UNIVERSIDADE FEDERAL DO AMAZONAS
CAMPUS UNIVERSITÁRIO SEN. ARTHUR VIRGÍLIO FILHO
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

ALGORITMOS BASEADAS EM PADRÕES DE BLOCOS E MULTIPLOS DICIONÁRIOS PARA COMPRESSÃO DE CÓDIGO EM SISTEMAS EMBARCADOS

WANDERSON ROGER AZEVEDO DIAS

MANAUS
2013

UNIVERSIDADE FEDERAL DO AMAZONAS
CAMPUS UNIVERSITÁRIO SEN. ARTHUR VIRGÍLIO FILHO
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

WANDERSON ROGER AZEVEDO DIAS

ALGORITMOS BASEADAS EM PADRÕES DE BLOCOS E MULTIPLOS DICIONÁRIOS PARA COMPRESSÃO DE CÓDIGO EM SISTEMAS EMBARCADOS

Tese de doutorado apresentada ao Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal do Amazonas, como parte de requisito para obtenção do título de Doutor em Informática, área de concentração: Engenharia da Computação.

Orientador: Prof. Dr. Edward David Moreno Ordoñez

MANAUS

2013

WANDERSON ROGER AZEVEDO DIAS

ALGORITMOS BASEADAS EM PADRÕES DE BLOCOS E MULTIPLOS DICIONÁRIOS PARA COMPRESSÃO DE CÓDIGO EM SISTEMAS EMBARCADOS

Tese de doutorado apresentada ao Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal do Amazonas, como parte de requisito para obtenção do título de Doutor em Informática, área de concentração: Engenharia da Computação.

BANCA EXAMINADORA

Prof. Dr. Edward David Moreno Ordoñez, Presidente
Universidade Federal de Sergipe (UFS)

Prof. Dr. Siang Wun Song, Membro
Universidade de São Paulo (USP)

Prof. Dr. Daniel Gomes Mesquita, Membro
Universidade Federal de Uberlândia (UFU)

Prof. Dr. Carlos Maurício Seródio Figueiredo, Membro
Universidade Federal do Amazonas (UFAM)

Prof. Dr. José Reginaldo Hughes Carvalho, Membro
Universidade Federal do Amazonas (UFAM)

ALGORITMOS BASEADAS EM PADRÕES DE BLOCOS E MULTIPLOS DICIONÁRIOS PARA COMPRESSÃO DE CÓDIGO EM SISTEMAS EMBARCADOS

Este exemplar corresponde à redação final da Tese do doutorando Wanderson Roger Azevedo Dias para ser aprovada pela Banca Examinadora.

Manaus – AM, 05 de julho de 2013.

Prof. Dr. Edward David Moreno Ordoñez
Orientador

Prof. Dr. Siang Wun Song
Membro

Prof. Dr. Daniel Gomes Mesquita
Membro

Prof. Dr. Carlos Maurício Seródio Figueiredo
Membro

Prof. Dr. José Reginaldo Hughes Carvalho
Membro

À Deus,
À minha esposa,
Aos meus pais,
Minhas irmãs e sobrinha,
Ao meu orientador,
Aos meus professores,
Meus parentes e amigos...

Agradecimentos

“O coração do entendido adquire o conhecimento, e o ouvido dos sábios busca a ciência". Pv.18:15.

Resumo

As atuais aplicações embarcadas têm exigido cada vez mais dos sistemas embarcados que por sua vez apresentam inúmeras limitações físicas e de recursos computacionais, sendo a memória um dos recursos mais críticos, devido a sua capacidade de armazenamento ser limitada de acordo com sua área ocupada no sistema. Assim justifica-se o esforço para otimizar o seu uso. Pesquisas têm mostrado que as técnicas de compressão de código servem como uma alternativa para resolver alguns problemas como: espaço, desempenho e consumo de energia nos sistemas embarcados. Esta tese trata da compressão de código dos programas para execução em sistemas embarcados baseados em processadores RISC. Na tese mostra-se que a utilização dos quatro novos métodos propostos e desenvolvidos neste trabalho (CPB-ARM, HDPB, CCHPB e CC-MLD), resulta em boas taxas de compressão. Implementações eficientes e simples do hardware descompressor também são apresentados. Além disto, um novo tipo de dicionário dividido em níveis também é introduzido por esta tese. O paradigma aplicado por esse novo dicionário consiste em armazenar instruções unitárias e padrões de blocos encontrados no código dos programas ao mesmo tempo e por isto é chamado de *Dicionário Multi-Nível*. Nas simulações realizadas com os métodos desenvolvidos nesta tese usaram-se alguns programas do *benchmark MiBench*. As taxas de compressão média obtidas nos métodos variaram de 24,2% a 32%. Assim, os algoritmos propostos oferecem uma melhor exploração da tríade compressão-desempenho-consumo.

Palavras-Chave: Sistemas Embarcados, Compressão de Código, Instrução Unitária, Padrão de Bloco, Dicionário Multi-Nível.

Abstract

Current embedded applications require more and more from embedded system which may present many physical and computational limitations, memory being one of the most critical due to its storage capacity being limited according to the area occupied in the system. Thus, the effort to optimize its use is justified. This thesis is about code compression of programs that run in embedded systems based on RISC processors. It is shown in the thesis that the use of four new methods proposed and developed in this work (CPB-ARM, HDPB, CCHPB and CC-MLD) leads to good compression rates. Efficient and simple implementations of decompressor hardware are also presented. Furthermore, a new dictionary type which is divided in levels is also introduced by these thesis. The applied paradigm by this new dictionary consists of storing unitary instructions and block patterns found in program's code at the same time and that is called Multi-Level Dictionary. In simulations performed with the developed methods in this thesis some programs from MiBench benchmark were used. The obtained average compression rates of the methods ranged from 24.2% to 32%. Thus, the proposed algorithms offer a better exploitation of the compression-performance-consumption triad.

Keywords: Embedded Systems, Code Compression, Unitary Instruction, Block Pattern, Multi-Level Dictionary.

LISTA DE FIGURAS

LISTA DE GRÁFICOS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

ALC	*Adaptive Loop Cache*
ARM	*Advanced RISC Machine*
BBM	*Basic Block Mapping*
BPA	*Bitstream Placement Algorithm*
CAD	*Computer-Aided Design*
CBAT	*Compressed Block Address Table*
CCA	*Compression Code Architecture*
CCRP	*Compressed Code RISC Processor*
CCTP	*Code Compressed THUMB Processor*
CDM	*Cache Decompressor Memory*
CLB	*Cache Line Address Look-aside Buffer*
CLBs	*Configurable Logic Blocks*
CPSR	*Current Program Status Register*
CPU	*Central Processing Unit*
CW	*Control Words*
DBB	*Decompressed Block Buffer*
DCE	*Decompression Engine for Bitmask Encoding*
DE	*Decompression Engine*
DFC	*Dynamic Frequency based Compression*
DISE	*Dynamic Instruction Stream Editing*
DRAM	*Dynamic Random Access Memory*
EEMBC	*EDN Embedded Microprocessor Benchmark Consortium*
ELF	*Executable and Linking Format*
EPROM	*Erasable Programmable Read Only Memory*

FPGA	*Field Programmable Gate Arrays*
GCC	*GNU Compiler C*
IBM	*International Business Machines Corporation*
ICW	*Instruction Code Words*
ISA	*Instruction Set Architecture*
LAT	*Line Address Table*
MIPS	*Microprocessor without Interlocked Pipeline Stages*
MTF	*Move-to-Front coding*
NOP	*No Operation*
PC	*Program Counter*
PDAs	*Personal Digital Assistants*
PDC	*Processor Decompressor Cache*
PISA	*Portable ISA*
PLC	*Preloaded Loop Cache*
RAM	*Random Access Memory*
RISC	*Reduced Instruction Set Computer*
ROM	*Read Only Memory*
RUU	*Register Update Unit*
SCE	*Execution Driven Simulation*
SCP	*Program Driven Simulation*
SCR	*Trace Driven Simulation*
SCW	*Sequence Code Words*
SFC	*Static Frequency based Compression*
SIMD	*Single Instruction Multiple Data*
SoC	*System-on-Chip*
SPARC	*Scalable Processor Architecture*
SPEC	*Standard Performance Evaluation Corporation*
TLB	*Translation Look-aside Buffer*
ULA	*Arithmetic Logic Unit*
VHDL	*Very High Speed Integrated Circuit Hardware Description Language*

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO

Em pouco mais de três décadas, os computadores causaram uma grande revolução em nossos hábitos diários, gerando uma nova era onde em poucos cliques é possível termos um mundo de informações ao nosso alcance.

Os sistemas de processamento de informação que estão incorporados em um produto maior e que normalmente não estão diretamente visíveis aos usuários são chamados de sistemas embarcados [48] e colaboram com esta nova tendência da computação, chamada de *disappearing computer* [42a], que tem o princípio de que a computação acontece em todo lugar (computação ubíqua), porém de forma invisível, sendo realizados em dispositivos com aparências que fogem do tradicional "gabinete e monitor" e cuja presença muitas vezes não se consegue identificar.

Então, defini-se que os sistemas embarcados são quaisquer sistemas digitais que estejam incorporados a outros sistemas com a finalidade de acrescentar ou otimizar funcionalidades [42, 48, 61]. Assim, os sistemas embarcados têm por função monitorar e/ou controlar o ambiente no qual esteja inserido, conforme OLIVEIRA & ANDRADE [48]. Esses ambientes podem estar presentes em dispositivos eletrônicos (celulares, *smartphones*, console de jogos portáteis, aparelhos de som, DVD player, televisores, telefone sem-fio), eletrodomésticos (forno microondas, geladeira, máquina de lavar roupa, ar-condicionado), veículos (controle de freio ABS, computador de bordo, injeção eletrônica), motores (controladores de grupo geradores, controladores de pressão), máquinas (controle de esteiras, braços mecânicos), ambientes físicos (módulos de sensoriamento em uma rede de sensores sem-fio monitorando um habitat), medicina (controle de equipamentos de ultrasom, tomografia), entre outros [48].

Devido à crescente demanda pelo uso dos sistemas embarcados nos dias atuais, tem se tornado cada vez mais comum a implementação de complexos sistemas em um único *chip*, os chamados *System-on-Chip* (SoC). No entanto, o processador embarcado é um dos principais componentes dos sistemas computacionais embarcados, segundo BENINI *et al* [4, 5]. Esses processadores para aplicações embarcadas, geralmente eram simples, ou seja, CPUs de apenas 8 ou 16 *bits*. Mas com a crescente complexidade das atuais aplicações embarcadas, os processadores embarcados tiveram que se modernizar para uma arquitetura de alto-desempenho (arquitetura RISC de 32 *bits*) que garante um melhor desempenho computacional para as tarefas a serem executadas. Desse modo, o projeto de sistemas embarcados para processadores de alto-desempenho não é uma tarefa simples, uma vez que as arquiteturas RISC geraram um aumento significativo no tamanho dos códigos das aplicações.

Os sistemas embarcados apresentam inúmeras limitações físicas e de recursos computacionais, sendo a memória um dos recursos mais críticos. Nos projetos de sistemas embarcados este recurso usualmente representa um dos componentes mais caros. Então, tudo isto justifica o esforço para otimizar o seu uso. Uma das técnicas desenvolvidas para isso é a compressão do código.

Nas duas últimas décadas muito se estudou acerca do uso da compressão de código para satisfazer a um tamanho maior das aplicações atuais (em termos de códigos) e/ou a uma menor área de silício dedicada à memória do sistema. Em 1992, WOLFE & CHANIN [62] propuseram o *Compress Code RISC Processor* (CCRP) e introduziram a compressão de código para as arquiteturas RISC. Dezenas de pesquisas se sucederam, mas, a maioria se dedicou prioritariamente apenas em reduzir o tamanho do código das aplicações, ou seja, a taxa de compressão obtida pela técnica.

À medida que os sistemas embarcados tornam-se mais heterogêneos os mesmos admitem maior complexidade em seu desenvolvimento; Por outro lado, os softwares que neles operam também ampliam o seu grau de complexidade, causando assim o aumento significativo em seu código na memória. Nesse sentido, surgiu uma técnica de alto nível que procura comprimir o código em tempo de compilação (modo *offline*) e a respectiva descompressão, que por sua vez, é feita em tempo de execução (modo *on-the-fly*), segundo WOLFE & CHANIN [62]. A Figura 1.1 mostra uma visão geral da compressão e descompressão de código.

A técnica de compressão foi desenvolvida com o intuito de reduzir o tamanho do código das instruções de uma aplicação [47]. Mas com o passar do tempo, grupos de pesquisadores verificaram que essa técnica também poderia trazer benefícios para o desempenho e o

consumo de energia nos sistemas de propósitos gerais e nos sistemas embarcados [45, 46]. De acordo com NETTO [44] a partir do momento que o código em memória está comprimido é possível em cada requisição do processador, buscar uma quantidade maior de instruções contidas na memória. Assim, haverá uma diminuição nas atividades de transição nos pinos de acesso à memória, levando a um possível aumento no desempenho do sistema e a uma possível redução no consumo de energia do sistema.

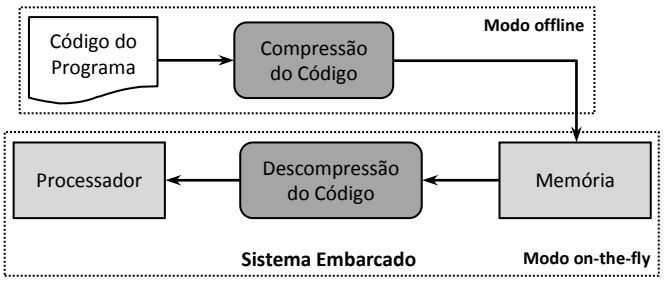


Figura 1.1 – Visão geral da compressão e descompressão de código

Na busca pelo aperfeiçoamento das técnicas existentes, surge a necessidade de criar novas estratégias capazes de aumentar a capacidade de armazenamento de códigos na hierarquia de memória, visando melhorar o desempenho computacional em termos de tempo de execução assim também como reduzir o consumo de energia nos sistemas embarcados.

Portanto, a principal contribuição desta tese é propor e implementar novos métodos de compressão/descompressão de código para sistemas embarcados que sejam eficientes computacionalmente na tríade compressão-desempenho-consumo, gerando menos *overhead* possível com a inclusão do hardware descompressor. Para isso, criou-se um novo tipo de dicionário, dividido em níveis, que permite armazenar instruções unitárias e padrões de blocos.

1.1 Motivação

A compressão de código tem sido objeto de vários estudos e pesquisas nos últimos anos. Atualmente existem dois tipos básicos de arquiteturas usadas na compressão de código chamadas de CDM (*Cache Decompressor Memory*) e PDC (*Processor Decompressor Cache*). Na arquitetura CDM o hardware descompressor é inserido entre as memórias *cache* e principal, enquanto que na arquitetura PDC o hardware descompressor é inserido entre o processador e a memória *cache*.

