificação direta (*direct encoding*), pois a lógica utilizada para decodificar um determinado campo depende apenas dos valores armazenados no próprio campo. Outra forma de se reduzir a largura do microcódigo é através de codificação indireta (*indirect encoding*). Nesse modelo a decodificação depende de valores armazenados em outros campos e também é conhecido como "codificação em dois níveis" ou *bit steering* [4]. A Figura 2.6 mostra exemplos de microinstruções sem codificação (a), com codificação direta (b) e com codificação indireta (c). Os campos A e B da Figura 2.6 (c) compartilham o valor do campo C. Observe que os decodificadores dos campos A e B utilizam o valor armazenado no campo C.

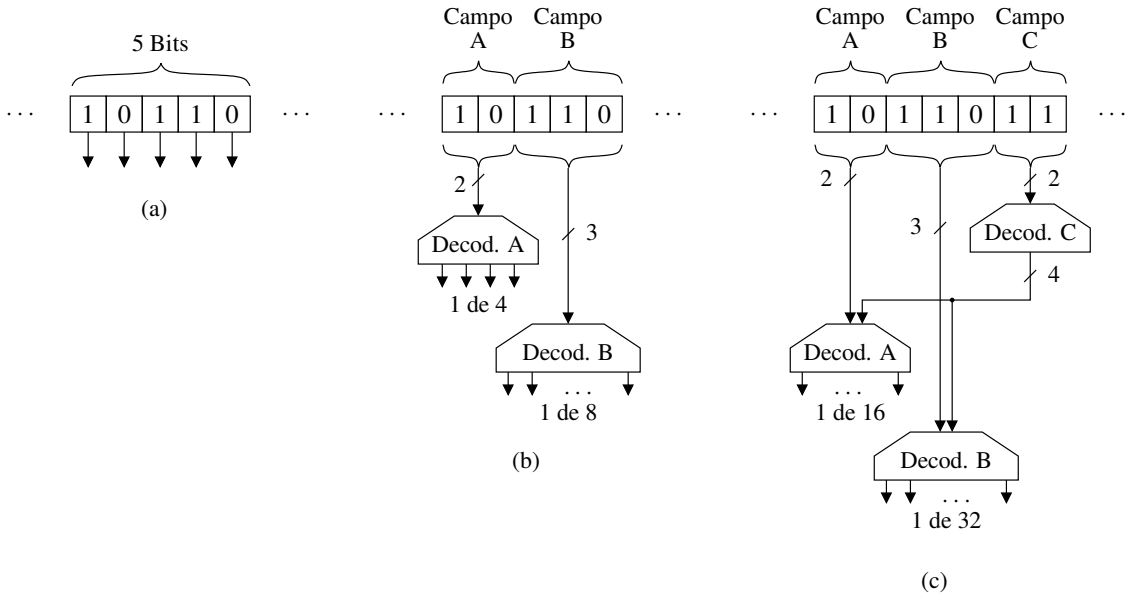


Figura 2.6: Exemplos de codificação de microinstruções. (a) Sem codificação. (b) Codificação direta. (c) Codificação indireta.

A divisão do conjunto de microinstruções em formatos, onde cada formato define a maneira como os *bits* serão interpretados, é uma forma de codificação indireta, ou *bit steering*. A codificação indireta pode gerar uma lógica de decodificação complexa e é utilizada de forma restrita no microcódigo. Mathialagan e Biswas [95] propuseram um algoritmo para detectar conjuntos de campos, diretamente codificados, que possam ter parte dos *bits* compartilhados. Biswas [22] propôs um método mais simples para detectar esses conjuntos, mas a complexidade do algoritmo ainda inviabiliza a utilização em projetos grandes.

Geração de Colunas

Halatsis e Gaitanis [58] propuseram um método para reduzir a largura do microcódigo que limita a quantidade de lógica adicionada junto à saída da μROM. Seja S o conjunto de colunas, ou micro-operações, do microcódigo, o objetivo é armazenar na μROM apenas um "subconjunto gerador" $S^\star \in S$ e gerar as colunas restantes ($S - S^\star$) a partir de $S^\star$ utilizando portas lógicas *AND* ou *OR* na saída da μROM. A Figura 2.7 mostra um exemplo de geração das colunas a partir do conjunto gerador $S^\star$. As colunas a, c, d e f compõem o conjunto gerador e são armazenadas diretamente na μROM e as colunas restantes são geradas a partir do conjunto gerador.

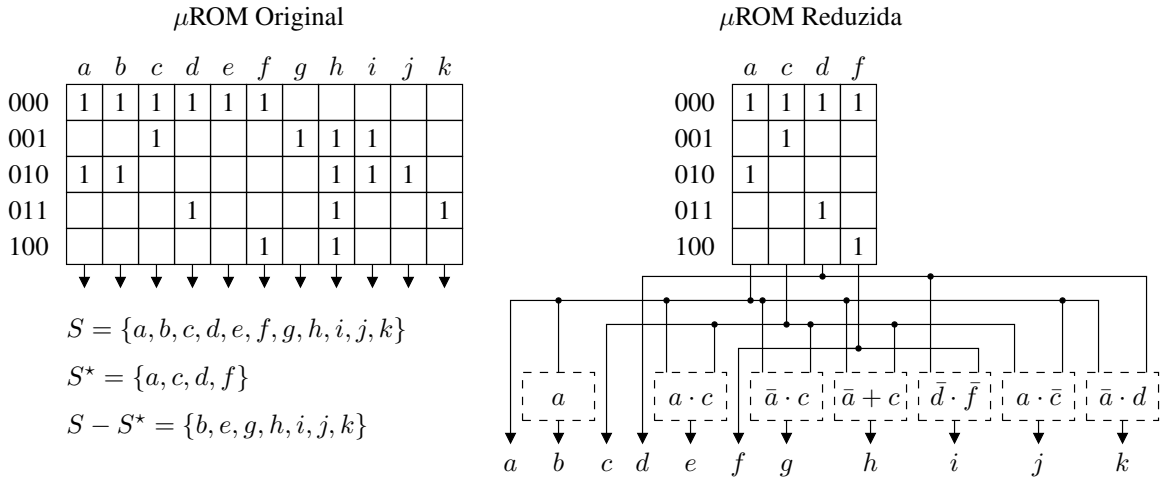


Figura 2.7: Geração de colunas a partir do conjunto gerador $S^\star$. As colunas b, e, g, h, i, j e k são geradas a partir das colunas a, c, d, f.

Martinez e Powers [94] também investigaram a técnica de geração de colunas e propuseram heurísticas e um algoritmo *branch-and-bound* para encontrar conjuntos geradores mínimos.

Apesar desse método manter a lógica de geração de colunas simples, pequenas modificações no microcódigo podem invalidar o conjunto gerador de colunas e a largura da μROM codificada pode variar expressivamente, dificultando a previsão da área final. Assim sendo, o método não é considerado flexível.

Microcódigo em Dois Níveis

Em 1962, Grasselli [53] propôs um modelo que dividia o microcódigo em dois níveis. O microcódigo era armazenado em duas memórias, a *Pathfinder Memory* (PFM) e a *Control Memory* (CM). A PFM armazenava seqüências de *micro-strings* e a CM armazenava as microinstruções[6]

[6]No artigo o autor utiliza o termo *micro-orders* para denotar microinstruções.

únicas. Uma *micro-string* é um vetor com k elementos (E_1, E_2, $\cdots$, E_k), onde cada elemento armazena o endereço de uma microinstrução da CM. Dessa forma, uma seqüência de microcódigo com m microinstruções, era representada por $\lceil m/k \rceil$ *micro-strings*. A Figura 2.8 ilustra o esquema de compressão em dois níveis proposto por Grasselli.

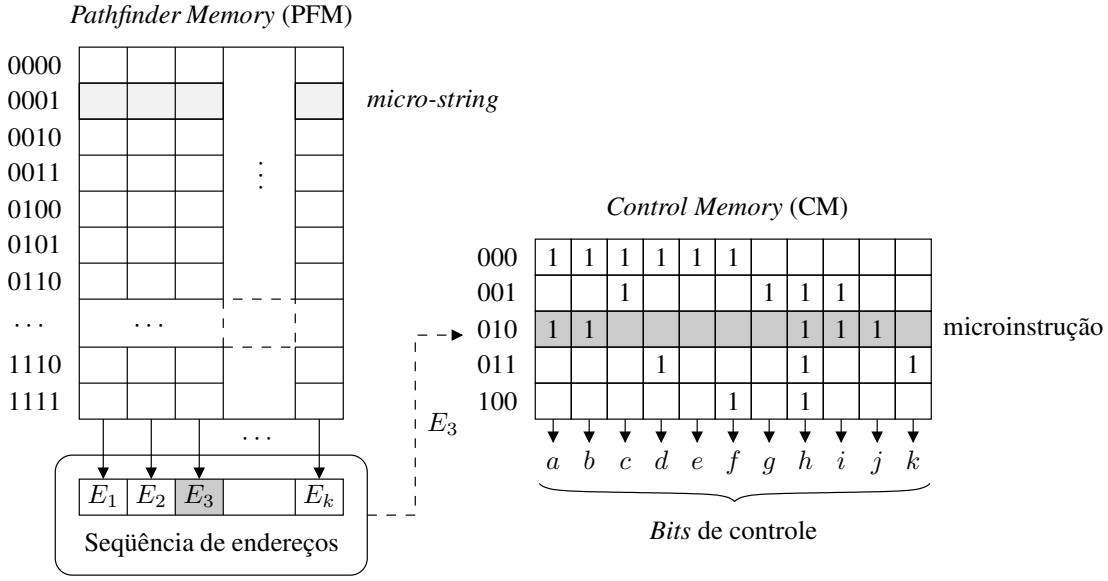


Figura 2.8: Esquema de compressão em dois níveis proposto por Grasselli [53].

No esquema da Figura 2.8, a CM seria implementada com uma ROM, ou mesmo uma rede lógica de decodificação, mas a PFM deveria ser modificável de forma que o seu conteúdo pudesse ser alterado durante a operação da máquina. Registradores posicionados na saída da PFM funcionavam como *caches* para as *micro-strings* que, uma vez carregadas da PFM, forneciam a seqüência de endereços para a CM. Naquela época, o tempo de leitura de memórias modificáveis era muito longo e o objetivo de Grasselli não era reduzir o tamanho do microcódigo, e sim propor um modelo com microcódigo modificável que não sacrificasse o desempenho do processador.

Além da baixa utilização dos campos, a replicação de microinstruções é outro fator que agrava o tamanho de microcódigos horizontais. Stritter e Tredennick [55, 119, 122] investigaram a organização do microcódigo em dois níveis para reduzir o tamanho do microcódigo horizontal utilizado no microprocessador MC68000 da Motorola. Nesse modelo, as microinstruções possuem poucos *bits* e são utilizadas como apontadores para as "nanoinstruções". As nanoinstruções são largas, e armazenam os *bits* de controle da unidade de execução. A Figura 2.9 mostra um exemplo de microcódigo em dois níveis onde a μROM e a *nano*ROM armazenam as microinstruções e as nanoinstruções respectivamente. A microinstrução possui um campo com o endereço da próxima microinstrução, que auxilia no seqüenciamento do mi-

crocódigo, e um campo que aponta para a nanoinstrução, evitando assim a replicação dos *bits* de controle.

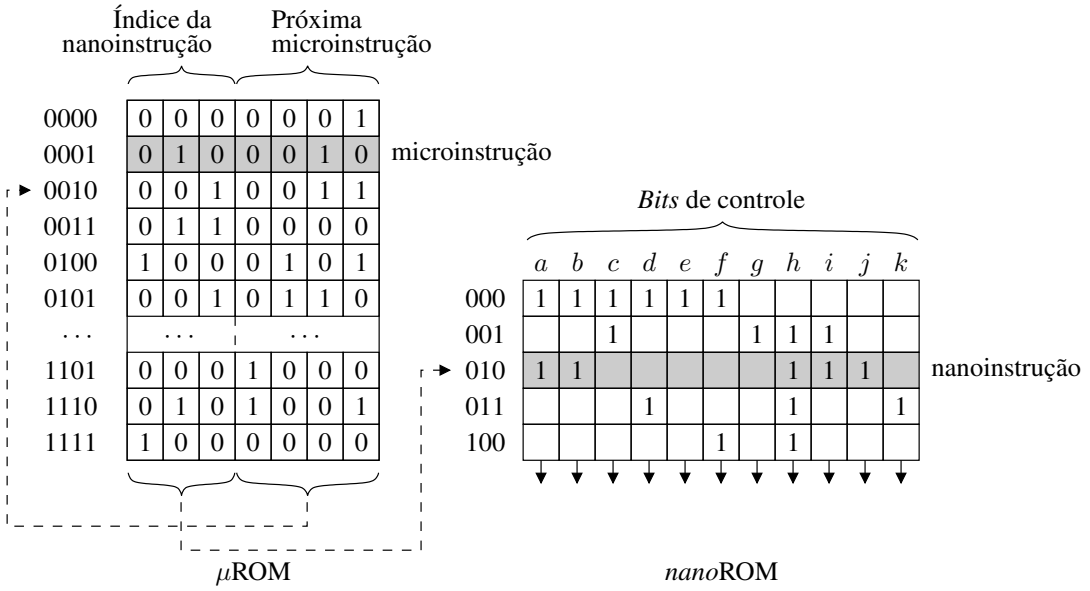


Figura 2.9: Organização do microcódigo em dois níveis: μROM e *nano*ROM.

Da mesma forma que na codificação máxima, o número de *bits* necessário no campo que aponta para as nanoinstruções é $\lceil log_2 M \rceil$, onde M, nesse caso, é o número de nanoinstruções na *nano*ROM. A organização do microcódigo em dois níveis e a técnica de codificação máxima são semelhantes no sentido em que a *nano*ROM pode ser vista como um decodificador que converte ponteiros, ou *codewords*, em *bits* de controle da unidade de execução. De fato, a lógica de decodificação adicionada na codificação máxima poderia ser implementada através de uma "tabela verdade" armazenada em uma ROM [108].

Apesar da semelhança funcional, a *nano*ROM e o decodificador da técnica de codificação máxima diferem em aspectos como área e flexibilidade. A *nano*ROM ocupa uma área maior do que a lógica de decodificação mas pode ser facilmente modificada através de alterações nos *bits* das nanoinstruções ou através da inclusão de novas nanoinstruções.

O processador QM-1 [4, 46, 100, 114], da empresa Nanodata, também utilizava uma organização de microcódigo em dois níveis. As microinstruções e nanoinstruções do QM-1 possuíam 18 e 360 *bits*, respectivamente. Além do seqüenciamento das microinstruções, o QM-1 permitia o seqüenciamento das nanoinstruções, em outras palavras, a microinstrução apontava para um "nanoprograma". A microinstrução possuía um apontador para os nanoprogramas (7 *bits*) e um campo com parâmetros (11 *bits*). Essa funcionalidade evitava a replicação de nanoprogramas semelhantes. O *hardware* de seqüenciamento de nanoinstruções ocupava mais área e

tornava a unidade de controle complexa. Essas características inviabilizam a implementação dessa técnica em processadores de alto desempenho.

Menzilcioglu [97] investigou a utilização de microcódigo em dois níveis para aumentar a capacidade de armazenamento do processador Warp [8]. O autor observou que alguns campos do microcódigo apresentavam mais regularidade do que outros e utilizou diversas *nano*ROMs para agrupar diferentes partes do microcódigo. A Figura 2.10 mostra um exemplo de organização em dois níveis com mais de uma *nano*ROM. Nesse esquema, os campos com pouca regularidade também podem ser armazenados pela própria microinstrução, que é o caso dos campos a e b na Figura 2.10.

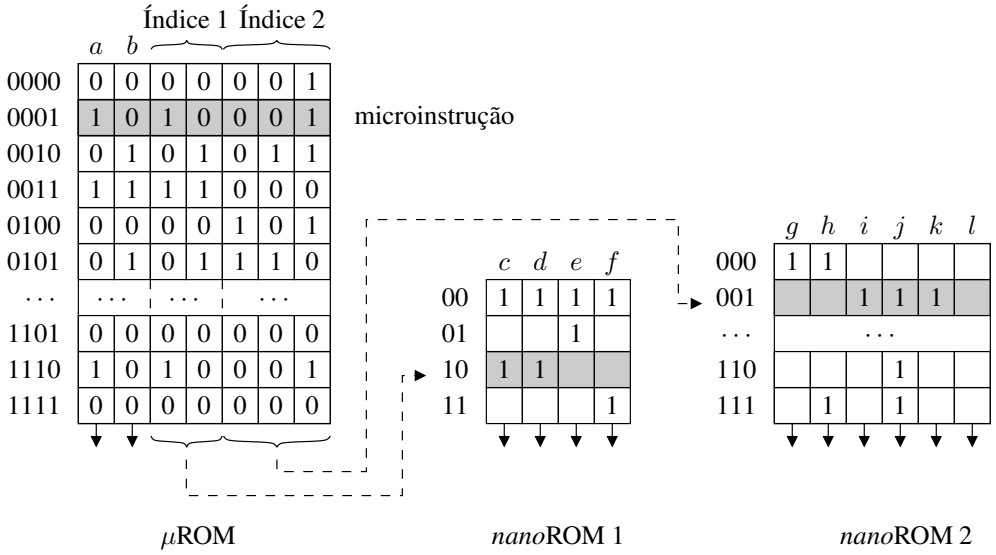


Figura 2.10: Microcódigo em dois níveis com mais de uma *nano*ROM.

De acordo com a operação, alguns campos das microinstruções do Warp não eram utilizados e podiam ser modificados sem alterar a semântica do microprograma. O conteúdo desses campos poderia ser modificado para aumentar o número de padrões idênticos no microcódigo. Menzilcioglu utilizou essa informação, e a troca de operandos simétricos[7], para minimizar o tamanho do microcódigo. Entretanto, o autor não apresentou um método sistemático para identificar e agrupar os campos do microcódigo dentro das *nano*ROMs.

Zhao e Papachristou [136] propuseram uma organização do microcódigo em memórias distribuídas muito similar ao modelo com mais de uma *nano*ROM utilizado por Menzilcioglu [97] (Figura 2.10). No modelo de Zhao e Papachristou, a μROM e a *nano*ROM são chamadas de *index memory* e *microcode memory module*, respectivamente. Os autores ainda apresentaram uma heurística para identificar e agrupar campos semelhantes dentro das *microcode memory modules*.

[7] Operandos simétricos são operandos que, se trocados, não alteram a semântica da microinstrução.

Ishiura e Yamaguchi [67] utilizaram codificação máxima para reduzir o tamanho de aplicações VLIW para *Application Specific VLIW Processors*. Os *bits*, ou colunas, do programa VLIW são agrupados em campos e, para cada campo, adiciona-se um decodificador à saída da memória que armazena o programa comprimido. Os decodificadores são implementados com memórias ROMs, o que torna a técnica muito similar à organização de microcódigo em dois níveis. Os autores também propuseram uma heurística para agrupar as colunas da instrução VLIW em campos que leva em conta o tamanho dos decodificadores.

Hum *et alii* [65] patentearam um esquema de compressão semelhante à organização do microcódigo em dois níveis. As colunas do microcódigo são agrupadas em tabelas, similares às *nano*ROMs, e o microcódigo comprimido, contendo os apontadores para as tabelas, é armazenado na μROM.

Borin *et alii* [25] utilizaram uma variação do modelo patenteado por Hum *et alii* [65] e propuseram três algoritmos para maximizar a compressão do microcódigo através do agrupamento de colunas. Esses algoritmos estão descritos em detalhes no Capítulo 4.

Apesar da diversidade dos termos empregados nos trabalhos acima, todos organizam o microcódigo em dois níveis e se identificam quanto à idéia de agrupar padrões de *bits* repetidos em tabelas e substituir as microinstruções originais por apontadores para essas tabelas.

A organização do código em dois níveis se adequa às restrições impostas em processadores de alto desempenho. A μROM e a *nano*ROM podem ser naturalmente posicionadas em estágios de *pipeline* distintos, evitando assim problemas de temporização no projeto do processador. Alterações no microcódigo podem ser realizadas através de modificações nos *bits* da μROM e da *nano*ROM. Além disso, alguns *bits* extras podem ser mantidos pela μROM e pela *nano*ROM para permitir modificações mais expressivas. A utilização de várias *nano*ROMs não aumenta significativamente a complexidade do *hardware* e pode reduzir a área total, explorando a entropia dos campos do microcódigo [25].

O Capítulo 3 detalha o esquema de compressão baseado na organização do microcódigo em dois níveis e o Capítulo 4 apresenta os algoritmos propostos para agrupar as colunas do microcódigo de forma a maximizar a redução da área.

2.3.3 Outras Abordagens

Guttag [56] utilizou técnicas de codificação de ROM e truques de projeto VLSI para reduzir o tamanho da ROM que armazenava o microprograma. A técnica modifica a estrutura interna da ROM e eventuais modificações no microcódigo podem alterar significativamente o projeto do *hardware*. Portanto, a técnica não é considerada flexível.

Um processador microprogramado pode ser visto como um par de autômatos que interagem entre si [51]. O "autômato operacional", também chamado de OP (*Operational Part*), representa a via de dados e o "autômato de controle", também conhecido como CP (*Control Part*),

modela a unidade de controle. Os sinais de saída (entrada) do autômato CP são conectados diretamente à entrada (saída) do OP. Observe que os sinais de saída do CP são as micro-operações. Glushkov [51] propôs uma técnica para transformar o autômato CP em uma máquina de estados incompletamente especificada e utilizou algoritmos de simplificação de máquinas incompletamente especificadas [74] para reduzir o número de estados de CP. Agerwala [3] descreve o método de Glushkov em detalhes e mostra que, mesmo para uma máquina muito simples (com apenas três estados), o procedimento de redução de estados é complexo, o que torna a técnica inapropriada para problemas mais complexos.

As técnicas descritas até então consideram a utilização de memórias ROMs para armazenar o microcódigo. Entretanto, o microcódigo também pode ser armazenado em um *Programmable Logic Array* (PLA) [108]. A forma de acesso aos dados em PLAs é orientada a conteúdo, ao invés de endereço, e essa característica permite que os campos do microcódigo que são pouco utilizados sejam armazenados de forma mais compacta.

Papachristou e Reuter [104] propuseram técnicas para particionar e reduzir a área do PLA que armazena o microcódigo. Em outro trabalho, Papachristou e Pandya [103] apresentaram um esquema de particionamento que considera a área e o atraso do sinal proporcionado pelo PLA. Otimizações para redução da área do PLA reduzem a flexibilidade de alterações do microcódigo. PLA *folding* [47, 57] e minimização *booleana* [26] são otimizações que modificam o circuito do *hardware* e alterações no conteúdo do microcódigo podem provocar mudanças grandes na área e na organização deste circuito.

2.3.4 Quadro Comparativo

A Tabela 2.1 mostra um quadro comparativo com os métodos de compressão de microcódigo apresentados nesta seção. As técnicas são classificadas em Bom, Médio e Ruim, de acordo com a capacidade de compressão da técnica e as restrições de projeto descritas na Seção 1.2.

As técnicas de compactação de microcódigo, descritas na Seção 2.3.1, são aplicadas na fase de compilação do microcódigo e não requerem *hardware* de descompressão. Assim sendo, os critérios de flexibilidade, desempenho e simplicidade do *hardware* não se aplicam a elas.

Como mencionamos anteriormente, a lógica de decodificação adicionada na técnica de codificação máxima poderia ser implementada através de uma "tabela verdade" armazenada em uma ROM [108], entretanto, essa abordagem torna a codificação máxima equivalente à organização do microcódigo em dois níveis. Assim sendo, para fins de comparação, consideramos que o decodificador da técnica de codificação máxima é implementado com lógica aleatória [108]. Esse tipo de circuito é geralmente complexo e alterações no microcódigo podem causar modificações significativas no *hardware*, o que torna a técnica Ruim no quesito flexibilidade. O *hardware* complexo também aumenta o tempo de propagação do sinal que, no entanto, é sempre menor do que o atraso causado pela técnica de microcódigo em dois níveis e pode ser enqua-

Tabela 2.1: Avaliação das técnicas de compressão de microcódigo.

Seção	Técnica	Flexibilidade	Desempenho [a]	Simplicidade do *Hardware* [b]	Capacidade de Compressão [c]
2.3.1	Compactação de Microcódigo [d]	Bom	N.A.[e]	N.A.	N.D.[f]
2.3.2	Codificação Máxima	Ruim	Médio	Ruim	Bom
	Codificação Mínima	Bom	Bom	Bom	Médio
	Codificação Indireta	Bom	Médio	Bom	N.D.
	Geração de Colunas	Ruim	Bom	Médio	Bom
	Microcódigo em Dois Níveis	Bom	Médio	Bom	Bom
2.3.3	Guttag [56]	Ruim	N.D.	Ruim	Bom
	Glushkov [51]	Ruim	N.A.	N.A.	N.D.
	Microcódigo em PLA	Ruim	Bom	Ruim / Bom[g]	Bom

[a] Atraso na propagação do sinal. Bom denota pouco atraso e Médio indica que um estágio extra de *pipeline* pode ser necessário.
[b] Complexidade do *hardware* para descomprimir o microcódigo. Bom indica que o *hardware* é simples.
[c] Bom indica melhores razões de compressão.
[d] Realizada durante a compilação do microcódigo.
[e] Não se aplica a esta técnica.
[f] Informação não disponível.
[g] Algumas otimizações em PLA são feitas com a ajuda de ferramentas de CAD, o que pode tornar o projeto e a manutenção do *hardware* simples.

drado em um novo estágio de *pipeline*.

A codificação mínima alia flexibilidade, desempenho e simplicidade do *hardware*. Entretanto, para manter o *hardware* simples, apenas micro-operações mutuamente exclusivas são agrupadas em um mesmo campo e essa restrição limita a capacidade de compressão da técnica. De fato, o microcódigo de alguns processadores modernos utiliza codificação mínima para agrupar micro-operações em campos e, como veremos no Capítulo 5, a organização do microcódigo em dois níveis consegue reduzir ainda mais esses microcódigos, atingindo razões de compressão de até 50%.

Os métodos automáticos para codificação indireta [22, 95] são muito custosos e, assim como na codificação mínima, os projetistas de microprocessadores dividem as microinstruções manualmente em formatos. Dessa forma, a técnica torna-se flexível e o *hardware* relativamente simples.

A técnica de geração de colunas limita a quantidade de lógica adicionada junto à saída da μROM, o que mantém o atraso da propagação do sinal e o *hardware* pequenos. Entretanto, cada coluna é gerada por combinações de portas lógicas diferentes. Desse modo, o *hardware*

de geração de colunas não é padronizado e a complexidade do *hardware* é considerada Média. Modificações no microcódigo podem invalidar a lógica de geração e o conjunto gerador de colunas, portanto a técnica não é flexível. Os resultados apresentados por Martinez e Powers [94] mostram que a técnica tem capacidade de compressão melhor do que a codificação mínima, porém, os microcódigos utilizados nos experimentos são pequenos, menores do que 64×36 *bits*, e não fica claro se a técnica pode ser aplicada com eficiência em microcódigos maiores.

O esquema de organização do microcódigo em dois níveis possui boa flexibilidade, pois o microcódigo pode ser modificado através de alterações nos *bits* da μROM e da *nano*ROM. O acesso indireto à *nano*ROM causa atraso na propagação do sinal, no entanto, a *nano*ROM pode ser posicionada em outro estágio de *pipeline*. No Capítulo 3 veremos que a perda de desempenho causada pelo estágio extra no *pipeline* é muito pequena. O *hardware* adicionado é considerado simples, consistindo em blocos de memórias ROM e interconexões entre estes blocos. Por fim, os resultados apresentados no Capítulo 5 e em [25] mostram que a capacidade de compressão dessa técnica é Boa.

A técnica de compressão apresentada por Guttag [56] modifica a estrutura interna da μROM e eventuais modificações no microcódigo podem alterar significativamente o projeto do *hardware*. Apesar dos resultados apresentados por Guttag serem promissores, as modificações dentro da μROM tornam o *hardware* complexo. O desempenho da técnica não é apresentado e não fica claro se a μROM pode ser dividida em dois estágios de *pipeline*.

A técnica de Glushkov [51] modela a unidade de controle com um "autômato de controle", que é otimizado com algoritmos de simplificação de máquinas incompletamente especificadas [74]. Modificações no modelo podem invalidar as otimizações e modificar significativamente o *hardware*, portanto, a técnica não é flexível. Assim como a técnica de compactação de microcódigo, as otimizações são feitas antes da implementação do microcódigo no *hardware*, logo, informações como desempenho e complexidade do *hardware* não se aplicam.

Por fim, as técnicas de otimização de microcódigo em PLA [103, 104] podem proporcionar pouco atraso na propagação do sinal e boas razões de compressão. Entretanto, assim como na abordagem de Guttag [56], modificações no conteúdo do microcódigo podem invalidar algumas otimizações [47, 57] e alterar significativamente o tamanho e o desempenho do *hardware*.

Técnica Escolhida - Microcódigo em Dois Níveis

A análise apresentada nesta seção indica que, dentre as técnicas existentes, a mais adequada para se comprimir o microcódigo de processadores de alto desempenho é a organização do microcódigo em dois níveis, como explicado abaixo.

A flexibilidade da técnica de compressão é uma característica essencial no projeto de processadores de alto desempenho. De acordo com a Tabela 2.1, as técnicas de compressão que possuem boa flexibilidade são: compactação de microcódigo, codificação mínima, codificação indireta e microcódigo em dois níveis. A compactação de microcódigo, quando aplicada, é feita

em tempo de compilação e, a não ser pela mudança na entropia do microcódigo, não influi na utilização das outras técnicas. A codificação mínima e a codificação indireta já são utilizadas pelos projetistas para reduzir o tamanho do microcódigo. A codificação mínima é utilizada para agrupar operações e operadores em campos da microinstrução e a codificação indireta é utilizada para dividir o conjunto de microinstruções em formatos, onde cada formato define a maneira como os *bits* serão interpretados. A capacidade de compressão da codificação mínima é Média e, como mostramos em [25], mesmo em microcódigos que utilizam essas técnicas, a organização do microcódigo em dois níveis é capaz de comprimir o microcódigo em até 50% do seu tamanho. Além disso a análise realizada no Capítulo 3 mostra que o *hardware* adicionado na organização do microcódigo em dois níveis é simples e a perda de desempenho, causada por um possível estágio extra de *pipeline*, é pequena. Assim sendo, escolhemos a técnica de organização do microcódigo em dois níveis para comprimir o microcódigo.

2.4 Conclusões

Recentemente, a compressão de código foi utilizada para reduzir o tamanho do código das aplicações em sistemas embarcados [13, 21, 31, 101]. A descompressão de trechos do programa sob demanda e o padrão de acesso aleatório a esses trechos tornam as técnicas tradicionais de compressão de dados inapropriadas para a compressão de código, o que incentivou o desenvolvimento de diversas técnicas de compressão de código em processadores embarcados nos últimos anos. Processadores de alto desempenho, por outro lado, possuem características que limitam o tamanho e do desempenho do *hardware* do descompressor. Essas restrições aliadas ao fato do microcódigo apresentar pouca regularidade no microprograma inviabilizam a utilização das técnicas de compressão de código para comprimir o microcódigo.

Desde a criação do conceito de microprogramação em 1951 [128], diversas técnicas foram propostas para a redução do tamanho do microcódigo. Essas técnicas utilizavam codificação de *bits* e escalonamento de operações para reduzir a largura e o número de microinstruções do microcódigo. Assim como as instruções, as microinstruções de arquiteturas modernas são sistematicamente codificadas em campos (codificação mínima) e formatos (codificação indireta) para reduzir o tamanho do microcódigo. Entretanto, testes com a organização do microcódigo em dois níveis [25] mostram que esses microcódigos ainda podem ter seu tamanho reduzido em até 50%.

A análise apresentada neste capítulo mostra que a organização do microcódigo em dois níveis é a técnica mais apropriada para comprimir o microcódigo em processadores de alto desempenho. O Capítulo 3 detalha o esquema de compressão baseado na organização do microcódigo em dois níveis e o Capítulo 4 apresenta os algoritmos propostos para agrupar as colunas do microcódigo de forma a maximizar a redução da área.

Capítulo 3

Compressão de Microcódigo em Dois Níveis

A análise apresentada no Capítulo 2 mostra que a técnica mais adequada para se comprimir o microcódigo de processadores de alto desempenho é a organização do microcódigo em dois níveis. Este capítulo apresenta em detalhes o esquema de compressão de microcódigo em dois níveis.

A organização do microcódigo em dois níveis foi explorada em diversos trabalhos com pequenas variações em torno da microarquitetura do descompressor [25, 46, 65, 97, 114, 119, 136]. Apesar da semelhança das estruturas e da organização do descompressor, os termos usados para definir os componentes da microarquitetura, como a μROM e a *nano*ROM, diferem na maioria desses trabalhos. Para descrever o esquema de compressão de microcódigo em dois níveis utilizaremos os termos definidos em [25]. Assim sendo, a μROM comprimida é chamada de "vetor de apontadores" e as *nano*ROMs são identificadas como "dicionários".

A Seção 3.1 descreve o esquema básico da compressão de microcódigo, a Seção 3.2 introduz a compressão com particionamento do microcódigo e a Seção 3.3 detalha a compressão baseada no agrupamento de colunas. A Seção 3.4 discute o desempenho do esquema de compressão, a Seção 3.5 compara a redução do número de *bits* do microcódigo com a redução da área da μROM e, finalmente, a Seção 3.6 apresenta as considerações finais.

3.1 Esquema de Compressão Básico

A idéia básica da compressão é identificar padrões de *bits* no microcódigo e armazená-los em um dicionário. A Figura 3.1 (b) ilustra esse esquema de compressão do microcódigo. No formato original, o "endereço" acessa diretamente a μROM para carregar a microinstrução ("μInstrução"). Na forma comprimida, os padrões únicos são armazenados no dicionário e as microinstruções do microcódigo original são substituídas por apontadores para o dicionário.

Dessa forma, a μROM original é substituída por um vetor de apontadores e um dicionário.

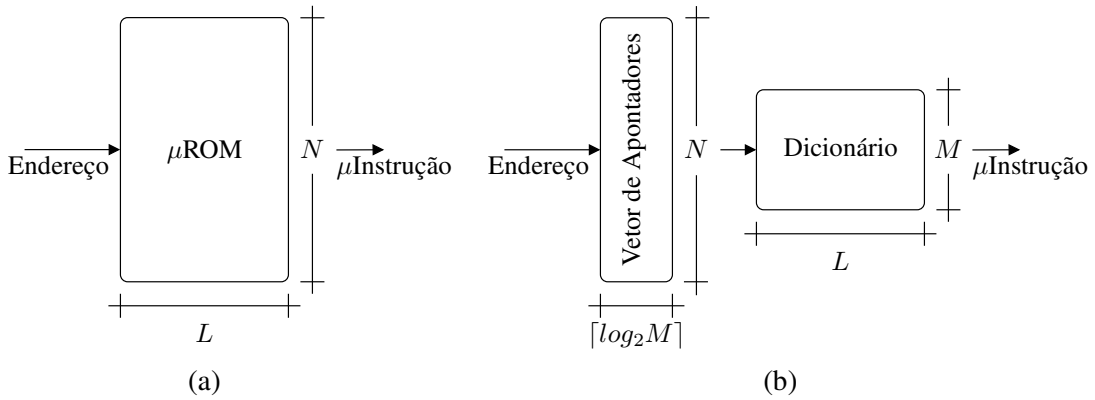


Figura 3.1: Idéia básica da compressão de microcódigo. (a) μROM original. (b) μROM comprimida.

Para exemplificar a compressão, vamos assumir que a μROM original tem N microinstruções (N linhas), com L *bits* (L colunas) cada uma, e que há um total de M microinstruções únicas. A μROM original ocupa $N \times L$ *bits*, enquanto que a μROM comprimida gasta $N \times \lceil log_2 M \rceil + M \times L$ *bits*, onde $M \times L$ é o tamanho do dicionário de padrões e $\lceil log_2 M \rceil$ é o número de *bits* necessários para indexá-lo. Para $N = 20\,000$, $M = 12\,000$ e $L = 70$ a μROM comprimida usa $1\,120\,000$ *bits*, contra $1\,400\,000$ da μROM original. Isso significa uma redução de 20% no número de *bits*.

A eficiência da técnica de compressão depende da quantidade de padrões únicos (M) no microcódigo. Quanto maior for o valor de M, pior será a razão de compressão. De fato, se não existirem microinstruções repetidas, ou seja, se o número de padrões únicos for igual ao número de microinstruções ($M = N$), o microcódigo comprimido será maior do que o original. Observe que para $M = N$ o dicionário tem o mesmo tamanho ($L \times M$) do microcódigo original ($L \times N$). Desse modo, o vetor de apontadores juntamente com o dicionário de padrões é maior do que o microcódigo original.

Uma forma de reduzir o tamanho de M e aumentar a eficiência da técnica de compressão é particionar o microcódigo em campos, ou sub-palavras, e comprimir as partes de forma independente. A próxima seção apresenta um esquema de compressão baseado em particionamento do microcódigo.

3.2 Esquema de Compressão com Particionamento

A quantidade de padrões únicos (M_i) em cada campo do microcódigo é sempre menor ou igual a M. O objetivo da compressão com particionamento é dividir o microcódigo em campos, para reduzir a quantidade de padrões únicos, e comprimir cada campo separadamente. O resultado da compressão com particionamento é um vetor de apontadores e um dicionário para cada campo. A Figura 3.2 mostra um exemplo onde cada microinstrução é dividida exatamente ao meio, em duas sub-palavras. Sendo M_1 e M_2 o número de padrões únicos nas duas metades, a μROM comprimida ocupará $N \times (\lceil log_2 M_1 \rceil + \lceil log_2 M_2 \rceil) + M_1 \times (L \div 2) + M_2 \times (L \div 2)$ *bits*. Para $N = 20\,000$, $M_1 = 5\,000$, $M_2 = 5\,000$ e $L = 70$, a μROM comprimida usa $870\,000$ *bits*, contra $1\,400\,000$ da μROM original. Nesse caso, a redução é de 38%.

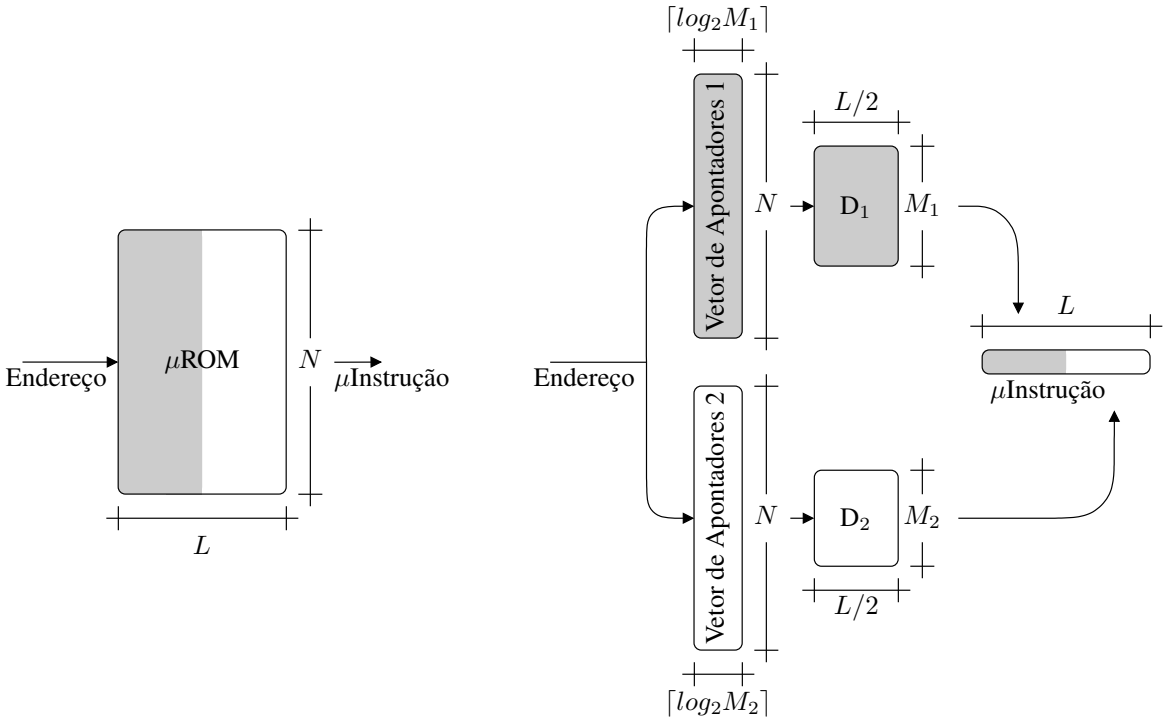


Figura 3.2: Compressão do microcódigo particionado. (a) μROM original. (b) μROM comprimida.

O esquema da Figura 3.2 (b) pode ser estendido para K partições, cada uma com um vetor de apontadores e um dicionário. Para isso, basta que o endereço de entrada seja distribuído para cada vetor de apontadores e os padrões de *bits* carregados dos dicionários sejam reorganizados para compor a microinstrução original.

Seja P_i o conjunto de padrões de *bits* da partição i, então, o conjunto de padrões do microcó-

digo completo (P) é formado pela concatenação dos padrões em $P_1, P_2, \cdots, P_K$ que ocorrem na mesma linha do microcódigo. Assim sendo, $P \subseteq P_1 \times P_2 \times \cdots \times P_K$. Para $M_i = |P_i|$ temos que $M \leq M_1 \times M_2 \times \cdots \times M_K$. Também podemos afirmar que o número de padrões do microcódigo completo (M) é sempre maior ou igual à quantidade de padrões em cada partição do microcódigo ($M_1, M_2, \cdots, M_K$), portanto, o número de padrões no microcódigo respeita a seguinte desigualdade:

$$1 \leq Máximo(M_1, M_2, \cdots, M_K) \leq M \leq (M_1 \times M_2 \times \cdots \times M_K) \leq N \tag{3.1}$$

O tamanho do dicionário na compressão básica é $L \times M$, onde L é a largura, em *bits*, do microcódigo e M é o número de padrões do dicionário. Analogamente, o tamanho dos dicionários das partições é $M_i \times L_i$. Com base na desigualdade 3.1 concluímos que:

$$\underset{(1)}{M \times L} = M \times \sum_{i=1}^{K} L_i = \sum_{i=1}^{K} (L_i \times M) \geq \underset{(2)}{\sum_{i=1}^{K} (L_i \times M_i)} \tag{3.2}$$

ou seja, a soma do tamanho dos dicionários das partições (2) é sempre menor ou igual ao tamanho do dicionário na compressão básica (1). Desse modo, podemos afirmar que o particionamento não aumenta a quantidade de *bits* usada nos dicionários.

Por outro lado, o somatório dos tamanhos dos vetores de apontadores tende a crescer com o particionamento. Seja N o número de linhas do microcódigo e $\lceil log_2 M_i \rceil \times N$ o tamanho do vetor de apontadores da partição i, a quantidade de *bits* utilizada pelos vetores de apontadores é:

$$\sum_{i=1}^{K} (\lceil log_2 M_i \rceil \times N) = N \times \sum_{i=1}^{K} \lceil log_2 M_i \rceil \tag{3.3}$$

A desigualdade 3.1 aponta que $M \leq M_1 \times M_2 \times \cdots \times M_K$. Como M_i é sempre maior ou igual a 1, temos que:

$$log_2 M \leq log_2 (M_1 \times M_2 \times \cdots \times M_K) \tag{3.4}$$

Analogamente:

$$\begin{array}{rcl} \lceil log_2 M \rceil & \leq & \lceil log_2 (M_1 \times M_2 \times \cdots \times M_K) \rceil \\ & = & \lceil log_2 M_1 + log_2 M_2 + \cdots + log_2 M_K \rceil \\ & \leq & \lceil log_2 M_1 \rceil + \lceil log_2 M_2 \rceil + \cdots + \lceil log_2 M_K \rceil \\ & = & \sum_{i=1}^{K} \lceil log_2 M_i \rceil \end{array} \tag{3.5}$$

Assim sendo:

$$\underset{(3)}{N \times \lceil log_2 M \rceil} \leq N \times \underset{(4)}{\sum_{i=1}^{K} \lceil log_2 M_i \rceil} \tag{3.6}$$

ou seja, o somatório dos tamanhos dos vetores de apontadores (4) é sempre maior ou igual ao tamanho do vetor de apontadores na compressão básica (3).

As equações anteriores indicam que sempre que particionamos o microcódigo existe a tendência de se reduzir o tamanho total dos dicionários e aumentar o tamanho total dos vetores de apontadores. Portanto, o particionamento deve ser feito de forma que a redução dos dicionários seja sempre maior do que o aumento dos vetores de apontadores.

O particionamento da Figura 3.2 divide o microcódigo em duas sub-palavras com colunas consecutivas. Entretanto, podemos explorar o fato de que colunas distantes, não adjacentes, podem ser semelhantes e, dessa forma, agrupá-las em uma mesma partição, sem a restrição de serem consecutivas. A próxima seção apresenta o esquema de compressão com agrupamento de colunas.

3.3 Esquema de Compressão com Agrupamento

A compressão baseada em agrupamento de colunas agrupa colunas similares, indo além do particionamento simples da microinstrução em sub-palavras compostas por *bits* consecutivos. Por exemplo, a Figura 3.3 mostra um particionamento simples, onde cada palavra é dividida em duas sub-palavras. Nesse particionamento, as duas metades têm três padrões únicos cada, sendo necessário dois *bits* para indexar cada dicionário. Portanto, a μROM comprimida gasta $10 \times (2 + 2) + 3 \times 3 + 3 \times 3 = 58$ *bits*. Esse valor equivale a uma redução menor que 4% do tamanho original (60 *bits*).

Col 1	Col 2	Col 3	Col 4	Col 5	Col 6
1	0	1	0	1	0
0	1	0	1	0	1
1	0	1	0	1	0
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	1	0
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	1	0
0	1	0	1	0	1

Figura 3.3: Método de partição simples. O primeiro conjunto contém as colunas de 1 a 3 e o segundo conjunto, as colunas de 4 a 6.

A mesma μROM da Figura 3.3 pode ter suas colunas reorganizadas em dois conjuntos, como mostrado na Figura 3.4. Observe que as colunas foram agrupadas sem a restrição de serem seqüenciais. Com a nova organização, ambos os conjuntos possuem apenas dois padrões únicos e necessitam apenas um *bit* para indexá-los. Como resultado, a μROM comprimida gasta $10 \times (1 + 1) + 2 \times 3 + 2 \times 3 = 32$ *bits*, ou seja, próximo a 50% de redução. O que é uma

melhora significativa quando comparado com o método de partição simples[1].

Col 1	Col 3	Col 5	Col 2	Col 4	Col 6
1	1	1	0	0	0
0	0	0	1	1	1
1	1	1	0	0	0
0	0	0	0	0	0
0	0	0	1	1	1
1	1	1	0	0	0
0	0	0	0	0	0
0	0	0	1	1	1
1	1	1	0	0	0
0	0	0	1	1	1

Figura 3.4: Método de agrupamento de colunas. O primeiro conjunto contém as colunas 1, 3 e 5 e o segundo conjunto, as colunas 2, 4 e 6.

A Figura 3.5 mostra como o microcódigo é acessado no método de agrupamento de colunas. Nesse caso, há um novo componente chamado "espalhador", que reorganiza os *bits* dos padrões únicos na disposição original das colunas. O "espalhador" é apenas uma reorganização na conexão dos fios que ligam os dicionários à via de dados e não deve aumentar a área ou o consumo de energia do *chip*.

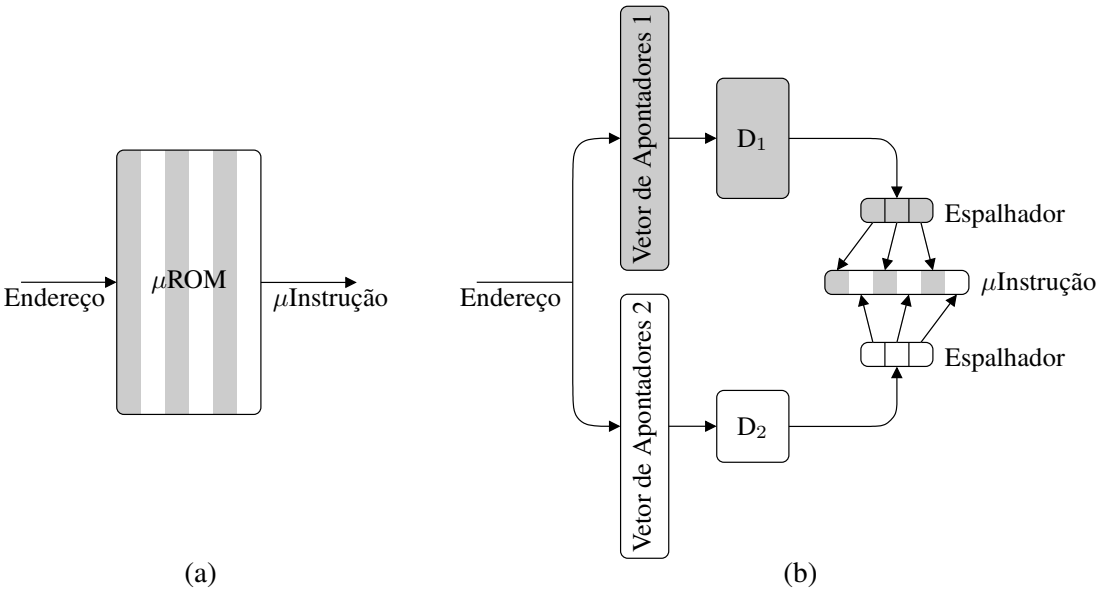


Figura 3.5: Compressão baseada no agrupamento de colunas.

[1]O objetivo deste exemplo é essencialmente didático. Existe uma solução melhor para este exemplo, dado que as colunas 1, 3 e 5, assim como as colunas 2, 4 e 6, são iguais.

3.3.1 Colunas Não-Comprimidas

Como vimos na Seção 3.2, quando o número de padrões é grande, o esquema de compressão pode aumentar o tamanho do microcódigo. Essa mesma análise é válida para os conjuntos de colunas. Entretanto, no caso da compressão com agrupamento de colunas, é possível que apenas parte dos conjuntos ocupe mais *bits* no formato comprimido. Dessa forma, comprimimos apenas os conjuntos que apresentam redução no tamanho.

Os conjuntos não comprimidos podem ser armazenados em memórias com tamanho $L_i \times N$ *bits*. Da mesma maneira que nos dicionários, os *bits* dos conjuntos não comprimidos são reposicionados para compor a microinstrução original. A Figura 3.6 mostra o esquema de compressão com dois conjuntos comprimidos e um conjunto com colunas não-comprimidas.

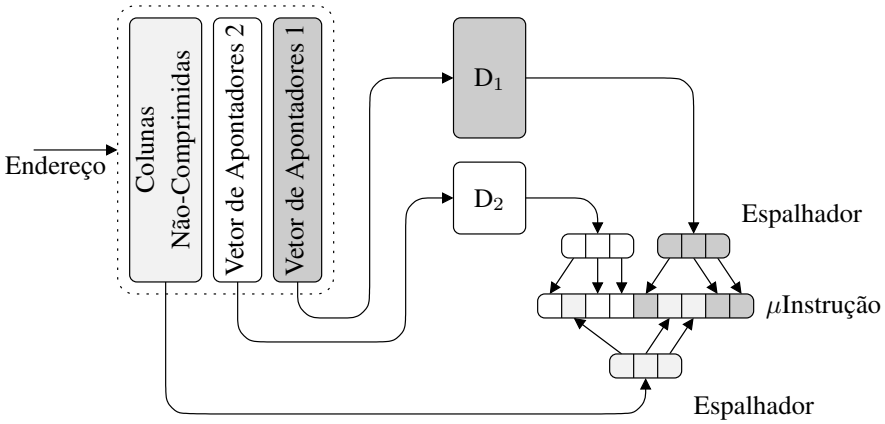


Figura 3.6: Esquema de compressão com parte das colunas não-comprimidas.

Observe que os vetores de apontadores e as colunas não-comprimidas podem ser agrupados em um mesmo bloco de memória, pois utilizam o mesmo endereço de entrada e possuem o mesmo número de linhas (N). Desse modo, o esquema de compressão utiliza um único bloco de memória para armazenar os vetores de apontadores e as colunas não-comprimidas mais um bloco de memória para cada dicionário.

3.3.2 Conversão de Endereços

A conversão dos endereços das instruções originais para os endereços das instruções comprimidas é um problema comum na área de compressão de código, entretanto, a técnica de compressão descrita aqui não requer esse tipo de tradução. Observe que o vetor de apontadores nas Figuras 3.1 (b), 3.2 (b), 3.5 (b) e 3.6 tem o mesmo número de linhas, ou microinstruções, que a μROM original. A diferença está apenas no número de *bits* por microinstrução, ou seja, na largura da μROM. A flexibilidade no projeto da μROM permite modificá-la de forma a reduzir

o tamanho das palavras, portanto, os endereços das instruções no microcódigo comprimido e original são iguais e a tradução de endereços não é necessária.

3.4 Análise de Desempenho

O esquema de compressão do microcódigo apresentado neste capítulo envolve acessos indiretos às memórias dos dicionários. Apesar dos dicionários serem acessados em paralelo, os endereços utilizados para indexá-los são carregados dos vetores de apontadores. Essa indireção aumenta o tempo de carga da microinstrução e pode causar problemas no projeto do microprocessador.

Na Seção 1.2 discutimos os problemas do atraso da propagação do sinal em processadores de alto desempenho e observamos que registradores de *pipeline* podem ser utilizados para amenizar esses problemas. De fato, se a latência do descompressor for maior do que a duração do ciclo do relógio do processador, podemos dividir o esquema de compressão em dois estágios de *pipeline*. A Figura 3.7 mostra o esquema de compressão dividido em dois estágios de *pipeline*.

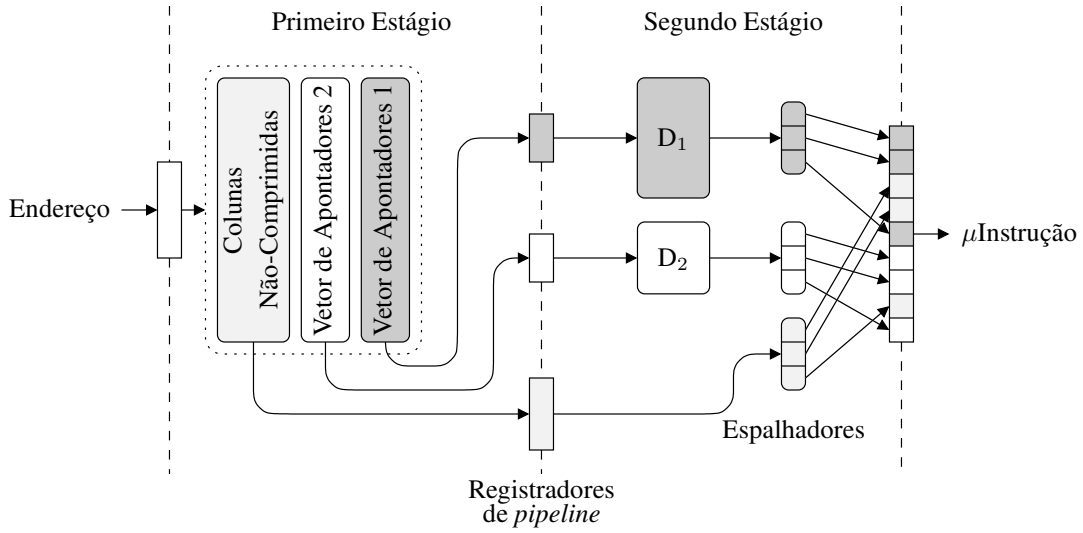


Figura 3.7: Esquema de compressão com *pipeline*.

O primeiro estágio do *pipeline* contém o bloco de memória com os vetores de apontadores e as colunas não-comprimidas. Como esse bloco de memória é menor do que a μROM original, a sua latência não ultrapassa a duração do ciclo do relógio do processador e a freqüência alvo do relógio do processador não é afetada. A mesma análise pode ser feita para o segundo estágio do *pipeline*. Nesse caso, os padrões armazenados nos dicionários são carregados em paralelo e o tempo para se completar a tarefa é determinado pela latência do maior dicionário. Novamente, o maior dos dicionários é menor do que a μROM original e, conseqüentemente, a sua latência é menor do que a duração do ciclo.

A introdução dos registradores de *pipeline* entre os vetores de apontadores e os dicionários aumenta em um ciclo o tempo necessário para se carregar a primeira microinstrução. No modelo original, sem compressão, cada microinstrução é carregada em um único ciclo e uma seqüência com d microinstruções pode ser carregada em d ciclos. Na compressão com *pipeline*, por outro lado, a primeira microinstrução é carregada após dois ciclos e as microinstruções subseqüentes tomam apenas um ciclo cada. Portanto, a mesma seqüência de microinstruções é carregada em $d + 1$ ciclos. Esse ciclo extra, consumido pela primeira microinstrução, também é conhecido como "bolha" no *pipeline* [108].

A geração do endereço da próxima microinstrução pode depender do conteúdo da microinstrução lida. Nesse caso, existiria uma dependência de dados entre o segundo e o primeiro estágios do *pipeline*. Essa dependência aumentaria em um ciclo o tempo de leitura de todas as microinstruções, e não somente o da primeira. Uma forma de resolver esse problema é armazenar o endereço da próxima microinstrução no vetor de apontadores, como colunas não-comprimidas, de forma que o seqüenciamento das microinstruções dependa apenas de dados no primeiro estágio do *pipeline*.

A degradação de desempenho do processador, causada pela "bolha" no *pipeline*, depende do tamanho das seqüências de microinstruções. Quanto maior for o tamanho da seqüência de microinstruções, menor será a degradação do desempenho. Por exemplo, para uma seqüência com 2 microinstruções, o aumento no tempo de execução é de $(2 + 1) \div 2 = 1, 5$ vezes, ou seja, a redução do desempenho é de 50%, por outro lado, se a seqüência possui 10 microinstruções, o aumento é de $(10 + 1) \div 10 = 1, 1$ vezes, causando uma perda de desempenho de 10%.

Outro fator que influencia na degradação do desempenho é a freqüência com que o microcódigo é utilizado. Processadores modernos utilizam decodificadores rápidos, implementados com lógica aleatória, para decodificar as instruções mais freqüentes e relegam para o microcódigo a decodificação de instruções complexas, geralmente implementadas para manter a compatibilidade com código legado. Por exemplo, o guia de otimização de código para processadores da Intel [66] recomenda a substituição de instruções complexas, traduzidas em mais de quatro microinstruções, por seqüências de instruções simples, de forma que não seja necessária a execução do microcódigo. Portanto, se uma aplicação despende pouco tempo, digamos 1%, executando instruções que necessitam do microcódigo, a degradação do desempenho causada pelo ciclo extra é ainda menor. De fato, testes com um processador de alto desempenho, realizados por engenheiros da Intel [24], indicam que a adição do ciclo extra causa uma degradação de desempenho média em torno de 0,4%. Assim sendo, o esquema de compressão com *pipeline* se adequa às restrições de desempenho impostas em projetos de processadores modernos.

3.5 Redução de *Bits* versus Redução da Área

Os exemplos de compressão de microcódigo apresentados neste capítulo mostram reduções no número de *bits* do microcódigo. Entretanto, a redução da área da μROM nem sempre é proporcional à redução no número de *bits* do microcódigo. Isso se dá devido à presença de circuitos periféricos, anexos ao núcleo da ROM, que não diminuem proporcionalmente com o tamanho da ROM. A lógica de controle, os decodificadores de endereço e os *drivers* e amplificadores são exemplos desses circuitos [110]. A Figura 3.8 ilustra a organização de uma ROM.

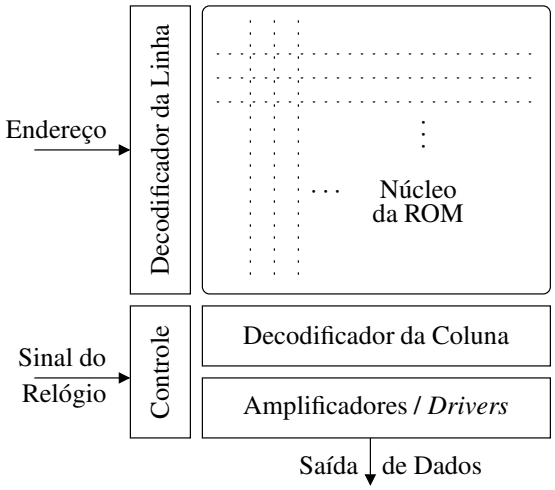


Figura 3.8: Organização de uma ROM.

Outro fator que pode influenciar na área da μROM é a regularidade e os formatos retangulares dos blocos de memória. Por exemplo, suponha blocos de memória com oito *bits* de largura, ou seja, oito colunas. Para armazenar um microcódigo com 15 colunas são necessários dois blocos de memória e a redução da quantidade de blocos só é possível se removermos pelo menos sete das 15 colunas do microcódigo. Portanto, a redução do número de *bits* do microcódigo não é necessariamente proporcional à redução da área ocupada pela μROM.

Projetistas de circuito da Intel utilizaram o esquema de compressão básico e particionado para comprimir um microcódigo com $75 \times 22\,528$ *bits* [25]. A Tabela 3.1 mostra os resultados da compressão comparando a redução do número de *bits* e a redução da área. Observe que, na compressão básica, a redução da área é melhor do que a redução do número de *bits*. Por outro lado, a redução da área na compressão particionada é pior do que a redução de *bits*. Essa diferença é normal e ocorre devido aos fatores mencionados anteriormente.

A regularidade, o formato e os circuitos periféricos que compõem a ROM são determinados em parte pela tecnologia de fabricação do circuito. Assim sendo, a redução da área da μROM depende diretamente da tecnologia de fabricação do circuito. Entretanto, assim como na Tabela 3.1, resultados anteriores [136] indicam que a redução do número de *bits* através

Tabela 3.1: Redução de *bits* versus redução da área.

Organização	Bloco de Memória	Tamanho em		Razão de compressão	
		Bits	Unidades de Área	*Bits*	Área
Original	μROM	1 689 600	$711 \times 800 = 568\,800$	100%	100%
Compressão Básica	Vet. Apont.	315 392	$484 \times 214 = 103\,576$	79,66%	71,1%
	Dicionário	1 030 425	$291 \times 1034 = 300\,894$		
Compressão Particionada	Vet. Apont.	563 200	$484 \times 349 = 168\,916$	57%	68,4%
	Dic. D_1	127 642	$155 \times 286 = 44\,330$		
	Dic. D_2	272 172	$277 \times 634 = 175\,618$		

desse esquema de compressão também reduz a área total da μROM. Dessa forma, utilizaremos a redução do número de *bits* como métrica para maximizar a compressão do microcódigo.

3.6 Conclusões

Neste capítulo apresentamos três esquemas de compressão baseado na organização do microcódigo em dois níveis: a compressão básica, a compressão com particionamento e a compressão com agrupamento de colunas. Esses esquemas são implementados com blocos de memória ROM interconectados entre si e se destacam pela simplicidade na organização e eficiência na compressão do microcódigo [25].

A descompressão do microcódigo envolve acessos indiretos às ROMs, o que pode aumentar o tempo de propagação do sinal. Entretanto, observamos que é possível utilizar registradores de *pipeline* para particionar o descompressor em dois estágios e diminuir o atraso da propagação do sinal. Também vimos que o ciclo extra, introduzido pelo preenchimento do *pipeline*, causa pouco impacto no desempenho do processador.

Por fim, analisamos a diferença entre redução do número de *bits* do microcódigo e redução da área da μROM. A redução da área da ROM depende, em parte, de características relacionadas à tecnologia de fabricação do circuito e nem sempre é proporcional à redução da área da ROM. Entretanto, observamos que a redução do número de *bits* do microcódigo geralmente leva à redução da área da μROM [25, 136] e pode ser utilizada como métrica para maximizar a compressão do microcódigo.

O esquema de compressão com agrupamento permite que colunas similares, não adjacentes, sejam agrupadas em um mesmo conjunto. O agrupamento de colunas similares reduz o número de padrões nos conjuntos e melhora a razão de compressão do microcódigo. No próximo capítulo propomos algoritmos para identificar e agrupar colunas similares do microcódigo.

Capítulo 4

Algoritmos para Agrupamento de Colunas

A técnica de compressão de microcódigo introduzida no Capítulo 3 permite o agrupamento de colunas não adjacentes. Entretanto, para tirarmos proveito dessa funcionalidade precisamos identificar as colunas similares do microcódigo. Este capítulo propõe novos algoritmos para identificar e agrupar essas colunas.