Uma das vantagens de se usar a arquitetura CDM é a possibilidade de obter um método de compressão de código que alcance uma razão de compressão consideravelmente maior,

mas, a descompressão nessa arquitetura geralmente é mais lenta. Já a arquitetura PDC sempre apresenta melhores resultados no desempenho e no consumo de energia do sistema, isso ocorre devido à alta taxa de acertos (*hit rates*) na *cache* de instrução. Por outro lado, a inserção do descompressor nessa arquitetura geralmente acarreta um aumento no *cycle-time* e alterações no núcleo do processador.

Analisando de forma resumida alguns métodos de compressão de código encontrados na literatura, constatamos que o método CCRP (*Compress Code RISC Processor*) [62] é considerado como o primeiro método de compressão implementado para um processador RISC (MIPS R2000) e que usa as falhas de acessos à memória *cache* para acionar o mecanismo de descompressão. Detectamos como um dos problemas desse método a falta de expressividade nos resultados obtidos, ou seja, a razão de compressão de 73% não é um valor real, mas sim apenas uma estimativa inicial deduzida pelos autores. Outro problema é que a área de silício usada pelo hardware descompressor não foi computada na razão de compressão, o que nos dá uma visão distorcida sobre o resultado final da compressão. Outro ponto desfavorável no método CCRP é a complexidade exigida na inclusão do mecanismo de descompressão – composto pela tabela chamada de LAT (*Line Address Table*), pelo *buffer* de memória chamado de CLB (*Cache Line Address Look-aside Buffer*) e pela máquina de descompressão conhecida por *Cache Refill Engine* – na arquitetura do processador usado por esse método.

O algoritmo do método *Instruction Splitting* [11] realiza uma análise exaustiva em cada *bit* das instruções do código na busca de encontrar os padrões repetitivos, é usada a Codificação *Canônica* de *Huffman* para comprimir os padrões repetitivos encontrados no código. Um dos problemas apresentados por esse método é a variação no tamanho das instruções pertencentes à tabela de decodificação e dos índices, tornando difícil o processo de decodificação quando implementado em hardware. Além disso, o tamanho necessário para armazenar a tabela de decodificação da memória é grande.

O método Compressão por *Codeword* [34] realiza a compressão de código baseado na codificação do programa usando um dicionário de códigos. Apenas as instruções mais frequentes no código são comprimidas, sendo as mesmas substituídas por uma palavra codificada (*codeword*). As instruções correspondentes às instruções comprimidas são armazenadas em um dicionário no hardware de descompressão e o código final consiste de *codewords* misturadas com instruções não comprimidas. A razão de compressão obtida pelos autores não reflete a realidade, uma vez que não foi computada a área usada pelo dicionário.

Outro problema é a necessidade de alterações no núcleo do processador para inserção de um hardware extra que trata os problemas de desvio para endereços alinhados a 4 *bits*.

Ainda destacamos que o método *Bitmasks Compression* [54, 55] realiza a compressão de código utilizando máscaras de *bits* para melhorar a razão de compressão, sem que haja muito *overhead* no sistema com o uso do hardware descompressor. Um dos problemas nesse método é a necessidade da limitação no tamanho do dicionário, pois um dicionário maior que 256 entradas aumenta significativamente o tempo de acesso ao mesmo e, como consequência, reduz a eficiência da descompressão. Já o método *Merge Bitmask and DISE* [58] é formado pela combinação dos métodos *Bitmasks* e *DISE* (*Dynamic Instruction Stream Editing*), esse método tende a reduzir o tamanho do dicionário e das *codewords*. No entanto, o algoritmo implementado possui uma elevada complexidade computacional, o que possivelmente o torna não aplicável em alguns projetos de sistemas embarcados que utilizam técnicas de compressão de código. O método *Bitstream Compression* [50] é uma nova técnica que usa *bitstream* (instrução binária) comprimida para realizar a descompressão paralela, sem prejudicar a eficiência da compressão. Um dos problemas apresentados nesse método é que o início de cada campo da instrução comprimida é desconhecido, assim é preciso descomprimir todos os campos anteriores da instrução, pois sem que isso seja feito não é possível decodificar simultaneamente todos os campos da mesma instrução.

Existem vários outros métodos de compressão de código, mais detalhes destes e de outros métodos são apresentados nos Capítulos 2 e 3 dessa tese. Detectamos que em média a razão de compressão obtida ficou em torno de 70%, o desempenho final do sistema em 90% e o consumo de energia em 75%.

Um dos problemas mais comuns quando se trabalha com a compressão de código é a falta de um método que obtém resultados satisfatórios para diferentes programas de diversas classes de aplicações e que seja aplicado em vários tipos de processadores embarcados, além da facilidade na implementação e implantação nos projetos de sistemas embarcados.

Portanto, a principal motivação desta pesquisa é a exploração de uma área de conhecimento e de interesse, aliada à oportunidade de desenvolver novos métodos de compressão de código que apresentem uma boa tríade compressão-desempenho-consumo, e que seja aplicado em projetos de sistemas embarcados que fazem o uso de técnicas de compressão de código.

1.2 Contribuições da Tese

Nesse trabalho são propostos quatro métodos de compressão de código baseados em dicionário, sendo denominados de CPB-ARM (*ComPatternBlocks-ARM*), HDPB (*Code Compression using Huffman and Dictionary-Based Pattern Blocks*), CCHPB (*Compressed Code using Huffman + Pattern Blocks and Multi-Level Dictionary*) e CC-MLD (*Compressed Code using Huffman-Based Multi-Level Dictionary*), onde a principal característica analisada para o processo da compressão é a redundância das instruções que geram os padrões de blocos de instruções no código dos programas. Um dos principais objetivos em comprimir os padrões de blocos é evitar múltiplos acessos à memória principal do sistema e também uma redução mais significativa no tamanho do código dos programas.

O método CPB-ARM é otimizado apenas para o conjunto de instruções do processador embarcado ARM e os outros métodos são independentes das características específicas do conjunto de instruções, podendo ser aplicados em quaisquer processadores embarcados.

Um novo tipo de dicionário é criado e usado pelos métodos HDPB, CCHPB e CC-MLD. Até então a maioria dos dicionários continham apenas instruções unitárias. Em nossa abordagem mostramos que usar apenas esse tipo de instrução torna alguns resultados ineficientes, por isto propomos alguns métodos que comprimem tanto instruções unitárias quanto padrões de blocos. Assim em um único dicionário denominado de "multi-nível" é possível armazenar esses dois tipos de instruções em níveis diferentes. Em resumo, as contribuições dessa tese são:

- Quatro novos métodos de compressão de código baseados em dicionário que são compostos basicamente por padrões de blocos de instruções;
- Um novo formato de dicionário dividido em níveis, que armazena em níveis diferentes do mesmo dicionário instruções do tipo unitárias e padrões de blocos;
- A arquitetura de um hardware decompressor implementado em VHDL e prototipado em uma FPGA que se caracteriza por ser simples, economico e eficiente.

1.3 Organização da Tese

O texto dessa tese está organizado em sete capítulos que fornecem toda a base conceitual e empírica para o entendimento completo dos novos métodos (algoritmos) de compressão/descompressão de códigos propostos e desenvolvidos nessa tese. Assim o Capítulo 2 apresenta uma visão geral em termos de compressão de código bem como a descrição de alguns trabalhos relacionados encontrados na literatura; o Capítulo 3 apresenta

detalhadamente alguns trabalhos correlatos desenvolvidos recentemente e que usaram a codificação baseada em dicionário; o Capítulo 4 apresenta a metodologia de avaliação, a ferramenta *SimpleScalar*, os *benchmarks* destacando o *MiBench*, as características do processador embarcado ARM, e por fim, a infraestrutura utilizada nessa tese. O Capítulo 5 primeiramente apresenta uma análise na redundância das instruções nos programas do *MiBench* justificando a compressão das instruções em padrões de blocos, em seguida apresenta em detalhes o método CPB-ARM. O Capítulo 6 apresenta um novo tipo de dicionário múltiplo proposto e desenvolvido nessa tese, em seguida apresenta detalhadamente os método HDPB, CCHPB e CC-MLD que também são contribuições centrais dessa tese, e por fim, faz uma análise comparativa entre os métodos desenvolvidos na tese e os principais métodos estudados nos Capítulos 2 e 3. E finalmente, o Capítulo 7 apresenta o fechamento do trabalho destacando as principais contribuições e também sugerindo possíveis trabalhos futuros.

CAPÍTULO 2

ARQUITETURAS DE COMPRESSÃO DE CÓDIGO

Nos anos 70 iniciaram os primeiros estudos sobre compressão de código usando-se de um conjunto de instruções que foi projetado para reduzir a utilização de memória que na época era escassa e cara. Em 1972, os projetistas do *Burroughs* B1700 [60] desenvolveram uma abordagem baseada na utilização das instruções para determinarem o tamanho dos campos das instruções, ou seja, as instruções mais frequentes foram associadas a campos menores e as instruções menos frequentes a campos maiores, usando assim o método de codificação de *Huffman* [27].

O primeiro método de compressão de código usando a arquitetura RISC (*Reduced Instruction Set Computer*) foi proposto em 1992 por WOLFE & CHANIN [62]. Desde então, diversos métodos vêm sendo propostos.

Este capítulo, em primeiro lugar, apresenta conceitos sobre compressão de dados e código (Seção 2.1), que apesar dos métodos de compressão de dados não serem apropriados para a compressão de códigos, formam a base para diversos métodos propostos e estudados neste capítulo. Em seguida, descrevemos as técnicas de compressão de código (Seção 2.2); adiante, enfatizamos quais são seus reais custos e benefícios (Seção 2.3); Na Seção 2.4, apresentamos as arquiteturas existentes na literatura, tais como: arquitetura CDM e PDC juntamente com os seus principais trabalhos correlatos. Em seguida, na Seção 2.5 fazemos um comparativo entre os principais métodos de compressão de código mencionados nas Subseções 2.4.1 e 2.4.2. E por fim, uma conclusão do capítulo.

2.1 Compressão de Dados e Código

Nesta seção apresentamos uma introdução à compressão de dados e de código, apontando as características que tornam a compressão de código diferente da compressão de dados.

2.1.1 Compressão de Dados

É uma área vastamente explorada. Os avanços 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 [40]) desenvolveram um método sistemático para atribuir *codewords* baseado em probabilidade a blocos de informações. Os blocos, ou subtrechos, de informações mais frequentes eram substituídos por *codewords* menores. Ao final, a informação era representada por um conjunto de *codewords* pequenos, que poderiam, então, ser substituídas novamente pelos blocos associados previamente para recuperar o conteúdo original. HUFFMAN [27] propôs um método ótimo para escolha desses *codewords*. Na década de 70, LEMPEL & ZIV [64, 65] 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 imagem) 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 sequência de acesso às instruções pode ser aleatória, 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 sequências de instruções aleatórias de forma eficiente. Essas características inviabilizam ou dificultam a utilização das técnicas de compressão de dados para comprimir códigos.

2.1.2 Compressão de Código

Constitui-se na criação de uma imagem do programa original, ou seja, uma cópia do programa formada com menos *bits*. Porém, para a descompressão do código é necessário usar uma rotina de descompressão (via software) ou um mecanismo implementado em hardware que realiza a descompressão do código comprimido.

Existe um grande número de métodos de compressão proposto na literatura. A compressão de código pode ser obtida no próprio projeto do processador, utilizando campos menores para codificar instruções mais frequentes e campos maiores para codificar as instruções pouco frequentes, o projeto *Burroughs* de WILNER [60] é um exemplo desse método. Na compressão/descompressão de código é essencial a escolha de um método de compressão no qual não seja gerado um código descomprimido errôneo, ou seja, sem perdas de instruções.

A compressão de código geralmente é realizada em tempo de compilação e a descompressão de código é realizada em tempo de execução [4, 34, 45]. Porém, na descompressão de código não é possível aplicar os métodos de compressão de dados (tais como: *Shannon-Fano*, Aritmético, *Elias-Bentley*, *Lempel-Ziv* e outros) [56, 20, 6, 64], uma vez que os métodos de compressão de dados só permitem a descompressão do código de forma sequencial e não permitem a descompressão do código de forma aleatória.

A descompressão de código nem sempre pode ser realizada de forma sequencial devido às instruções de desvios (controle de fluxos) existentes nos programas. A Figura 2.1 mostra um exemplo de um trecho de código que contém instrução de desvio.

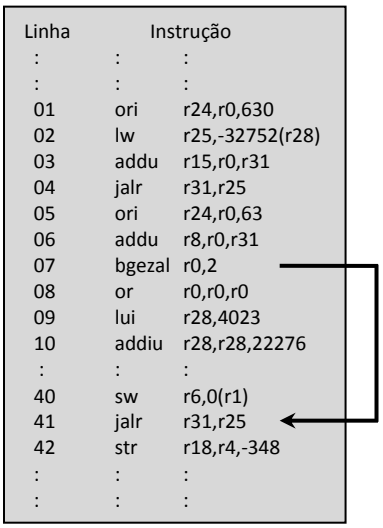


Figura 2.1 – Exemplo de trecho de código que contém instrução de desvio [2]

No exemplo da Figura 2.1, supondo que a instrução de desvio da linha 07 tenha que ser executada e que a instrução alvo do desvio seja a instrução da linha 41, para que a execução do programa seja feita corretamente é necessário enviar ao processador as instruções a partir da linha 41. Porém, se o trecho de código da Figura 2.1 fosse comprimido utilizando qualquer método de descompressão de dados, a próxima linha a ser descomprimida seria a linha de instrução subsequente à linha da instrução 07, passando ao processador uma informação errônea.

2.2 Técnicas de Compressão de Código

Existem diferentes técnicas de compressão de código como mostrado em [7]. Dentre as diversas classificações e características podemos citar três tipos de técnicas que se destacam na compressão de código executável, sendo:

2.2.1 Codificação por Frequência

Os símbolos com maior frequência são codificados com sequências menores de *bits* e, dependendo de como se faz a codificação pode-se classificá-la como estática ou dinâmica. Na forma estática os códigos dos símbolos são fixos para toda a informação a ser comprimida, já na dinâmica os códigos podem variar de acordo com as frequências em um determinado instante da codificação. Dentro desta técnica destacamos o método de *Huffman* [27] e suas variações.

2.2.2 Codificação Aritmética

Esta codificação usa probabilidade diretamente no lugar das frequências. A ideia é ter uma linha de probabilidades [0-1) e associar a cada símbolo uma faixa desta linha onde uma maior probabilidade do símbolo corresponde a uma faixa maior. A codificação é feita por transformações lineares e colocação dos símbolos em suas faixas de frequência. A saída é um número real mas com algumas modificações é possível transformá-los em números inteiros para o processamento digital [9].