O agrupamento de colunas pode ser influenciado por restrições impostas pela tecnologia utilizada para implementar a μROM e pela organização interna do microprocessador. Por exemplo, se a tecnologia de fabricação só permite blocos de ROM com 40 *bits* de largura, é interessante gerarmos dicionários com 40 colunas pois, do contrário, estaríamos desperdiçando área com *bits* não utilizados.

Restrições como o número de dicionários, a largura ou mesmo a altura destes dividem o problema de agrupamento de colunas em diversas categorias. A Tabela 4.1 identifica as principais categorias do problema em função dessas restrições.

Tabela 4.1: Categorias do problema de agrupamento de colunas.

Categoria	Número de Dicionários	Número de Colunas por Dicionário	Número de Linhas por Dicionário
1	Livre	Livre	Livre
2	Livre	Restrito	Livre
3	Livre	Livre	Restrito
4	Livre	Restrito	Restrito
5	Restrito	Livre	Livre
6	Restrito	Restrito	Livre
7	Restrito	Livre	Restrito
8	Restrito	Restrito	Restrito

Observe que a primeira categoria da Tabela 4.1 não impõe restrições no número ou tamanho dos dicionários, portanto, o algoritmo para agrupar colunas pode explorar essa flexibilidade para

maximizar a compressão do microcódigo. A solução ótima do agrupamento nessa categoria determina o limitante inferior da razão de compressão do microcódigo. As outras categorias apresentam algum tipo de restrição, seja no número de dicionários, na quantidade de colunas ou de linhas por dicionário.

A Seção 4.1 descreve a terminologia utilizada. A Seção 4.2 propõe quatro algoritmos novos para realizar o agrupamento de colunas sem restrições. Por fim, a Seção 4.3 descreve o algoritmo proposto para agrupar colunas com restrições.

4.1 Terminologia

Os algoritmos apresentados neste capítulo agrupam as colunas de forma a minimizar o custo da compressão. Antes de definir o custo da compressão, introduzimos os seguintes termos:

- L: o número de colunas na μROM, ou o número de *bits* em cada microinstrução.
- N: o número de *bits* em cada coluna, ou o número de microinstruções.
- K: o número de conjuntos em que as L colunas serão agrupadas.
- $L_1, L_2, \cdots, L_K$: o número de colunas em cada conjunto $C_1, C_2, \cdots, C_K$.
- $M_1, M_2, \cdots, M_K$: o número de padrões únicos em cada conjunto $C_1, C_2, \cdots, C_K$.
- $c_1, c_2, \cdots, c_L$: as colunas do microcódigo.

O custo da compressão é definido pela função F na equação:

$$F = \sum_{i=1}^{K} \; [\; \underbrace{N \times \lceil log_2 M_i \rceil}_{Vetor\ de\ Apontadores_i} \; + \; \overbrace{M_i \times L_i}^{Dicionário_i} \;] \tag{4.1}$$

Para o exemplo da Figura 3.4, onde $N = 10$, $K = 2$, $L_1 = 3$, $L_2 = 3$, $M_1 = 2$ e $M_2 = 2$, $F = 10 \times \lceil log_2 2 \rceil + 3 \times 2 + 10 \times \lceil log_2 2 \rceil + 3 \times 2 = 32$. Este número corresponde ao tamanho em *bits* do microcódigo comprimido.

De acordo com a Equação 4.1, uma forma de se minimizar o custo da compressão é reduzir o número de padrões em cada dicionário (M_i). A técnica de compressão em dois níveis permite o agrupamento de colunas não adjacentes para reduzir o número de padrões nos dicionários. Neste texto, utilizamos o conceito de similaridade para denotar colunas que, quando agrupadas, produzam poucos padrões de *bits*. Assim sendo, uma forma de se minimizar o custo da compressão é através do agrupamento de colunas similares do microcódigo.

4.2 Agrupamento de Colunas sem Restrições

A ausência de restrições permite que o algoritmo de agrupamento de colunas tenha mais flexibilidade para maximizar a compressão do microcódigo. Entretanto, a quantidade de combinações possíveis pode ser muito grande. Por exemplo, seja A um microcódigo com três colunas (c_1, c_2 e c_3), os possíveis agrupamentos para estas colunas são:

1. $\{\{c_1, c_2, c_3\}\}$
2. $\{\{c_1\}, \{c_2, c_3\}\}$
3. $\{\{c_2\}, \{c_1, c_3\}\}$
4. $\{\{c_3\}, \{c_1, c_2\}\}$
5. $\{\{c_1\}, \{c_2\}, \{c_3\}\}$

O primeiro agrupamento possui um único conjunto com três colunas. Os próximos três agrupamentos possuem dois conjuntos e o último agrupamento possui três conjuntos, um para cada coluna.

A quantidade de possíveis agrupamentos, ou partições, de um conjunto com n elementos é definido pelo n-ésimo número de Bell [115], ou B_n. Este número satisfaz a seguinte recursão:

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k \tag{4.2}$$

Apesar do número de agrupamentos do exemplo anterior ser pequeno, apenas cinco, a quantidade de partições de um conjunto com n elementos cresce muito rápido em função de n. Mesmo se o número de conjuntos fosse fixado em dois, a quantidade de possíveis agrupamentos com n elementos seria $2^{n-1} - 1$, ou seja, a quantidade de partições é no mínimo exponencial em n.

Intuitivamente, o agrupamento de colunas sem restrições é um problema de otimização $\mathcal{NP}$-Difícil [50]. As próximas seções descrevem as heurísticas propostas para agrupar a colunas do microcódigo.

4.2.1 Colunas Seqüenciais

Como mencionamos anteriormente, o problema de agrupamento de colunas parece ser um problema $\mathcal{NP}$-Difícil e o número de possibilidades para agrupar as colunas é muito grande. Portanto, propusemos uma nova heurística dividindo o problema em duas partes:

- Criar conjuntos de colunas e atribuir a cada conjunto C_i um benefício B_i, onde B_i é o número de *bits* economizados se o conjunto C_i for selecionado. B_i é definido pela seguinte equação:

$$B_i = \underbrace{L_i \times N}_{Tam.\ Original} - (\underbrace{\lceil log_2 M_i \rceil \times N + M_i \times L_i}_{Tam.\ Comprimido}) \tag{4.3}$$

- Selecionar os melhores conjuntos disjuntos para maximizar o benefício total ($\sum_{i=1}^{K} B_i$).

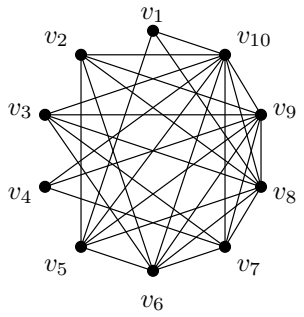
Dado que enumerar todos os possíveis conjuntos é uma tarefa impraticável, consideramos apenas os conjuntos formados por colunas consecutivas, ou seja, adjacentes. Por exemplo, os conjuntos $\{c_2, c_3, c_4\}$ e $\{c_8, c_9\}$ são conjuntos formados por colunas consecutivas. O número total de conjuntos distintos com colunas consecutivas é:

$$Q = \sum_{i=1}^{L} i = \frac{L^2 + L}{2} \tag{4.4}$$

Por exemplo, para um microcódigo com 75 colunas geramos 2850 conjuntos de colunas. Para selecionar os conjuntos que proporcionam a melhor razão de compressão, interpretamos os conjuntos como vértices de um grafo e inserimos arestas entre os vértices dos conjuntos que contêm colunas em comum. Atribuímos a cada vértice o benefício do conjunto e buscamos pelo conjunto independente de peso máximo no grafo. A Figura 4.1 (b) mostra um grafo representando os conjuntos gerados a partir de um microcódigo com 4 colunas.

$C_1 = \{c_1\}$ $C_5 = \{c_1, c_2\}$ $C_8 = \{c_1, c_2, c_3\}$
$C_2 = \{c_2\}$ $C_6 = \{c_2, c_3\}$ $C_9 = \{c_2, c_3, c_4\}$
$C_3 = \{c_3\}$ $C_7 = \{c_3, c_4\}$ $C_{10} = \{c_1, c_2, c_3, c_4\}$
$C_4 = \{c_4\}$

(a)



(b)

Figura 4.1: Grafo G (b) construído a partir de conjuntos de colunas (a). Cada vértice v_i representa um conjunto C_i. Os conjuntos C_i com colunas (c_j) em comum possuem arestas entre seus respectivos vértices (v_i) no grafo G.

Buscar pelo conjunto independente de peso máximo em grafos arbitrários é um problema $\mathcal{NP}$-Difícil [50]. Entretanto, como os conjuntos possuem colunas seqüenciais, podemos reduzir o problema para grafos intervalares e, conseqüentemente, resolver o problema em tempo

polinomial. Dessa forma, podemos encontrar a solução ótima quando consideramos apenas conjuntos com colunas seqüenciais.

No grafo intervalar, cada conjunto, ou vértice, é representado com três números: a coluna inicial, a última coluna e o benefício B_i do conjunto. Assim, o conjunto C_i é visto como um "intervalo" I_i com benefício B_i. A Figura 4.2 mostra os conjuntos de colunas da Figura 4.1 representados como intervalos.

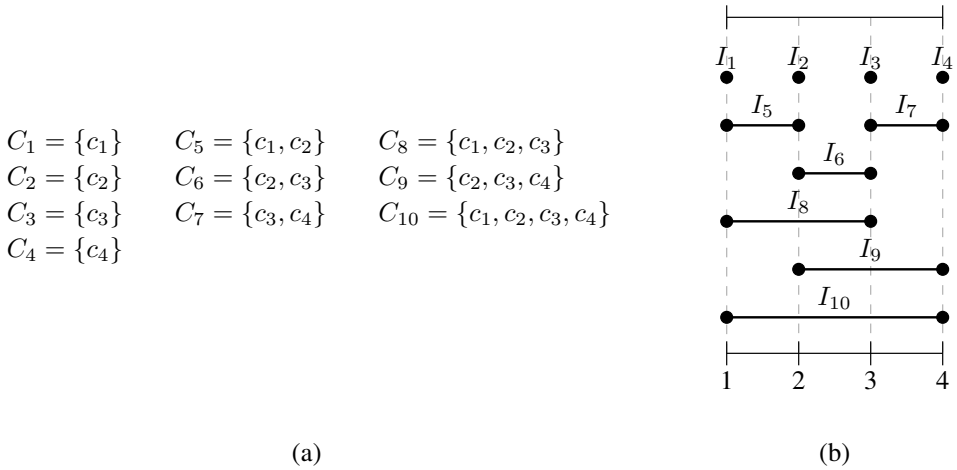


Figura 4.2: (a) Conjuntos de colunas seqüenciais (b) representados como intervalos. Cada conjunto C_i é representado por um intervalo I_i.

Seleção de Intervalos

O problema de seleção de conjuntos se reduz à seleção de intervalos disjuntos que proporcionem o melhor benefício acumulado.

Definimos $PIV(i)$ como o primeiro intervalo válido após a coluna c_i. Por exemplo, na Figura 4.2, $PIV(2) = I_3$[1] pois I_1, I_2, I_5, I_6, I_8, I_9 e I_{10} possuem colunas com índices menores ou igual a 2. Também definimos Ic_1, Ic_2, $\cdots$, Ic_n como os intervalos que conflitam[2] com o primeiro intervalo válido ($PIV(i)$) e contenham apenas colunas com índice maior que c_i. Por exemplo, para $i = 2$, $PIV(2) = I_3$ e $Ic_1 = I_7$, pois I_7 conflita com I_3 e não possui colunas com índices menores ou iguais a 2.

Definimos a solução parcial do problema $SP(i)$ como sendo o melhor benefício acumulado proporcionado por intervalos disjuntos que não possuam colunas com índices menores ou igual a i. Por exemplo, na Figura 4.2, $SP(1)$ não pode incluir os intervalos I_1, I_5, I_8 e I_{10}, pois estes possuem colunas com índices menores ou iguais a 1. A partir dessa definição, observamos

[1] O primeiro intervalo válido após a coluna c_2 pode ser I_3 ou I_7. No exemplo utilizamos I_3.

[2] Dois intervalos conflitam se ambos possuírem colunas em comum.

que a solução ótima para o problema de seleção de intervalos pode ser obtida com o cálculo da $SP(0)$.

Seja $I_i.fim$ o índice da última coluna do intervalo I_i, definimos $SPI(I_i)$ como sendo a solução parcial $SP(I_i.fim)$ adicionada ao benefício do intervalo I_i (B_i). Ou seja:

$$SPI(I_i) = B_i + SP(I_i.fim) \tag{4.5}$$

Por exemplo, na Figura 4.2, $SPI(I_5) = SP(I_5.fim) + B_5$, ou seja, $SPI(I_5) = SP(2) + B_5$. Assim sendo, a solução parcial ótima $SP(c_i)$ é computada pela seguinte equação:

$$SP(i) = \begin{cases} Máximo(\ SPI(PIV(i)), \\ \qquad SPI(Ic_1), \\ \qquad SPI(Ic_2), & \text{para } PIV(i) \neq \emptyset \\ \qquad \cdots, \\ \qquad SPI(Ic_n)) \\ \qquad 0 & \text{para } PIV(i) = \emptyset \end{cases} \tag{4.6}$$

A Equação 4.6 mostra que o cálculo da solução parcial ótima ($SP(i)$) envolve apenas o primeiro intervalo válido ($PIV(i)$) e os intervalos conflitantes com este intervalo (Ic_1, Ic_2, $\cdots$, Ic_n). De fato, para um intervalo I_j, que não conflita com $PIV(i)$, a solução $SPI(I_j)$ pode ser melhorada com a adição de $PIV(i)$ [3] e, neste caso, pela definição de $SPI(I_i)$ essa solução será pior ou igual a $SPI(PIV(i))$.

Sempre que a $SP(i)$ é calculada para um determinado índice i, o algoritmo atualiza um vetor com o resultado de forma que não seja necessário recalcular $SP(i)$. Observe que, dessa forma, $SP(i)$ será computada no máximo uma vez para cada coluna. Sempre que $SP(i)$ é computada, o algoritmo busca pelo primeiro intervalo válido após a coluna c_i ($PIV(i)$) e pelos intervalos conflitantes (Ic_1, Ic_2, $\cdots$, Ic_n). Os intervalos podem ser ordenados pelos índices das colunas final e inicial de forma que a busca possa ser feita em $O(logQ)$ (onde Q é o número de intervalos). Portanto, o tempo de execução do algoritmo de seleção de conjuntos é $O(Q \times logQ)$. Como $Q \leq L^2$, a complexidade do algoritmo é $O(L^2 \times logL)$.

Após a seleção dos intervalos, a compressão é realizada criando-se um dicionário para cada conjunto de colunas. As colunas que não estão presentes em nenhum dos intervalos selecionados são posicionadas diretamente no vetor de apontadores (Seção 3.3.1).

Como o algoritmo de colunas seqüenciais considera apenas conjuntos formados por colunas consecutivas só é possível agrupar colunas similares que estão afastadas se as colunas posicionadas entre essas também forem incluídas no mesmo conjunto. Entretanto, podemos reordenar as colunas do microcódigo de forma a posicionar as colunas similares próximas umas das outras

[3] Isso é verdade por que os conjuntos com benefícios negativos são removidos antes do passo de seleção de conjuntos.

e então utilizar o algoritmo de colunas seqüenciais para agrupar estas colunas em conjuntos. As próximas seções descrevem duas heurísticas para reorganizar as colunas do microcódigo antes de agrupar a colunas com o algoritmo de colunas seqüenciais.

4.2.2 Ordenação Linear e Colunas Seqüenciais

A ordenação linear é uma heurística que visa reorganizar as colunas para melhorar o resultado do algoritmo de colunas seqüenciais. A heurística não encontra qualquer agrupamento de colunas, mas apenas reorganiza o microcódigo de forma a colocar as colunas similares próximas umas das outras, assim, o algoritmo de colunas seqüenciais pode agrupar colunas similares que estavam distantes anteriormente.

Para determinar a nova ordem das colunas, o algoritmo possui um vetor de ordenação e uma lista de colunas (LC). A lista é inicializada com uma coluna e, a cada iteração, uma nova coluna, não pertencente à lista, é selecionada e incluída na lista. Durante a execução do algoritmo, o vetor de ordenação é preenchido com as colunas na ordem em que são adicionadas à LC. Quando todas as colunas forem adicionadas à lista, ou seja, quando o vetor de ordenação estiver preenchido, a execução do algoritmo termina. A ordem das colunas no vetor de ordenação determina a nova ordem das colunas do microcódigo. O Algoritmo 4.1 mostra a heurística de ordenação linear.

Algoritmo 4.1 Ordenação Linear

```
1:  LC ← {c_1} /* Inicie a lista de colunas com uma coluna arbitrária. Ex: c_1 */
2:  vetor_ordenação[1] ← c_1
3:  para i ← 2 até L faça
4:     para cada coluna c_j ∉ LC faça
5:        B_j ← Benefício(LC ∪ {c_j})
6:        se B_j > Melhor_Benefício então
7:           Melhor_Benefício ← B_j
8:           Melhor_Coluna ← c_j
9:        fim se
10:    fim para
11:    LC ← LC ∪ {Melhor_Coluna}
12:    vetor_ordenação[i] ← Melhor_Coluna
13: fim para
14: Reorganize o microcódigo de acordo com a ordem das colunas em vetor_ordenação[ ]
```

A quinta linha do Algoritmo 4.1 computa o benefício proporcionado pelo conjunto de colunas $LC \cup \{c_i\}$. O benefício é definido como sendo o número de *bits* economizados com a compressão do conjunto e é determinado a partir da Equação 4.3. O benefício é utilizado como métrica para avaliar a "similaridade" entre a coluna c_i e as colunas em LC. Após computar

o benefício de todas as colunas, o algoritmo insere a melhor coluna na LC e executa a próxima iteração. A intuição por trás da heurística é agrupar as colunas similares no início do microcódigo.

A complexidade do algoritmo é $O(L^2 \times NlogN)$, onde $NlogN$ é a complexidade da operação na linha 5 do Algoritmo 4.1. O tempo de execução do algoritmo é pequeno em microcódigos com 70 a 400 colunas. Assim sendo, para explorar a heurística, executamos a ordenação linear iniciando a LC com cada uma das L colunas do microcódigo. Dessa forma, executamos o algoritmo L vezes. A Figura 4.3 mostra as razões de compressão para um microcódigo com 75 colunas quando a LC é iniciada com cada uma das colunas.

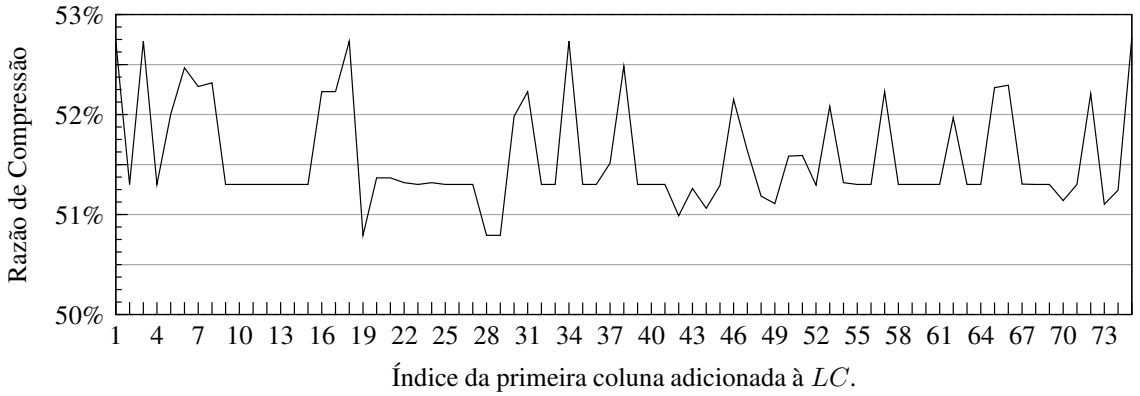


Figura 4.3: Razões de compressão para o microcódigo com as colunas reorganizadas. O gráfico mostra a razão de compressão final quando a LC é inicializada com cada uma das colunas do microcódigo.

Observe que há uma variação de quase 2% na razão de compressão quando iniciamos a lista de colunas do algoritmo com colunas diferentes. Essa variação ocorre porque a heurística é sensível à coluna selecionada para iniciar a LC.

Apesar da variação da razão de compressão, a maioria dos resultados obtidos geraram apenas dois conjuntos, com tamanhos em torno de 48 e 20 colunas, respectivamente. O maior desses conjuntos contém as colunas posicionadas no início do microcódigo. Isso acontece porque a heurística de ordenação linear tende a agrupar as colunas mais semelhantes no início do microcódigo. A próxima seção apresenta uma heurística de ordenação que visa agrupar de maneira mais uniforme as colunas similares.

4.2.3 Ordenação Circular e Colunas Seqüenciais

A heurística de ordenação circular é baseada na ordenação linear. A principal diferença está no fato de que na ordenação circular o vetor de ordenação é previamente inicializado com todas as colunas e a lista de colunas (LC) se move sobre este vetor, de forma circular, reordenando as

colunas.

Inicialmente, a lista LC cresce até um tamanho limite W. Nessa fase, a LC cresce da mesma forma que no algoritmo de ordenação linear. Após atingir o tamanho máximo (W), para cada coluna nova adicionada à LC, uma coluna antiga, adicionada a mais tempo, é removida. Assim, a lista se desloca sobre o vetor de ordenação reordenando as colunas. A Figura 4.4 ilustra como a lista cresce e se move sobre o vetor de ordenação quando $W = 3$. A Figura 4.4 (a) mostra o estado inicial do algoritmo, quando a LC possui apenas uma coluna (a região sombreada mostra as colunas pertencentes à LC). A Figura 4.4 (b) mostra a LC e o vetor de ordenação quando a lista LC atinge o tamanho máximo (W). Observe que o vetor de ordenação já foi reorganizado de acordo com a ordem de escolha das colunas na fase de crescimento da LC. A Figura 4.4 (c) mostra a lista LC se movendo sobre o vetor de ordenação. Nesse caso, a coluna c_1 foi removida da cauda da LC para que c_2 pudesse ser inserida na cabeça sem violar o tamanho máximo da LC. Na Figura 4.4 (d) a lista LC salta do final do vetor de ordenação para o início de forma circular.

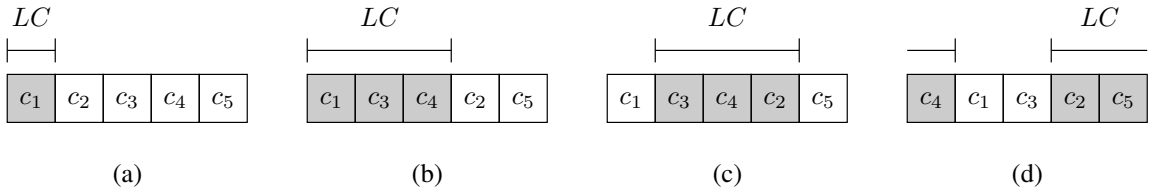


Figura 4.4: Movimento da lista sobre o vetor de ordenação para $W = 3$. (a) A lista crescendo ($|LC| < W$). (b) A lista atinge o tamanho máximo ($|LC| = W$). (c) A lista de colunas se deslocando sobre o vetor. (d) Quando a lista chega ao final do vetor, ela continua se deslocando para o início do vetor.

Como a lista se move de forma circular, limitamos a execução do algoritmo a um número máximo de iterações, ou ciclos, sobre o vetor de ordenação. Toda vez que a lista ultrapassa o ponto inicial, o algoritmo armazena a ordenação de colunas, que corresponde ao próprio vetor. Desta forma, a cada iteração o algoritmo produz uma nova ordenação de colunas. O Algoritmo 4.2 descreve a heurística de ordenação circular.

A lista de colunas LC é implementada com um par de ponteiros para o vetor de ordenação. O primeiro ponteiro ($LC_{Cabeça}$) aponta para a cabeça da lista e o segundo (LC_{Cauda}) para a cauda. A segunda linha do Algoritmo 4.2 inicializa os apontadores da lista. Nesse caso, a lista possui uma única coluna (Ex: Figura 4.4 (a)). A função $Cresce\_Lista(LC, W)$, na terceira linha do algoritmo, cresce a lista LC, incrementando $LC_{Cabeça}$ e reordenando as colunas, até que a LC atinja o tamanho máximo W. Após crescer a lista, o algoritmo executa um número máximo de iterações (linhas 4 a 11) e a cada iteração a ordem das colunas é alterada (linhas 5 a 10) e armazenada (linha 11).

Em nossos experimentos constatamos que valores pequenos de W produzem ordenações

Algoritmo 4.2 Ordenação Circular

1: *vetor_ordenação*$[...] \leftarrow \{c_1, c_2, c_3, \cdots, c_L\}$
2: $LC_{Cabeça} \leftarrow LC_{Cauda} \leftarrow 1$
3: $Cresce\_Lista(LC, W)$ /* Cresça a lista até possuir W colunas */
4: **para** cada iteração **faça**
5: **para** $i \leftarrow 1$ até L (número de colunas) **faça**
6: Escolha a melhor coluna $c_i \notin LC$ para agrupar com as colunas em LC
7: /* Desloque a lista de colunas */
8: $LC_{Cabeça} \leftarrow (LC_{Cabeça} + 1)$ resto L
9: $LC_{Cauda} \leftarrow (LC_{Cauda} + 1)$ resto L
10: Troque a cabeça da LC pela melhor coluna
11: **fim para**
12: Armazene a ordem das colunas
13: **fim para**

piores. Esse resultado indica que a qualidade da decisão da melhor coluna está associada ao número de colunas na LC.

Apesar da ordenação circular apresentar ligeira melhora sobre os resultados obtidos pela ordenação linear, a melhora não é significativa e o número de variáveis para explorar na ordenação circular é muito grande.

4.2.4 AKL - Agrupamento Baseado em Kernighan e Lin

Heurística de Kernighan e Lin

A heurística de Kernighan e Lin [71] foi originalmente proposta para resolver um problema $\mathcal{NP}$-Difícil de agrupamento de vértices em grafos. Nesse problema de agrupamento, as arestas do grafo são anotadas com valores que denotam o custo de comunicação entre os pares de vértices e o custo total de comunicação é a soma dos custos das arestas que possuem vértices em conjuntos diferentes. Em outras palavras, vértices em um mesmo conjunto possuem custo de comunicação zero. Além de reduzir o custo total de comunicação, o algoritmo deve respeitar restrições de tamanho nos conjuntos. Note que se o conjunto não tem restrições de tamanho, basta agruparmos todos os vértices em um mesmo conjunto, e o custo total será nulo.

Para exemplificar a heurística, vamos supor um grafo com seis vértices. O objetivo é agrupar os vértices em dois conjuntos com três vértices cada de forma a minimizar o custo de comunicação entre os conjuntos. A partir de um agrupamento inicial a heurística tenta trocar vértices dos dois conjuntos para minimizar o custo de comunicação entre os conjuntos. A Figura 4.5 (a) mostra um agrupamento com custo de comunicação igual a 50. A Figura 4.5 (b) mostra o novo agrupamento após a troca dos vértices v_1 e v_2. Observe que o custo de comunicação foi reduzido

para 10.

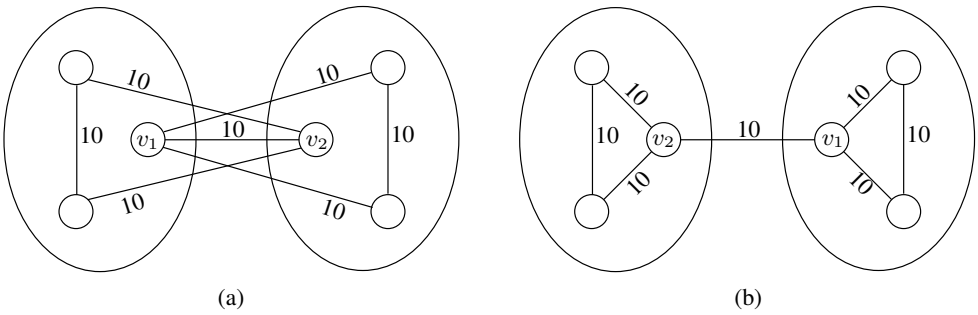


Figura 4.5: Trocando os vértices v_1 e v_2.

Em cada passo o algoritmo calcula a redução do custo proporcionado pela troca de cada par de vértices. A troca que proporciona a maior redução no custo é realizada e então o próximo passo é executado.

Em um determinado passo não existirá mais trocas que proporcionem ganhos. A abordagem mais simples interromperia a execução nesse ponto, visto que a idéia é minimizar o custo total. Como exemplo, vamos supor a seguinte seqüência de ganhos: $\{7, 2, -1, 2, -3, 2, \cdots\}$, onde cada valor representa o ganho da melhor troca em cada passo do algoritmo. Se o algoritmo finaliza no primeiro ganho negativo, o ganho total é $7 + 2 = 9$. No entanto, se formos mais adiante, e realizarmos a troca com ganho negativo (-1), os próximos passos podem apresentar trocas com ganhos que sobreponham essa perda. No caso da seqüência anterior, o ganho total poderia ser $7 + 2 - 1 + 2 = 10$. A principal contribuição da heurística de Kernighan e Lin é permitir que o algoritmo continue trocando vértices para escapar dos mínimos locais.

O Algoritmo 4.3 mostra o núcleo da heurística de Kernighan e Lin. A partir de um agrupamento inicial ($Agrupamento\_Inicial$), o algoritmo tenta trocar vértices para melhorar o custo do agrupamento. O bloqueio de vértices garante que cada vértice será trocado apenas uma vez, conseqüentemente, o algoritmo termina. Por fim, a condição da linha 11 garante que o melhor agrupamento encontrado será retornado.

Note que o Algoritmo 4.3 retorna um agrupamento melhor ou igual ao inicial. Se o agrupamento retornado é melhor que o inicial, o algoritmo é executado novamente sobre o resultado anterior. O algoritmo só termina quando o agrupamento não é mais melhorado pelo Algoritmo 4.3.

A heurística de Kernighan e Lin é vastamente utilizada em soluções para problemas $\mathcal{NP}$-Difíceis. Algoritmos para CAD (*Computer Aided Design*) e *High Level Synthesis* muitas vezes tiram proveito dessa técnica para encontrar soluções boas em tempo viável [41, 126]. A partir da idéia de continuar a troca de vértices, mesmo com ganhos negativos, para escapar dos mínimos locais, propusemos um algoritmo de agrupamento de colunas baseado na heurística de Kernighan e Lin.

Algoritmo 4.3 Heurística de Kernighan e Lin

```
1: Melhor_Agrupamento ← Agrupamento ← Agrupamento_Inicial
2: Agrupamento.desbloqueia_vértices()
3: repita
4:    para cada par de vértices desbloqueados (v_i,v_j) em conjuntos distintos faça
5:       Ganho ← Agrupamento.computa_ganho_com_a_troca(v_i, v_j)
6:    fim para
7:    /* Seja v_1 e v_2 os vértices da troca com melhor ganho */
8:    Agrupamento.troca_vértices(v_1, v_2)
9:    Agrupamento.bloqueia(v_1)
10:   Agrupamento.bloqueia(v_2)
11:   se Agrupamento.custo < Melhor_Agrupamento.custo então
12:      Melhor_Agrupamento ← Agrupamento
13:   fim se
14: até que não haja mais vértices desbloqueados
15: retorna Melhor_Agrupamento
```

AKL sem restrições

Diferente do problema de agrupamento abordado por Kernighan e Lin, o agrupamento de colunas para compressão de microcódigo não tenta reduzir o custo de comunicação entre os conjuntos, mas sim, o número de padrões dentro de cada conjunto de colunas.

Apesar da função de custo não ser diretamente aplicada ao problema de agrupamento de colunas, a idéia de continuar trocando vértices, mesmo com ganhos negativos, permite que o algoritmo escape dos mínimos locais.

Além da operação de troca de vértices, o agrupamento de colunas baseado em Kernighan e Lin, ou AKL, permite que colunas sejam movidas de um conjunto para outro. Da mesma forma que a heurística de Kernighan e Lin, o agrupamento de colunas parte de um agrupamento inicial, e a cada passo, o algoritmo descobre e realiza a melhor operação. O Algoritmo 4.4 mostra o núcleo do algoritmo proposto.

No Algoritmo 4.4 existem dois tipos de operações:

- TR: Troca dois vértices em conjuntos distintos.
- MV: Move um vértice para outro conjunto.

A melhor operação é aquela que provê o maior ganho quando executada. O ganho de uma operação op é a redução do número de *bits* no microcódigo comprimido após a execução da operação. Seja K_i e K_j os conjuntos modificados pela operação op e K'_i e K'_j os conjuntos K_i e K_j após a execução da operação op, o ganho de op é definido por:

$$ganho(op) = (tc(K_i) + tc(K_j)) - (tc(K'_i) + tc(K'_j)) \tag{4.7}$$

Algoritmo 4.4 AKL - Agrupamento de colunas baseado em Kernighan e Lin

1: $Melhor\_Agrupamento \leftarrow Agrupamento \leftarrow Agrupamento\_Inicial$
2: $Agrupamento.desbloqueia\_colunas()$
3: **repita**
4: $op \leftarrow Agrupamento.computa\_melhor\_op()$
5: **se** $op.tipo = TR$ **então**
6: $Agrupamento.troca\_colunas(op.c_1, op.c_2)$
7: $Agrupamento.bloqueia(op.c_1)$
8: $Agrupamento.bloqueia(op.c_2)$
9: **senão se** $op.tipo = MV$ **então**
10: $Agrupamento.move\_coluna(op.c_1, op.K_2)$
11: $Agrupamento.bloqueia(op.c_1)$
12: **fim se**
13: **se** $Agrupamento.custo < Melhor\_Agrupamento.custo$ **então**
14: $Melhor\_Agrupamento \leftarrow Agrupamento$
15: **fim se**
16: **até** que não haja mais operações válidas
17: **retorna** $Melhor\_Agrupamento$

onde $tc(K_i)$ é o tamanho comprimido do conjunto K_i. Seja L_i e M_i o número de colunas e padrões do conjunto K_i, o tamanho comprimido do conjunto K_i é definido pela equação:

$$tc(K_i) = N * \lceil log_2(M_i) \rceil + M_i * L_i \tag{4.8}$$

O método $computa\_melhor\_op()$, na quarta linha do Algoritmo 4.4, calcula o ganho de todas as operações válidas, ou seja, operações que não envolvam colunas bloqueadas, e retorna a operação que provê o melhor ganho.

Apesar do algoritmo tomar como entrada um agrupamento inicial, o número de conjuntos não é restrito por este. A operação MV permite que colunas migrem de um conjunto para outro e durante este processo alguns conjuntos podem ser esvaziados. Assim sendo, o número de conjuntos no agrupamento inicial pode ser diferente do número de conjuntos no agrupamento ao término do algoritmo.

4.2.5 Outras Heurísticas

Esta seção apresenta as heurísticas de agrupamento de colunas propostas por outros autores. Os resultados experimentais desses algoritmos podem ser encontrados na Seção 5.1.

Heurística de Ishiura e Yamaguchi

Ishiura e Yamaguchi [67] propuseram um método para compressão de código VLIW semelhante à técnica de compressão de microcódigo apresentada no Capítulo 3.

Nesse trabalho, os *bits* das instruções VLIW são particionados em campos e os padrões de *bits* dentro dos campos são substituídos por *codewords*. Durante a execução do programa os decodificadores, similares aos dicionários na compressão de microcódigo, transformam as *codewords* nos padrões de *bits* originais. Os decodificadores são implementados com ROMs e o método de partição agrupa os *bits* das instruções VLIW sem comprometer o tamanho dos decodificadores dos campos.

Para particionar os *bits* das instruções VLIW em campos, os autores propuseram uma heurística que identifica e agrupa campos similares. Seja F o particionamento dos *bits* em campos e f_i e f_j dois campos distintos em F, a função de ganho $G(F, f_i, f_j)$ determina o número de *bits* economizados se os *bits* dos campos f_i e f_j forem agrupados em um único campo f_{ij}. A idéia básica da heurística é encontrar o par de campos $(f_i, f_j) \in F$ que provê o melhor ganho $G(F, f_i, f_j)$ e agrupar as colunas desses campos em um novo campo f_{ij}. Esse processo é repetido até que não haja mais ganhos. O Algoritmo 4.5 mostra o pseudo-código para a heurística de Ishiura e Yamaguchi. Observe que no particionamento inicial cada campo possui exatamente um *bit* b_i.

Algoritmo 4.5 Particionamento das Instruções VLIW em Campos [67]

```
1:  F ← {{b_1}, {b_2}, ··· , {b_w}};
2:  repita
3:     encontre (f_i, f_j) ∈ F × F que maximize G(F, f_i, f_j);
4:     se G(F, f_i, f_j) > 0 então
5:        agrupe f_i e f_j;
6:     senão
7:        encontre (f_i, f_j) ∈ F × F que maximize G'(F, f_i, f_j);
8:        se G'(F, f_i, f_j) > 0 então
9:           agrupe f_i e f_j;
10:       senão
11:          encontre (f_i, f_j) ∈ F × F que maximize G''(F, f_i, f_j);
12:          se G''(F, f_i, f_j) > 0 então
13:             agrupe f_i e f_j;
14:          fim se
15:       fim se
16:    fim se
17: até que não haja mais agrupamentos
```

A quarta linha do algoritmo assegura que o par de campos (f_i, f_j) só será agrupado se o

ganho proporcionado pelo agrupamento for positivo. O objetivo é parar de agrupar os campos quando essa operação não for mais vantajosa. Entretanto, essa restrição pode fazer com que o algoritmo fique preso em soluções mínimas locais. De fato, o particionamento inicial tem grandes chances de não ser modificado por causa dessa restrição. Observe que é necessário um par de colunas que não produzam mais do que dois padrões de *bits* para que haja ganho no agrupamento de dois campos com uma coluna cada. Para contornar essa limitação, os autores propuseram o uso de duas funções de ganhos relaxadas, G' e G'', que são utilizadas quando o resultado de $G(F, f_i, f_j)$ é negativo. Mais detalhes sobre essas funções podem ser encontrados em [67].

Heurística de Zhao e Papachristou

Zhao e Papachristou [136] investigaram a compressão de microcódigo em *Application Specific Programmable Processors* (ASPPs). Os autores utilizaram um método de compressão semelhante ao descrito no Capítulo 3 e propuseram uma heurística para particionar os *bits* do microcódigo em campos.

A heurística proposta por Zhao e Papachristou busca e agrupa pares de campos similares. Da mesma forma que a heurística de Ishiura e Yamaguchi [67], a similaridade entre os campos é medida com base na redução de *bits* advinda do agrupamento dos campos. O Algoritmo 4.6 mostra o heurística proposta por Zhao e Papachristou.

Algoritmo 4.6 Particionamento de Microcódigo [136]

```
1:  enquanto f > 1 faça /* f é o número de campos no microcódigo. */
2:    para i ← 1 a f − 1 faça
3:      para j ← i + 1 a f faça
4:        calcule a similaridade[i][j]
5:        se similaridade[i][j] < temp então
6:          m ← i; n ← j;
7:          temp ← similaridade[i][j];
8:        fim se
9:      fim para
10:   fim para
11:   Agrupe_os_campos(m, n);
12:   f ← f − 1;
13:   Calcule o tamanho em bits do microcódigo;
14:   Armazene o melhor particionamento;
15: fim enquanto
16: retorna Melhor particionamento
```

Após identificar e agrupar o par de campos com maior similaridade, a heurística calcula o

tamanho do microcódigo e armazena o melhor particionamento, ou seja, o particionamento que resulta no menor tamanho em *bits*. O processo é repetido até que reste apenas um único campo. Essa abordagem permite que a heurística continue agrupando campos sem o risco de ficar presa em soluções mínimas locais. Ao final, o melhor particionamento encontrado é retornado.

4.3 Agrupamento de Colunas com Restrições

As heurísticas apresentadas na Seção 4.2 agrupam as colunas do microcódigo sem restrições no número e tamanho dos conjuntos. A ausência de restrições permite que as heurísticas tenham mais flexibilidade para explorar a entropia do microcódigo e alcançar melhores razões de compressão. Por outro lado, o projeto e a implementação do vetor de apontadores e dos dicionários são influenciados por restrições impostas pela tecnologia utilizada para implementar a μROM e pela organização interna do microprocessador. O formato da área alocada para a μROM é um dos fatores que influenciam na quantidade e no formato dos dicionários.

Apesar das restrições reduzirem o número total de agrupamentos válidos, o problema não se torna necessariamente mais fácil. No Apêndice A, provamos que, quando o número de colunas por dicionário e a quantidade de dicionários são fixos, o problema de agrupamento de colunas é $\mathcal{NP}$-Completo. Esse resultado indica a necessidade de heurísticas para realizarmos o agrupamento das colunas do microcódigo de forma eficiente.

A próxima seção descreve as modificações feitas para incluir restrições no algoritmo AKL.

4.3.1 AKL com Restrições

Restrições na quantidade de dicionários e número de colunas por dicionário

O algoritmo AKL, descrito na Seção 4.2.4, parte de um agrupamento inicial e a cada passo realiza uma operação de troca ou movimentação de colunas entre os conjuntos. A operação MV migra colunas de um conjunto para outro, modificando o tamanho dos conjuntos. Por outro lado, as operações de troca de colunas (TR) não afetam o número de colunas nem a quantidade de conjuntos no agrupamento. Assim sendo, se a heurística não realizar as operações MV, o agrupamento final terá o mesmo número de conjuntos e a mesma quantidade de colunas por conjunto que o agrupamento inicial.

Para inserir as restrições de quantidade de dicionários e número de colunas por dicionário, fizemos as seguintes alterações no Algoritmo 4.4:

- O algoritmo toma como entrada o número de conjuntos (K) e a quantidade mínima (Min_Cols_i) e máxima (Max_Cols_i) de colunas em cada conjunto C_i.

- Inserimos uma rotina para verificar se o agrupamento inicial satisfaz as restrições impostas pelo usuário. Essa verificação é necessária para garantir que em todos os passos do algoritmo o agrupamento de colunas é válido com relação às restrições impostas.

- Modificamos o método $computa\_melhor\_op()$ para ignorar operações que produzam agrupamentos que não satisfaçam as restrições. A priori, as operações MV que violam o número mínimo de colunas do conjunto fonte ou o número máximo de colunas do conjunto alvo são ignoradas.

A partir dessas modificações podemos restringir o número de conjuntos e, conseqüentemente, o número de dicionários, resolvendo assim, o problema de agrupamento da categoria 5 (Tabela 4.1). Para tanto, basta restringir o número mínimo de colunas de cada conjunto em 1 ($Min\_Cols_i = 1$) e o número máximo em L ($Max\_Cols_i = L$). Dessa forma, os conjuntos podem perder ou ganhar qualquer quantidade de colunas, mas não podem se tornar vazios. Na Seção 5.2.1, utilizamos essas restrições para avaliar o comportamento da razão de compressão em função do número de dicionários.

A mesma idéia pode ser utilizada para resolver o problema de agrupamento da categoria 6, onde o número de colunas por dicionário e o número de dicionários é restrito. No caso onde cada conjunto C_i tem sua quantidade de colunas fixa em L_i, basta fazermos com que $Min\_Cols_i = Max\_Cols_i = L_i$. Observe que só haverão operações de troca, pois qualquer operação de movimentação violaria as restrições impostas. Na Seção 5.2.2, utilizamos essas restrições para produzir dicionários com tamanhos similares durante a compressão do microcódigo.

O problema de agrupamento da categoria 2, onde apenas o número de colunas por dicionário é restrito, pode ser resolvido executando-se várias instâncias do algoritmo com restrições, cada uma com um número diferente de dicionários. Por exemplo, vamos supor que desejamos encontrar um agrupamento de colunas para um microcódigo com 36 colunas, onde cada dicionário tenha exatamente cinco colunas. Primeiro resolvemos o problema para um único conjunto com cinco colunas. Depois resolvemos com dois conjuntos com cinco colunas cada e assim por diante, até sete conjuntos com cinco colunas cada, o que totaliza 35 colunas agrupadas. Ao final, escolhemos o agrupamento que provê a melhor compressão. Dessa forma, garantimos que, independente da quantidade de dicionários, a compressão possuirá apenas dicionários com cinco colunas. Diferente das categorias 5 e 6, não encontramos problemas de agrupamento de colunas reais que se enquadrem na categoria 2.

Restrições no número de linhas por dicionário

Os problemas de agrupamento das categorias 3, 4, 7 e 8 restringem o número de linhas em cada dicionário. O número de linhas, por sua vez, é determinado pela quantidade de padrões (M_i) em cada conjunto C_i.

Modificamos o algoritmo de agrupamento com restrições para não realizar operações que violem o número máximo (Max_M_i) e mínimo (Min_M_i) de padrões em cada conjunto C_i. Assim sendo, se o agrupamento inicial satisfizer as restrições do número de padrões em cada conjunto, o agrupamento final também o satisfará. Observe que, no entanto, gerar um agrupamento inicial pode ser um problema a parte. Compor um agrupamento inicial com restrições no número de dicionários e na quantidade de colunas por dicionário é uma tarefa trivial e não depende da similaridade das colunas. Por outro lado, gerar um agrupamento inicial com restrições no número de padrões em cada conjunto pode ser um problema difícil. Se as restrições forem muito justas pode ser que poucos agrupamentos satisfaçam as restrições. No caso em que apenas um único agrupamento satisfaz as restrições, o problema se torna $\mathcal{NP}$-Completo (Apêndice A).

4.4 Conclusões

Neste capítulo apresentamos algoritmos para identificar e agrupar colunas semelhantes do microcódigo. Classificamos o agrupamento de colunas em oito categorias, de acordo com as restrições no número e tamanho dos dicionários, e dividimos os algoritmos em dois grupos principais: agrupamento sem restrições e agrupamento com restrições.

Na Seção 4.2 propomos quatro algoritmos novos para realizar o agrupamento de colunas sem restrições. A ausência de restrições permite que o algoritmo de agrupamento de colunas tenha mais flexibilidade para agrupar as colunas e maximizar a compressão do microcódigo. De fato, a solução ótima do agrupamento de colunas sem restrições determina o limitante inferior da razão de compressão do microcódigo. Por outro lado, o número de conjuntos e a quantidade de colunas por conjunto no resultado final não podem ser controlados pelo usuário.

Por fim, na Seção 4.2 identificamos a necessidade de se agrupar as colunas com restrições no número e tamanho dos dicionários e apresentamos uma heurística para agrupar colunas com restrições. Adicionalmente, provamos que, quando o número de dicionários e a quantidade de colunas por dicionários são fixos, o problema de agrupamento de colunas é $\mathcal{NP}$-Completo. A prova se encontra no Apêndice A.

O próximo capítulo compara as técnicas de agrupamento de colunas utilizando-se quatro microcódigos extraídos de processadores em desenvolvimento e produção.

Capítulo 5

Resultados Experimentais

Neste capítulo comparamos as técnicas de agrupamento de colunas quando aplicadas a microcódigos extraídos de processadores em produção e em estágios avançados de desenvolvimento.

O microcódigo de um processador é geralmente considerado informação confidencial. Nossos experimentos com microcódigos de produção foram possíveis devido à parceria do Laboratório de Sistemas de Computação (LSC) da Unicamp com o *Programming System Laboratory* (PSL/MTL, Santa Clara, California) da Intel Corporation. Nessa parceria, realizamos experimentos de compressão em quatro microcódigos. O primeiro microcódigo foi extraído de um processador com microarquitetura Netburst e contém 22 528 microinstruções com 75 *bits* cada. O segundo e o terceiro microcódigo são de processadores baseados na microarquitetura do Pentium-M e o último foi extraído de um processador usado em aplicações móveis. A Tabela 5.1 resume os microcódigos utilizados em nossos experimentos.

Tabela 5.1: Descrição dos microcódigos.

Microcódigo	# Colunas	# Linhas	Tamanho em *bits*	Classe
A	75	22 528	1 689 600	Netburst
B	80	6 144	491 520	Pentium-M
C	234	3 328	778 752	
D	236	5 632	1 329 152	Móvel

Assim como as técnicas de agrupamento de colunas do Capítulo 4, dividimos os experimentos em duas categorias: compressão sem restrições e compressão com restrições. A Seção 5.1 compara as heurísticas de agrupamento de colunas sem restrições e a Seção 5.2 mostra os resultados para a compressão com restrições.

5.1 Compressão sem Restrições

Para comparar as heurísticas de agrupamento de colunas sem restrições, implementamos os algoritmos apresentados da Seção 4.2, incluindo os algoritmos propostos por Ishiura e Yamaguchi [67] e Zhao e Papachristou [136], e comprimimos os quatro microcódigos da Tabela 5.1. Os resultados foram gerados utilizando os seguintes critérios:

- Dicionário Único: o microcódigo é comprimido utilizando-se um único dicionário, que inclui todas as colunas do microcódigo, como mostrado na Figura 3.1 (b).

- Ishiura e Yamaguchi [67]: as colunas do microcódigo são agrupadas com o Algoritmo 4.5 (Seção 4.2.5), proposto por Ishiura e Yamaguchi [67].

- Zhao e Papachristou [136]: as colunas do microcódigo são agrupadas com o Algoritmo 4.6 (Seção 4.2.5), proposto por Zhao e Papachristou [136].

- Colunas Seqüenciais: aplicamos o algoritmo de colunas seqüenciais (Seção 4.2.1) no microcódigo original.

- Ordenação Linear: para um dado microcódigo com L colunas, executamos o algoritmo de ordenação linear inicializando a lista de colunas (LC) com cada uma das L colunas. Como resultado, L ordenações de colunas são geradas. Para cada ordenação, as colunas do microcódigo são reorganizadas e o algoritmo de colunas seqüenciais é utilizado para agrupar as colunas do novo microcódigo.

- Ordenação Circular: da mesma forma que na ordenação linear, utilizamos o algoritmo de colunas seqüenciais para agrupar as colunas do microcódigo reorganizado. Além de inicializar a LC com cada uma das L colunas, variamos o valor do tamanho máximo da lista de colunas (W) e limitamos a execução do algoritmo a L iterações.

- AKL: o algoritmo AKL, descrito na Seção 4.2.4, toma como entrada um agrupamento inicial. Para cada microcódigo, executamos o AKL diversas vezes, cada uma com um agrupamento inicial distinto. Cada agrupamento inicial possui uma quantidade diferente de conjuntos e as colunas do microcódigo são distribuídas de forma uniforme nesses conjuntos. Por exemplo, para um microcódigo com 7 colunas, o agrupamento inicial com 3 conjuntos é construído da seguinte maneira: $\{\{c_1, c_2, c_3\}, \{c_4, c_5\}, \{c_6, c_7\}\}$.

A Figura 5.1 mostra os resultados obtidos com a compressão dos microcódigos da Tabela 5.1 utilizando as diferentes técnicas de agrupamento de colunas sem restrições.

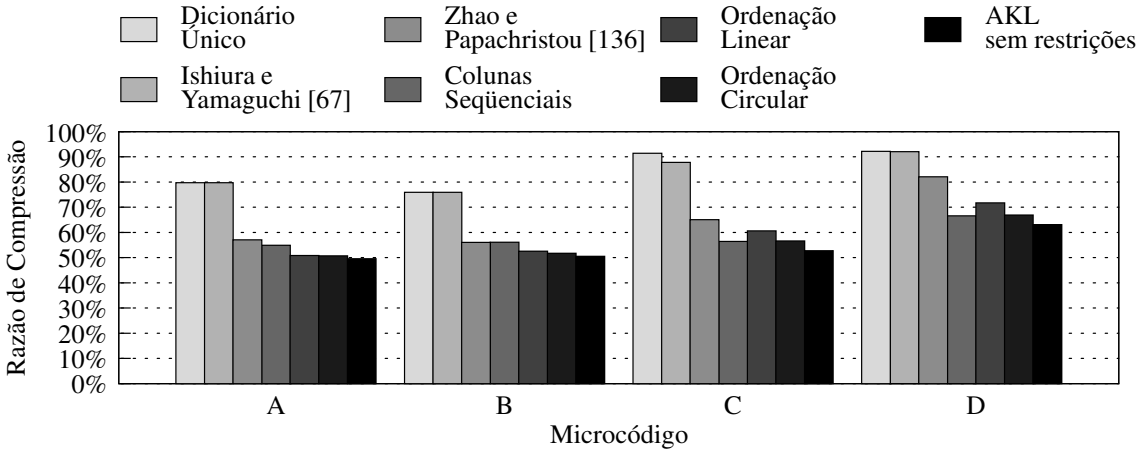


Figura 5.1: Razões de compressão produzidas pelas técnicas de agrupamento de colunas sem restrições.

A heurística de Ishiura e Yamaguchi não apresentou melhoras expressivas sobre a compressão com dicionário único. De fato, os resultados mostram apenas uma diminuição de aproximadamente 3,5% na razão de compressão do microcódigo C. A heurística de Zhao e Papachristou, por outro lado, alcançou razões de compressão significativamente melhores do que a compressão com dicionário único. O gráfico da Figura 5.1 mostra melhoras em torno de 20% nos microcódigos A, B e C e de 9,5% no microcódigo D. As técnicas propostas nesta tese, ou seja, colunas seqüenciais, ordenação linear, ordenação circular e AKL alcançaram resultados melhores do que as técnicas propostas anteriormente. O algoritmo AKL melhora de 6,1% (microcódigo B) a 19,8% (microcódigo D) as razões de compressão obtidas pela heurística de Zhao e Papachristou.

As heurísticas de ordenação linear e circular reordenam as colunas do microcódigo e, então, utilizam o algoritmo de colunas seqüenciais para comprimir o microcódigo. Como esperado, essas heurísticas obtiveram resultados melhores do que o algoritmo de colunas seqüenciais na compressão dos microcódigos A e B. No entanto, o mesmo não ocorreu nos resultados com os microcódigos C e D. A ordenação linear obteve razões de compressão em torno de 5% inferiores às razões de compressão obtidas pelo algoritmo de colunas seqüenciais nos microcódigos C e D. A degradação do resultado foi possivelmente causada por uma reordenação ruim das colunas do microcódigo.

Os resultados obtidos com a compressão dos microcódigos da Tabela 5.1 apresentam duas características que distinguem os microcódigos A e B dos microcódigos C e D: a quantidade de colunas e a quantidade de dicionários presente nos melhores resultados da compressão. Os microcódigos A e B têm 75 e 80 colunas, respectivamente, e os melhores resultados da compressão possuem dois ou três dicionários. Por outro lado, os microcódigos C e D têm 234 e 236 colunas, respectivamente, sendo que os melhores resultados da compressão possuem de 10 a 12 dicioná-

rios. Essas características, juntamente com os resultados apresentados na Figura 5.1, sugerem que a qualidade das heurísticas de ordenação de colunas depende da largura do microcódigo e/ou da quantidade de conjuntos dos agrupamentos com melhores razões de compressão. A próxima seção investiga a relação entre a capacidade de compressão das técnicas e a quantidade de colunas do microcódigo.

5.1.1 Compressão do Microcódigo versus Número de Colunas

Os resultados obtidos com a compressão dos microcódigos da Tabela 5.1 sugerem que a eficiência das heurísticas de ordenação linear e circular está relacionada ao número de colunas do microcódigo. Nesta seção, investigamos essa relação através da compressão de partes menores do microcódigo.

A idéia consiste em dividir as colunas do microcódigo em partes iguais e comprimir essas partes separadamente para avaliar o comportamento das técnicas de agrupamento quando o número de colunas do microcódigo varia. Por exemplo, podemos dividir o microcódigo D em dois microcódigos disjuntos, D' e D", cada um com 118 colunas, e comprimí-los separadamente. Nesse caso, a razão de compressão total será a média ponderada entre as razões de compressão de D' e D", ou seja, se a razão de compressão de D' é 70% e a de D" é 50%, a razão de compressão total é $((70\% \times 118) + (50\% \times 118)) \div 236 = 60\%$.

Comprimir isoladamente as partes de um microcódigo limita a capacidade das técnicas de agrupamento de colunas. Observe que as colunas em partes distintas não podem ser agrupadas em um mesmo conjunto. No exemplo anterior, as colunas em D' $(c_1, c_2, \cdots, c_{118})$ e D" $(c_{119}, c_{120}, \cdots, c_{236})$ não são agrupadas em um mesmo conjunto. Assim sendo, a compressão do microcódigo como um todo deve produzir resultados melhores ou equivalentes à compressão do microcódigo em partes. De fato, se o algoritmo de agrupamento de colunas sempre encontra o resultado ótimo, então o agrupamento de colunas no microcódigo completo é melhor ou equivalente ao agrupamento de colunas do microcódigo em partes.

As heurísticas apresentadas no Capítulo 4 não garantem o resultado ótimo no agrupamento de colunas. Dessa forma, os resultados da compressão do microcódigo em partes pode ser melhor do que a compressão do microcódigo como um todo. Por exemplo, quando o microcódigo C é dividido em três partes iguais (com 78 colunas cada), o algoritmo de Zhao e Papachristou [136] encontra um agrupamento de colunas que leva a uma razão de compressão total de $(58{,}0\% + 57{,}7\% + 58{,}7\%) \div 3 = 58{,}1\%$. Por outro lado, quando o microcódigo não é dividido, o mesmo algoritmo encontra um agrupamento que atinge uma razão de compressão de 64,9%. Essa degradação do resultado, quando o agrupamento é realizado com mais colunas, indica que a heurística é sensível ao tamanho da entrada do problema e pode produzir resultados inferiores quando o microcódigo possui muitas colunas.

Para avaliar as heurísticas de agrupamento de colunas dividimos os microcódigos A, B,

C e D em partes iguais e as comprimimos separadamente. A Figura 5.2 mostra a razão de compressão total produzida pelas heurísticas de agrupamento de colunas para os microcódigos particionados. Observe que o microcódigo A foi dividido em quatro configurações: a primeira possui 15 partes com cinco colunas cada (15x5), a segunda possui cinco partes com 15 colunas cada (5x15), a terceira possui três partes com 25 colunas (3x25) e a última possui uma única parte com 75 colunas (1x75). Para cada configuração, as partes são comprimidas individualmente e a razão de compressão total é computada através da média ponderada das razões de compressão em cada parte.

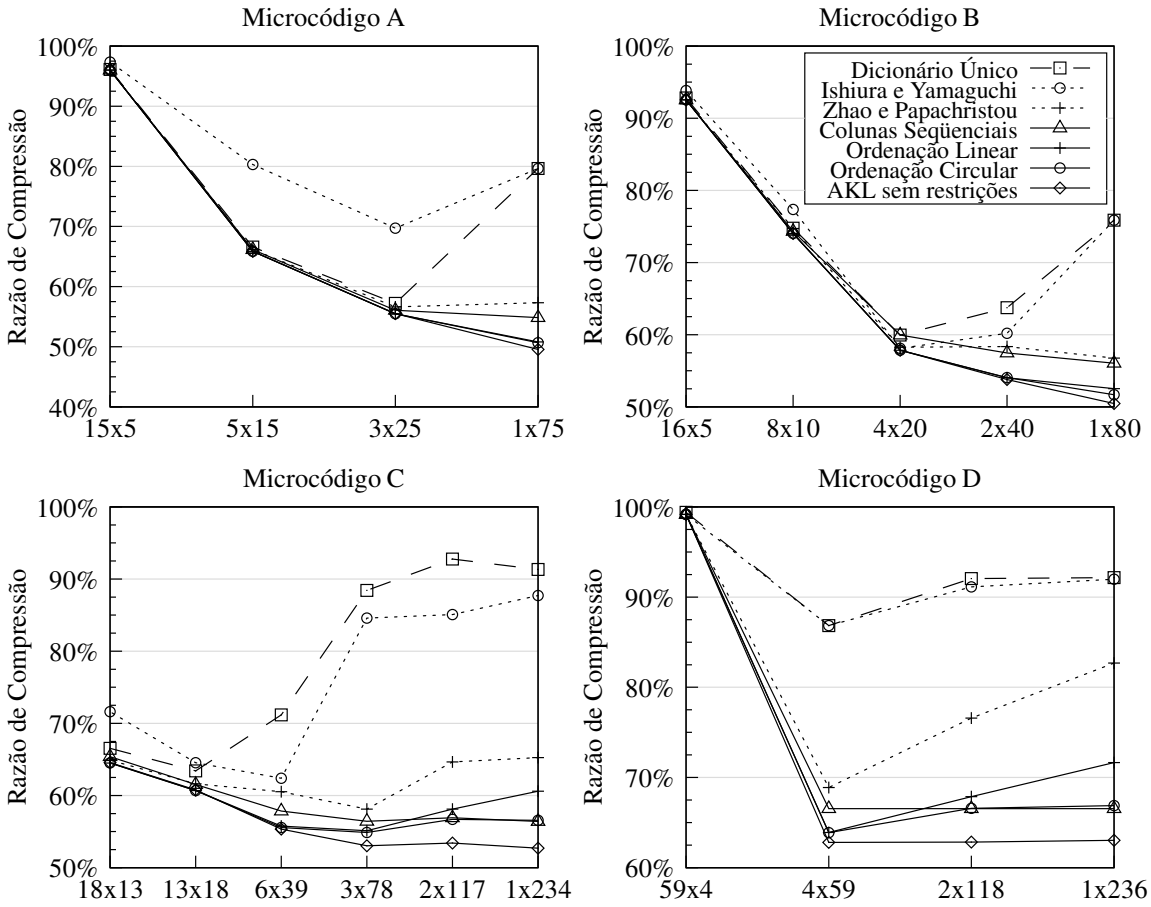


Figura 5.2: Resultado da compressão dos microcódigos em partes.

Nos resultados apresentados na Figura 5.2, quando o microcódigo é dividido em partes, a técnica de dicionário único utiliza um dicionário para cada parte. Assim sendo, para um microcódigo dividido em quatro partes, as colunas são agrupadas em quatro dicionários distintos. Por ser a abordagem mais simples, o resultado dessa técnica serve como base para avaliar o resultado produzido pelas outras técnicas. Por exemplo, observe que, no gráfico do microcódigo A, a técnica de Ishiura e Yamaguchi produz resultados piores que a técnica de dicionário único para

as três primeiras configurações. De fato, a técnica de Ishiura e Yamaguchi apresentou resultados próximos ou inferiores aos resultados obtidos com a técnica de dicionário único na maioria dos casos. Isso indica que essa heurística não consegue explorar de forma eficiente o agrupamento de colunas similares.

À medida que o número de partes diminui, a quantidade de colunas em cada parte aumenta e as heurísticas têm mais oportunidades para agrupar as colunas. Assim sendo, a tendência é que haja uma melhora na razão de compressão quando o número de partes diminui. De fato, a maioria das curvas nos gráficos da Figura 5.2 segue essa tendência. Entretanto, em alguns casos, esse efeito não é constatado. Por exemplo, no microcódigo A, os resultados produzidos pelas técnicas de Ishiura e Yamaguchi [67] e Zhao e Papachristou [136] são melhores quando o agrupamento de colunas é realizado na configuração com três partes (3x25) do que na configuração com o microcódigo completo (1x75). Esse comportamento indica que a qualidade dos resultados produzidos pela heurística é sensível ao número de colunas do microcódigo e quanto maior o número de colunas menor é a chance da heurística produzir bons resultados.

A heurística de Zhao e Papachristou apresentou resultados superiores à compressão com dicionário único em todas as configurações. Mesmo assim, em alguns casos, quando o número de partes do microcódigo diminui, podemos perceber uma degradação nos resultados da compressão. Essa degradação é visivelmente acentuada no gráfico do microcódigo D, onde a razão de compressão piora de 68,9%, na configuração com quatro partes de 59 colunas (4x59), para 82,7% na configuração com o microcódigo completo (1x236). Os gráficos dos microcódigos C e D mostram que, apesar de apresentarem resultados melhores do que a heurística de Zhao e Papachristou, as técnicas de ordenação linear e circular também apresentam degradação nos resultados quando o número de colunas do microcódigo aumenta. Assim sendo, a qualidade dos resultados gerados por essas heurísticas também é sensível ao número de colunas do microcódigo.

Os resultados produzidos pela heurística de colunas seqüenciais não sofrem degradação com o aumento do número de colunas do microcódigo. Esse efeito não acontece porque a heurística encontra a solução ótima para agrupamentos com colunas seqüenciais. As soluções encontradas para as partes do microcódigo também são compostas por conjuntos com colunas seqüenciais, dessa forma, esses conjuntos podem ser compostos para gerar um agrupamento de colunas no microcódigo completo. Por exemplo, um microcódigo com seis colunas pode ser dividido em duas partes com três colunas cada. Sendo $\{\{c_1\},\{c_2,c_3\}\}$ e $\{\{c_4,c_5\},\{c_6\}\}$ os agrupamentos encontrados para ambas as partes, o agrupamento de colunas para o microcódigo completo pode ser composto a partir da união dos dois agrupamentos, ou seja, $\{\{c_1\},\{c_2,c_3\},\{c_4,c_5\},\{c_6\}\}$. Por construção, o agrupamento composto é um agrupamento com colunas seqüenciais e o resultado é pior ou igual ao resultado ótimo encontrado pela técnica de colunas seqüenciais no microcódigo completo. Assim sendo, os resultados dessa técnica para a compressão do microcódigo completo são sempre melhores ou iguais aos resultados da compressão em partes.