2.2.3 Codificação Baseada em Dicionário

Nestas técnicas, a sequência de símbolos comuns são codificadas e substituídas pelos seus códigos menores. Há muitas variações desta estratégia, mas se destacam alguns métodos

como o MTF (*Move-to-Front Coding*) e a família de compressores *Lempel-Ziv* [64]. Mais detalhes sobre esta técnica são explicados no Capítulo 3.

2.3 Custos e Benefícios da Compressão de Código

A compressão de código, além de reduzir o tamanho do programa, pode produzir outros efeitos. Uma melhor avaliação desses possíveis efeitos deve ser feita antes de implementar um sistema que utiliza código comprimido. Os dois parâmetros mais avaliados são o tamanho do programa e o desempenho final da execução, mas outras duas medidas podem ser consideradas, que são o consumo de energia e a segurança do sistema. Detalhes sobre esses aspectos são descritos a seguir:

- **Tamanho do Programa**: o principal benefício da compressão é reduzir o tamanho da memória utilizada pelo programa. Para avaliar os métodos de compressão de código, uma medida comumente encontrada na literatura é a razão de compressão (ou taxa de compressão) [3, 52]. A razão entre o tamanho do código comprimido e o tamanho do código original (Equação 2.1) é uma métrica que implica em pensar que quanto menor melhor.

$$razão\ de\ compressão\ 1 = \frac{tamanho\ do\ código\ comprimido}{tamanho\ do\ código\ original}\ x\ 100\ \ (\%) \qquad (2.1)$$

  Conforme NETTO [44], o uso desta equação tem sido feito em duas vertentes diferentes, dificultando o trabalho de comparação entre os métodos de compressão. A primeira vertente, simplesmente desconsidera o tamanho do dicionário em si, apontando o código comprimido apenas como a sequência de *codewords*. Já a segunda vertente, leva em consideração o tamanho do dicionário criado pela compressão. A nova razão e a taxa de compressão são definidas como apresentadas nas Equações 2.2 e 2.3.

$$razão\ de\ compressão\ 2 = \frac{tamanho\ do\ código\ comprimido + tamanho\ do\ dicionário}{tamanho\ do\ código\ original}\ x\ 100\ \ (\%) \qquad (2.2)$$

$$taxa\ de\ compressão = 100 - razão\ de\ compressão\ \ (\%) \qquad (2.3)$$

  Em alguns casos, a diminuição do tamanho do programa pode não ser suficiente para promover a remoção de módulos de memória do sistema final, o que anula ou reduz o ganho da compressão;

- **Desempenho**: neste caso, o sistema de descompressão pode produzir tanto um ganho como uma perda de desempenho, dependendo do modelo a ser utilizado. Perdas podem vir da execução de um número maior de instruções (menores) necessárias para reduzir o tamanho do código, assim como o tempo gasto para descomprimir as instruções [41]. Já os ganhos podem ser encontrados quando o tempo de acesso à memória é muito grande e, devido à menor quantidade de leituras feitas da memória, o sistema como um todo pode executar mais rapidamente. Vale ressaltar que a existência de *caches* no sistema auxilia neste ganho por não exigir que as instruções sejam descomprimidas quando ocorre um *cache hit*. Portanto, para avaliar o desempenho do sistema é usada a Equação 2.4. A métrica de desempenho também implica em pensar que quanto menor melhor.

$$desempenho\ do\ sistema = \frac{tempo\ de\ execução\ (código\ comprimido)}{tempo\ de\ exeução\ (código\ original)}\ x\ 100\ \ (\%) \qquad (2.4)$$

- **Consumo de Energia**: a menor quantidade de memória utilizada e também uma redução no número de transações no barramento de memória podem ou não ser suficiente para compensar o consumo do módulo descompressor quando ele é implementado em hardware [2];
- **Segurança**: embora seja um aspecto pouco considerado, a compressão de programas torna o código executável inelegível e a simples leitura da memória ROM não é suficiente para decodificar (*disassembly*) o programa armazenado nela.

2.4 Arquiteturas de Compressão de Código

Na literatura são encontrados dois tipos básicos de arquiteturas de compressão de código, CDM (*Cache Decompressor Memory*) e PDC (*Processor Decompressor Cache*), que indicam o posicionamento do hardware descompressor em relação ao processador e subsistema de memória, como mostra a Figura 2.2. A arquitetura CDM indica que o descompressor está posicionado entre a *cache* e a memória principal, enquanto que a arquitetura PDC posiciona o descompressor entre o processador e a *cache*.

Como foi visto na Seção 2.1, o desenvolvimento de arquiteturas para compressão ou descompressão de código é feito de forma separada, ou seja, na grande maioria dos trabalhos desenvolvidos só é tratado o hardware descompressor porque a compressão das instruções geralmente é feita por meio de modificações no compilador. Assim, a compressão é realizada

em tempo de compilação e a descompressão é feita em tempo de execução usando um hardware específico para descompressão.

Quando uma arquitetura CDM é usada, o descompressor só é chamado quando há uma falha na *cache* e se a instrução buscada na memória estiver comprimida. Se a taxa de acerto na *cache* é alta, então o tempo de descompressão pode não interferir muito no desempenho do sistema. Por outro lado, se uma arquitetura PDC é utilizada, o código pode ficar comprimido na *cache* de instrução, elevando assim a taxa de acerto. Infelizmente, nesta arquitetura, a descompressão pode ocorrer a cada pedido de instrução do processador, o que exige um descompressor mais rápido em relação à arquitetura CDM.

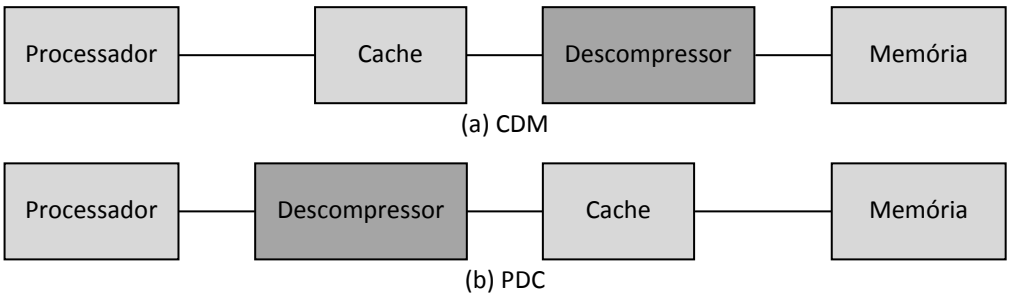


Figura 2.2 – Arquiteturas de descompressão de código (a) CDM e (b) PDC

A seguir, descrevemos detalhadamente os principais métodos de compressão de código em suas respectivas arquiteturas. No caso em que um método é descrito em mais de uma publicação, optou-se pela elaboração dessas publicações em conjunto, quando possível. Às vezes, uma publicação também contém descrições de mais de um método, em tais casos, nós separamos como diferentes métodos.

2.5 Métodos de Compressão com Arquitetura CDM

2.5.1 CCRP (*Compressed Code RISC Processor*)

WOLFE & CHANIN [62] desenvolveram o CCRP, que foi o primeiro hardware descompressor implementado em um processador RISC (MIPS R2000) e também foi a primeira técnica a usar as falhas de acesso à *cache* para acionar o mecanismo de descompressão. O CCRP tem uma arquitetura idêntica ao padrão do processador RISC e assim os modelos dos programas ficaram inalterados. Isso implica em que todas as ferramentas de desenvolvimento existentes para a arquitetura RISC, incluindo compiladores

otimizados, simuladores funcionais, bibliotecas gráficas e outras, também servem para a arquitetura CCRP.

A unidade de compressão usada é a linha da *cache* de instruções. A cada falha de acesso à *cache*, as instruções são buscadas na memória principal, descomprimidas e alimentam a linha da *cache* onde houve a falha [33, 62]. O fato de o CCRP já fazer a descompressão das instruções antes de armazená-las na *cache* é vantajoso, no sentido que os endereços de saltos contidos na *cache* são os mesmos do código original. Isto resolve a maioria dos problemas de endereçamento, não havendo necessidade de recorrer a artifícios como: (i) colocar hardware extra no processador para tratamento diferenciado dos saltos; e (ii) fazer *patches* de endereços de salto.

Porém, ainda existe um problema de endereçamento ao usar este tipo de técnica: no caso de uma falha de acesso à *cache*, é necessário buscar na memória principal as instruções requisitadas, as quais têm endereços diferentes das instruções da *cache*, já que a memória contém código comprimido, enquanto a *cache* contém código não comprimido. Este problema foi resolvido usando uma tabela, chamada de LAT (*Line Address Table*). A LAT foi implantada no hardware de preenchimento da *cache*, e contém os mapas dos programas com os endereços não comprimidos da *cache* em endereços na memória principal [62]. Os dados contidos na LAT são gerados por ferramentas de compressão e armazenado em conjunto com o programa.

Uma restrição imposta para que o mapeamento feito na LAT cubra todo o código original é que o endereço de início de uma linha comprimida seja reconhecido pelo sistema de memória, ou seja, se a memória em questão for alinhada a 32 *bits*, então cada endereço de início de uma linha comprimida também deve estar alinhado a 32 *bits*. Quando isso não ocorre, são usados *bits* de *padding* (preenchimento com zeros) para permitir o alinhamento.

Segundo WOLFE & CHANIN [62] quanto maior a taxa de falhas na *cache*, mais consultas deverão ser feitas à LAT e com isso, mais degradado será o desempenho em comparação ao processador original. Para minimizar esta perda, eles propuseram um mecanismo semelhante a uma TLB (*Translation Look-aside Buffer*), usado em sistemas de memórias virtuais que, neste caso, é chamado de CLB (*Cache Line Address Look-aside Buffer*), responsável por armazenar as últimas entradas consultada da LAT, minimizando significativamente o *overhead* causado nas falhas de acesso à cache.

O descompressor do sistema CCRP foi posicionado entre a *cache* de instrução (*I-cache*) e a memória principal (RAM - *Random Access Memory*), de maneira que as instruções armazenadas na *cache* já estão descomprimidas e prontas para serem usadas pelo processador.

Desta forma, quando o processador solicita a próxima instrução e há um acerto na *I-cache*, a instrução é imediatamente repassada, sem necessidade de acionamento do mecanismo de descompressão. Entretanto, no caso de uma falha, o mecanismo de descompressão (*Cache Refill Engine*) procura na CLB pelo endereço solicitado. Se houver um acerto na CLB, o endereço comprimido correspondente é usado para buscar a instrução na memória, que é então descomprimida e repassada ao *I-cache* e em seguida para o processador. No caso de falha na consulta à CLB, é necessário buscar o endereço na LAT. Após isso, o endereço comprimido obtido da LAT é atualizado na CLB e usado para recuperar a instrução na memória principal, que é então descomprimida e repassada ao *I-cache* e ao processador.

O descompressor conseguiu descomprimir 2 *bytes* por ciclo [62]. Portanto, para descomprimir os 32 *bytes* de uma linha da *cache*, são usados no mínimo 16 ciclos de *clock*, podendo este número aumentar no caso de memórias lentas, onde o mecanismo de descompressão não é o gargalo.

A técnica CCRP de [62] utilizou o método de *Huffman* [27] gerado através de um histograma de ocorrências de *bytes* de programa e mostrou uma razão de compressão de 73%, em média, para o pacote testado (composto pelos programas: *nasa1*, *nasa7*, *tomcatv*, *matrix25A*, *espresso*, *fpppp* e outros), mas esse valor não levou em consideração o tamanho da LAT, CLB e nem do descompressor. Para modelos de memórias DRAM mais lentos o desempenho do processador foi na maioria das vezes suavemente melhorado. Para modelos mais rápidos de memórias EPROM, o desempenho teve uma leve degradação. A Figura 2.3 mostra a arquitetura do CCRP, usada por WOLFE & CHANIN [62].

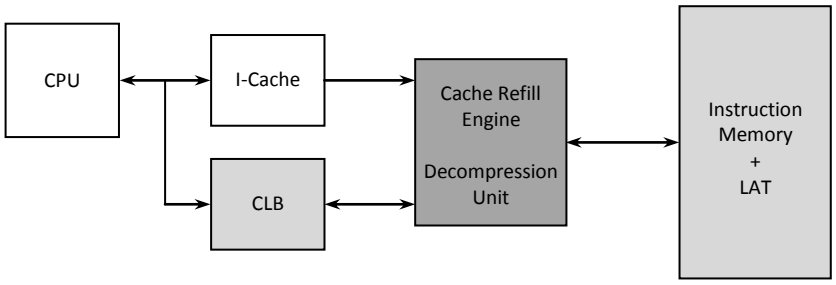


Figura 2.3 – Arquitetura da técnica CCRP [62]

2.5.2 PCACHE

KIROVSKI *et al* [32] propuseram uma abordagem onde um procedimento completo é comprimido e colocado em uma *cache* especial chamada de *pcache*, que é utilizada como repositório para descompressão. A Figura 2.4 mostra a arquitetura da *pcache*.

Quando um procedimento não se encontra descomprimido na *pcache*, ele passa por um descompressor que é embutido na própria *pcache*. Este modelo traz um conjunto de implicações, a saber [32]: mais de um procedimento pode estar armazenado na *pcache* (desde que a soma de seus tamanhos não ultrapasse a capacidade do repositório), isto significa que, em algumas situações, ao alocar um procedimento é necessário retirar outros. Ainda, é possível que haja espaço suficiente para alocação de um procedimento, mas devido ao algoritmo de substituição de código na *pcache*, este espaço pode não ser contíguo.

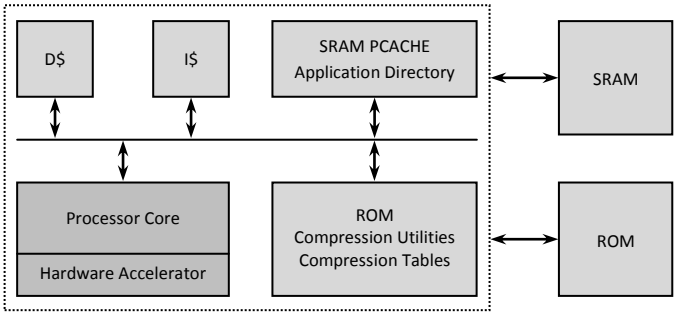


Figura 2.4 – Arquitetura do sistema embarcado para o método *PCACHE* [32]

Um compactador de *pcache* é utilizado fazendo com que os procedimentos não fiquem espaçados no repositório. Se o espaço livre total na *pcache* é menor que o necessário para alocar o procedimento chamado, então é necessário substituir alguns procedimentos. É preciso garantir também que a *pcache* tenha no mínimo o tamanho do maior procedimento.