O algoritmo AKL apresentou os melhores resultados de compressão. Apesar do gráfico do microcódigo D mostrar uma ligeira degradação (0,2%) do resultado entre as configurações 4x59 e 1x236, na grande maioria dos casos, a técnica se mostrou estável obtendo resultados melhores à medida em que o número de partes do microcódigo diminuía. A Figura 5.3 apresenta os melhores resultados obtidos após a divisão dos microcódigos em partes. As regiões sombreadas mostram os resultados da Figura 5.1, obtidos com os microcódigos completos. Observe que a técnica de Ishiura e Yamaguchi melhora significativamente os resultados após a divisão dos microcódigos em partes, no entanto, se mostra pior do que a técnica de dicionário único. As técnicas de Zhao e Papachristou, ordenação linear e ordenação circular também apresentam melhoras significativas nos microcódigos C e D, o que aponta a fragilidade dessas técnicas quando o número de colunas é grande.

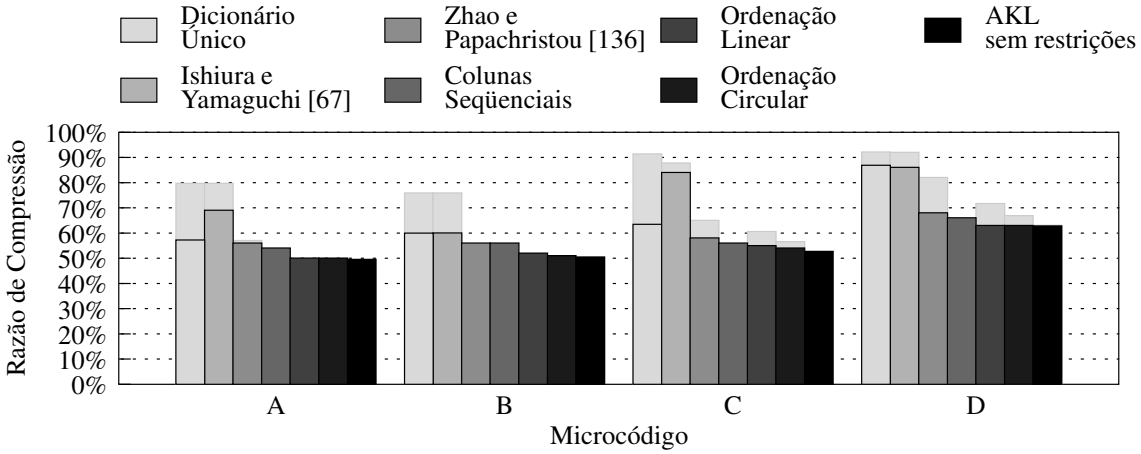


Figura 5.3: Melhores resultados após a divisão do microcódigo em partes.

5.1.2 Tempo de Execução dos Algoritmos

Durante o desenvolvimento do processador, o microcódigo é freqüentemente alterado. Conseqüentemente, o microcódigo deve ser comprimido diversas vezes para gerar um novo descompressor ou mesmo para verificar se a razão de compressão variou significativamente. Assim sendo, além da capacidade de comprimir o microcódigo, outro fator importante na escolha da técnica de compressão é o tempo de execução do algoritmo. Nesta seção apresentamos o tempo de execução das heurísticas de agrupamento de colunas. Os experimentos de tempo de execução foram realizados em um computador com processador Intel Pentium 4 2.8GHz e 1GB de memória RAM.

De acordo com os critérios apresentados na Seção 5.1, as heurísticas de ordenação linear, ordenação circular e o algoritmo AKL são executados diversas vezes, cada uma com uma configuração diferente. Por exemplo, a heurística de ordenação linear utiliza o Algoritmo 4.1 para

gerar L ordenações de colunas e, em seguida, executa o algoritmo de colunas seqüenciais para cada ordenação de colunas gerada. Como as diferentes configurações podem ser executadas em paralelo, apresentamos os resultados de tempo de execução em dois grupos: heurísticas com execução única e heurísticas com múltiplas execuções.

Heurísticas com execução única

As heurísticas de Ishiura e Yamaguchi, de Zhao e Papachristou e de colunas seqüenciais executam um único programa para agrupar as colunas do microcódigo. A Tabela 5.2 apresenta o tempo de execução dessas heurísticas quando agrupamos as colunas dos microcódigos da Tabela 5.1.

Tabela 5.2: Tempo de execução das heurísticas com execução única.

Microcódigo	Tamanho	Ishiura e Yamaguchi	Zhao e Papachristou	Colunas Seqüenciais
A	$75 \times 22\,528$	2m 02s	1m 58s	1m 19s
B	$80 \times 6\,144$	2m 38s	38s	20s
C	$234 \times 3\,328$	16m 59s	17m 16s	2m 25s
D	$236 \times 5\,632$	14m 48s	15m 23s	5m 45s

Os resultados da Tabela 5.2 mostram que o tempo de execução das heurísticas é sensível ao tamanho do microcódigo. Observe também que, além de executar em menos tempo, o algoritmo de colunas seqüenciais obteve razões de compressão melhores do que o algoritmo de Ishiura e Yamaguchi e Zhao e Papachristou (Figura 5.3).

Heurísticas com múltiplas execuções

As heurísticas de ordenação linear, ordenação circular e AKL são executadas diversas vezes para cada microcódigo. A maior parte dessas execuções podem ser realizadas em paralelo, utilizando-se diversos computadores. Dessa forma, o tempo de execução depende da quantidade de computadores disponíveis para realizar a compressão.

A heurística de ordenação linear utiliza o Algoritmo 4.1 para gerar L ordenações de colunas e, em seguida, executa o algoritmo de colunas seqüenciais para cada ordenação de colunas gerada. Dessa forma, o tempo total de execução da heurística de ordenação linear é o tempo de execução do Algoritmo 4.1 mais o tempo de execução do algoritmo de colunas seqüenciais nas L ordenações de colunas. A Tabela 5.3 mostra o tempo de execução do Algoritmo 4.1 e do algoritmo de colunas seqüenciais quando a heurística de ordenação linear é aplicada aos

microcódigos da Tabela 5.1. Novamente, o tempo de execução da heurística é sensível ao número de colunas e linhas do microcódigo.

Tabela 5.3: Tempo de execução da heurística de ordenação linear.

Microcódigo	Tamanho	Algoritmo 4.1	Colunas Seqüenciais		Total
			# de Execuções	Tempo Médio	
A	$75 \times 22\,528$	16m	75	1m 27s	2h 04m
B	$80 \times 6\,144$	5m	80	19s	30m
C	$234 \times 3\,328$	1h 26m	234	2m 18s	10h 24m
D	$236 \times 5\,632$	3h 24m	236	6m 01s	27h 05m

No pior caso, utilizando-se apenas um computador, a heurística de ordenação linear agrupa as colunas do microcódigo D em 27 horas. Observe que são necessárias aproximadamente três horas e meia para gerar as 236 ordenações de colunas (tempo do Algoritmo 4.1) e 23 horas e meia para executar o algoritmo de colunas seqüenciais em cada uma das ordenações de colunas. Por outro lado, o algoritmo de colunas seqüenciais pode ser executado em paralelo, utilizando-se diversos computadores, e o tempo de execução pode ser reduzido para até três horas e meia.

No algoritmo de ordenação circular, além de iniciar a LC com cada uma das L colunas, variamos o tamanho máximo da lista de colunas (W) e limitamos a execução do algoritmo a L iterações. Dessa forma, para cada W, o algoritmo gera $L \times L$ ordenações de colunas. Apesar dos resultados de compressão da ordenação circular serem ligeiramente melhores do que os resultados da ordenação linear, o número de ordenações de colunas geradas é muito grande e, se o número de computadores utilizados for pequeno, a execução do algoritmo de colunas seqüenciais em todas as ordenações de colunas pode levar meses, ou até anos.

O algoritmo AKL toma como entrada um agrupamento de colunas inicial. Em nossos experimentos, geramos diversos agrupamentos iniciais variando o número de conjuntos e distribuindo uniformemente as colunas entre esses conjuntos. Por exemplo, para um microcódigo com 7 colunas, o agrupamento inicial com 3 conjuntos é construído da seguinte maneira: $\{\{c_1, c_2, c_3\}, \{c_4, c_5\}, \{c_6, c_7\}\}$. A Tabela 5.4 mostra o tempo de execução do algoritmo AKL quando comprimimos os microcódigos da Tabela 5.1. O microcódigo D foi comprimido utilizando-se 20 agrupamentos iniciais e o tempo total da compressão desse microcódigo foi de dez dias sete horas e trinta minutos.

Da mesma forma que na ordenação linear, o algoritmo AKL pode ser executado em paralelo para cada um dos agrupamentos iniciais gerados. Os resultados da Tabela 5.4 apresentam o pior tempo de execução e o tempo total, somando-se o tempo de todas as execuções. Observe que o tempo total de compressão pode ser reduzido através da execução do algoritmo AKL em vários computadores. Dessa forma, o microcódigo A poderia ser comprimido em uma hora e

Tabela 5.4: Tempo de execução da heurística AKL.

Microcódigo	Tamanho	# de Agrupamentos Iniciais	Pior Tempo	Total
A	$75 \times 22\,528$	8	1h 38m	7h 46m
B	$80 \times 6\,144$	12	1h 31m	11h 49m
C	$234 \times 3\,328$	20	14h 54m	6d 6h 38m
D	$236 \times 5\,632$	20	30h 46m	10d 7h 30m

38 minutos e o microcódigo D poderia ser comprimido em até 30h e 46 minutos.

Os resultados apresentados nesta seção mostram que as heurísticas com execução única comprimem o microcódigo em tempo significativamente menor do que as heurísticas com múltiplas execuções. Dentre as heurísticas com execução única, a heurística de colunas seqüenciais apresentou os melhores resultados em tempo de execução e razão de compressão. Dessa forma, em um cenário onde o tempo de compressão do microcódigo é um fator importante, essa técnica é a mais adequada para comprimir o microcódigo.

Apesar das heurística com múltiplas execuções executarem em um tempo maior, elas apresentam razões de compressão melhores. Dentre essas heurísticas, o algoritmo AKL apresentou os melhores resultados de compressão e tempos de execução compatíveis com a ordenação linear. Assim sendo, o algoritmo AKL é o mais adequado quando o tempo de compressão do microcódigo não é um fator crítico.

5.1.3 Estabilidade da Razão de Compressão

A Seção 1.2 aponta a necessidade da previsão e da manutenção do tamanho da μROM durante o desenvolvimento do microprocessador. Apesar do desenvolvimento do microcódigo ocorrer de forma paralela ao desenvolvimento do microprocessador, a quantidade de *bits* e o número de microinstruções no microcódigo final podem ser estimados com base no microcódigo de processadores da mesma família ou semelhantes. Dessa forma, a área a ser ocupada pela μROM no final do projeto pode ser estimada com certo grau de precisão.

A compressão do microcódigo introduz uma nova variável na previsão do tamanho final da μROM. Além da quantidade de microinstruções, o projetista deve prever a razão de compressão do microcódigo no estágio final de desenvolvimento do microprocessador. Observe que, se essa previsão estiver errada, a área previamente alocada à μROM pode ser insuficiente no final do projeto. Por exemplo, vamos supor que, no estágio inicial de desenvolvimento, a razão de compressão do microcódigo é de 40% e que, durante o desenvolvimento do microprocessador, a entropia do microcódigo seja alterada, aumentando a razão de compressão para 60%. Se a área alocada à μROM for estimada com base na razão de compressão inicial, de 40%, pode ser que

o espaço alocado para a μROM seja insuficiente no final do projeto. Assim sendo, é importante prever o que acontece com a razão de compressão durante o desenvolvimento do microcódigo.

Definimos a estabilidade da razão de compressão como sendo o comportamento da razão de compressão em função do crescimento, ou desenvolvimento, do microcódigo. Se a razão de compressão varia bruscamente quando o microcódigo cresce, dizemos que a razão de compressão não é estável. Por outro lado, se a razão de compressão aumenta ou diminui suavemente em função do crescimento do microcódigo, dizemos que a razão de compressão é estável. A estabilidade da razão de compressão é importante para a manutenção e a previsão da área da μROM no início e durante o projeto do microprocessador.

Para medir a estabilidade da razão de compressão, precisamos comprimir o microcódigo em suas diversas etapas de desenvolvimento. Entretanto, essa informação é confidencial e não temos acesso a esses dados. Vamos assumir que o microcódigo em um estágio inicial de desenvolvimento corresponde a um subconjunto das microinstruções pertencentes ao microcódigo final. Assim sendo, utilizamos seqüências de microinstruções, extraídas de um microcódigo completo ou em estágio avançado de desenvolvimento, para simular os diversos estágios de desenvolvimento do microcódigo.

Um "estágio de desenvolvimento virtual" é um estágio de desenvolvimento do microcódigo criado artificialmente para simularmos a evolução do microcódigo e analisarmos a razão de compressão durante a sua evolução. Cada estágio de desenvolvimento virtual é caracterizado por um número de microinstruções, sendo que, o microcódigo nos estágios de desenvolvimento virtuais mais avançados possui mais microinstruções. Por exemplo, para um microcódigo com 2048 microinstruções, uma seqüência com 256 microinstruções representa o microcódigo em seu estágio inicial de desenvolvimento, enquanto que uma seqüência com 1792 microinstruções representa o microcódigo nos estágios finais de desenvolvimento.

Para avaliar a estabilidade da razão de compressão de um dado microcódigo comprimimos o microcódigo em diferentes estágios de desenvolvimento virtual da seguinte forma:

- Removemos as seqüências de microinstruções repetidas do microcódigo original. Apesar de estarem em estágios de desenvolvimento avançados, alguns microcódigos ainda possuem seqüências de endereços com microinstruções repetidas. Essas repetições afetam significativamente a razão de compressão em seqüências pequenas de microinstruções, portanto, não podem ser utilizadas para representar o microcódigo nos diferentes estágios de desenvolvimento virtual.

- Determinamos a quantidade e o tamanho das seqüências em cada estágio de desenvolvimento virtual de acordo com o número de microinstruções do microcódigo completo e o tempo de execução do algoritmo de compressão.

- Para cada estágio de desenvolvimento virtual, extraímos e comprimimos 10 seqüências de microinstruções de posições aleatórias do microcódigo.

Utilizamos a metodologia descrita acima para avaliar a estabilidade da razão de compressão nos microcódigos da Tabela 5.1. Após a remoção das seqüências de microinstruções repetidas, restaram 21 077 microinstruções no microcódigo A, de onde extraímos seqüências com 256, 768, $\cdots$, 20 736 e 21 077 microinstruções, totalizando 42 estágios de desenvolvimento virtual. Para cada estágio, extraímos e comprimimos 10 seqüências de microinstruções de posições aleatórias do microcódigo. A Figura 5.4 mostra as razões de compressão mínima, média e máxima obtidas em cada estágio de desenvolvimento virtual do microcódigo A.

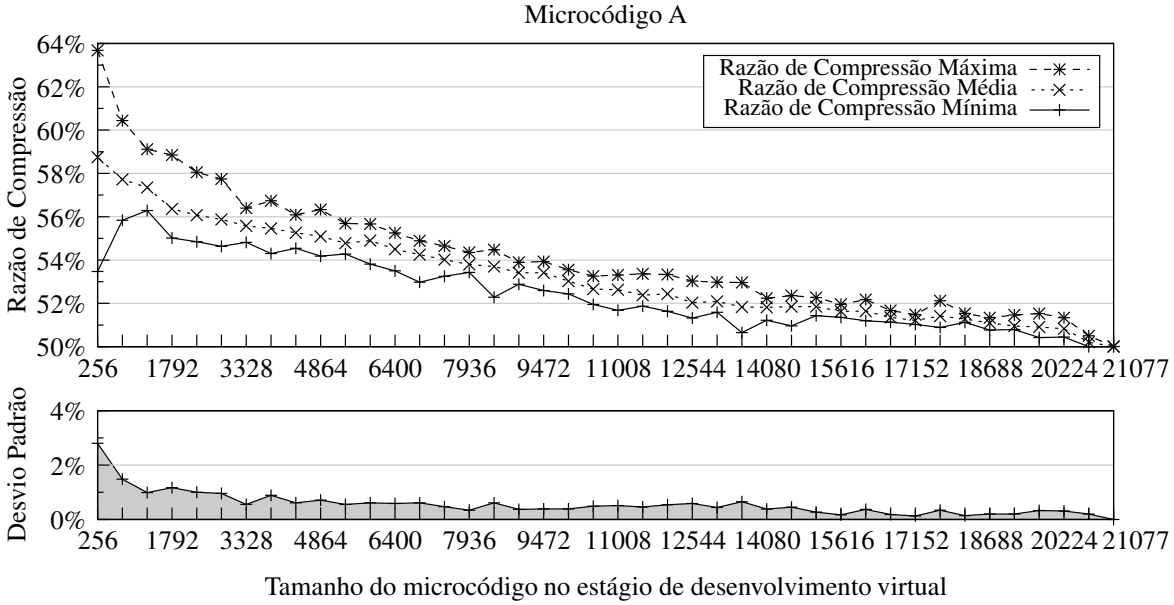


Figura 5.4: Razões de compressão mínima, média e máxima do microcódigo A em seus diversos estágios de desenvolvimento virtual.

O gráfico da Figura 5.4 mostra que a razão de compressão média decresce suavemente à medida que o microcódigo A aumenta de tamanho. Assim sendo, há uma tendência de que a razão de compressão do microcódigo nos primeiros estágios de desenvolvimento seja maior do que na versão final, o que favorece o projetista no processo de previsão da área final da μROM. Mesmo no pior caso, onde a menor razão de compressão obtida no estágio inicial (53,5%) seria utilizada para prever a compressão do microcódigo no estágio final, a previsão é maior do que a razão de compressão real no estágio final. Para o último estágio de desenvolvimento virtual, com 21 077 microinstruções, geramos apenas uma seqüência com todas as microinstruções. Conseqüentemente, as razões de compressão máxima, média e mínima possuem o mesmo valor.

Outra característica observada no gráfico da Figura 5.4 é que, à medida que o microcódigo cresce, as razões de compressão máxima e mínima se aproximam. Um dos motivos para esse efeito é o aumento da quantidade de microinstruções em comum nas seqüências extraídas. À medida que aumentamos o tamanho das amostras extraídas do microcódigo, a quantidade de

microinstruções em comum também aumenta. As microinstruções em comum nas seqüências produzem padrões de *bits* semelhantes, conseqüentemente, as razões de compressão se tornam parecidas. Por exemplo, sejam S_1 e S_2 duas seqüências com 19 200 microinstruções extraídas do microcódigo A. Se S_1 possui as microinstruções contidas entre os endereços 215 e 19 414 e S_2 entre os endereços 1 820 e 21 019, pelo menos 17 595 microinstruções em S_1 são iguais às microinstruções contidas em S_2. As microinstruções em comum nas seqüências S_1 e S_2 produzem padrões de *bits* semelhantes e tornam as razões de compressão parecidas. Outro motivo para que as razões de compressão máxima e mínima se aproximem é que, à medida que as amostras aumentam de tamanho, a representatividade do conjunto de microinstruções se torna maior e os conjuntos de padrões de *bits* gerados tornam-se mais próximos do microcódigo final.

Os resultados para o microcódigo B, apresentados na Figura 5.5, também apresentam uma melhora suave na razão de compressão média à medida que o microcódigo aumenta de tamanho.

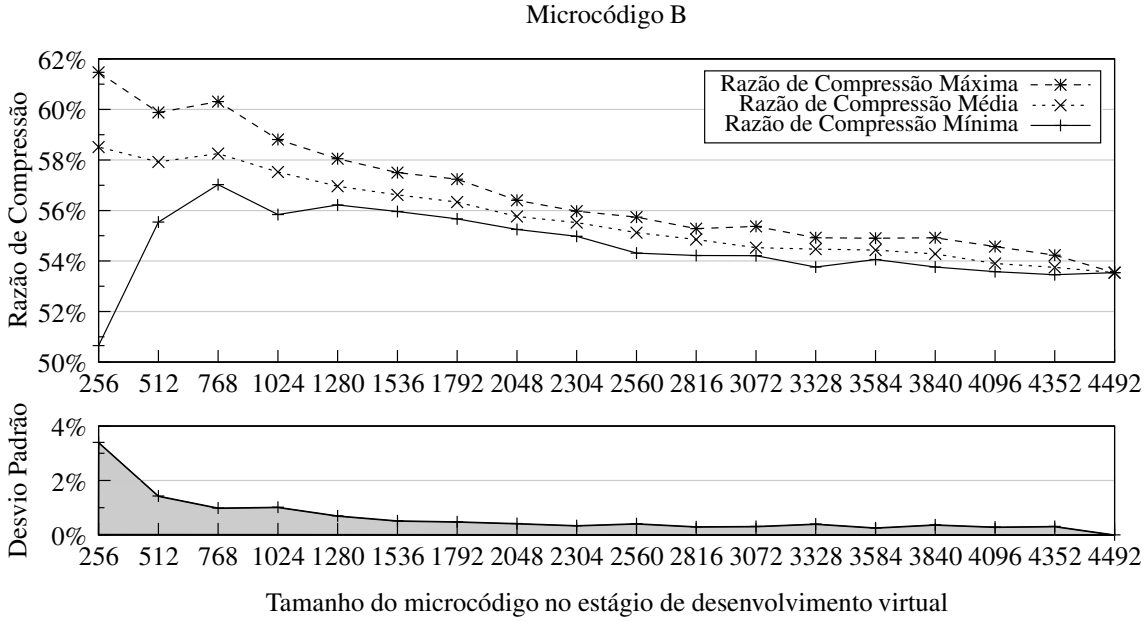


Figura 5.5: Razões de compressão mínima, média e máxima do microcódigo B em seus diversos estágios de desenvolvimento virtual.

Observe que a melhor razão de compressão obtida no estágio inicial no microcódigo B (50,65%) é menor do que a razão de compressão do microcódigo final (53,54%). Nesse cenário, a previsão da área da μROM realizada nos estágios iniciais do projeto pode ser menor do que a área necessária ao final do projeto. Entretanto, a diferença da razão de compressão é de apenas 2,9%. Adicionalmente, de acordo com o desvio padrão das razões de compressão (Figura 5.5), a maioria das razões de compressão encontradas no primeiro estágio de desenvolvimento virtual dista de 3,4% da média, ou seja, está entre 61,9% e 55,1%. Assim sendo, há uma grande chance

do projetista encontrar uma razão de compressão inicial maior do que a final. Também podemos observar que a partir do segundo estágio de desenvolvimento virtual, ou seja, ainda no início do projeto, todas as razões de compressão são maiores do que a razão de compressão final.

O gráfico da Figura 5.6 apresenta os resultados para o microcódigo C. Da mesma forma que nos microcódigos A e B, os resultados do microcódigo C apontam uma melhora suave na razão de compressão média à medida que o microcódigo aumenta de tamanho. Observe também que a melhor razão de compressão no estágio inicial é maior do que a razão de compressão no estágio final de desenvolvimento.

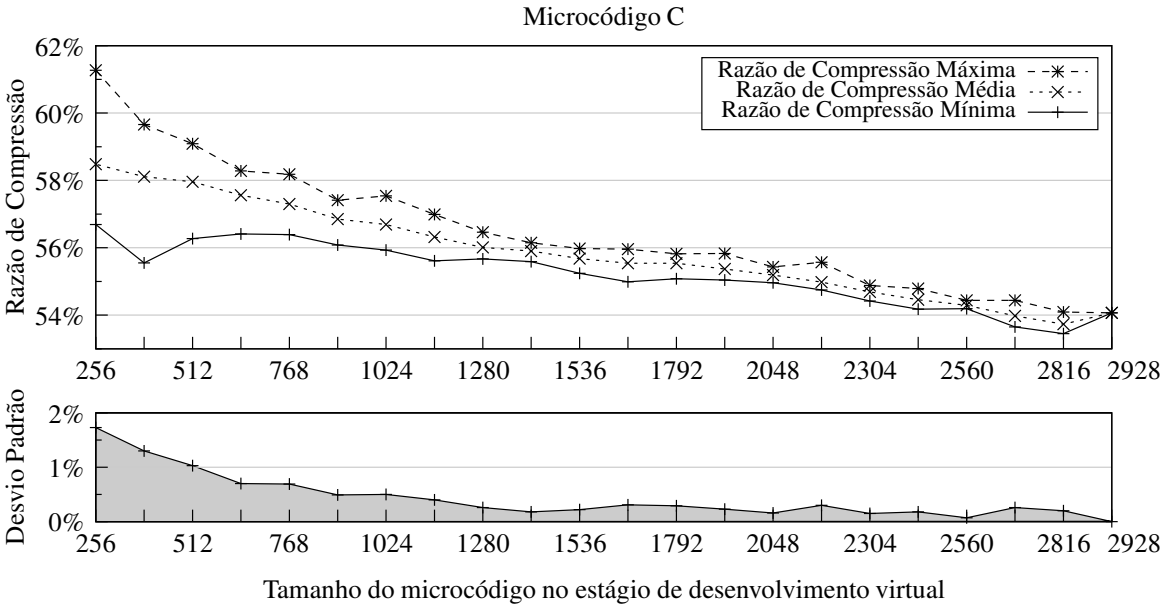


Figura 5.6: Razões de compressão mínima, média e máxima do microcódigo C em seus diversos estágios de desenvolvimento virtual.

A Figura 5.7 apresenta os resultados para o microcódigo D. Diferente dos resultados obtidos com os outros microcódigos, podemos observar um pequeno aumento da razão de compressão média nos últimos estágios de desenvolvimento. Esse aumento ocorre porque as microinstruções no início e no fim desse microcódigo possuem poucos padrões de *bits* em comum e as seqüências de microinstruções nos estágios finais de desenvolvimento são grandes e incluem microinstruções do início e do fim do microcódigo. A razão de compressão média é modificada suavemente à medida que o microcódigo cresce e o aumento nos últimos estágios de desenvolvimento é pequeno, em torno de 1,1%.

Outra característica que difere os resultados do microcódigo D dos outros resultados é que a melhor razão de compressão no estágio inicial (54,5%) é quase 10% menor do que a razão de compressão no estágio final (64,4%). Se esse valor for utilizado pelo projetista para alocar a área final do descompressor, haveria o risco do microcódigo comprimido não caber dentro da

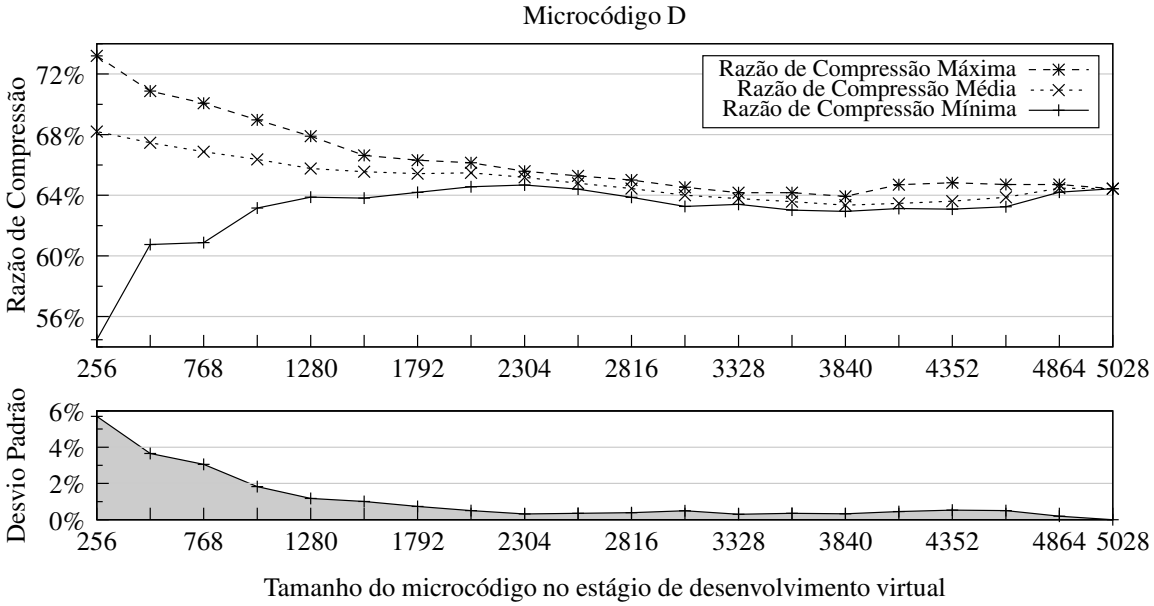


Figura 5.7: Razões de compressão mínima, média e máxima do microcódigo D em seus diversos estágios de desenvolvimento virtual.

área alocada.

Apesar da melhor razão de compressão no estágio inicial ser quase 10% menor do que a razão de compressão no estágio final, uma análise dos resultados revelou que apenas uma seqüência de microcódigo apresenta uma razão de compressão tão baixa e na grande maioria dos casos as razões de compressão não são menores do que a razão de compressão final (64,4%). De fato, se considerarmos uma margem de tolerância de 1,5% para a previsão da razão de compressão final, ou seja, um limitante mínimo de 63%, apenas 5 resultados, um no primeiro estágio e dois no segundo e no terceiro estágio de desenvolvimento, estariam abaixo do limitante mínimo de 63%. Esses resultados apontam que a partir do quarto estágio de desenvolvimento virtual, ou seja, ainda no início do projeto, as razões de compressão obtidas seriam no pior caso 1,5% menor do que a razão de compressão do microcódigo final e uma possível previsão da área final estaria dentro do limite seguro de 1,5%.

Os resultados apresentados nesta seção indicam que a técnica de compressão do microcódigo em dois níveis é estável e não altera significativamente a razão de compressão em função de modificações no conteúdo do microcódigo. Outra característica importante é a capacidade de se estimar, com segurança, a razão de compressão final. Mesmo no pior cenário, apresentado a partir dos resultados do microcódigo D, a razão de compressão do microcódigo final é próxima ou menor do que as razões de compressão obtidas em estágios iniciais de desenvolvimento.

5.2 Compressão com Restrições

O projeto e a implementação do vetor de apontadores e dos dicionários são influenciados por restrições impostas pela tecnologia utilizada para implementar a μROM e pela organização interna do microprocessador. O formato da área alocada para a μROM é um dos fatores que influenciam na quantidade e no formato ideal dos dicionários.

Na compressão sem restrições, as heurísticas têm mais flexibilidade para explorar a entropia do microcódigo e alcançar melhores razões de compressão. Por outro lado, os algoritmos de agrupamento de colunas geram conjuntos com tamanhos arbitrários, com número de colunas e padrões variáveis. Dessa forma, a quantidade e o tamanho dos dicionários, gerados a partir dos conjuntos de colunas, não podem ser controlados e o posicionamento desses dentro da área alocada para a μROM pode se tornar um desafio a parte.

O algoritmo AKL com restrições, descrito na Seção 4.3.1, nos permite impor restrições no número e no tamanho dos conjuntos de colunas durante o agrupamento das colunas do microcódigo. Essas restrições podem ser utilizadas para moldar o formato final do descompressor. Por exemplo, baseado no formato da área disponível para a μROM, o projetista de um microprocessador decide comprimir o microcódigo com dois dicionários, sendo que um deles tem 10 e o outro 60 colunas. Assim sendo, o algoritmo AKL com restrições pode ser utilizado para agrupar as colunas do microcódigo em dois conjuntos com 10 e 60 colunas.

As restrições utilizadas para moldar o formato do descompressor no processo de compressão do microcódigo são impostas principalmente pelo projeto do microprocessador. Dessa forma, para realizar experimentos reais de compressão com restrições nos microcódigos da Tabela 5.1, deveríamos utilizar informações do projeto do microprocessador, como a área disponível para a μROM e a área ocupada pelo *hardware* em função do número de *bits* dos dicionários. Entretanto, essas informações são confidenciais e não tivemos acesso a elas. Assim sendo, não realizamos experimentos de compressão com restrições para moldar o descompressor desses microcódigos. Por outro lado, realizamos estudos de casos utilizando o algoritmo de agrupamento com restrições no número de dicionários e na similaridade dos dicionários.