Quanto ao endereçamento dos procedimentos, um serviço de diretório foi utilizado ligando o identificador do procedimento ao endereço comprimido. O algoritmo utilizado para compactar os procedimentos foi uma versão do clássico *Lempel-Ziv* [64, 65] que atingiu 60% de compressão (razão de compressão de 40%) usando *bytes* como símbolos para a arquitetura SPARC. Com uma *pcache* de 64 *KBytes*, o impacto negativo no desempenho chegou a 11% ou, se incluso o *gcc* e o *go* no conjunto de *benchmarks* usados, 166%. Para uma *pcache* de 32 *KBytes*, este número foi de 36% e 600% respectivamente. A razão de compressão não considerou o tamanho da *pcache* nem o hardware associado para executar as políticas de substituição de código.

2.5.3 CodePack

A IBM em 1998, desenvolveu o CodePack [28, 29] que utilizava uma codificação baseada em ocorrências e implementada com dicionários para integrar a família de microprocessadores embarcados como o PowerPC 401 e 405. A Figura 2.5 mostra a arquitetura do CodePack. Cada instrução de 32 *bits* é dividida em duas partes de 16 *bits* (meia

palavra) e um histograma de cada uma delas é montado. Com base no número de ocorrências de cada parte, a instrução é codificada. O esquema de codificação requer um prefixo para cada conjunto de 16 *bits* e a partir daí os *bits* seguintes são conhecidos.

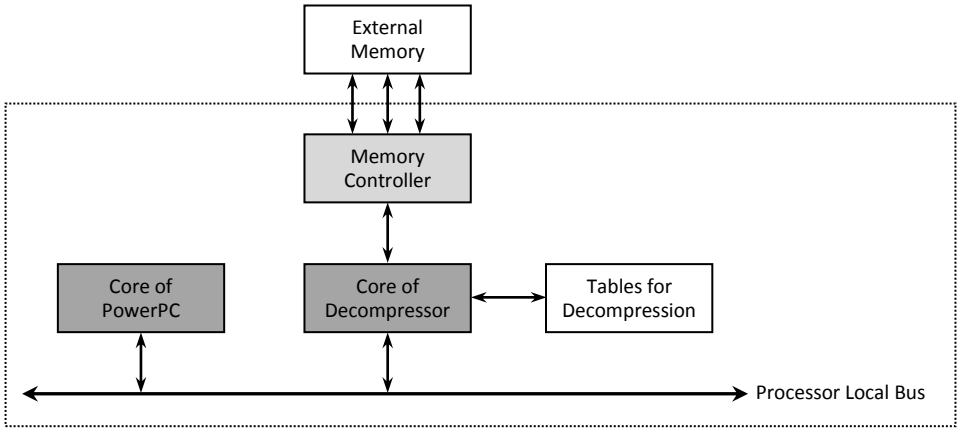


Figura 2.5 – Arquitetura do CodePack [28]

O prefixo não tem tamanho fixo, portanto, a decodificação se torna mais complexa. As árvores na Figura 2.6 mostram os prefixos usados na codificação de cada uma das partes da instrução. Por exemplo, uma instrução que tenha sua codificação da parte alta como 00111_2 significa ter dois *bits* de prefixo (00_2) e três de índice (AAA=111_2) apontando para uma tabela de 8 posições contendo as mais frequentes ocorrências destes 16 *bits*. Ao encontrar uma codificação 0100000_2 significa que aponta-se para meia palavra de índice 00000_2 numa outra tabela que guarda as próximas 32 meia palavras mais frequentes. O 16xY significa uma meia palavra original.

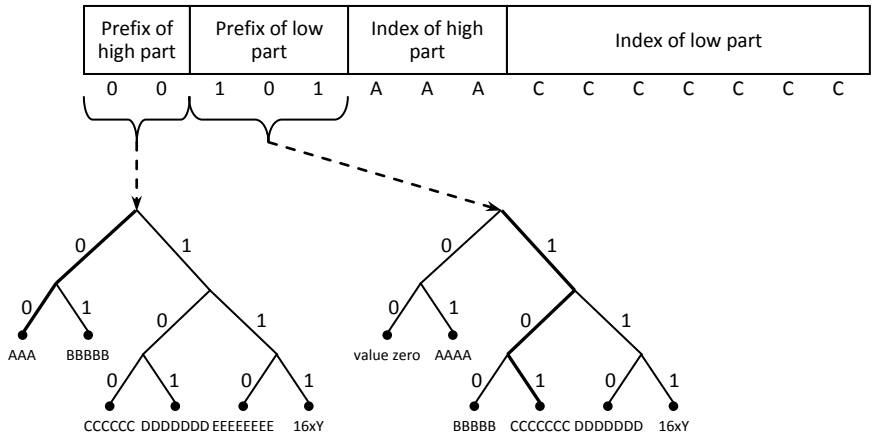


Figura 2.6 – Árvore de prefixos usadas no CodePack da IBM [28]

Neste método, a instrução de maior ocorrência na parte baixa e na parte alta pode ser codificada com 7 *bits*. A instrução comprimida é montada com a seguinte sequência de *bits*:

prefixo da parte alta, prefixo da parte baixa, índice da parta alta, índice da parte baixa. No pior caso, uma instrução pode ser codificada com 38 *bits*, sendo: 6 *bits* de prefixos e os 32 *bits* originais. É interessante notar que a distribuição de ocorrências da parte alta e da parte baixa das instruções é bem diferente, isto porque na parte alta ficam os *opcodes* e registradores e na parte baixa ficam, normalmente, os endereços imediatos. Assim, dois dicionários distintos são utilizados.

Para encontrar uma instrução comprimida na memória a solução foi usar tabelas, desta vez formando uma hierarquia, mas com o mesmo objetivo de tradução de endereços. A descompressão ocorre sempre que houver uma falha na *cache*, como no CCRP. A razão de compressão reportada pela IBM chegou a 60%, incluindo a tabela de tradução de endereços, enquanto o desempenho variou 10% para mais ou menos. Um estudo interessante deste método foi feito por LEFURGY *et al* [35], que analisou a quantidade de falhas na *cache*, para processadores do tipo: *single issue*; 4-*issue*; 8-*issue* e *out-of-order* e usou diversas otimizações. A conclusão evidenciada é que muitas vezes a compressão de código propicia uma melhoria no desempenho da máquina, além da já esperada redução no tamanho do código.

2.5.4 IBC (*Instruction Based Compression*)

AZEVEDO [2] propôs um método chamado de IBC, que tem por função realizar a divisão do conjunto de instruções do processador em classes, levando em consideração a quantidade de ocorrências juntamente com o número de elementos de cada classe. AZEVEDO [2] mostrou melhores resultados na compressão de 4 classes de instruções. A técnica de compressão desenvolvida consiste em agrupar pares no formato [prefixo, *codeword*] que substituem o código original. Nos pares formados, o prefixo indica a classe da instrução e o *codeword* serve como um índice para a tabela de instruções. Como nos outros métodos vistos anteriormente, este também necessita de uma tabela LAT.

O processo de descompressão foi realizado em 4 estágios de *pipeline*. O primeiro estágio é chamado de Input onde é convertido o endereço do processador (código não comprimido) em endereço da memória principal. O segundo estágio é chamado de Fetch, que é responsável pela busca da palavra comprimida na memória principal. O terceiro estágio é conhecido como Decode onde verdadeiramente é realizada a decodificação dos *codewords*. E finalmente no quarto estágio, chamado de Output, é realizada a consulta no dicionário de instrução para ser fornecida a instrução ao processador. Nos testes realizados em [2] o autor obteve uma taxa de compressão de 53,6% para o processador MIPS [68] e 63,9% para o

processador SPARC usando programas do *benchmark SPECint95*. Quanto ao desempenho, constatou-se uma perda de 5,9% utilizando o método IBC para o processador SPARC. A Figura 2.7 mostra a arquitetura do descompressor de código IBC proposta por AZEVEDO [2] para o processador SPARC v8 LEON.

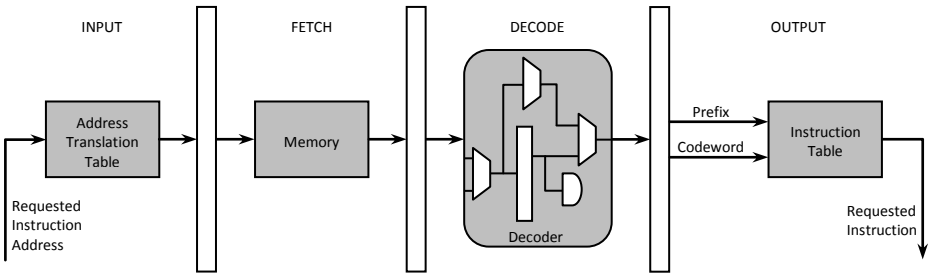


Figura 2.7 – Arquitetura do descompressor IBC para o processador SPARC v8 LEON [2]

2.5.5 Instruction Splitting

BONNY & HENKEL [11] desenvolveram um método de compressão de código onde as instruções originais são divididas em padrões de comprimento variável e depois é aplicada a codificação de *Huffman* para os padrões obtidos. As instruções obtidas após a sua divisão podem ser classificadas em dois tipos: (1) instruções que não se repetem dentro do aplicativo (frequência = 1), chamadas de "instruções únicas"; (2) instruções que se repetem dentro do aplicativo (frequência > 1), também chamadas de "instruções repetidas".

A Figura 2.8 mostra como as instruções originais são divididas. A primeira instrução não pertence ao tipo de instruções únicas porque de fato se repete no exemplo. Portanto, ela é deixada sem dividir. A segunda instrução é uma instrução única. O algoritmo de divisão considera que a melhor forma de divisão é (3,5), ou seja, um padrão de 3 *bits* seguido por um padrão de 5 *bits* (porque a primeira parte da instrução "11101" é repetida 3 vezes e a segunda parte da instrução "000" é repetida 5 vezes dentro de todos os padrões das instruções no exemplo). A terceira instrução é dividida na forma (2,2,3), também por razões de frequências de repetição vantajosa em todos os padrões e assim por diante.

O algoritmo de divisão encontra a maioria dos padrões repetitivos de instruções entre todas as outras instruções. Os padrões podem ter comprimentos diferentes, sendo entre 2 *bits* até 32 *bits*. Depois disso, os novos códigos são comprimidos usando a codificação de *Huffman*.

No método de compressão de BONNY & HENKEL [11], as entradas da tabela de decodificação dos códigos de comprimento são variáveis e os índices (ou seja, as instruções codificadas) para essas entradas também possuem comprimento variável. Isto tornou difícil a

decodificação quando foi implementado em hardware. Além disso, o tamanho necessário para armazenar a tabela de decodificação da memória é grande. Para resolver estes problemas, os autores usaram a *Codificação Canônica de Huffman*, como feito em [10, 12]. Esta técnica de codificação (Canônica) "recodifica" as instruções codificadas de tal forma que as instruções com o mesmo comprimento são representações binárias de números inteiros consecutivos [43]. Dessa maneira, os autores constataram que a decodificação tornou-se mais eficiente em espaço e tempo [30], porque requer que menos informações sejam armazenadas para a decodificação.

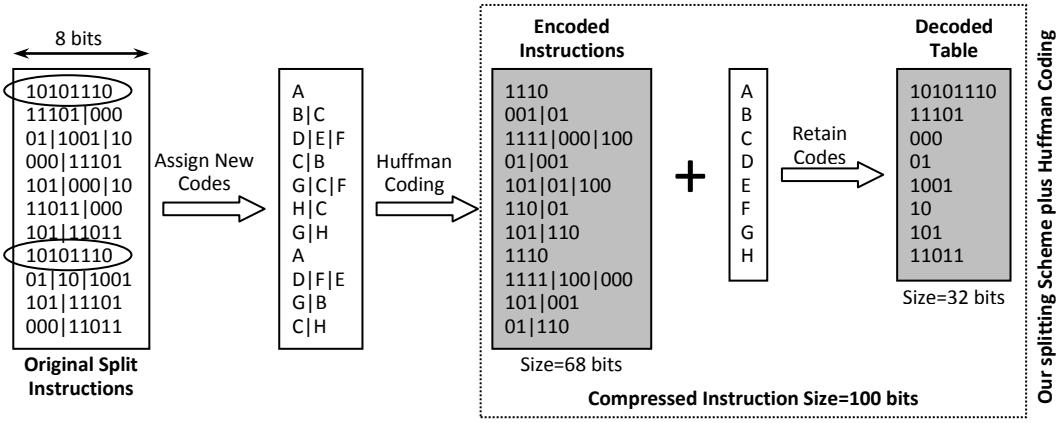


Figura 2.8 – Exemplo da divisão das instruções originais [11]

A Figura 2.9 mostra a aplicação do método de *Codificação Canônica de Huffman* usado pelos autores, porém, observa-se que os índices (as instruções codificadas) para a tabela de decodificação têm 3 comprimentos diferentes (2, 3 e 4 *bits*).

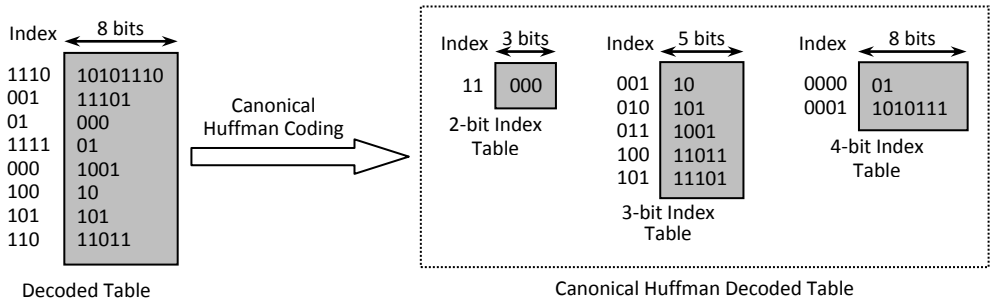


Figura 2.9 – Exemplo da aplicação do método de compressão com a *Codificação Canônica de Huffman* [11]

Observa-se na Figura 2.9 que a tabela de decodificação foi dividida em 3 sub-tabelas, uma para cada comprimento de índice diferente [43], precisando recodificar esses índices de forma que eles se tornem contínuos em cada tabela. A recodificação das instruções codificadas muda estas instruções, mas não muda seus tamanhos determinados usando o

Código de *Huffman*, porque a *Codificação Canônica de Huffman* mantém o tamanho das instruções codificadas geradas a partir da Codificação de *Huffman*. Para fazer com que o comprimento das entradas em cada tabela seja uniforme, a tabela é dividida em tabelas adicionais, uma para cada comprimento de entradas diferentes. Na Figura 2.9, a tabela de índice de 5 *bits*, por exemplo, é dividida em 4 tabelas diferentes, uma vez que tem 4 entradas de diferentes comprimentos.

A arquitetura do hardware descompressor é mostrada na Figura 2.10. Segundo BONNY & HENKEL [11], ela é composta por 2 registradores de deslocamento (*Shift Register*), 1 demultiplexador e 2 grupos de comparadores. O primeiro registrador recebe a palavra de instrução de 32 *bits* comprimida da memória. Esta palavra de instrução pode conter uma ou mais palavras de instruções comprimidas ou a instrução original completa ou partes dela. A tarefa principal deste registrador é manter o segundo registrador de deslocamento cheio e cada vez que o seu conteúdo é reduzido, desloca-se a palavra de instrução comprimida em série para ele. O segundo registrador tem um comprimento igual à maior instrução codificada (L), ou seja, o maior índice (na Figura 2.9, L=4). Ele (o segundo registrador) transfere o índice da tabela de L *bits* para o primeiro grupo dos comparadores. A tarefa dos comparadores é decodificar o comprimento das instruções codificadas a partir dos *bits* de entrada L. O número de comparadores neste grupo é igual ao comprimento *k* de índices diferentes da tabela (na Figura 2.9, *k*=3), ou seja, a quantidade de tabelas formadas ou *Table Decoded* (3, sendo de 2, 3 e 4 *bits*). Cada comparador faz comparação dos *bits* de entrada L com o índice mínimo da tabela correspondente. Se os *bits* de entrada L são maiores ou iguais ao índice mínimo da tabela, o comparador correspondente de saída é "1", caso contrário, "0". O seletor da tabela observa as saídas do comparador para encontrar o menor comparador que gera um "1". O número deste comparativo refere-se à tabela correspondente que contém a instrução original (ou parte dele). O segundo grupo dos comparadores especifica o comprimento da instrução original dentro da tabela que foi selecionada a partir do primeiro grupo de comparadores.

Todo o descompressor foi descrito em VHDL e sintetizado utilizando a ferramenta ISE-8.1 da Xilinx para o Virtex-II e implementado em uma plataforma de FPGA escalável "*Platinum*" da *Pro-Design*. O tempo médio de acesso foi de 4ns e usou aproximadamente 1.200 CLBs (*Configurable Logic Blocks*). O método desenvolvido em [11] é independente de qualquer arquitetura e de seu conjunto de instruções. Os autores usaram o *benchmark MiBench* para obter as medições de desempenho e as razões de compressão alcançadas para os processadores ARM [1, 53], MIPS [59a] e PowerPC [42b] foram 59%, 60% e 62% respectivamente.

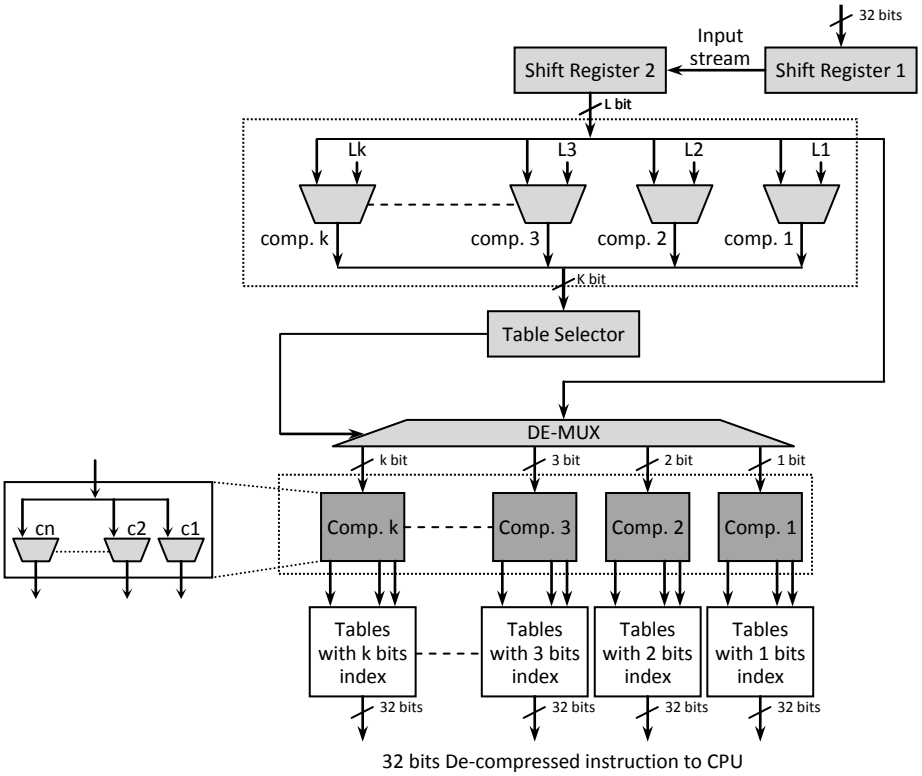


Figura 2.10 – Arquitetura do hardware descompressor proposto em [11]

2.6 Métodos de Compressão com Arquitetura PDC

2.6.1 Agrupamento por Características Próprias

LEKATSAS *et al* [37] desenvolveram um método de compressão de código onde as instruções foram agrupadas separadamente por suas características próprias. As instruções da plataforma SPARC foram utilizadas para a criação dos grupos de instruções, sendo [37]:

- **Grupo 1 - Instruções com Endereços Imediatos**: é utilizada a codificação aritmética. Sabe-se que este tipo de codificação possui vantagens significativas sobre o método de compressão de *Huffman*, especialmente apropriada quando as ocorrências das instruções com endereços imediatos são bastante espaçadas. Na maioria das vezes, a tabela de codificação é criada de forma mais genérica e produz uma razão de compressão melhor do que a codificação de *Huffman* [3];
- **Grupo 2 - Instruções de Salto**: são comprimidas usando-se da compactação do campo de deslocamento. Para obter a compressão, codifica-se apenas a quantidade mínima de *bits* necessários que representa o deslocamento (salto) em questão;

- **Grupo 3 - Instruções de Acesso Rápido**: são os índices para um dicionário de instruções, portanto, essas instruções podem ser descomprimidas com um simples acesso a um dicionário de código e usando apenas um único ciclo do processador;
- **Grupo 4 - Instruções Não Comprimidas**: são as instruções do código que não foram comprimidas.

A identificação dos grupos é realizada usando uma sequência de *bits*, sendo que [37]: (i) Grupo 1 = 0_2; (ii) Grupo 2 = 11_2; (iii) Grupo 3 = 100_2 e (iv) Grupo 4 = 101_2.

Desta forma, o hardware descompressor consegue realizar de forma correta a descompressão do código de acordo com o seu grupo de instrução, visto que cada grupo possui a sua particularidade, ou seja, uma codificação própria.

O desenvolvimento do descompressor é ajustado com o *pipeline* do processador conforme mostra a Figura 2.11. Foram adicionados quatro *pipelines*, um para cada grupo de instruções descrito em [37]. Conforme a Figura 2.11 observa-se que antes do hardware descompressor existe um *buffer* de entrada (*I-Buf*), que tem por função armazenar as instruções comprimidas vindas da *cache* de instruções (*I-cache*) e após o hardware descompressor existe um *buffer* de saída (*O-Buf*), que tem por função armazenar as instruções descomprimidas e fornecê-las ao *pipeline* do processador à medida que forem requisitadas.

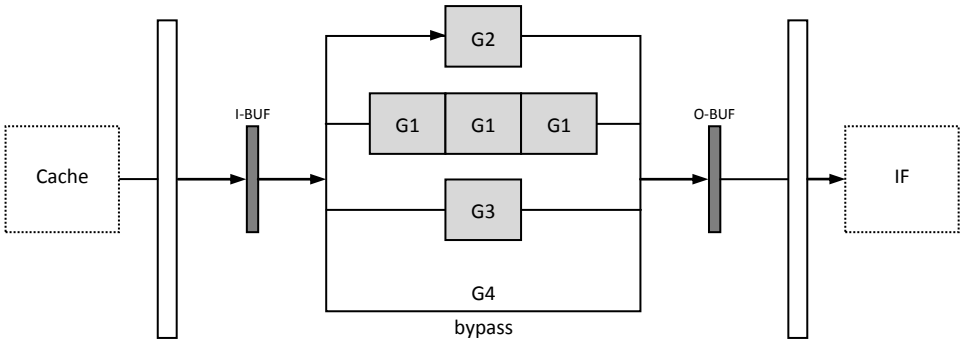


Figura 2.11 – Arquitetura do descompressor *pipelined* proposto por LEKATSAS *et al* [38]

LEKATSAS *et al* [38] obtiveram os seguintes resultados para os programas *compress*, *diesel*, *i3d*, *key*, *mpeg*, *smo* e *trick*: 53% das instruções pertencem ao grupo 1; 26,7% das instruções ao pertencem ao grupo 2; 19,7% das instruções formam o grupo 3 e apenas 0,6% das instruções faz parte do grupo 4, ou seja, são instruções que não são comprimidas. Quanto à razão de compressão ficou em torno de 65%. Já a estimativa de redução no consumo de energia apontou em média um índice de 28% de economia e um ganho de desempenho de

25% em média. Ainda vale ressaltar que nestes valores obtidos não foi levado em conta o *overhead* do hardware descompressor, que é complexo.

Em outro trabalho [36, 39], LEKATSAS *et al* desenvolveram uma unidade de descompressão com um único ciclo utilizando como plataforma alvo o processador *Xtensa*-1040 [31]. A descompressão pode ser aplicada para instruções de qualquer tamanho de um processador RISC (16, 24 ou 32 *bits*). A única aplicação específica é a parte do interfaceamento entre o processador e a memória (principal ou *cache*). O mecanismo de descompressão é capaz de descomprimir uma ou duas instruções por ciclo para atender a demanda da CPU sem aumentar o tempo de execução.

Os autores criaram um dicionário que contém as instruções que aparecem com mais frequência. O dicionário de código refere-se a uma classe de métodos de compressão que substitui sequências de símbolos com os índices de uma tabela. Essa tabela é chamada de "dicionário" e os índices são os "*codewords*" no programa compactado [39]. A principal vantagem dessa técnica é que os índices geralmente são de comprimento fixo, e assim, simplifica-se a lógica da descompressão em acessar o dicionário e também reduz a latência da descompressão. A Figura 2.12 mostra a arquitetura básica proposta por LEKATSAS *et al* em [36].

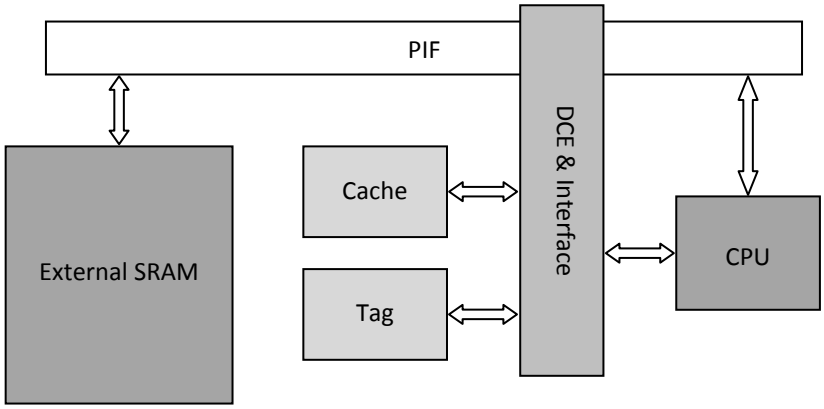


Figura 2.12 – Arquitetura básica proposta por LEKATSAS *et al* [36]

As instruções de 24 *bits* do processador *Xtensa*-1040 podem receber dois tipos de codificação, sendo elas: 8 *bits* e 16 *bits*. Quando a codificação é 8 *bits*, o código completo é usado para endereçar o dicionário. No caso da codificação de 16 *bits*, apenas os 8 *bits* mais significativos são usados para endereçamento do dicionário, enquanto os outros 8 *bits* menos significativos são usados para identificar o tipo de instrução (se é comprimida e o seu tamanho). Após comprimido, é feito *patching* no código para correção dos *offsets* dos saltos,

sendo que, para efeitos de simplificação, os alvos dos saltos são forçados a se manterem alinhados a palavras.

A parte mais significativa do algoritmo de descompressão para essa arquitetura é formada por uma árvore lógica. Porém, na Figura 2.13 é representada a ordem de descompressão das palavras. O nó raiz da árvore recebe instruções comprimidas de 32 *bits* e é apontado pela seta de entrada (*32-bit input*). Daí em diante, este *stream* de *bits* vai percorrendo os nós da árvore até chegar a uma ponta da árvore, ou seja, uma folha da árvore que contém realmente uma instrução comprimida. Em cada nó (ou bifurcação) é possível gerar duas ou mais instruções, realizando assim a descompressão (mediante as consultas ao dicionário) e em cada nó da árvore também se mostra o tamanho das instruções que possivelmente podem ocorrer. Portanto, o processador espera receber um *bitstreams* de 32 *bits* das instruções originais.

Desta forma, a estrutura de decodificação da árvore proposta em [36, 39] garante que sempre que um nó folha for atingido, pelo menos 32 *bits* de instruções estarão disponíveis para o processador. Então, os *bits* que forem excedentes aos 32 *bits* requeridos pelo processador são armazenados em um *buffer* para serem usados no próximo ciclo.

Os resultados obtidos nos testes realizados demonstraram que houve um ganho médio de 25% no tempo de execução dos aplicativos usando a compressão de código e uma média de 35% na redução do tamanho do código. Segundo os autores essa tecnologia desenvolvida não está limitada a apenas um único processador, uma vez que pode ser aplicada e obter resultados similares em outros processadores.

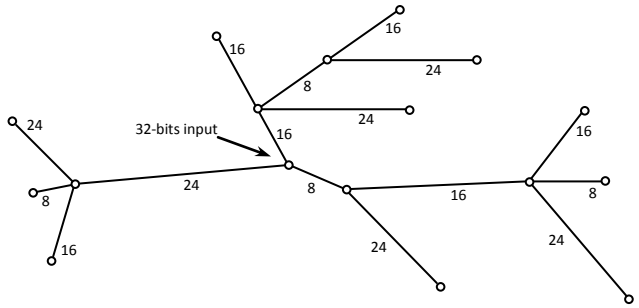


Figura 2.13 – Árvore lógica para hardware de descompressão de 1 ciclo [36]

2.6.2 Compressão por Linhas de *Cache*

BENINI *et al* [4] desenvolveram um algoritmo de compressão que é adaptado para execução eficiente do hardware (descompressor). As instruções são compactadas em grupos que têm o tamanho de uma linha da *cache* e a sua descompactação ocorre no instante que são

extraídas da *cache*. Experimentos foram realizados com o processador DLX, devido o mesmo ter uma arquitetura simples de 32 *bits* e também ser uma arquitetura RISC. Além disso, o processador DLX é semelhante a vários processadores comerciais da família ARM e MIPS. Uma tabela de 256 posições foi utilizada para guardar as instruções mais executadas. Cada linha da *cache* é formada por 4 instruções originais ou um conjunto de instruções comprimidas e possivelmente intercaladas com outras não comprimidas, prefixado por uma palavra de 32 *bits*. A palavra não comprimida tem um posicionamento fixo na linha da *cache* e serve para diferenciar uma linha de *cache* com instruções comprimidas das outras linhas com as instruções originais. De fato, uma linha de *cache* comprimida não contém necessariamente todas as instruções comprimidas, mas sempre deve ter um número entre 5 e 12 instruções comprimidas na linha da *cache* para ser vantajoso o uso da compressão [4]. A estrutura detalhada de uma linha comprimida de *cache* é mostrada na Figura 2.14.