A Seção 5.2.1 utiliza as restrições no número de conjuntos para avaliar o comportamento da razão de compressão em função do número de dicionários. A Seção 5.2.2 discute o problema do projeto do *hardware* dos dicionários e propõe a reutilização de um mesmo projeto de memória para implementar o *hardware* dos diferentes dicionários da compressão.

5.2.1 Compressão do Microcódigo versus Número de Dicionários

A quantidade de dicionários utilizada na compressão influi diretamente no tamanho final do microcódigo comprimido. Nesta seção, avaliamos a compressão dos microcódigos da Tabela 5.1 em função do número de dicionários utilizados.

Além de prover os melhores resultados de compressão (Seção 5.1.1), o algoritmo AKL permite restringir o número de conjuntos durante o agrupamento de colunas. Desse modo, utilizamos esse algoritmo para avaliar a compressão dos microcódigos em função do número de dicionários.

Para restringir o número de conjuntos no algoritmo AKL, fornecemos um agrupamento inicial, com o número desejado de conjuntos, e restringimos a quantidade mínima de colunas por conjunto a uma coluna. Essa restrição não permite que os conjuntos sejam esvaziados durante a movimentação de colunas. Dessa forma, o agrupamento final tem o mesmo número de conjuntos que o agrupamento inicial. A Figura 5.8 compara o tamanho dos dicionários e dos vetores de apontadores quando variamos a quantidade de dicionários (K) na compressão dos microcódigos A, B, C e D.

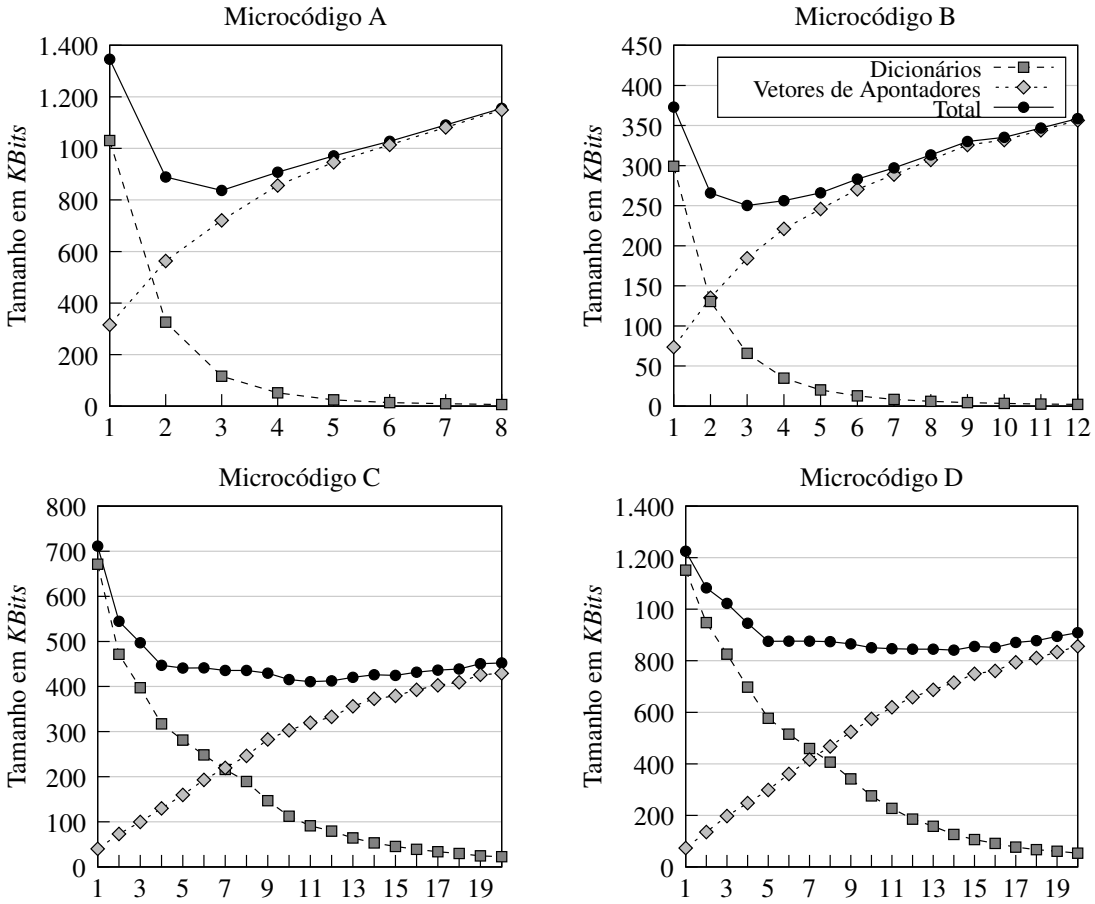


Figura 5.8: Tamanho em *bits* dos dicionários e dos vetores de apontadores em função do número de dicionários (K).

Os gráficos apresentados na Figura 5.8 mostram que, à medida que aumentamos o número de dicionários, a quantidade de *bits* utilizada pelos vetores de apontadores aumenta e o tamanho

acumulado dos dicionários diminui. De fato, a análise realizada na Seção 3.2 já sugeria esse efeito durante o particionamento do microcódigo.

A curva com a linha sólida, no gráfico da Figura 5.8, representa o tamanho total do microcódigo comprimido, ou seja, é a soma dos tamanhos dos vetores de apontadores e dos dicionários. Observe que, inicialmente, à medida que aumentamos o número de dicionários, o tamanho do microcódigo comprimido diminui e, após atingir um ponto mínimo, o tamanho volta a aumentar. Por exemplo, quando usamos até três dicionários, o tamanho do microcódigo A diminui e, a partir desse ponto, o tamanho volta a aumentar. Essa queda se deve a uma redução mais acentuada no tamanho dos dicionários no início do gráfico. De fato, no microcódigo A, os dicionários têm seu tamanho reduzido em 704 781 *bits* quando aumentamos de um para dois a quantidade de dicionários utilizados na compressão enquanto que, quando aumentamos de dois para três a quantidade de dicionários, o tamanho é reduzido em apenas 209 567 *bits*. A redução acentuada no tamanho dos dicionários no início do gráfico acontece porque as colunas que não são similares podem ser posicionadas em dicionários distintos quando aumentamos o número de dicionários. Por outro lado, a partir de um certo ponto, quando já existem dicionários suficientes para separar essas colunas, a adição de novos dicionários não reduz significativamente a quantidade total de padrões de *bits*. Como conseqüência, a redução do tamanho dos dicionários passa a ser menor do que o aumento dos vetores de apontadores e o tamanho do microcódigo comprimido volta a aumentar.

Além de variar o número de dicionários, investigamos a influência das colunas não-comprimidas na compressão do microcódigo. A Figura 5.9 mostra a razão de compressão dos microcódigos A, B, C e D quando o algoritmo AKL permite que algumas colunas não sejam comprimidas (# colunas não-comprimidas > 0) e quando todas as colunas devem ser comprimidas (# colunas não-comprimidas = 0). De acordo com o modelo de compressão apresentado na Seção 3.3.1, as colunas não-comprimidas são colocadas diretamente no vetor de apontadores.

Os gráficos da Figura 5.9 apresentam diferenças na razão de compressão quando o algoritmo permite que algumas colunas não sejam comprimidas e quando todas as colunas devem ser comprimidas. Por exemplo, no microcódigo A, quando todas as colunas são comprimidas em um único dicionário a razão de compressão é de 80% e quando parte das colunas não é comprimida a razão de compressão é de 56%. Essa diferença de 24% acontece porque, quando todas as colunas são comprimidas em um único dicionário, as colunas que não são similares muitos padrões de *bits* e aumentam o tamanho do dicionário. Por outro lado, na compressão com algumas colunas não-comprimidas, o algoritmo de agrupamento de colunas posiciona as colunas que não são similares diretamente no vetor de apontadores, mantendo o dicionário pequeno, com poucos padrões de *bits*.

À medida em que aumentamos o número de dicionários e o tamanho do microcódigo comprimido diminui, há uma aproximação das razões de compressão obtidas na compressão com e sem colunas não-comprimidas. A aproximação dos resultados acontece porque as colunas que

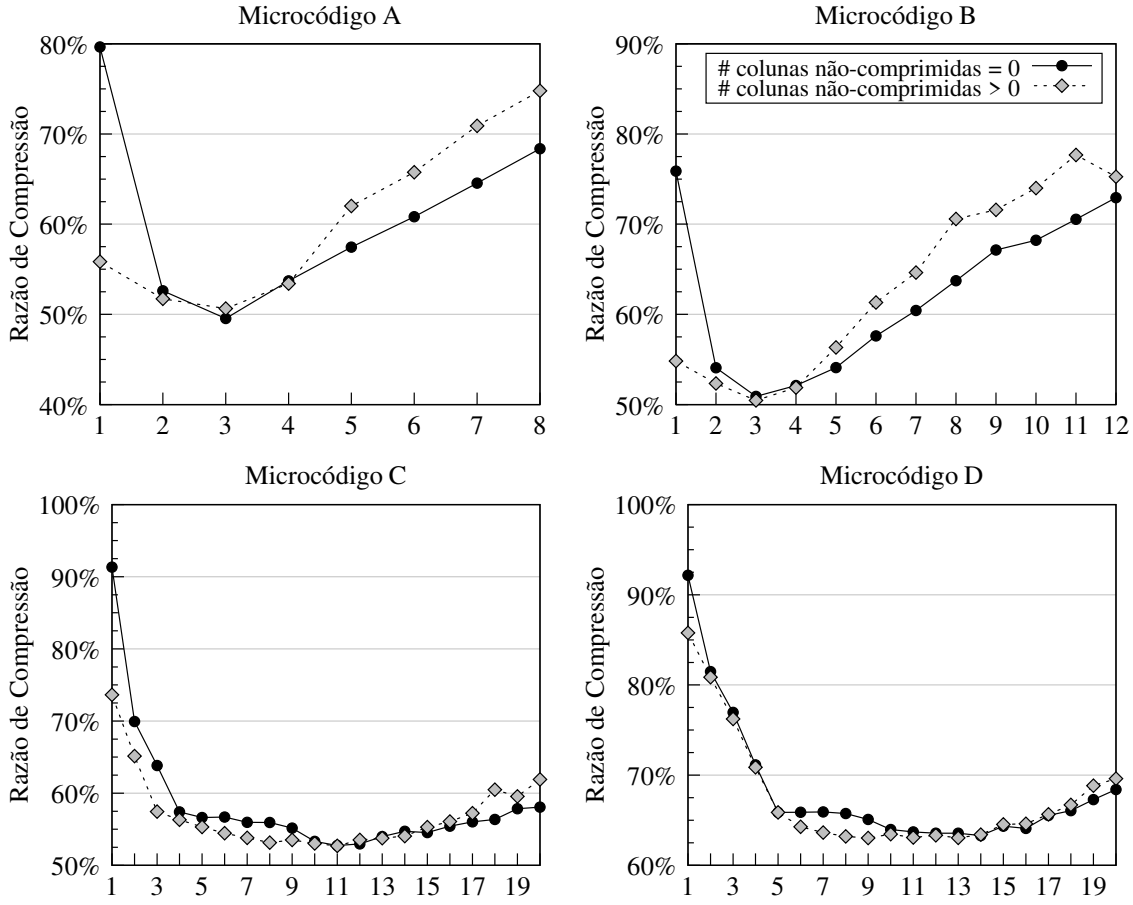


Figura 5.9: Razão de compressão versus quantidade de conjuntos (K).

não seriam comprimidas, por não serem similares a outras colunas, agora podem ser comprimidas em outros dicionários. De fato, verificamos uma redução no número de colunas não-comprimidas com o aumento do número de conjuntos, o que significa que o algoritmo AKL posicionou essas colunas em algum dicionário. Quando o tamanho do microcódigo volta a aumentar, com a adição de novos dicionários, a razão de compressão com colunas não-comprimidas se torna pior do que a razão de compressão sem colunas não-comprimidas. Esse efeito inverso ocorre porque, mesmo com um número suficiente de dicionários para separar as colunas que não são similares, o algoritmo AKL não comprimiu algumas das colunas do microcódigo.

Os resultados apresentados nesta seção indicam que a capacidade de compressão da técnica é sensível ao número de dicionários utilizados para se comprimir o microcódigo. À medida que adicionamos dicionários ao descompressor, o tamanho dos vetores de apontadores aumenta e o tamanho dos dicionários diminui. Inicialmente, a adição de dicionários provê uma redução mais significativa no tamanho dos dicionários e resulta em uma redução do tamanho total do microcódigo comprimido. Por outro lado, a partir de um certo ponto, quando já existem dicio-

nários suficientes para separar as colunas que não são similares, a adição de novos dicionários não reduz significativamente a quantidade total de padrões de *bits* e o aumento dos vetores de apontadores sobrepõe a redução do tamanho dos dicionários, causando um aumento no tamanho do microcódigo comprimido. Também observamos que a capacidade de se posicionar colunas diretamente no vetor de apontadores, sem comprimí-las, permite reduzir a quantidade de padrões de *bits* na compressão com poucos dicionários.

5.2.2 Regularidade no Projeto do Descompressor

O projeto e a implementação da máquina de descompressão do microcódigo requerem a utilização de blocos de memória para o vetor de apontadores e para os dicionários. Apesar das ferramentas de CAD serem capazes de gerar esses blocos de maneira automática a partir de descrições de alto nível, em alguns casos o resultado não é satisfatório e os blocos são desenvolvidos manualmente por projetistas especializados. Processadores com restrições críticas de desempenho e consumo de potência são exemplos de projetos que ainda exigem o projeto manual desses blocos.

Infelizmente o projeto manual de *hardware* é extremamente custoso em termos de recursos humanos e tempo. Para minimizar esse custo, os projetistas muitas vezes recorrem à reutilização de módulos previamente desenvolvidos. Por exemplo, é possível criar um bloco de memória com 1024×32 *bits* a partir de dois blocos de 512×32 *bits*, previamente projetados e testados, e um pequeno circuito combinacional.

Para se reduzir o custo e o tempo de projeto do *hardware* do descompressor propomos reutilizar um mesmo projeto de memória para implementar os diferentes dicionários. Dessa forma, uma única ROM seria projetada para implementar os dicionários. A desvantagem dessa abordagem é que o descompressor pode ter diversos dicionários com tamanhos diferentes e o *hardware* pode ser subutilizado, afetando a razão de compressão do microcódigo e a área final do descompressor.

Como exemplo, suponha que um determinado microcódigo comprimido possua dois dicionários, D_1 e D_2. Se D_1 tem 28 colunas e 300 padrões e D_2 tem 30 colunas e 200 padrões, poderíamos projetar um bloco de memória com 30 colunas e 300 linhas e reutilizá-lo para implementar os dicionários D_1 e D_2. Observe que D_1 não utilizaria duas das 30 colunas do bloco de memória e D_2 só utilizaria 200 das 300 linhas. Essa sobra representa área extra não utilizada e uma conseqüente piora na razão de compressão. A Figura 5.10 mostra um descompressor onde os dois dicionários são implementados com o mesmo bloco de *hardware*. A área hachurada mostra a região do bloco de *hardware* não utilizada pelos dicionários.

Para minimizar a perda de área causada pelo reuso do bloco de *hardware*, é necessário gerar dicionários, ou conjuntos de colunas, com tamanhos similares. A próxima seção utiliza o algoritmo de agrupamento de colunas AKL para gerar grupos de colunas com tamanhos similares.

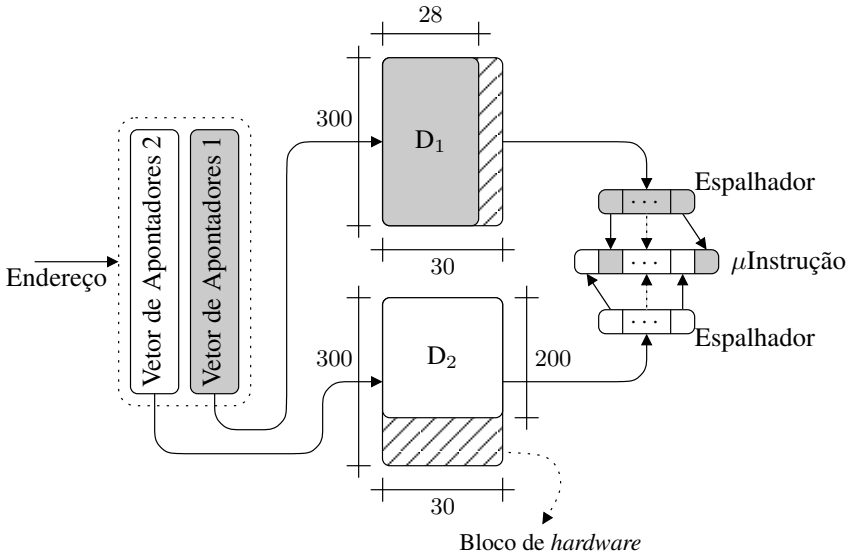


Figura 5.10: Dicionários implementados com o mesmo bloco de *hardware*.

5.2.3 Compressão de Microcódigo Padronizando Dicionários

O algoritmo AKL com restrições, apresentado na Seção 4.3.1, permite realizar o agrupamento de colunas com restrições no número de conjuntos (K) e na quantidade mínima (Min_Cols_i) e máxima (Max_Cols_i) de colunas em cada conjunto C_i. A partir dessas restrições, forçamos os conjuntos a terem o mesmo número de colunas, conseqüentemente, os dicionários terão a mesma largura.

Para medir a capacidade de padronização dos dicionários do algoritmo AKL com restrições, utilizamos duas medidas de razão de compressão: *razão de compressão comum* (RC_{com}) e *razão de compressão padronizada* (RC_{pad}). A razão de compressão padronizada utiliza o tamanho do maior dicionário para computar o tamanho do microcódigo comprimido.

Por exemplo, supondo que o microcódigo da Figura 5.10 tenha 2 000 linhas e 58 colunas, a razão de compressão padronizada é:

$$RC_{pad} = \frac{\overbrace{(\lceil log_2(300) \rceil + \lceil log_2(200) \rceil) \times 2\,000}^{\text{Tam. do Vetor de Apontadores}} + \overbrace{\mathbf{300 \times 30 + 300 \times 30}}^{\text{Tam. dos Dicionários}}}{58 \times 2\,000} = 44,83\% \quad (5.1)$$

enquanto que a razão de compressão comum é:

$$RC_{com} = \frac{\overbrace{(\lceil log_2(300) \rceil + \lceil log_2(200) \rceil) \times 2\,000}^{\text{Tam. do Vetor de Apontadores}} + \overbrace{\mathbf{300 \times 28 + 200 \times 30}}^{\text{Tam. dos Dicionários}}}{58 \times 2\,000} = 41,72\% \quad (5.2)$$

Note que, no exemplo anterior, existe uma diferença de 3% entre a razão de compressão padronizada e a comum devido à diferença no tamanho dos dicionários. Essa diferença in-

dica a área em silício desperdiçada devido à subutilização do *hardware* quando os dicionários reutilizam o mesmo projeto de memória.

Da mesma forma que no exemplo anterior, os resultados obtidos com a compressão dos microcódigos da Tabela 5.1 também apresentam diferenças nos tamanhos dos dicionários. A Tabela 5.5 mostra as melhores razões de compressão comum (obtidas a partir da compressão sem restrições) e as respectivas razões de compressão padronizada para os microcódigos da Tabela 5.1.

Tabela 5.5: Melhores razões de compressão comum e as respectivas razões de compressão padronizada para a compressão sem restrições dos microcódigos A, B, C e D.

Microcódigo	Compressão Sem Restrições		
	RC_{com}	RC_{pad}	Diferença
A	49,54%	52,83%	3,30%
B	50,46%	58,93%	8,47%
C	52,70%	61,29%	8,59%
D	63,03%	85,65%	22,62%

A última coluna da Tabela 5.5 apresenta a diferença entre a razão de compressão comum e a padronizada. Observe que, no microcódigo D, a razão de compressão padronizada é quase 23% maior do que a razão de compressão comum. Novamente, essa diferença ocorre em função dos tamanhos irregulares dos dicionários. O microcódigo D foi comprimido com nove dicionários, sendo que, o segundo dicionário possui o maior número de colunas (33) e o terceiro dicionário possui o maior número de padrões (1728). Dessa forma, a razão de compressão padronizada foi computada utilizando-se dicionários com 33×1728 *bits*. A Tabela 5.6 mostra a configuração dos dicionários resultante da compressão sem restrições desse microcódigo.

Resultados da Padronização de Dicionários

A partir do algoritmo de agrupamento modificado para padronizar os dicionários, comprimimos os microcódigos A, B, C e D utilizando restrições no número de dicionários e na quantidade de colunas por dicionário. Seja K o número de dicionários utilizados na compressão e L_{fixo} a quantidade fixa de colunas em cada dicionário, a tupla $\{K, L_{fixo}\}$ determina a configuração da compressão com restrições. Por exemplo, a configuração $\{2, 40\}$ restringe o agrupamento de colunas a dois conjuntos com 40 colunas cada, ou seja, o descompressor possuirá dois dicionários com 40 colunas cada. Assim sendo, utilizamos o algoritmo AKL para agrupar as colunas dos microcódigos da seguinte maneira:

- Com base nos resultados da compressão do microcódigo versus o número de dicionários (Figura 5.8) limitamos o número máximo de dicionários ($K_{Máximo}$) para comprimir o

Tabela 5.6: Configuração dos dicionários para a compressão do microcódigo D sem padronização dos dicionários.

Dicionário	# Colunas	# Padrões	*Bits* para endereçamento
1	30	1018	10
2	**33**	923	10
3	20	**1728**	11
4	18	501	9
5	18	968	10
6	30	995	10
7	20	999	10
8	32	863	10
9	14	940	10
Colunas Não-Comprimidas	21	—	—

microcódigo com restrições. Dessa forma, comprimimos os microcódigos A e B com até 10 dicionários e os microcódigos C e D com até 20 dicionários.

- Para i variando de 2 até $K_{Máximo}$, e j de um até $L \div i$, geramos as configurações $\{i, j\}$. Por exemplo, para o microcódigo B, com $L = 80$ colunas, geramos as configurações $\{2,1\}, \{2,2\}, \{2,3\}, \cdots, \{2,40\}, \{3,1\}, \cdots, \{3,26\}, \{4,1\}, \cdots, \{10,8\}$.

- A partir das configurações geradas, agrupamos as colunas do microcódigo utilizando o algoritmo AKL com restrições. As colunas não agrupadas são posicionadas diretamente no vetor de apontadores.

A Tabela 5.7 mostra as melhores razões de compressão padronizada, obtidas a partir da compressão com restrições no número de colunas por dicionário. As respectivas razões de compressão comum e o número de colunas dos dicionários também são apresentados. Por exemplo, o microcódigo A foi comprimido com dois dicionários com 32 colunas cada e a diferença entre a razão de compressão padronizada e a comum é de apenas 0,52%. Observe que, na compressão com restrições, a razão de compressão padronizada dos microcódigos é melhor do que a razão de compressão padronizada obtida na compressão sem restrições (Tabela 5.5). Esse resultado indica que o algoritmo AKL com restrições foi capaz de produzir conjuntos de colunas similares, melhorando a padronização dos dicionários.

As restrições utilizadas para comprimir os microcódigos não limitam o número de padrões em cada dicionário. No entanto, os resultados da Tabela 5.7 apontam que a restrição no número de colunas de cada dicionário já é suficiente para produzir dicionários com quantidades de padrões aproximadas. De fato, a pequena diferença entre as razões de compressão padronizada

Tabela 5.7: Melhores razões de compressão padronizada e as respectivas razões de compressão comum após a compressão com padronização dos dicionários.

Microcódigo	# Dicionários	# Colunas por Dicionário	Compressão Com Restrições		
			RC_{pad}	RC_{com}	Diferença
A	2	32	51,67%	51,15%	0,52%
B	3	24	51,15%	51,12%	0,03%
C	8	26	55,07%	53,66%	1,42%
D	9	22	66,50%	66,33%	0,17%

e comum, nos resultados da Tabela 5.7, indicam que o número de padrões nos dicionários é parecido.

Definimos o "custo da padronização dos dicionários" como sendo a diferença entre a melhor razão de compressão padronizada e a melhor razão de compressão comum. Esse custo indica a área extra necessária para se comprimir o microcódigo utilizando um mesmo projeto de memória para os diferentes dicionários. A Tabela 5.8 resume os resultados obtidos a partir da compressão com e sem restrições e mostra o custo da padronização dos dicionários em cada microcódigo.

Tabela 5.8: Custo da padronização dos dicionários.

Microcódigo	Compressão Sem Restrições		Compressão Com Restrições		Custo da Padronização dos Dicionários
	RC_{com}	RC_{pad}	RC_{com}	RC_{pad}	
A	49,54%	52,83%	51,15%	51,67%	2,13%
B	50,46%	58,93%	51,12%	51,15%	0,69%
C	52,70%	61,29%	53,66%	55,07%	2,37%
D	63,03%	85,65%	66,33%	66,50%	3,47%

Os resultados da Tabela 5.8 indicam que o custo da padronização dos dicionários é pequeno. Isso mostra que, além de padronizar os dicionários, o algoritmo AKL com restrições foi capaz de comprimir os microcódigos com boas razões de compressão.

Os resultados apresentados nesta seção mostram que, apesar da compressão sem restrições produzir as melhores razões de compressão, o microcódigo comprimido pode possuir muita variação no tamanho dos dicionários. Conseqüentemente, a reutilização do projeto do bloco de memória para implementar os dicionários causa uma subutilização dos *bits* dos dicionários, o que implica no desperdício de área em silício. Também mostramos que, através do algoritmo AKL com restrições, é possível comprimir os microcódigos de forma que os dicionários produzidos sejam padronizados com a mesma quantidade de colunas e um número parecido de padrões. Essa padronização permite que o projeto de um mesmo bloco de memória seja reutilizado para implementar os diferentes dicionários sem que haja desperdício significativo

de área em silício. As diferenças entre as razões de compressão padronizada, na compressão com restrições, e as razões de compressão comum, na compressão sem restrições, mostram que, além de padronizar os dicionários, o algoritmo AKL com restrições foi capaz de comprimir os microcódigos com boas razões de compressão.

5.3 Conclusões

Neste capítulo comparamos as técnicas de agrupamento de colunas utilizando quatro microcódigos, extraídos de processadores em produção e em fases avançadas de desenvolvimento. Assim como as técnicas de agrupamento de colunas do Capítulo 4, dividimos os experimentos em duas categorias: compressão sem restrições e compressão com restrições.

Na compressão sem restrições, as novas heurísticas, propostas na Seção 4.2, foram comparadas com as outras heurísticas encontradas na literatura [67, 136]. Os resultados da Seção 5.1 mostram que, o algoritmo AKL produziu os melhores resultados de compressão e, em todos os cenários, as heurísticas propostas nesta tese produziram resultados melhores do que as heurísticas dos outros trabalhos.

Na Seção 5.1.1, avaliamos a qualidade das heurísticas de agrupamento de colunas em função do número de colunas no microcódigo. Os resultados apresentados indicam que as heurísticas de ordenação linear, ordenação circular e as heurísticas dos outros autores são sensíveis à quantidade de colunas do microcódigo e, à medida que o número de colunas aumenta, essas heurísticas produzem resultados piores. Por outro lado, o algoritmo AKL se mostrou estável e obteve resultados melhores à medida em que o número de colunas do microcódigo aumentava.

Ainda na compressão sem restrições, fizemos uma análise da estabilidade da razão de compressão em função do crescimento do microcódigo. Os resultados apresentados na Seção 5.1.3 indicam que a técnica de compressão do microcódigo em dois níveis é estável e não altera significativamente a razão de compressão em função de modificações no conteúdo do microcódigo. Outra característica importante é a capacidade de se estimar, com segurança, a razão de compressão final do microcódigo a partir da compressão do microcódigo em seus estágios iniciais de desenvolvimento.

Na compressão com restrições, utilizamos o algoritmo AKL com restrições para avaliar a compressão do microcódigo em função do número de dicionários. Os resultados apresentados na Seção 5.2.1 indicam que a capacidade de compressão da técnica de compressão em dois níveis é sensível ao número de dicionários utilizados para se comprimir o microcódigo. À medida que adicionamos dicionários ao descompressor, o tamanho dos vetores de apontadores aumenta e o tamanho dos dicionários diminui. Inicialmente, a adição de dicionários provê uma redução mais acentuada no tamanho dos dicionários e resulta em uma redução do tamanho total do microcódigo comprimido. Por outro lado, a partir de um certo ponto, quando já existem dicionários suficientes para separar as colunas que não são similares, a adição de novos dicionários

não reduz significativamente a quantidade total de padrões de *bits* e o aumento dos vetores de apontadores sobrepõe a redução do tamanho dos dicionários, causando um aumento no tamanho do microcódigo comprimido. Também observamos que a capacidade de se posicionar colunas diretamente no vetor de apontadores, sem comprimí-las, permite reduzir a quantidade de padrões de *bits* na compressão com poucos dicionários.

A Seção 5.2.2 propõe a reutilização de um mesmo projeto de memória para implementar os diferentes dicionários do descompressor. A reutilização do mesmo projeto de memória permite a redução de custos do projeto em termos de tempo e recursos humanos. Por outro lado, quando os dicionários do descompressor possuem tamanhos diferentes, a reutilização do mesmo projeto de memória causa uma subutilização das memórias que armazenam os dicionários, o que implica no aumento da razão de compressão e no desperdício de área em silício. De fato, a razão de compressão do microcódigo D aumenta de 63,03% para 85,65% quando o mesmo projeto de memória é utilizado para implementar os diferentes dicionários do descompressor. Dessa forma, é importante que o microcódigo comprimido possua dicionários com tamanhos parecidos. Os resultados apresentado na Seção 5.2.3 mostram que, através do algoritmo AKL com restrições, é possível comprimir os microcódigos de forma que os dicionários produzidos sejam padronizados com a mesma quantidade de colunas e um número parecido de padrões. Essa padronização permite que o projeto de um mesmo bloco de memória seja reutilizado para implementar os diferentes dicionários sem que haja desperdício significativo de área em silício. As diferenças entre as razões de compressão padronizada, na compressão com restrições, e as razões de compressão comum, na compressão sem restrições, mostram que, além de padronizar os dicionários, o algoritmo AKL com restrições foi capaz de comprimir os microcódigos com boas razões de compressão.

Capítulo 6

Conclusões

Neste trabalho, investigamos a compressão do microcódigo em processadores de alto desempenho. A partir de uma colaboração entre o Laboratório de Sistemas de Computação (LSC) da Unicamp e o *Programming Systems Lab* (PSL/MTL, Santa Clara, California) da Intel Corporation, foi possível investigar os principais problemas na compressão do microcódigo e propor soluções para estes problemas.

Além de uma boa capacidade de compressão, o projeto de processadores de alto desempenho exige que a técnica de compressão do microcódigo tenha características como: flexibilidade na modificação do microcódigo, pouco atraso de propagação do sinal e um circuito de *hardware* simples. Assim sendo, utilizamos essas restrições como critério para fazer uma análise qualitativa das técnicas de compressão de código e microcódigo. A análise realizada no Capítulo 2 mostra que a técnica de organização do microcódigo em dois níveis é a mais adequada para se comprimir o microcódigo em processadores de alto desempenho.

Na organização do microcódigo em dois níveis, as microinstruções são substituídas por apontadores para dicionários que armazenam os padrões de *bits* extraídos do microcódigo. Os apontadores são armazenados em uma ROM denominada "vetor de apontadores" e os padrões de *bits* residem em ROMs distintas, denominadas "dicionários". Durante a execução, o descompressor utiliza os apontadores armazenados no vetor de apontadores para recuperar padrões de *bits* dos dicionários e reconstruir a microinstrução original. O processo de descompressão do microcódigo inclui acessos indiretos às ROMs dos dicionários, o que poderia prejudicar o desempenho do processador, entretanto, é possível dividir o descompressor com registradores de *pipeline* e minimizar o impacto no desempenho do microprocessador. De fato, testes com um processador de alto desempenho, realizados por engenheiros da Intel, indicam que o descompressor dividido com registradores de *pipeline* causa uma degradação de desempenho média menor do que 0,5%.

A capacidade de compressão da técnica de compressão em dois níveis é sensível ao número de dicionários e à quantidade de colunas e padrões de *bits* em cada dicionário. A técnica também

permite que as colunas do microcódigo sejam agrupadas em conjuntos de forma a reduzir o número de padrões de *bits* nos dicionários. O agrupamento de colunas similares é fundamental para minimizar o número de padrões de *bits* nos dicionários e, conseqüentemente, maximizar a redução do tamanho da μROM.