Para evitar o uso das tabelas de tradução de endereços, BENINI *et al* exigem que os endereços de destino estejam sempre alinhados a 32 *bits* (palavra). A primeira palavra (32 *bits*) da linha de *cache* contém uma marca L e um conjunto de *bits* de *flags*. A marca é um *opcode* de instrução não utilizada, ou seja, um *opcode* inválido que sinaliza uma linha comprimida (no processador DLX os *opcodes* são de 6 *bits*). Segue-se um conjunto de 12 *bits* de *flags*, que forma um subconjunto com pares de 2 *bits* para sinalizar se o restante, ou seja, os 3 *bytes* correspondentes (B0 a B11) contém instruções comprimidas ou não.

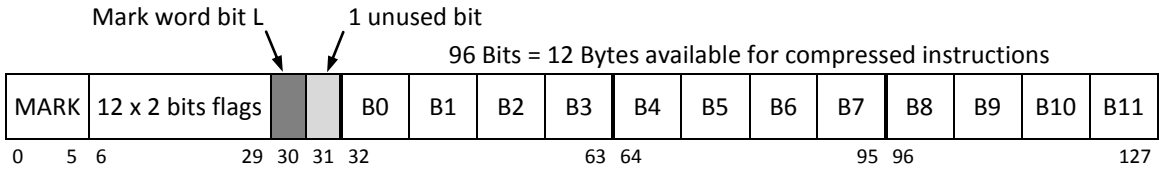


Figura 2.14 – Estrutura da linha comprimida sugerido por BENINI *et al* [4]

Os valores do conjunto dos *bits flags* são atribuídos da seguinte forma [4]:

- 00_2 se o *byte* correspondente contém uma instrução comprimida;
- 01_2 se o *byte* correspondente contém 8 *bits*, ou seja, parte de uma instrução não comprimida;
- 11_2 se o *byte* correspondente é deixado vazio por motivo do alinhamento de instruções;
- 10_2 serve para assinalar a última instrução comprimida da linha;
- O *bit* de *flag* L indica se a linha é comprimida (L = 0) ou não (L = 1).

O algoritmo de compressão desenvolvido por BENINI *et al* [4], analisa o código sequencialmente, a partir da primeira instrução (supondo que cada linha da *cache* já esteja alinhada) e tenta acondicionar instruções adjacentes em linhas comprimidas. O procedimento da compressão garante que o código comprimido nunca seja maior do que o código original. Além disso, o número dos *bits* transferidos da memória para a *cache* ao ser executada a compressão do código, nunca seja maior do que para o caso não comprimido.

Um detalhe adicional é que nenhuma instrução pode ultrapassar o limite da linha de *cache*. E ainda, nem todas as instruções que pertencem ao dicionário estarão representadas de forma comprimida no código, dado que a compressão depende da sua vizinhança e da viabilidade de formação de uma linha comprimida. O hardware de descompressão proposto em [4] é executado *on-the-fly* e está embutido no controlador principal.

Os experimentos realizados em vários pacotes de código C do *benchmark* fornecido pelo projeto *Ptolemy* [19], comprovaram que houve uma redução média no tamanho do código de 28% e uma economia média no consumo de energia de 30%. A Figura 2.15 mostra a arquitetura do hardware descompressor proposto por BENINI *et al* [4].

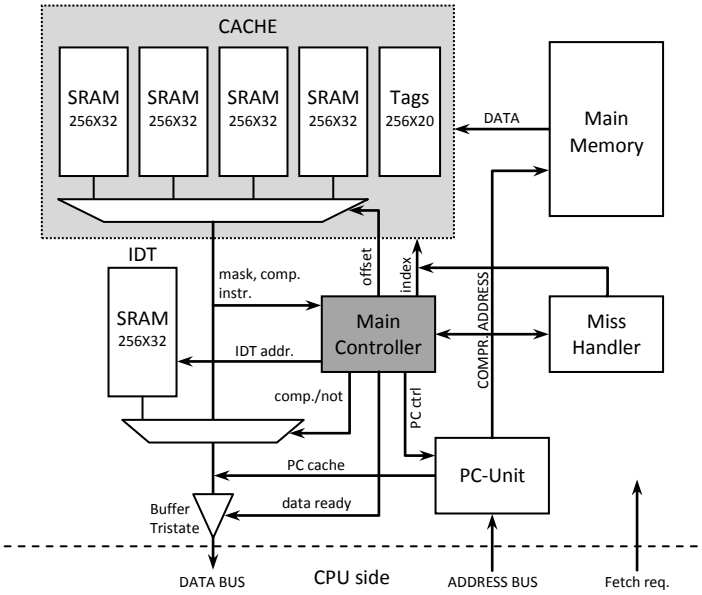


Figura 2.15 – Arquitetura do hardware descompressor sugerido por BENINI *et al* [4]

2.6.3 Compressão por *Codeword*

LEFURGY *et al* [34] propuseram uma técnica de compressão de código baseada na codificação do programa usando um dicionário de códigos. A Figura 2.16 mostra a arquitetura desta abordagem. Assim, a compressão é realizada após a compilação do código fonte, porém, o código objeto é analisado e as sequências comuns de instruções são substituídas por uma

palavra codificada (*codeword*), como na compressão de texto. Apenas as instruções mais frequentes são comprimidas. Um *bit* (*escape bit*) é utilizado para distinguir uma palavra comprimida (codificada) de uma instrução não comprimida. As instruções correspondentes às instruções comprimidas são armazenadas em um dicionário no hardware de descompressão. As instruções comprimidas são usadas para indexar as entradas do dicionário. O código final consiste de *codewords* misturadas com instruções não comprimidas. A Figura 2.17 exemplifica o mecanismo de compressão e indexação no dicionário.

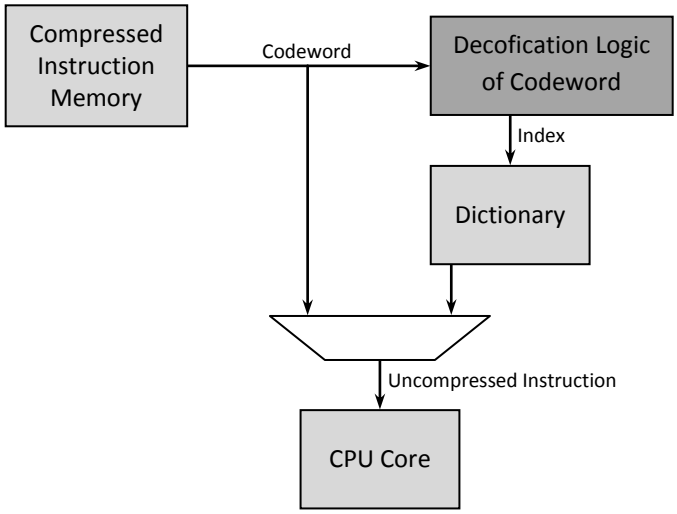


Figura 2.16 – Arquitetura do descompressor sugerido por LEFURGY *et al* [34]

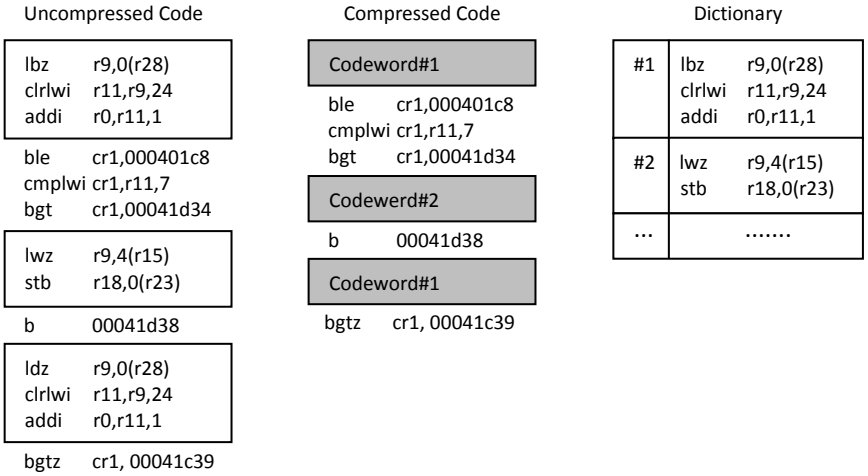


Figura 2.17 – Esquema de compressão de LEFURGY *et al* [34]

Observa-se que um dos problemas mais comuns encontrados na compressão de código se refere à determinação dos endereços alvo das instruções de salto. Normalmente este tipo de instrução (desvio direto) não é codificado para evitar a necessidade de reescrever as palavras de códigos que representam estas instruções [27]. Já os desvios indiretos podem ser codificados normalmente, pois, como seus endereços alvos estão armazenados em

registradores, apenas as palavras de códigos necessitam ser rescritas. Neste caso, é necessária apenas uma tabela para mapear os endereços originais armazenados no registrador para os novos endereços comprimidos.

Este método diverge dos demais métodos vistos na literatura no sentido que, os endereços alvos estejam sempre alinhados a 4 *bits* (tamanho de um *codeword*) e não ao tamanho da palavra do processador (32 *bits*). Como vantagem destaca-se uma melhor compressão; mas como desvantagem verifica-se a necessidade de alterações no *core* do processador (um hardware extra) para tratar desvios para endereços alinhados a 4 *bits*. Entretanto, os detalhes sobre a interação do hardware descompressor com os processadores experimentados (PowerPC, ARM e i386) não foram esclarecidos.

O funcionamento do hardware descompressor é realizado basicamente da seguinte maneira: A instrução é buscada da memória, caso seja um *codeword*, a lógica de decodificação dos *codewords* obtém o deslocamento e o tamanho do *codeword* que servirá como índice para acessar a instrução não comprimida no dicionário e repassar ao processador. No caso de instruções não comprimidas, elas são repassadas diretamente ao processador. Com o método proposto em [34], foram obtidas razões de compressão de 61% para o processador PowerPC, 66% para o processador ARM e 74% para o processador i386. As métricas de desempenho e consumo de energia não foram oferecidas pelos autores.

2.6.4 PDC-ComPacket

NETTO [44], desenvolveu um método baseado nos formatos das instruções do SPARC versão 8, processador escolhido para implementação do método. No SPARC v8, todos os formatos de instruções têm 32 *bits*. No caso do formato de instrução 2, mostrado na Figura 2.18, o padrão reserva uma sequência de *bits* para representar instruções inválidas para o processador: OP = 00_2 e OP2 = $00X_2$. É justamente esta sequência de *bits* que é aproveitada pelo ComPacket. Ele a usa para identificar uma instrução comprimida. Isto permite que o descompressor, ao identificar que a sequência acima não foi usada, possa repassar ao processador diretamente a instrução sem necessidade de descompressão, já que se trata de uma instrução descomprimida, o que diminui o *overhead* imposto, minimizando também o consumo de energia, na medida em que o hardware de descompressão só será ativado quando efetivamente necessário.

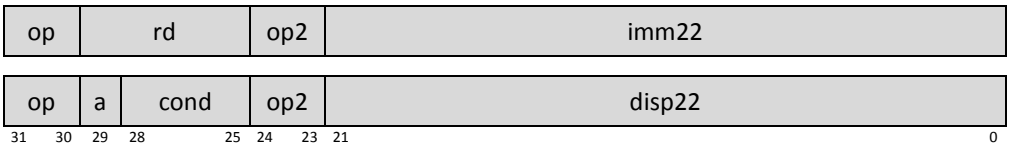


Figura 2.18 – Formato 2 do padrão SPARC v8 [44]

Palavras comprimidas (OP = 00_2, OP2 = $00X_2$), ou ComPacket, nada mais são do que conjuntos de índices para o dicionário de instruções. Conforme NETTO *et al* [45], existem quatro formatos de instruções comprimidas, permitindo de dois a quatro índices, como mostrado na Figura 2.19. Todos os formatos têm em comum uma sequência de escape ESC que caracteriza o ComPacket. Além dos índices para identificação da compressão, OP e OP2 apresentados anteriormente, a sequência de escape prevê o par de *bits* TT usado para identificar um alvo, no caso de um salto para uma instrução comprimida. O *bit* S indica o tamanho dos índices. Para S = 0_2, os índices têm 6 *bits*. Para S = 1_2, são 8 *bits*. O *bit* B indica se há alguma instrução de salto, dentre as instruções apontadas pelos índices do ComPacket.

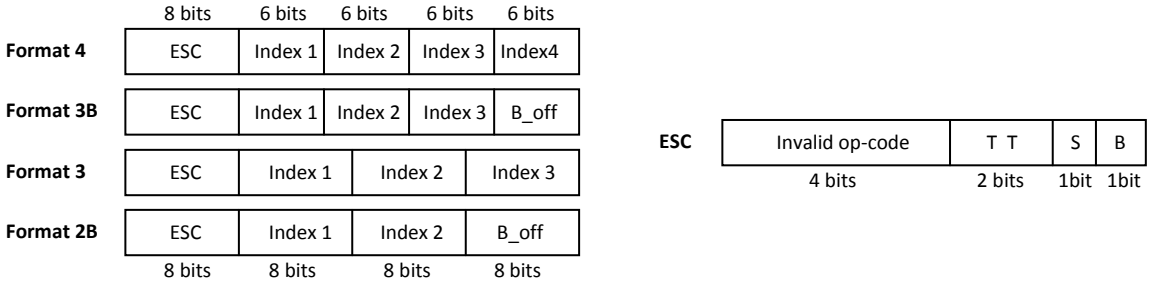


Figura 2.19 – Codificação usada nos ComPackets [44]

O primeiro formato de compressão (Formato 4), requer quatro instruções adjacentes, todas localizadas nas 64 primeiras posições do dicionário (já que os índices têm 6 *bits*) e que nenhuma das instruções seja um salto. O segundo formato (Formato 3) envolve três instruções adjacentes, podendo estar localizadas em qualquer posição do dicionário. Neste formato, instruções de salto também não são permitidas. O terceiro formato (Formato 3B) envolve três instruções adjacentes situadas nas 64 primeiras posições do dicionário, porém uma delas pode ser uma instrução de salto. Finalmente o quarto formato (Formato 2B), envolve duas instruções adjacentes localizadas em qualquer posição do dicionário, onde uma das instruções pode ser um salto. O algoritmo de compressão do ComPacket foi executado usando os seguintes passos, [46]:

- **Construir o Dicionário de Instruções**: é feita mediante a ordenação das instruções tanto pela classificação estática quanto pela dinâmica, [45]. Então, insere-se uma

porcentagem das instruções respeitando a classificação estática e as demais respeitando a classificação dinâmica, até completar o dicionário. Antes de inserir uma instrução no dicionário é verificado se ela já foi inserida antes. Em caso positivo, a próxima instrução da classificação é usada. O percentual de instruções estáticas e/ou dinâmicas inseridas no dicionário permite que o usuário configure o sistema de acordo com suas necessidades de compressão e desempenho;

- **Marcar o Código**: fazer a marcação no código original das instruções que pertencem ao dicionário, das instruções que são alvos de saltos e aquelas que precisam de *patching*;
- **Atribuir os ComPackets**: varre o código original e, de acordo com o dicionário e algumas restrições, procura marcar as instruções prioritariamente para compressão pelo primeiro formato (Formato 4). Se não for possível, tenta o segundo formato (Formato 3). E assim por diante, até o quarto formato (Formato 2B). Se ainda assim não conseguir, a instrução fica no seu formato original, não comprimida. Ou seja, a preferência é dada aos índices que ocupam menos *bits*, para conseguir a melhor razão de compressão possível;
- **Compressão do Código**: uma única passada do código e os ComPackets são atribuídos, substituindo as instruções originais. Nesta fase também é gerada uma tabela de mapeamento entre os endereços convencionais e os endereços comprimidos;
- ***Patching* dos Endereços**: finalmente é feito o *patch* das instruções de salto, com base na tabela de mapeamento gerada no passo anterior.