Intuitivamente, o agrupamento de colunas similares é um problema $\mathcal{NP}$-Difícil. De fato, provamos que, quando o número de colunas por dicionário e a quantidade de dicionários são fixos, o problema é $\mathcal{NP}$-Completo. Esse resultado indica a necessidade de heurísticas para realizarmos o agrupamento das colunas do microcódigo de forma eficiente. Assim sendo, propusemos quatro heurísticas para agrupar as colunas do microcódigo: colunas seqüenciais, ordenação linear, ordenação circular, e o algoritmo AKL.

As heurísticas propostas, juntamente com as outras heurísticas encontradas na literatura [67, 136], foram implementadas e avaliadas utilizando-se quatro microcódigos, extraídos de processadores em produção e em fases avançadas de desenvolvimento. Em todos os experimentos as nossas heurísticas alcançaram resultados superiores aos resultados das heurísticas propostas pelos outros autores.

Ainda neste trabalho, identificamos a necessidade de se comprimir o microcódigo com restrições no número de dicionários e na quantidade de colunas por dicionário. Também modificamos o algoritmo AKL para agrupar colunas sob estas restrições. Os resultados experimentais mostram que, mesmo sob restrições, o algoritmo AKL é capaz de produzir bons resultados de compressão.

6.1 Trabalhos Futuros

Juntamente com a redução da área da μROM, a compressão de microcódigo produz diversos efeitos que influenciam no projeto e na implementação de um microprocessador de alto desempenho. O consumo de energia, a previsibilidade da área da μROM, o desempenho do descompressor ou mesmo a flexibilidade na modificação do microcódigo sem alterar a estrutura do descompressor são exemplos de fatores que influenciam o projeto do microprocessador. Apesar de explorarmos alguns desses fatores nesta tese, a quantidade de experimentos e variáveis envolvidas é muito grande e ainda há muito para ser investigado. Adicionalmente, existem outras áreas, como compressão de código e dados, que poderiam tirar proveito da técnica de compressão em dois níveis. Assim sendo, reservamos esta parte da tese para indicar possíveis trabalhos futuros.

Na Seção 6.1.1, apresentamos os trabalhos relacionados à técnica de compressão de microcódigo em dois níveis. A estabilidade da razão de compressão, a redução de *bits* versus a área da μROM, o consumo de energia e até mesmo o desempenho do descompressor são exemplos de problemas que podem ser estudados. A Seção 6.1.2 indica trabalhos relacionados às técnicas de agrupamento de colunas e propõe experimentos para verificar a qualidade dos resultados das

heurísticas. Por fim, a Seção 6.1.3 sugere a utilização da técnica de compressão em dois níveis para comprimir código e dados.

6.1.1 Compressão de Microcódigo em Dois Níveis

Estabilidade da razão de compressão

Na Seção 1.2, apontamos a necessidade da previsão e da manutenção do tamanho da μROM durante o desenvolvimento do microprocessador. O microcódigo é desenvolvido em paralelo com o microprocessador e cresce de tamanho à medida que o projeto do microprocessador se desenvolve. Apesar do desenvolvimento do microcódigo ocorrer de forma paralela ao desenvolvimento do microprocessador, a quantidade de *bits* e o número de microinstruções no microcódigo final podem ser estimados com base no microcódigo de processadores da mesma família ou semelhantes. Dessa forma, a área a ser ocupada pela μROM no final do projeto pode ser estimada com certo grau de precisão.

A compressão do microcódigo introduz uma nova variável na previsão da área ocupada pela μROM. Além da quantidade de microinstruções, o projetista deve prever a razão de compressão do microcódigo no estágio final de desenvolvimento do microprocessador. Entretanto, durante o processo de desenvolvimento, o microcódigo sofre modificações que podem alterar a sua entropia e, conseqüentemente, a sua razão de compressão. Identificamos dois processos que modificam a entropia do microcódigo: o crescimento do microcódigo, através da adição de novas microinstruções, e a alteração do microcódigo, através de correções nos *bits* das microinstruções.

Crescimento do Microcódigo: Na Seção 5.1.3 investigamos a relação entre a razão de compressão e o crescimento do microcódigo e constatamos que à medida que o microcódigo cresce de tamanho, os padrões se repetem com mais freqüência e a razão de compressão melhora.

Os experimentos realizados na Seção 5.1.3 consideram a estrutura do descompressor como sendo livre, ou seja, é possível modificar a estrutura do descompressor sempre que necessário. Entretanto, em alguns casos, durante o projeto do processador, a organização do descompressor deve ser congelada, ou seja, a quantidade e o tamanho dos dicionários não podem mais sofrer modificações. Este congelamento ocorre antes mesmo do microcódigo ser finalizado. Nesse caso, existe a possibilidade do microcódigo ser modificado após o congelamento do descompressor e a compressão do novo microcódigo não caber dentro do *hardware* de descompressão.

Um possível trabalho, seria investigar o que acontece com a razão de compressão quando a estrutura do descompressor é congelada antes da versão final do microcódigo.

Alterações no Microcódigo: Na Seção 5.1.3, avaliamos a razão de compressão do microcódigo em função do seu crescimento. Entretanto, a entropia do microcódigo pode ser modificada por causa de alterações nos *bits* do microcódigo, sem que haja crescimento. Essas alterações

podem ocorrer com correções no microprograma ou mesmo com alterações no modelo da microarquitetura. Os resultados da Seção 5.1.3 indicam que a razão de compressão varia muito pouco com o crescimento do microcódigo. Apesar do mesmo resultado ser esperado para alterações no microcódigo, poderíamos avaliar a sensibilidade da técnica de compressão a alterações no microcódigo, com mais precisão, utilizando heurísticas para simular alterações no microcódigo. Por exemplo, poderíamos alterar *bits* do microprograma aleatoriamente ou mesmo substituir microinstruções escolhidas aleatoriamente por outras microinstruções válidas, pertencentes a um banco de microinstruções.

Assim como o crescimento do microcódigo, se houver congelamento das estruturas do descompressor as alterações no microcódigo podem fazer com que o microcódigo comprimido não caiba no *hardware* de descompressão. Dessa forma, os mesmos experimentos propostos para avaliar o impacto do crescimento do microcódigo no descompressor poderiam ser feitos para avaliar as alterações no microcódigo.

Compressão com Espaço Extra

Observe que, em algumas das situações anteriores, quando o microcódigo é modificado após o congelamento das estruturas, o número de padrões nos dicionários pode aumentar e o microcódigo comprimido pode não caber dentro do descompressor. Para resolver esse problema poderíamos projetar o descompressor com folga, ou espaço extra, para comportar possíveis alterações na entropia do microcódigo.

Uma forma de comportar os novos padrões, produzidos pela modificação no microcódigo, seria adicionar espaços vazios nos dicionários do descompressor, de forma que, os novos padrões pudessem ser alojados sem a necessidade de se aumentar o *hardware* dos dicionários. A Figura 6.1 mostra um descompressor com espaço extra reservado para a adição de novos padrões. A área hachurada representa o espaço extra acrescentado nos dicionários.

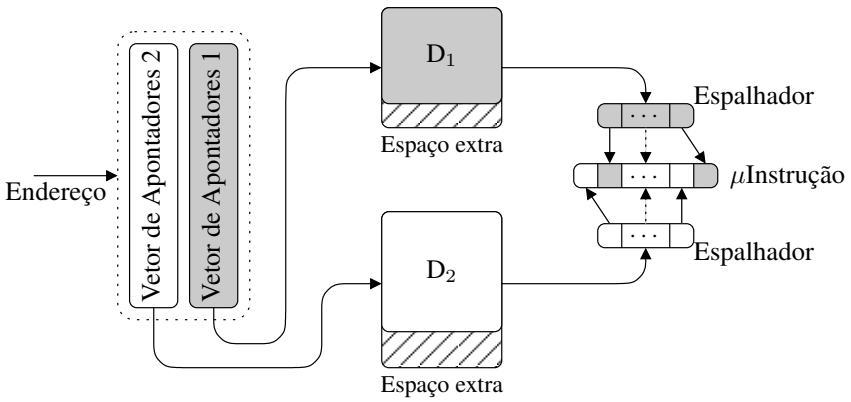


Figura 6.1: Descompressor com espaço extra reservado para a adição de novos padrões.

A adição de espaço extra nos dicionários pode influenciar no tamanho do vetor de apontadores. Isso acontece porque a quantidade de padrões no dicionário determina o número de *bits* necessário para se indexar o dicionário. Por exemplo, vamos supor que após a compressão do microcódigo, o dicionário D_1 possua 500 padrões e o projetista do descompressor deseje adicionar 50 entradas vazias para comportar futuras alterações no microcódigo. A compressão do microcódigo havia reservado 9 *bits* no vetor de apontadores para indexar os 500 padrões do dicionário D_1. Entretanto, se o número de padrões aumentar e ultrapassar 512 unidades, serão necessários 10 *bits* para a indexação. Dessa forma, além de aumentar o número de entradas no dicionário o projetista deve aumentar a quantidade de *bits* no vetor de apontadores.

A introdução de *bits* no vetor de apontadores, após a compressão, pode levar a soluções subótimas. Por exemplo, vamos supor que durante a compressão, o algoritmo de agrupamento de colunas encontrou um agrupamento com dois conjuntos de colunas, cada um com 500 padrões. Os vetores de apontadores possuirão, no total, 18 *bits*, sendo 9 *bits* para indexar cada dicionário. Quando o projetista adicionar espaço extra nos dicionários, digamos 50 entradas vazias, os vetores de apontadores terão sua largura total aumentada para 20 *bits*, ou seja, 10 *bits* por dicionário. Por outro lado, se esse espaço extra fosse considerado pelo algoritmo de agrupamento de colunas, o algoritmo poderia ter encontrado um agrupamento com 700 e 450 padrões que, apesar de parecer pior do que o agrupamento anterior, necessitaria apenas 10+9=19 *bits* no vetor de apontadores, pois a adição de 50 entradas no dicionário com 450 padrões não ocasionaria o aumento do vetor de apontadores. De acordo com o número de microinstruções (N), essa diferença de um *bit* na largura do vetor de apontadores pode ter um impacto grande na razão de compressão e tornar o agrupamento com 700 e 450 padrões melhor do que o agrupamento com 500 e 500 padrões.

Essa diferença no resultado ocorre porque a função objetivo F, definida na Equação 4.1, não reflete necessariamente a minimização do tamanho do microcódigo quando há a adição de espaço extra. Como trabalho futuro, poderíamos estender os algoritmos de agrupamento de colunas para considerar o espaço extra durante o agrupamento das colunas.

Se o espaço extra adicionado for pequeno, existe o risco do microcódigo modificado não caber no descompressor, por outro lado, se o espaço extra for muito grande, a razão de compressão, e conseqüentemente, a área da μROM podem ser comprometidas. Assim sendo, seria interessante a realização de experimentos empíricos para avaliar a melhor métrica de adição de espaço extra.

Redução de *Bits* versus Área

Na Seção 3.5 mostramos que a redução da área da μROM nem sempre é proporcional à redução do número de *bits* do microcódigo. Isso se dá devido à presença de circuitos periféricos, anexos ao núcleo da ROM, que não diminuem proporcionalmente com o tamanho da ROM.

Apesar de não termos acesso à tecnologia de síntese de *hardware* da Intel, poderíamos uti-

lizar ferramentas de síntese de *hardware* comerciais (Leonardo, Design Compiler, entre outros) para avaliar a área do descompressor em função do número de *bits* do microcódigo comprimido.

Outra idéia, seria investigar a possibilidade de se mudar a função objetivo F para minimizar diretamente a área dos vetores de apontadores e dicionários, ao invés do número de *bits*.

Consumo de Energia

O consumo de energia da μROM está diretamente relacionado à área ocupada pela μROM. Dessa forma, um efeito da compressão do microcódigo seria a redução da energia consumida pela μROM. Possivelmente, a quantidade e o tamanho dos dicionários influenciam na quantidade de energia total consumida pelo descompressor. Essa relação poderia ser investigada com o uso de técnicas ou ferramentas que estimam o consumo de energia.

Outro fator que determina o consumo de energia da μROM é a quantidade de *bits* do microcódigo com o valor '1' [7, 32]. Assim sendo, a redução da quantidade de *bits* com o valor '1' poderia reduzir o consumo de energia da μROM. Em [25], propusemos uma técnica que reduz a quantidade de *bits* com o valor '1' através da compressão do microcódigo.

Os padrões de *bits* podem ser posicionados em qualquer endereço dentro dos dicionários. Para isso, basta que os valores armazenados no vetor de apontadores sejam ajustados com o endereço correto do padrão referenciado. A técnica proposta em [25] consiste em reordenar os padrões de *bits* no dicionário, de forma que, a quantidade de *bits* com o valor '1' no vetor de apontadores seja minimizado. Por exemplo, o padrão mais freqüente, ou seja, mais referenciado pelo vetor de apontadores, poderia ser posicionado no endereço zero do dicionário de forma que os apontadores que referenciam este padrão no vetor de apontadores possuam o valor zero. O algoritmo proposto em [25] consiste basicamente em ordenar os padrões do dicionário pela ordem decrescente de freqüência e posicionar os padrões com maior freqüência nos endereços com a menor quantidade de *bits* com o valor '1', ou seja, "0000", "0001", "0010", $\cdots$, "1111".

O microcódigo A, da Tabela 5.1, contém 273 264 *bits* com o valor '1'. A técnica de reordenação de padrões, descrita acima, foi capaz de reduzir o número de *bits* com o valor '1' para 114 549 *bits*, ou seja, obteve uma redução de 58% na quantidade de *bits* com o valor '1'.

A eficiência dessa técnica na redução do consumo de energia também poderia ser investigada com o uso ferramentas de estimativa de consumo de energia.

Desempenho

Na Seção 3.4, apresentamos um esquema de compressão onde registradores de *pipeline* são adicionados entre os vetores de apontadores e os dicionários. Esse esquema evita que o tempo de carga da microinstrução, determinado pelo acesso indireto aos dicionários, influencie no tempo do ciclo do relógio do microprocessador. Por outro lado, a introdução dos registradores de *pipeline*, entre os vetores de apontadores e os dicionários, aumenta em um ciclo o tempo

necessário para se carregar a primeira microinstrução da μROM. Esse ciclo extra, consumido pela primeira microinstrução, também é conhecido como "bolha" no *pipeline* [108].

Testes com um processador de alto desempenho, realizados por engenheiros da Intel, indicam que a "bolha" no *pipeline* causa uma degradação de desempenho média em torno de 0,4%. Apesar do número ser pequeno, é mais um fator negativo para ser pesado contra o benefício proporcionado, em economia de área e energia, pela compressão do microcódigo. Assim sendo, seria importante investigar um modelo de descompressor que não adicionasse a "bolha" no *pipeline*.

No descompressor com *pipeline*, enquanto o endereço da primeira instrução é utilizado para carregar o apontador do vetor de apontadores, os dicionários estão ociosos, ou seja, não são acessados. Uma idéia seria projetar o microprocessador de forma que o endereço fornecido à μROM para carga da seqüência de microinstruções seja um par de endereços que, no primeiro ciclo, carregue simultaneamente a primeira microinstrução dos dicionários e o apontador para a segunda microinstrução do vetor de apontadores. Essa idéia permite remover a "bolha" do *pipeline*. Note que, no entanto, o formato do endereço fornecido à μROM deve ser modificado, alterando parte da estrutura do microprocessador.

Testes com o desempenho do descompressor poderiam ser realizados com simuladores de arquiteturas x86 com precisão de ciclos, como o PTLsim [134].

6.1.2 Agrupamento de Colunas

Dividir o Microcódigo Antes da Compressão

Uma forma de se modificar a geometria da μROM é dividir o conjunto de linhas ao meio e reposicionar as duas metades lado a lado. Dessa forma, uma μROM com $L \times N$ *bits* passa a ter $2L \times N/2$ *bits*. A Figura 6.2 mostra um exemplo onde uma μROM é dividida ao meio e suas metades são reposicionadas lado a lado. Após a reorganização da μROM, duas microinstruções são carregadas ao mesmo tempo em cada acesso à memória e um multiplexador é necessário para selecionar a microinstrução correta, de acordo com o endereço solicitado.

A reorganização apresentada na Figura 6.2 (b) aumenta o número de colunas do microcódigo e cria novas oportunidades para se agrupar colunas parecidas. Outro efeito da reorganização, é a divisão dos padrões em dois grupos distintos. Por exemplo, os padrões da primeira metade da μROM na Figura 6.2 (b) não precisam mais ser armazenados no mesmo dicionário em que os padrões da segunda metade são armazenados.

A divisão da μROM na Figura 6.2 (b) separa as microinstruções com endereços maiores das microinstruções com endereços menores. As microinstruções também poderiam ser separadas de acordo com a paridade do endereço, ou seja, um conjunto possuiria as microinstruções com endereço par e o outro as microinstruções com endereço ímpar. A diferença entre essas duas abordagens é o *bit* de endereço utilizado para classificar as microinstruções. No primeiro caso, o

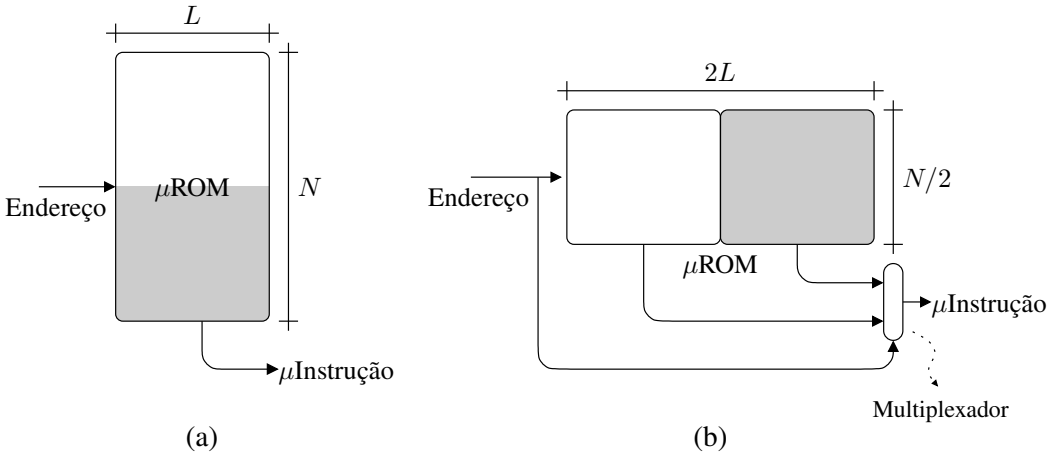


Figura 6.2: Reorganização da μROM para aumentar o número de colunas. (a) μROM original. (b) μROM reorganizada.

bit mais significativo é utilizado e as microinstruções com endereços menores são separadas das microinstruções com endereços maiores. No segundo caso, o *bit* menos significativo é utilizado e as microinstruções com endereços pares são separadas das microinstruções com endereços ímpares. De fato, poderíamos utilizar qualquer *bit* do endereço para separar as microinstruções em dois conjuntos.

Observe que, apesar do exemplo anterior dividir a μROM em duas partes, essa quantidade não precisa ser necessariamente 2 e a μROM poderia ser dividida em 4, 8, 16 ou mais partes.

Um possível trabalho futuro seria investigar o impacto da reorganização da μROM na razão de compressão do microcódigo.

Qualidade dos Resultados dos Algoritmos de Compressão

Na Seção 5.1 avaliamos os algoritmos de agrupamento de colunas sem restrições. Nesse estudo, mostramos que alguns dos algoritmos apresentados no Capítulo 4 não produzem bons resultados quando a quantidade de colunas do microcódigo aumenta. De fato, apenas o algoritmo AKL não produziu resultados piores quando o microcódigo possui mais colunas. Por outro lado, apesar dos resultados de compressão serem promissores, não mostramos o quão distante do ótimo os resultados obtidos se encontram. Um possível trabalho futuro seria determinar o resultado ótimo, ou mesmo um limitante inferior teórico para estimar a qualidade dos resultados das heurísticas.

6.1.3 Outros Trabalhos

Comprimir Dados com a Técnica de Compressão de Microcódigo em Dois Níveis

Alguns circuitos implementados em silício utilizam ROMs para armazenar tabelas de dados. Por exemplo, o *chip* de decodificação de áudio MP3 [9], desenvolvido no Laboratório de Sistemas de Computação (LSC) da Unicamp, utiliza três tabelas para armazenar dados que auxiliam na decodificação do áudio. Assim como a μROM, essas ROMs ocupam área em silício e a técnica de compressão em dois níveis poderia ser utilizada para comprimí-las.

Comprimir Código com a Técnica de Compressão de Microcódigo em Dois Níveis

Na Seção 2.2.2 apresentamos técnicas de compressão de código que utilizam o auxílio de *hardware* especializado para reduzir a perda de desempenho durante a descompressão do código. Dentre essas técnicas, a tecnologia de compressão *CodePack* [49, 83], desenvolvida pela IBM e implementada em alguns processadores da linha PowerPC, divide as instruções do código original (32 *bits*) ao meio e utiliza dois dicionários de 16 *bits* para armazenar os padrões encontrados em cada metade.

Assim como a compressão de microcódigo em dois níveis, a tecnologia *CodePack* explora a entropia do código através do particionamento das colunas. Entretanto, o particionamento é simples e não garante o agrupamento de colunas similares em um mesmo dicionário. Um possível trabalho seria utilizar as heurísticas de agrupamento para agrupar colunas similares do código e melhorar a capacidade de compressão da tecnologia *CodePack*. O algoritmo AKL com restrições poderia ser utilizado para buscar a quantidade e o formato ideal dos dicionários na compressão de código de processadores PowerPC. O espalhador, posicionado na saída dos dicionários, poderia ser implementado utilizando-se um *hardware* reconfigurável para comportar o agrupamento de colunas em diferentes programas.

O modelo de compressão de microcódigo utiliza a repetição de padrões de *bits* únicos para comprimí-lo. Por outro lado, a técnica de fatoração de operandos, proposta por Araújo *et alii* [11], busca por seqüências de instruções, de forma que o número de dados referenciado por um único apontador seja maximizado. Para aumentar o número de seqüências iguais, Araújo *et alii* propuseram uma técnica que fatora o código em operandos e operadores antes de buscar pelas seqüências repetidas. Os autores mostram que, com a fatoração, mais de 80% do programa é coberto por apenas 20% das seqüências. A fatoração do código em operandos e operadores é equivalente à divisão das colunas em conjuntos. Nesse caso específico, o código é dividido em dois conjuntos, um com as colunas dos operadores e outro com as colunas dos operandos das instruções. Um possível trabalho seria utilizar os algoritmos de agrupamento de colunas para investigar novos particionamentos do código e maximizar o número de seqüências repetidas.

A técnica de compressão em dois níveis também poderia ser utilizada para comprimir o código de processadores embarcados. Esses processadores são utilizados em sistemas com restri-

ções críticas de espaço e a área ocupada pela memória é um dos componentes que mais ocupam espaço no *chip*. Dessa forma, a capacidade de se comprimir o código desses processadores, com a técnica de compressão em dois níveis, poderia ser investigada.

Diferente da μROM dos microprocessadores, o formato da memória nesses sistemas é geralmente fixo e a largura do vetor de apontadores não poderia ser ajustada de acordo com a compressão. Assim sendo, a memória teria que armazenar mais de um apontador por palavra e os endereços das instruções comprimidas seriam diferentes dos endereços das instruções originais. Por outro lado, como o tamanho dos apontadores é uniforme, a relação entre os endereços das instruções originais e comprimidas segue uma função linear, o que facilita a tradução dos endereços em tempo de execução. Por exemplo, se os apontadores possuem 15 *bits* e cada palavra da memória possui 32 *bits*, a relação entre o endereço da instrução original ($E_{Original}$) e o endereço da instrução comprimida ($E_{Comprimida}$) é determinada pela seguinte equação:

$$E_{Comprimida} = \frac{E_{Original} \times 15}{32} \tag{6.1}$$

Como o tamanho da palavra da memória não é múltiplo do tamanho dos apontadores, o conteúdo de um apontador pode ser armazenado no meio de uma palavra da memória, bem como começar em uma palavra e terminar em outra. A Equação 6.2 determina a posição do *bit* inicial do apontador na palavra da memória.

$$\text{Posição do } \textit{bit} \text{ inicial} = (E_{Original} \times 15)\ resto\ (32) \tag{6.2}$$

Como trabalho futuro, as Equações 6.1 e 6.2 poderiam ser implementadas em *hardware* para realizar a conversão de endereços de forma eficiente em tempo de execução.

Referências Bibliográficas

[1] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu e John Wiegert. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3), 2006.

[2] Advanced Micro Devices (AMD). *Software Optimization Guide for AMD64 Processors*, setembro de 2005.

[3] Tilak Agerwala. Microprogram optimization: A survey. *IEEE Transactions on Computers*, C-25(10):962–973, outubro de 1976.

[4] Ashok K. Agrawala e Tomlinson G. Rauscher. Microprogramming: Perspective and status. *IEEE Transactions on Computers*, C-23(8):817–837, agosto de 1974.

[5] Alfred V. Aho, Ravi Sethi e Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[6] Francois Anceau. *The architecture of microprocessors*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[7] Edwin de Angel e Earl E. Swartzlander Jr. Survey of low power techniques for ROMs. Em *ISLPED '97: Proceedings of the 1997 international symposium on Low power electronics and design*, pp. 7–11, agosto de 1997, Monterey, CA, USA.

[8] Marco Annaratone, Emmanuel Arnould, Thomas Gross, H. T. Kung e Monica S. Lam. Warp architecture and implementation. Em *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, pp. 346–356, junho de 1986, Tokyo, Japan.

[9] Guido Araujo, Edna Barros, Elmar Melcher, Rodolfo Azevedo, Karina Silva, Bruno Prado e Manoel Eusebio. A SystemC-only design methodology and the CINE-IP multimedia platform. *Design Automation for Embedded Systems*, 10(2-3):181–202, September de 2005.

[10] Guido Araujo, Paulo Centoducatte, Rodolfo Azevedo e Ricardo Pannain. Expression-tree-based algorithms for code compresion on embedded RISC architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(5):530–533, 2000.

[11] Guido Araujo, Paulo Centoducatte, Mario Cortes e Ricardo Pannain. Code compression based on operand factorization. Em *MICRO '31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pp. 194–201, novembro de 1998, Dallas, TX, USA.

[12] Tamarah Arons, Elad Elster, Limor Fix, Sela Mador-Haim, Michael Mishaeli, Jonathan Shalev, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi e Lenore D. Zuck. Formal verification of backward compatibility of microcode. Em *CAV '05: Proceedings of the 17th International Conference on Computer Aided Verification*, pp. 185–198, julho de 2005, Edinburgh, Scotland, UK.

[13] Rodolfo Jardim de Azevedo. *Uma Arquitetura para Execução de Código Comprimido em Sistemas Dedicados*. Tese de Doutorado, Instituto de Computação - Unicamp, Campinas, SP, Brasil, junho de 2002.

[14] Jean-Loup Baer e Barbara Koyama. On the minimization of the width of the control memory of microprogrammed processors. *IEEE Transactions on Computers*, C-28(4):310–316, abril de 1979.

[15] Martin Beneš, Steven M. Nowick e Andrew Wolfe. A fast asynchronous huffman decoder for compressed-code embedded processors. Em *ASYNC '98: Proceedings of the 4th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 43–56, março de 1998, San Deigo, CA, USA.

[16] Martin Beneš, Andrew Wolfe e Steven M. Nowick. A high-speed asynchronous decompression circuit for embedded processors. Em *ARVLSI '97: Proceedings of the 17th Conference on Advanced Research in VLSI*, pp. 219–236, setembro de 1997, Ann Arbor, MI, USA.

[17] Luca Benini, Alberto Macii e Alberto Nannarelli. Cached-code compression for energy minimization in embedded processors. Em *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*, pp. 322–327, agosto de 2001, Huntington Beach, CA, USA.

[18] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc e Konsta Karsisto. Survey of code-size reduction methods. *ACM Computing Surveys*, 35(3):223–267, 2003.

[19] Sunil Bharitkar, Kazuhiro Tsuchiya e Yoshiyasu Takefuji. Microcode optimization with neural networks. *IEEE Transactions on Neural Networks*, 10(3):698–703, maio de 1999.

[20] Eduardo Billo, Rodolfo Azevedo, Guido Araujo, Paulo Centoducatte e Eduardo Wanderley Netto. Design of a decompressor engine on a SPARC processor. Em *SBCCI '05: Proceedings of the 18th annual symposium on Integrated circuits and system design*, pp. 110–114, setembro de 2005, Florianópolis, SC, Brasil.

[21] Eduardo Afonso Billo. Projeto e implementação de um descompressor PDC-ComPacket em um processador SPARC. Dissertação de Mestrado, Instituto de Computação - Unicamp, Campinas, SP, Brasil, abril de 2005.

[22] Nripendra N. Biswas. On bit steering in the minimization of the control memory of microprogrammed processors. *IEEE Transactions on Computers*, C-34(11):1057–1061, novembro de 1985.

[23] Darrell Boggs, Aravindh Baktha, Jason Hawkins, Deborah T. Marr, J. Alan Miller, Patrice Roussel, Ronak Singhal, Bret Toll e K.S. Venkatraman. The microarchitecture of the Pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8(1):1–18, 2004.

[24] Edson Borin. Degradação do desempenho do processador. Comunicação interna, julho de 2006.

[25] Edson Borin, Mauricio Breternitz Jr., Youfeng Wu e Guido Araujo. Clustering-based microcode compression. Em *ICCD '06: Proceedings of the XXIV IEEE International Conference on Computer Design*, pp. 189–196, outubro de 2006, San Jose, CA, USA.

[26] Robert King Brayton, Gary D. Hachtel, Curtis T. McMullen e Alberto L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Norwell, MA, USA, 1984.

[27] Mauricio Breternitz Jr. e Roger Smith. Enhanced compression techniques to simplify program decompression and execution. Em *ICCD '97: Proceedings of the 1997 International Conference on Computer Design*, pp. 170–176, outubro de 1997, Austin, TX, USA.

[28] John Bunda, Don Fussell, W. C. Athas e Roy Jenevein. 16-bit vs. 32-bit instructions for pipelined microprocessors. Em *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pp. 237–246, maio de 1993, San Diego, CA, USA.

[29] Adrian Carbine e Derek Feltham. Pentium pro processor design for test and debug. *IEEE Design & Test of Computers*, 15(3):77–82, julho de 1998.

[30] Brian Case. Intel reveals pentium implementation details. *Microprocessor Report*, 7(4):1–9, março de 1993.

[31] Paulo Cesar Centoducatte. *Compressão de Programas Usando Árvores de Expressão*. Tese de Doutorado, Instituto de Computação - Unicamp, Campinas, SP, Brasil, fevereiro de 2000.

[32] You-Sung Chang, Bong-Il Park e Chong-Min Kyung. Conforming inverted data store for low power memory. Em *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*, pp. 91–93, agosto de 1999, San Diego, CA, USA.

[33] Dave Christie. Developing the AMD-K5 architecture. *IEEE Micro*, 16(2):16–27, abril de 1996.

[34] Lars Raeder Clausen, Ulrik Pagh Schultz, Charles Consel e Gilles Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22(3):471–489, 2000.

[35] Keith D. Cooper e Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. Em *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pp. 139–149, maio de 1999, Atlanta, GA, USA.

[36] Subrata Dasgupta. The organization of microprogram stores. *ACM Computing Surveys*, 11(1):39–65, março de 1979.

[37] Saumya K. Debray, William Evans, Robert Muth e Bjorn De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.

[38] David Johns Dewitt. *A machine independent approach to the production of optimized horizontal microcode*. Tese de Doutorado, University of Michigan, Ann Arbor, MI, USA, 1976.

[39] EEMBC – the embedded microprocessor benchmark consortium. http://www.eembc.org. Acessado no dia 20 de novembro de 2006.

[40] Jens Ernst, William Evans, Christopher W. Fraser, Todd A. Proebsting e Steven Lucco. Code compression. *SIGPLAN Notices*, 32(5):358–365, 1997.

[41] Charles M. Fiduccia e Robert M. Mattheyses. A linear-time heuristic for improving network partitions. Em *DAC '82: Proceedings of the 19th conference on Design automation*, pp. 175–181, junho de 1982, Las Vegas, NV, USA.

[42] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, julho de 1981.

[43] Christopher W. Fraser. An instruction for direct interpretation of LZ77-compressed programs. Relatório técnico, Microsoft Research, setembro de 2002.

[44] Christopher W. Fraser, Eugene W. Myers e Alan L. Wendt. Analyzing and compressing assembly code. Em *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pp. 117–121, junho de 1984, Montreal, Canada.

[45] Christopher W. Fraser e Todd A. Proebsting. Custom instruction sets for code compression. http://www.cs.arizona.edu/people/todd/papers/pldi2.ps.

[46] Gideon Frieder e Jill Miller. An analysis of code density for the two level programmable control of the Nanodata QM-1. Em *MICRO '10: Proceedings of the 10th annual workshop on Microprogramming*, pp. 26–32, outubro de 1977, Niagara Falls, NY, USA.

[47] Bill W. Fung, Timothy J. Macnee e Gheorghe Rugila. A high-density PLA macro and its layout generator. Em *IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, volume XXV, pp. 58–59, fevereiro de 1982.

[48] Jiri Gaisler. *Leon2 processor user's manual 1.0.24*. Gaisler Research, setembro de 2004.

[49] Mark Game e Alan Booker. CodePack: Code compression for PowerPC processors. *MicroNews*, 5(1), 1999.

[50] Michael R. Garey e David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[51] Victor M. Glushkov. Minimization of microprograms and algorithm schemes. *Kibernetika*, 2(5):1–3, 1966.

[52] Michael Golden, Steve Hesley, Alisa Scherer, Matthew Crowley, Scott C. Johnson, Stephan Meier, Dirk Meyer, Jerry D. Moench, Stuart Oberman, Hamid Partovi, Fred Weber, Scott White, Tim Wood e John Yong. A seventh-generation x86 microprocessor. *IEEE Journal of Solid-State Circuits*, 34(11):1466–1477, novembro de 1999.

[53] Antonio Grasselli. The design of program-modifiable micro-programmed control units. *IRE Transactions on Electronic Computers*, EC-11(6):336–339, junho de 1962.

[54] Antonio Grasselli e Ugo Montanari. On the minimization of READ-ONLY memories in microprogrammed digital computers. *IEEE Transactions on Computers*, C-19(11):1111–1114, novembro de 1970.

[55] Thomas G. Gunter e Harry L. Tredennick. Two-level control store for microprogrammed data processor. U.S. Patent n. 4,325,121, abril de 1982.

[56] Karl M. Guttag. Compressing control ROM for VLSI microprogrammed microprocessors. Em *MICRO '13: Proceedings of the 13th annual workshop on Microprogramming*, pp. 115–121, novembro de 1980, Springs, CO, USA.

[57] Gary D. Hachtel, Arthur Richard Newton e Alberto L. Sangiovanni-Vincentelli. An algorithm for optimal PLA folding. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1(2):63–77, abril de 1982.

[58] Constantine Halatsis e Nikolaos Gaitanis. On the minimization of the control store in microprogrammed computers. *IEEE Transactions on Computers*, C-27(12):1189–1192, dezembro de 1978.

[59] V. Carl Hamacher, Zvonko G. Vranesic e Safwat G. Zaky. *Computer Organization*. McGraw-Hill, segunda edição, 1984.

[60] John L. Hennessy e David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2003.

[61] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[62] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker e Patrice Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, 5(1):1–13, 2001.

[63] Se-Kyoung Hong, In-Cheol Park e Chang-Min Kyung. An $O(n^3logn)$-heuristic for microcode bit optimization. Em *ICCAD '90: Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 180–183, novembro de 1990, Santa Clara, CA, USA.

[64] David A. Huffman. A method for construction of minimum redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40:1098–1101, setembro de 1952.

[65] Herbert Hum, Mauricio Breternitz Jr., Youfeng Wu e Sangwook Kim. Compressing microcode. U.S. Patent n. 7,095,342, agosto de 2006.

[66] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, novembro de 2006.

[67] Nagisa Ishiura e Masayuki Yamaguchi. Instruction code compression for application specific VLIW processors based on automatic field partitioning. Em *The Seventh Workshop on Synthesis and System Integration of Mixed technologies*, pp. 105–109, dezembro de 1997, Osaka, Japan.

[68] Sadahiro Isoda, Yoshizumi Kobayaski e Toru Ishida. Global compaction of horizontal microprograms based on generalized data dependency graph. *IEEE Transactions on Computers*, C-32(10):922–933, outubro de 1983.

[69] David Jagger. *ARM Architectural Reference Manual*. Prentice Hall, 1996.

[70] Totadri Jayasri e Dhruba Basu. An approach to organizing microinstructions which minimizes the width of control store words. *IEEE Transactions on Computers*, C-25(5):514–521, maio de 1976.

[71] Brian Wilson Kernighan e Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–207, fevereiro de 1970.

[72] Darko Kirovski, Johnson Kin e William H. Mangione-Smith. Procedure based program compression. Em *MICRO '30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pp. 204–213, dezembro de 1997, Research Triangle Park, NC, USA.

[73] Kevin D. Kissell. MIPS16: High-density MIPS for the embedded market. Em *Proceedings of Real Time Systems*, pp. 559–571, janeiro de 1997.

[74] Zvi Kohavi. *Switching and Finite Automata Theory: Computer Science Series*. McGraw-Hill Higher Education, 1990.

[75] Sandeep Koranne. A "bit-bucket" data structure to optimize local search for microword length minimization. Em *Proceedings of the 1999 IEEE Canadian Conference on Electrical and Computer Engineering*, volume 1, pp. 507–512, maio de 1999, Edmonton, Alta, Canada.

[76] Michael Kozuch e Andrew Wolfe. Compression of embedded system programs. Em *ICCD '94: Proceedings of the 1994 IEEE International Conference on Computer Design: VLSI in Computer & Processors*, pp. 270–277, outubro de 1994, Cambridge, MA, USA.

[77] Kevin Krewell. AMD takes Hammer to Itanium. *Microprocessor Report*, 15(11):1–4, novembro de 2001.

[78] Jehkwan Lah e Daniel E. Atkins. Tree compaction of microprograms. *SIGMICRO Newsletter*, 14(4):23–33, 1983.

[79] David Landskov, Scott Davidson, Bruce Shriver e Patrick W. Mallett. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3):261–294, 1980.

[80] James R. Larus. A comparison of microcode, assembly code, and high-level languages on the VAX-11 and RISC I. *SIGARCH Computer Architecture News*, 10(5):10–15, 1982.

[81] Charles Lefurgy, Peter Bird, I-Cheng Chen e Trevor Mudge. Improving code density using compression techniques. Em *MICRO '30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pp. 194–203, dezembro de 1997, Research Triangle Park, NC, USA.

[82] Charles Lefurgy, Eva Piccininni e Trevor Mudge. Evaluation of a high performance code compression method. Em *MICRO '32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pp. 93–102, novembro de 1999, Haifa, Israel.

[83] Charles Lefurgy, Eva Piccininni e Trevor Mudge. Reducing code size with run-time decompression. Em *Proceedings of Sixth International Symposium on High-Performance Computer Architecture*, pp. 218–227, janeiro de 2000, Toulouse, France.

[84] Haris Lekatsas, Jörg Henkel e Venkata Jakkula. Design of an one-cycle decompression hardware for performance increase in embedded systems. Em *DAC '02: Proceedings of the 39th conference on Design automation*, pp. 34–39, junho de 2002, New Orleans, LO, USA.

[85] Haris Lekatsas, Jörg Henkel e Wayne Wolf. Code compression as a variable in hardware/software co-design. Em *CODES '00: Proceedings of the 8th International Workshop on Hardware/software Codesign*, pp. 120–124, maio de 2000, San Diego, CA, USA.

[86] Haris Lekatsas, Jörg Henkel e Wayne Wolf. Code compression for low power embedded system design. Em *DAC '00: Proceedings of the 37th conference on Design automation*, pp. 294–299, junho de 2000, Los Angeles, CA, USA.

[87] Haris Lekatsas e Wayne Wolf. Code compression for embedded systems. Em *DAC '98: Proceedings of the 35th annual conference on Design automation*, pp. 516–521, junho de 1998, San Francisco, CA, USA.

[88] Haris Lekatsas, Wayne Wolf e Jörg Henkel. Arithmetic coding for low power embedded system design. Em *DCC '00: Proceedings of the Conference on Data Compression*, pp. 430–439, março de 2000, Snowbird, UT, USA.

[89] Debra A. Lelewer e Daniel S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, 1987.

[90] Abraham Lempel e Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, janeiro de 1976.

[91] Stan Liao, Srinivas Devadas e Kurt Keutzer. A text-compression-based method for code size minimization in embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 4(1):12–38, 1999.

[92] Stan Yi-Huang Liao. *Code generation and optimization for embedded digital signal processors*. Tese de Doutorado, Massachusetts Institute of Technology, Cambridge, MA, USA, janeiro de 1996.

[93] Joseph L. Linn. SRDAG compaction: a generalization of trace scheduling to increase the use of global context information. *SIGMICRO Newsletter*, 14(4):11–22, 1983.

[94] Jorge Francisco Martinez-Carballido e V. Michael Powers. General microprogram width reduction using generator sets. Em *MICRO '14: Proceedings of the 14th annual workshop on Microprogramming*, pp. 144–153, dezembro de 1981, Chatham, MA, USA.

[95] Ayakannu Mathialagan e Nripendra N. Biswas. Bit steering in the minimization of control memory in microprogrammed digital computers. *IEEE Transactions on Computers*, 30(2):144–147, fevereiro de 1981.

[96] James McKevitt e John Bayliss. New options from big chips. *IEEE Spectrum*, 16(3):28–34, março de 1979.

[97] Onat Menzilcioglu. A case study in using two-level control stores. Em *MICRO '20: Proceedings of the 20th annual workshop on Microprogramming*, pp. 142–146, dezembro de 1987, Colorado Springs, CO, USA.

[98] Andrew W. Nagle, Richard Cloutier e Alice C. Parker. Synthesis of hardware for the control of digital systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1(4):201–212, outubro de 1982.

[99] Sang-Joon Nam, In-Cheol Park e Chong-Min Kyung. Improving dictionary-based code compression in VLIW architectures. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E82-A(11):2318–2324, 1999.

[100] Nanodata Corporation, 2457 Wehrie Drive, Williamsville, New York. *QM-1 Hardware Level User's Manual*, segunda edição, agosto de 1974.

[101] Eduardo Bráulio Wanderley Netto. *Compressão de Código Baseada em Multi-Profile*. Tese de Doutorado, Instituto de Computação - Unicamp, Campinas, SP, Brasil, maio de 2004.

[102] Eduardo Wanderley Netto, Rodolfo Azevedo, Paulo Centoducatte e Guido Araujo. Multi-profile instruction based compression. Em *SBAC-PAD '04: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pp. 23–29, outubro de 2004, Foz do Iguaçu, PR, Brasil.

[103] Christos A. Papachristou e Anil L. Pandya. A design scheme for PLA-based control tables with reduced area and time-delay cost. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(5):453–472, maio de 1990.

[104] Christos A. Papachristou e James M. Reuter. Microassembly and area reduction techniques for PLA microcode. *SIGMICRO Newsletter*, 15(4):86–94, dezembro de 1984.

[105] David B. Papworth. Tuning the Pentium Pro microarchitecture. *IEEE Micro*, 16(2):8–15, 1996.

[106] In-Cheol Park, Se-Kyoung Hong e Chong-Min Kyung. Two complementary approaches for microcode bit optimization. *IEEE Transactions on Computers*, 43(2):234–239, 1994.

[107] David A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, 1985.

[108] David A. Patterson e John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2004.

[109] Ruchir Puri e Jun Gu. Microword length minimization in microprogrammed controller synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(10):1449–1457, outubro de 1993.

[110] Jan M. Rabaey, Anantha Chandrakasan e Borivoje Nikolić. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, Upper Saddle River, NJ, USA, segunda edição, 2002.

[111] Chamarty D.V.P. Rao e Nripendra N. Biswas. On the minimization of wordwidth in the control memory of a microprogrammed digital computer. *IEEE Transactions on Computers*, C-32(9):863–868, setembro de 1983.

[112] S. S. Ravi e Dechang Gu. On approximation algorithms for microcode bit minimization. Em *MICRO '21: Proceedings of the 21st annual workshop on Microprogramming and microarchitecture*, pp. 67–69, dezembro de 1988, San Diego, CA, USA.

[113] Edward L. Robertson. Microcode bit optimization is NP-Complete. *IEEE Transactions on Computers*, C-28(4):316–319, abril de 1979.

[114] Robert F. Rosin, Gideon Frieder e Richard H. Eckhouse Jr. An environment for research in microprogramming and emulation. *Communications of the ACM*, 15(8):748–760, 1972.

[115] Gian-Carlo Rota. The number of partitions of a set. *American Mathematical Monthly*, 71(5):498–504, maio de 1964.

[116] Scott J. Schwartz. An algorithm for minimizing read only memories for machine control. Em *Conference Record of 1968 Ninth Annual Symposium on Switching and Automata Theory*, pp. 28–33, outubro de 1968, Schenectady, NY, USA.

[117] Michael Slater. K6 to boost AMD's position in 1997. *Microprocessor Report*, 10(14):1–2, outubro de 1996.

[118] Alex Van Someren e Carol Atack. *The ARM RISC Chip. A Programmer's Guide*. Addison Wesley, 1993.

[119] Skip Stritter e Nick Tredennick. Microprogrammed implementation of a single chip microprocessor. Em *MICRO '11: Proceedings of the 11th annual workshop on Microprogramming*, pp. 8–16, novembro de 1978, Pacific Grove, CA, USA.

[120] Bogong Su, Shiyuan Ding e Lan Jin. An improvement of trace scheduling for global microcode compaction. *SIGMICRO Newsletter*, 15(4):78–85, 1984.

[121] L.-F. Sun, J.-M. Liaw e T.-M. Parng. Automated synthesis of microprogrammed controllers in digital systems. *IEE Proceedings*, 135(4):231–240, julho de 1988.

[122] Harry L. Tredennick e Thomas G. Gunter. Microprogrammed control apparatus having a two-level control store for data processor. U.S. Patent n. 4,307,445, dezembro de 1981.

[123] Nick Tredennick. The "cultures" of microprogramming. Em *MICRO 15: Proceedings of the 15th annual workshop on Microprogramming*, pp. 79–83, Piscataway, NJ, USA, outubro de 1982. Palo Alto, California, United States.

[124] Stuart G. Tucker. Microprogram control for system/360. *IBM Systems Journal*, 6(4):222–241, 1967.

[125] James L. Turley. Thumb squeezes ARM code size. *Microprocessor Report*, 9(4):1–5, março de 1995.

[126] Frank Vahid. Modifying min-cut for hardware and software functional partitioning. Em *CODES '97: Proceedings of the 5th International Workshop on Hardware/Software Co-design*, pp. 43–48, março de 1997, Braunschweig, Germany.

[127] Neil H. E. Weste e Kamran Eshraghian. *Principles of CMOS VLSI design: a systems perspective*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, segunda edição, outubro de 1994.

[128] Maurice Vincent Wilkes. The best way to design an automatic calculating machine. Em *Manchester University Computer Inaugural Conference*, pp. 16–18, julho de 1951, Manchester, England.

[129] Maurice Vincent Wilkes. EDSAC 2. *IEEE Annals of the History of Computing*, 14(4):49–56, abril de 1992.

[130] Andrew Wolfe e Alex Chanin. Executing compressed programs on an embedded RISC architecture. Em *MICRO '25: Proceedings of the 25th annual international symposium on Microarchitecture*, pp. 81–91, dezembro de 1992, Portland, OR, USA.

[131] Youfeng Wu, Mauricio Breternitz Jr., Herbert Hum, Ramesh Peri e Jay Pickett. Enhanced code density of embedded CISC processors with echo technology. Em *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 160–165, setembro de 2005, Jersey City, NJ, USA.

[132] Yuan Xie, Wayne Wolf e Haris Lekatsas. Compression ratio and decompression overhead tradeoffs in code compression for VLIW architectures. Em *Proceedings of the 4th International Conference on ASIC*, pp. 337–340, outubro de 2001, Shanghai, China.

[133] Yuan Xie, Wayne Wolf e Haris Lekatsas. Code compression for VLIW processors using variable-to-fixed coding. Em *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pp. 138–143, outubro de 2002, Kyoto, Japan.

[134] Matt T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. Aceito para publicação no ISPASS '97: IEEE International Symposium on Performance Analysis of Systems and Software, abril de 2007.

[135] Michael Joseph Zastre. Compacting object code via parameterized procedural abstraction. Dissertação de Mestrado, University of Victoria, Victoria, BC, Canada, 1995.

[136] Wei Zhao e Christos A. Papachristou. Architectural partitioning of control memory for application specific programmable processors. Em *ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pp. 521–526, novembro de 1995, San Jose, CA, USA.

[137] Jacob Ziv e Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transaction on Information Theory*, 23(3):337–343, maio de 1977.

Apêndice A

Provas de $\mathcal{NP}$-Completude

A.1 Compressão com um único dicionário de tamanho fixo

Seja D_1 um dicionário com L_1 colunas, a compressão de um microcódigo B com um único dicionário de tamanho fixo consiste em selecionar L_1 colunas do microcódigo B e comprimí-las utilizando-se o dicionário D_1. Sendo N a quantidade de microinstruções e L o número de *bits* por microinstrução no microcódigo B, e M_1 o número de padrões no dicionário D_1, o tamanho do microcódigo comprimido é definido pela seguinte equação:

$$\begin{matrix} Tamanho \\ Comprimido(B) \end{matrix} = \underbrace{N \times (L - L_1)}_{Colunas\ não-comprimidas} + \overbrace{\lceil log_2 M_1 \rceil \times N}^{Vetor\ de\ apontadores} + \underbrace{M_1 \times L_1}_{Dicionário\ D_1} \quad \text{(A.1)}$$

O "problema da melhor compressão de microcódigo com um único dicionário de tamanho fixo" é um problema de otimização onde a função objetivo é o tamanho do microcódigo comprimido. Em outras palavras, o objetivo é minimizar o tamanho do microcódigo comprimido. Para mostrar que o problema da melhor compressão de microcódigo é difícil, o convertemos em um problema de decisão e mostramos que este é $\mathcal{NP}$-Completo.

O problema de decisão da compressão com um único dicionário de tamanho fixo consiste em determinar se é possível comprimir uma matriz de *bits* com um único dicionário D_1 de L_1 colunas com no máximo sz *bits*. Observe que introduzimos a variável sz para transformar o problema de otimização em um problema de decisão. Assim sendo, o problema de decisão da compressão com um único dicionário pode ser resumido da seguinte forma:

Seja B uma matriz de *bits* com L colunas e N linhas, determinar se é possível comprimir B, utilizando-se um dicionário D_1 de L_1 colunas, com no máximo sz *bits*.

Para provar que o problema acima é $\mathcal{NP}$-Completo, reduzimos o problema de determinar se existe uma clique de tamanho k em um grafo qualquer G para o problema da compressão. A

idéia é representar o grafo com uma matriz de *bits* e mostrar que essa matriz pode ser representada de forma comprimida, utilizando-se um dicionário de $L_1 = k$ colunas, com no máximo sz *bits* se e somente se G tem uma clique com k vértices.

$\mathcal{NP}$

Dado um conjunto de L_1 colunas, para serem comprimidas no dicionário D_1, podemos computar o número de padrões M_1 e verificar em tempo polinomial se o tamanho da matriz B, determinado pela Equação A.1, é menor do que sz. Portanto, o problema é $\mathcal{NP}$.

Redução do problema da clique em grafos

Determinar se um grafo simples G possui ou não uma clique de tamanho k (para $k \geq 3$) é um problema $\mathcal{NP}$-Completo [50]. Para provar que o problema da compressão com um dicionário de tamanho fixo é $\mathcal{NP}$-Difícil vamos reduzir o problema da clique para o problema da compressão.

O problema da clique é definido da seguinte forma: Seja $G = (V, E)$ um grafo com n vértices e m arestas, determinar se existe uma clique com k vértices.

Seja $\overline{m}$ o número de arestas ausentes no grafo G e $\frac{n^2-n}{2}$ o número de arestas no grafo completo, temos que:

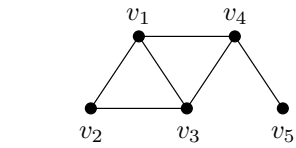
$$\overline{m} = \frac{n^2 - n}{2} - m \tag{A.2}$$

A partir do grafo G, construímos uma matriz de *bits* com n colunas e $n + \overline{m} + 1$ linhas da seguinte forma:

- Cada vértice v_i do grafo G é representado por uma coluna v_i na matriz de *bits*. Portanto, a matriz possui n colunas.
- Para cada vértice v_i, criamos uma linha lv_i denominada "padrão identidade". Esta linha possui o valor '1' na coluna v_i e o valor '0' nas outras colunas.
- Para cada aresta não presente no grafo, criamos uma linha $la_{i\,j}$ com o valor '1' nas colunas v_i e v_j, que representam os extremos da aresta.
- Por fim, criamos uma linha com o padrão zero, ou seja, todas as colunas possuem o valor '0'.

A Figura A.1 (b) mostra um exemplo da matriz construída a partir do grafo G da Figura A.1 (a). Observe que a matriz possui $n = 5$ colunas e $n + \overline{m} + 1 = 10$ linhas.

Seja C' um subconjunto com k colunas da matriz de *bits*. Para cada coluna v_i em C', existe um padrão distinto, formado pelo "padrão identidade" (lv_i) da coluna v_i. Por exemplo, o subconjunto $\{v_1, v_2, v_3\}$ da matriz de *bits* na Figura A.1 possui os padrões 100, 010 e 001,



$V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{a_{12}, a_{13}, a_{14}, a_{23}, a_{34}, a_{45}\}$

$\overline{E} = \{a_{15}, a_{24}, a_{25}, a_{35}\}$

(a)

	v_1	v_2	v_3	v_4	v_5	
lv_1	1	0	0	0	0	n
lv_2	0	1	0	0	0	
lv_3	0	0	1	0	0	
lv_4	0	0	0	1	0	
lv_5	0	0	0	0	1	
la_{15}	1	0	0	0	1	$\overline{m}$
la_{24}	0	1	0	1	0	
la_{25}	0	1	0	0	1	
la_{35}	0	0	1	0	1	
padrão zero	0	0	0	0	0	1

n

(b)

Figura A.1: Exemplo de uma matriz de *bits* construída a partir de um grafo.

decorrentes do padrão identidade de cada coluna. Esses padrões mais o padrão zero, na última linha da matriz, fazem com que o subconjunto C' tenha no mínimo $k+1$ padrões.

De fato, se o subconjunto C' representar uma clique, ou seja, se não houver nenhuma aresta faltando neste subconjunto de vértices, o número de padrões vai ser $k+1$. Observe que não haverá padrão de *bits* com mais de um *bit* com o valor '1' no subconjunto C'.

Por outro lado, se o subconjunto C' possui duas colunas v_i e v_j tal que não haja aresta entre os respectivos vértices no grafo G, então existe um padrão de *bits* que contém o valor '1' nas colunas v_i e v_j, gerado a partir da linha la_{ij}, ou seja, da aresta que falta. Por exemplo, considere o subconjunto $\{v_1, v_2, v_4\}$ da Figura A.1, não há aresta ligando os vértices v_2 e v_4 no grafo G, portanto, existe o padrão 011, com dois *bits* '1', derivado da linha la_{24}. Esse padrão mais o padrão zero e os k padrões identidade totalizam $k+2$ padrões. Assim sendo, se o subconjunto não é uma clique, o número de padrões é maior do que $k+1$.

Portanto, existe uma clique de tamanho k no grafo se e somente se existe um subconjunto C' com $L_1 = k$ colunas na matriz de *bits* com exatamente $k+1$ padrões de *bits*. Dessa maneira, podemos determinar o tamanho sz como:

$$sz = \underbrace{N \times (L-k)}_{Colunas\ não-comprimidas} + \overbrace{\lceil log_2(k+1) \rceil \times N}^{Vetor\ de\ apontadores} + \underbrace{(k+1) \times k}_{Dicionário} \tag{A.3}$$

e perguntar se é possível comprimir a matriz de *bits* com no máximo sz *bits* utilizando-se um dicionário de k colunas. Se for possível, então existe uma clique de tamanho k no grafo G, do contrário, a clique não existe.

□

A.2 Compressão com j dicionários de tamanhos fixos

Na compressão com j dicionários de tamanhos fixos, j conjuntos de colunas são selecionados e comprimidos separadamente, cada um com um dicionário com um número fixo de colunas. O problema da compressão com j dicionários com tamanhos fixos pode ser resumido da seguinte forma:

Seja B uma matriz de *bits* com L colunas e N linhas, determinar se é possível comprimir B com no máximo sz *bits* utilizando-se j dicionários, tal que cada dicionário D_i tenha exatamente L_i colunas.

Cada dicionário comprime um conjunto distinto de colunas da matriz e utiliza o próprio vetor de apontadores. Assim sendo, o tamanho da matriz B comprimida é definida pela seguinte equação:

$$Tamanho(B) = \underbrace{N \times (L - \sum_{i=1}^{j} L_i)}_{Colunas\ não{-}comprimidas} + \sum_{i=1}^{j} (\overbrace{\lceil log_2 M_i \rceil \times N}^{Vetor\ de\ apontadores\ i} + \underbrace{M_i \times L_i}_{Dicionário\ i}) \quad \text{(A.4)}$$

$\mathcal{NP}$

Dado j conjuntos de colunas para serem comprimidos com j dicionários, podemos computar o número de padrões em cada conjunto e verificar em tempo polinomial se o tamanho da matriz B, determinado pela Equação A.4, é menor do que sz. Portanto, o problema é $\mathcal{NP}$.

Redução do problema da clique em grafos

Da mesma forma que na Seção A.1, utilizaremos o problema da clique em grafos para mostrar que a compressão com j dicionários é $\mathcal{NP}$-Difícil.

O problema da clique pode ser definido da seguinte forma: Seja $G = (V, E)$ um grafo com n vértices e m arestas, determinar se existe uma clique com k vértices.

Para provar que o problema da compressão com j dicionários de tamanhos fixos é $\mathcal{NP}$-Difícil seguiremos os seguintes passos:

1. Gerar um grafo $G' = (V', E')$ a partir de G.
2. Gerar uma matriz de *bits* B a partir de G'.
3. Calcular o tamanho sz em função de k.
4. Mostrar que a matriz B pode ser comprimida com no máximo sz *bits*, utilizando-se os j dicionários, se e somente se G possui uma clique de tamanho k.

Seja $L_1, L_2, \cdots, L_j$ a largura (número de colunas) de cada dicionário D_i. Sem perda de generalidade, assumimos que $L_1 = k$ e o tamanho dos demais dicionários ($L_2, L_3, \cdots, L_j$) é arbitrário. Assim sendo, geramos $G' = (V', E')$ da seguinte forma:

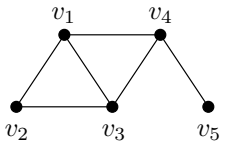
- $V' = V_{D_2} \cup V_{D_3} \cup \cdots \cup V_{D_j}$
- $E' = E_{D_2} \cup E_{D_3} \cup \cdots \cup E_{D_j}$

onde:

- $V_{D_i} = \{x_{i,1}, x_{i,2}, x_{i,3}, \cdots, x_{i,L_i}\}$
- $E_{D_i} = \{\{x_{i,1}, x_{i,2}\}, \{x_{i,1}, x_{i,3}\}, \cdots, \{x_{i,1}, x_{i,L_i}\}, \{x_{i,2}, x_{i,3}\}, \cdots, \{x_{i,L_i-1}, x_{i,L_i}\}\}$

Ou seja, G' é basicamente G adicionado de $j-1$ cliques, cada uma com L_i vértices. O exemplo da Figura A.2 mostra um grafo G' gerado a partir de G. Observe que $j = 4$, $L_1 = k$, $L_2 = 2$, $L_3 = 5$ e $L_4 = 4$.

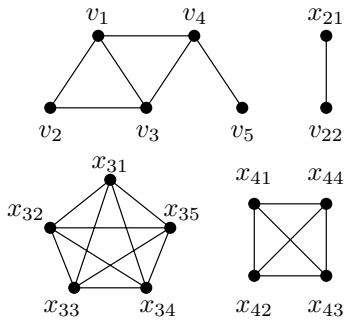
O grafo G' da Figura A.2 possui quatro cliques disjuntas de tamanhos k, 2, 5 e 4 se e somente se G possui uma clique de tamanho k.



$V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_5\}\}$

(a)



$V' = \{v_1, v_2, v_3, v_4, v_5, x_{21}, x_{22}, x_{31}, x_{32}, x_{33}, x_{34}, x_{35}, x_{41}, x_{42}, x_{43}, x_{44}\}$

$E' = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_5\}, \{x_{21}, x_{22}\}, \{x_{31}, x_{32}\}, \{x_{31}, x_{33}\}, \{x_{31}, x_{34}\}, \{x_{31}, x_{35}\}, \{x_{32}, x_{33}\}, \{x_{32}, x_{34}\}, \{x_{32}, x_{35}\}, \{x_{33}, x_{34}\}, \{x_{33}, x_{35}\}, \{x_{34}, x_{35}\}, \{x_{41}, x_{42}\}, \{x_{41}, x_{43}\}, \{x_{41}, x_{44}\}, \{x_{42}, x_{43}\}, \{x_{42}, x_{44}\}, \{x_{42}, x_{45}\}, \{x_{43}, x_{44}\}, \{x_{43}, x_{45}\}, \{x_{44}, x_{45}\}\}$

(b)

Figura A.2: Geração de um grafo G' a partir de G para $j = 4$, $L_1 = k$, $L_2 = 2$, $L_3 = 5$ e $L_4 = 4$.

Assim como na redução apresentada na Seção A.1, geramos uma matriz de *bits* B a partir do grafo G' da seguinte forma:

- Cada vértice v_i do grafo G' (incluindo os vértices do tipo x_{jk}) é representado por uma coluna v_i na matriz de *bits*. Portanto, a matriz possui n' colunas.
- Para cada vértice v_i, criamos uma linha lv_i denominada "padrão identidade". Esta linha possui o valor '1' na coluna v_i e '0' nas outras colunas.
- Para cada aresta não presente no grafo G', criamos uma linha $la_{i\,j}$ com o valor '1' nas colunas v_i e v_j, que representam os extremos da aresta.
- Por fim, criamos uma linha com o padrão zero, ou seja, todas as colunas possuem o valor '0'.

Pelo mesmo argumento utilizado na Seção A.1, podemos observar que qualquer subconjunto de colunas C' com $|C'|$ colunas possui no mínimo $|C'|+1$ padrões. Observe que um conjunto com $|C'|$ colunas possui $|C'|$ padrões identidades (um para cada coluna) e o padrão zero. Para todo par de colunas (c_1, c_2) representando dois vértices v e u em G' tal que $\{u, v\} \notin E'$, existe um padrão de *bits* com o valor '1' nas colunas c_1 e c_2. Portanto, se as colunas em C' não representam um conjunto de vértices em G' que formam uma clique, o número de padrões em C' é maior que $|C'|+1$. Por outro lado, se o subconjunto C' representar uma clique, não existirá padrões com mais de um *bit* com o valor '1' no subconjunto, pois não faltam arestas entre os vértices representados pelas colunas em C'. Assim sendo, a quantidade de padrões é $|C'|+1$, ou seja, o número de padrões com um único *bit* com o valor '1' e o padrão zero.

Sendo M_i e L_i o número de padrões e o número de colunas, respectivamente, em cada dicionário D_i, o tamanho mínimo da matriz B comprimida é:

$$\begin{matrix} Tamanho \\ Mínimo(B) \end{matrix} = \underbrace{N \times (L - \sum_{i=1}^{j} L_i)}_{Colunas\ não{-}comprimidas} + \sum_{i=1}^{j} (\overbrace{\lceil log_2(L_i+1) \rceil \times N}^{Vetor\ de\ apontadores\ i} + \underbrace{(L_i+1) \times L_i}_{Dicionário\ i}) \quad \text{(A.5)}$$

Podemos observar também que este tamanho só é possível se e somente se o grafo G' tiver j cliques disjuntas com tamanhos $L_1 = k, L_2, L_3, \cdots, L_j$. Da mesma forma, o grafo G' tem j cliques disjuntas com tamanhos $L_1 = k, L_2, L_3, \cdots, L_j$ se e somente se o grafo G tem uma clique de tamanho k. Assim sendo, o problema da compressão com j dicionários de tamanhos fixos é mais difícil que o problema da clique.

□