A Figura 2.20 mostra um exemplo do código marcado com o método ComPacket e na Figura 2.21 é mostrada a arquitetura do descompressor sugerida por NETTO [44]. A arquitetura foi validada usando programas pertencentes a dois *benchmarks*: *MediaBench* [8] e *MiBench* [24], além de uma demonstração prática envolvendo a decodificação e reprodução de arquivos MP3 (*libmad*). Conforme NETTO [44], os resultados foram satisfatórios, com ganhos de desempenho chegando até 45% e redução de energia de até 46% e, naturalmente, permitindo que o código fosse comprimido, com razões de compressão variando entre 72% a 88%.

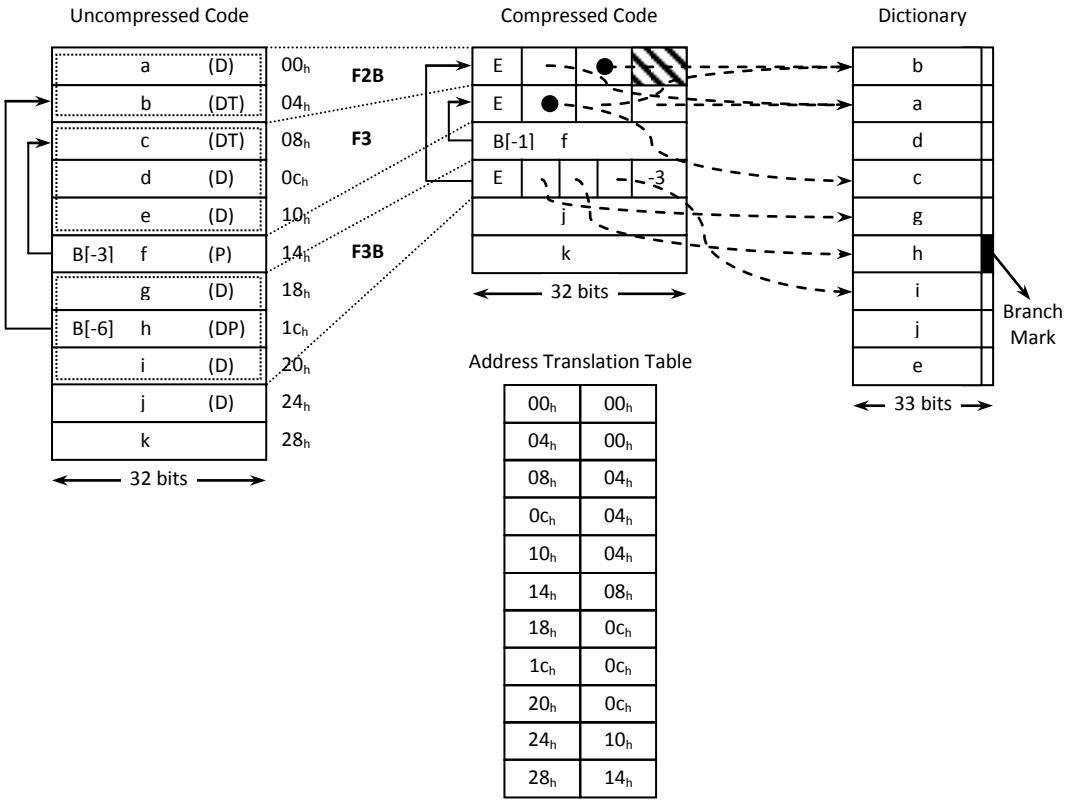


Figura 2.20 – Exemplo do algoritmo de compressão marcado com o ComPacket [44]

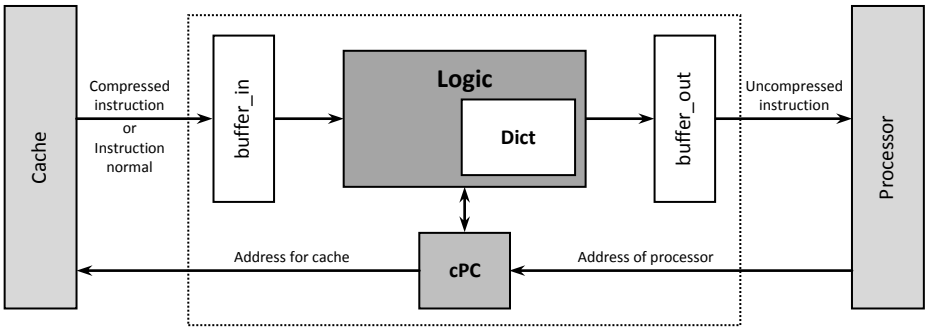


Figura 2.21 – Arquitetura sugerida por NETTO [44]

2.7 Comparativo entre os Principais Métodos de Compressão de Código

A seguir é realizada uma análise comparativa entre os métodos de compressão de código descritos nas seções anteriores (Seção 2.5 e 2.6). Para comparar os diferentes métodos de compressão de código, os seguintes parâmetros são usados:

- **Razão de Compressão**: esse é o indicador de quanto o programa foi reduzido. Quanto menor a razão de compressão, menor a área de memória ocupada pelo programa final e melhor o resultado do método. A razão de compressão é calculada usando a Equação 2.2 mostrada na Seção 2.3;

- **Desempenho**: embora a redução do tamanho do código executável seja o principal objetivo, não se pode deixar de lado o desempenho resultante após a implementação do método. Alguns desses métodos utilizaram uma quantidade maior de instruções comprimidas para conseguir uma diminuição do tamanho final do código. Essa maior quantidade de instruções influenciou na quantidade de ciclos que o processador precisou para executar o programa. Houve ganho suficiente em outras partes do sistema, como um aumento no *hit ratio* da *cache* ou na redução do número de acessos à memória;
- **Consumo de Energia**: este parâmetro indica qual foi o consumo energético da plataforma com o módulo de descompressão do código. Quanto menor o consumo, melhor é para o sistema, uma vez que o consumo de energia é um dos pontos importantes no desenvolvimento de sistemas embarcados.

A Tabela 2.1 mostra um comparativo entre os diversos métodos de compressão de código descritos anteriormente, apresentado: o método de compressão utilizado; a técnica de compressão; o tipo de arquitetura; a plataforma base; o *benchmark* testado; a respectiva razão de compressão obtida; o desempenho final do sistema medido pelo ganho no tempo de execução do programa e o ganho no consumo de energia usando o método. Nesta Tabela, considera-se como 100% a razão de compressão e o tempo de execução do programa original, ou seja, descomprimido.

Para comparar as razões de compressão em cada um dos métodos apresentados, é necessário avaliar com muito cuidado como estas foram obtidas. Alguns fatores afetam fortemente os benefícios da compressão de um método, tais como:

- **Plataforma Base**: a plataforma utilizada na implementação é um grande diferencial entre dois métodos distintos ou mesmo entre duas implementações de um mesmo método. Como pode ser observado nos trabalhos de LEFURGY *et al* [34, 35], o código para o PowerPC permitiu mais compressão que o do ARM que por sua vez mostrou resultados melhores que o i386, pois o i386 utiliza instruções de tamanho variável enquanto os outros dois processadores utilizam instruções de tamanho fixo (32 *bits*). Desta forma, as razões de compressão só devem ser comparadas se a arquitetura base utilizada for a mesma entre os dois métodos;
- **Custo do Descompressor**: o custo do descompressor, ou ao menos uma estimativa dele, deve ser incluído na razão de compressão, uma vez que o tamanho e a latência do descompressor influenciam na área final ocupada pelo sistema. Esse custo nem

sempre foi considerado nos trabalhos correlatos, como no caso do WOLFE & CHANIN [62] e LEFURGY *et al* [34].

Observando a Tabela 2.1, os trabalhos de LEFURGY *et al* [34, 35] (Compressão por *Codeword*) e LEKATSAS & WOLF [39] (Agrupamento por Características Próprias) mostraram que existem formas diferentes de obter uma razão de compressão similar. Entretanto, em nenhum dos trabalhos de Lefurgy foram obtidos resultados de desempenho do hardware e do consumo de energia, sendo apenas realizados testes de compressão do código.

Tabela 2.1 – Sumarização dos primeiros métodos de compressão de código

Método	Técnica	Arquitetura	Plataforma	Benchmark	Razão de Compressão	Desempenho	Consumo de Energia
CCRP [62]	Frequência	CDM	MIPS	lex, pswarp, yacc, eightq, matrix25, lloop01, xlisp, espresso, spim	73%	---	---
PCACHE [32]	Dicionário	CDM	SPARC	gcc, go	60%	111%	---
CodePack [28]	Dicionário	CDM	PowerPC	MediaBench, SPEC95	60%	90% a 110%	---
IBC [2]	Dicionário	CDM	MIPS	SPECint95	53,6%	88% a 184%	---
			SPARC		63,9%	105,9%	
Instruction Splitting [11]	Frequência	CDM	ARM	MiBench	59%	---	---
			MIPS		60%		
			PowerPC		62%		
Agrupamento por Características Próprias [37]	Aritmética	PDC	SPARC	compress, diesel, i3d, key, mpeg, smo, trick	65%	75%	72%
			Xtensa-1040				---
Compressão por Linhas de *Cache* [4]	Dicionário	PDC	DLX	Ptolomy	72%	---	70%
Compressão por *Codeword* [34]	Dicionário	PDC	PowerPC	SPECint95	61%	---	---
			ARM		66%		
			i386		74%		
PDC-ComPacket [44]	Dicionário	PDC	SPARC	MediaBench, MiBench	72% a 88%	55%	54%

Dos trabalhos que utilizaram a plataforma MIPS, o método IBC [2] apresentou maiores resultados na razão de compressão e desempenho. Porém, as métricas de desempenho e consumo de energia dos métodos CCRP [62] e Instruction Splitting [11] não foram relatadas pelos autores. Ainda vale ressaltar que a razão de compressão do CCRP é apenas uma estimativa inicial e não uma medição real. No entanto, observando a Tabela 2.1 pode-se verificar que o desempenho final do método IBC [2] usando a plataforma MIPS apresentou uma variação de aproximadamente 96%, mostrando assim uma instabilidade no desempenho do sistema quando usado código comprimido.

O trabalho de KIROVSKI *et al* [32], ou seja, o método PCACHE apresentou um resultado satisfatório na razão de compressão, mas é importante lembrar que não teve uma implementação real. Então, para a plataforma SPARC o método IBC também apresentou boa razão de compressão por não ser apenas resultado estimado, mas sim real. Já as estimativas de desempenho e consumo de energia do método PDC-ComPacket [44] mostrou-se mais eficiente entre os demais métodos implementados na plataforma SPARC. Os métodos IBC e PCACHE tiveram um acréscimo de 5,9% e 11% respectivamente no desempenho final do sistema.

Ainda observando os resultados da Tabela 2.1, verifica-se que entre os métodos que usaram a plataforma PowerPC a razão de compressão foi quase idêntica para todos, ou seja, uma diferença de apenas 2% e 1% a mais para os métodos *Instruction Splitting* [11] e Compressão por *Codeword* [34] respectivamente quando comparado com o método *CodePack* [28], que por sinal apresentou a menor razão de compressão, sendo 60%.

Já no uso da plataforma ARM o método *Instruction Splitting* [11] apresentou-se mais eficiente na razão de compressão do que o método desenvolvido por Charles Lefurgy [34] (Compressão por *Codeword*). E por fim, o método de BENINI *et al* [5] (Compressão por Linhas de *Cache*) não foi comparado nessa análise por não ter outro método que fosse implementado usando a plataforma DLX. Só que mesmo assim, destaca-se que o método obteve resultado satisfatório no consumo de energia, sendo uma redução de 30%.

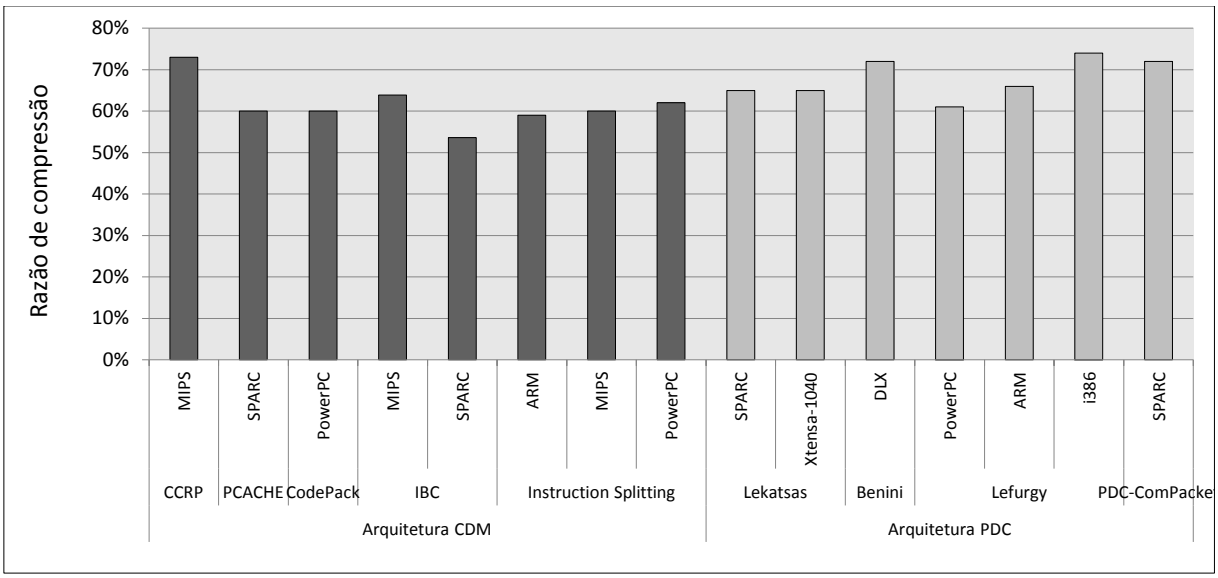
Os Gráficos 2.1, 2.2 e 2.3, mostram respectivamente a razão de compressão, o desempenho final do sistema e o consumo de energia dos métodos apresentados nas Seções 2.5 e 2.6. As arquiteturas CDM e PDC são representadas nos Gráficos pelas barras de cor cinza escuro e cinza claro respectivamente.

No Gráfico 2.1 observa-se a razão de compressão obtida pelos métodos descritos neste capítulo. Nota-se que o método IBC usando a plataforma SPARC apresentou o maior resultado, sendo 53,6%. Em seguida aparece o método *Instruction Splitting* usando a plataforma ARM com uma razão de compressão de 59%, e os métodos *PCACHE, CodePack* e novamente o *Instruction Splitting* (usando a plataforma MIPS) com 60% cada. Outra vez reforçamos que isso não garante que esses métodos sejam melhores ou piores do que os outros, devido à utilização de plataformas, dos *benchmarks*, arquiteturas e técnicas diferentes.

Os métodos que foram implementados usando a arquitetura CDM apresentaram, em média uma eficiência de aproximadamente 6,5% a mais no que diz respeito à métrica de razão de compressão quando comparados aos métodos implementados usando a arquitetura PDC,

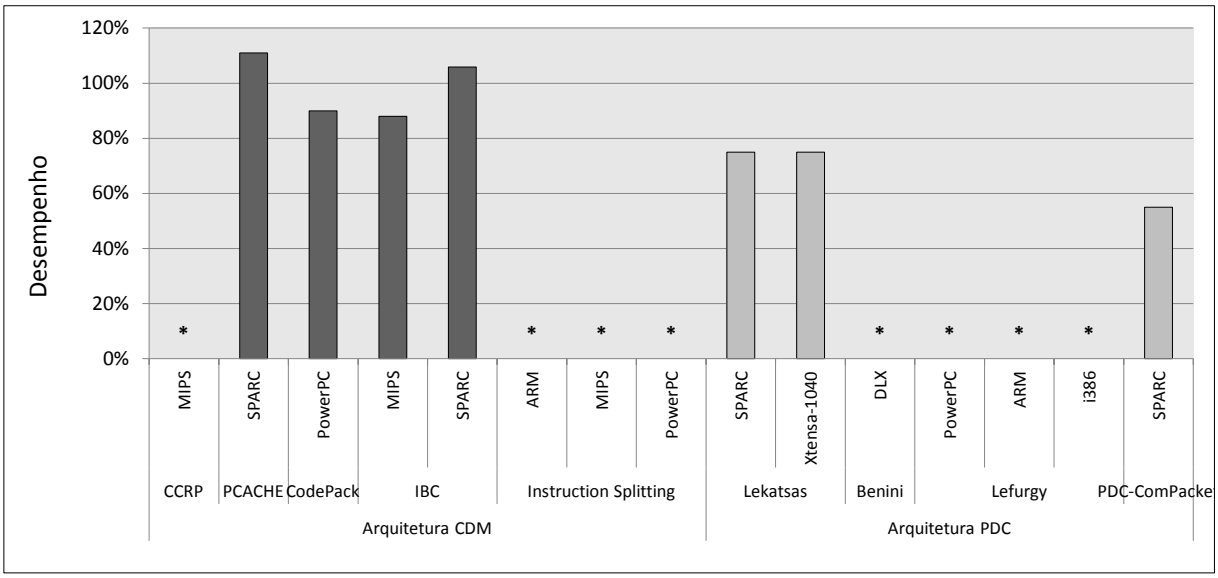
confirmando assim o que foi escrito em [44], “a razão de compressão para os métodos que usam a arquitetura CDM costuma ser maior que a dos métodos que usam a arquitetura PDC”.

Gráfico 2.1 – Razão de compressão dos primeiros métodos de compressão



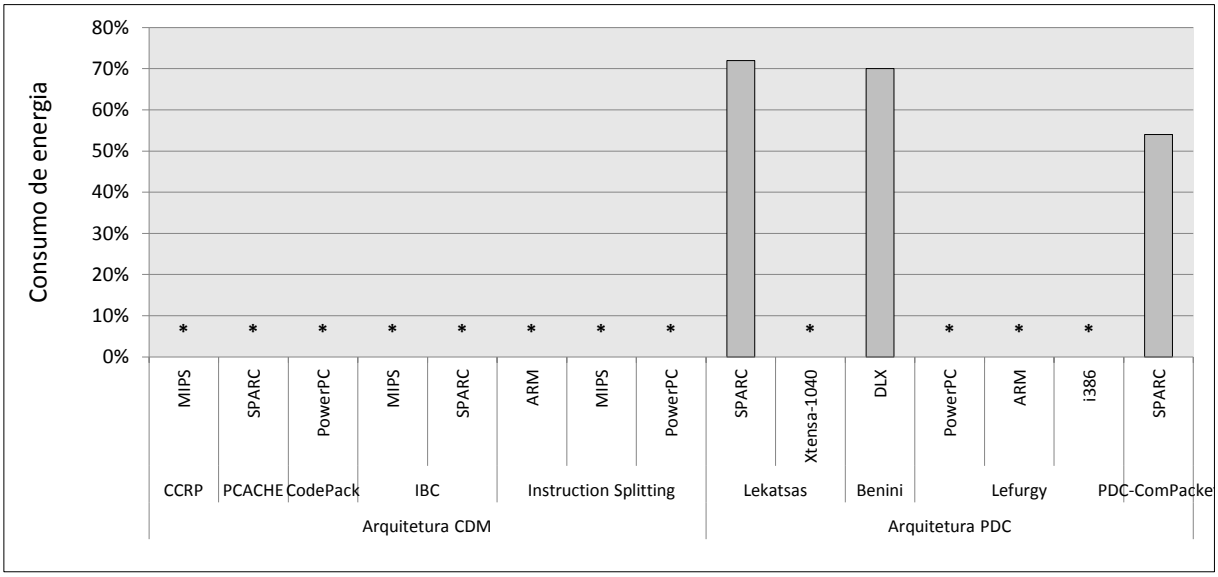
No Gráfico 2.2 apresenta-se o desempenho final do sistema com os métodos de compressão de código descritos anteriormente. Constata-se também que os métodos que usam a arquitetura PDC (barras em cinza claro) apresentaram desempenho abaixo dos 100%, ou seja, possuem melhoras significativas na execução de programas que usaram a compressão de código. O sinal de * em cima das barras demonstrativas dos métodos identifica a ausência de valores que não foram informados pelos autores nos artigos.

Gráfico 2.2 – Desempenho dos primeiros métodos de compressão



Finalmente, no Gráfico 2.3 pode-se ver o consumo de energia dos métodos apresentados neste capítulo. Embora o grande objetivo da técnica de compressão de código seja obter um melhor desempenho do sistema através da redução do código dos programas, o consumo de energia também pode ser um fator na justificativa do uso dessa técnica no projeto de sistemas embarcados, porém, nem todos os autores conseguiram realizar e mostrar esta medida.

Gráfico 2.3 – Consumo de energia dos primeiros métodos de compressão



2.8 Considerações Finais do Capítulo

Neste capítulo apresentamos conceitos sobre as diferenças entre a compressão de dados e código, em seguida descrevemos as principais técnicas de compressão de código, e ainda enfatizamos quais são os reais custos e benefícios da compressão de código. Posteriormente, apresentamos as arquiteturas existentes na literatura (CDM e PDC) e seus principais métodos. Estudamos nove métodos para a redução do tamanho do código das instruções, publicados em cerca de trinta artigos, desde 1992 até a presente data. Eles diferem em muitas maneiras, porque muitas vezes têm motivações e áreas de aplicação diferentes, que vão desde a minimização do tráfego de rede ao consumo de energia em sistemas embarcados.

O uso das arquiteturas CDM e PDC também é diferente nos métodos de compressão. No entanto, observando os resultados da Tabela 2.1 concluímos que os métodos que usam a arquitetura CDM podem ter um tempo de descompressão mais lento e costumam ter uma razão de compressão maior, mas o maior ganho em termos de desempenho e economia de energia está nos métodos que usam a arquitetura PDC.

Também analisamos de forma detalhada os principais métodos de compressão de código, apresentando as suas propriedades, limitações e desempenhos. Notamos que em muitos casos, as publicações não fornecem informações suficientes sobre os métodos e, quando o fazem, os dados muitas vezes não podem ser comparados devido às diferenças de plataformas, *benchmarks*, arquiteturas e técnicas utilizadas para os experimentos.

Assim, forneceu-se uma base comparativa entre os mesmos para auxiliarmos na escolha do melhor e/ou mais adequado método de compressão. No próximo capítulo, apresentamos alguns métodos de compressão de código desenvolvidos recentemente e que são baseados em dicionário.

CAPÍTULO 3

MÉTODOS DE COMPRESSÃO BASEADOS EM DICIONÁRIO

Analisando o código executável gerado pelo compilador, observamos que em um programa muitas instruções e sequências de instruções são repetidas. Por exemplo, em *loop* de *for*, chamadas recursivas e outras, sempre apresentam a mesma estrutura e aparecem repetidamente em pequenos trechos do código. Então, a técnica baseada em dicionário aproveita dessa característica e a usa para reduzir o tamanho do código, gerando assim uma maior taxa de compressão. A Figura 3.1 mostra um exemplo da compressão usando a técnica baseada em dicionário. O algoritmo de compressão encontra sequências de instruções que são frequentemente repetidas no código, e em seguida substitui toda essa sequência por uma *codeword* única. Na Figura 3.1 a sequência das instruções "`ldr r0,[r2,r3]`", "`mov r2,#1,20`", "`add r3,r1,r2,lsr #1`" e "`cmp r0,r1`" é substituída pela *`codeword`* `#1`. Já a sequência das instruções "`and r0,r1,#0xf0`" e "`stmed sp!,{r0-r2,r14}`" é substituída pela *`codeword`* `#2`. O restante das instruções é deixado sem compressão.

Todas as sequências das instruções reescritas (ou codificadas) são mantidas em um dicionário que é usado pelo descompressor em tempo de execução para expandir as *codewords* na sequência original das instruções. No entanto, todas as *codewords* atribuídas pelo algoritmo de compressão são meramente um índice para o dicionário de instruções. O índice tem um tamanho fixo igual a log_2 (do número de sequências comprimidas). O programa final comprimido consiste em *codewords* intercaladas com instruções não comprimidas e o dicionário de instruções.

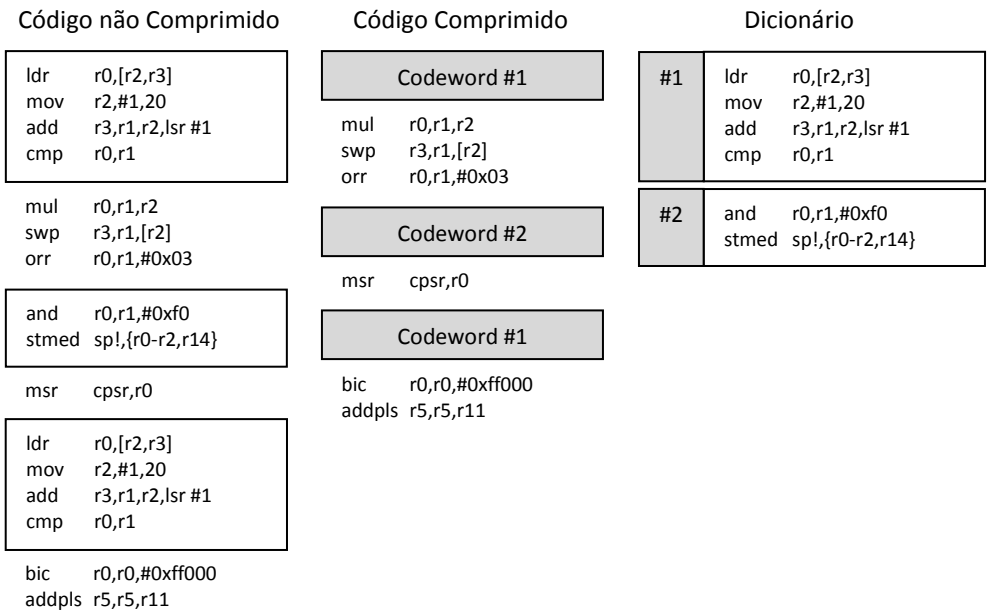


Figura 3.1 – Exemplo de técnica de compressão baseada em dicionário

A razão de compressão é calculada conforme a Equação 2.2 mostrada no Capítulo 2. Porém, dimensionando o tamanho de cada termo apresentado na Equação 2.2 temos:

- $Tamanho(Código\_Original) = W\,x\,N$
- $Tamanho(Código\_Comprimido) = W\,x\,U + n\,x\,log_2(n)$
- $Tamanho(Dicionário) = W\,x\sum_{i=1}^{n} Ci$

Substituindo estes termos, obtemos a Equação 3.1, que mostra o tamanho final da razão de compressão para a sequência de instruções da técnica baseada em dicionário.

$$razão\ de\ compressão = \frac{W\,x\,U + n\,x\log_2(n) + W\,x\,\sum_{i=1}^{n} Ci}{W\,x\,N}\ x\ 100\ \ (\%) \qquad (3.1)$$

Onde,

- W - comprimento da palavra de instrução;
- N - número de instruções originais;
- n - número de sequências;
- U - número de instruções restantes não comprimidas;
- Ci - número de instruções em sequência i.

Neste capítulo apresentamos alguns métodos de compressão de código desenvolvidos recentemente e que são baseados em dicionário. Neste capítulo, não levamos em consideração o tipo da arquitetura (CDM ou PDC) utilizada por estes métodos. Outro ponto importante a

destacar na escolha desses artigos para análises está no fato de que quase todos estes métodos usaram a ferramenta de simulação arquitetural *SimpleScalar* (descrita no Capítulo 4 desta tese) para realizar as simulações.

3.1 Bitmasks Compression

SEONG & MISHRA [54, 55] desenvolveram um método de compressão de código no qual utilizaram *bitmasks* para melhorar a taxa de compressão sem que houvesse muita penalidade (*overhead*) com o hardware descompressor. Primeiramente, realizaram um estudo sobre os custos/benefícios na utilização dos *bitmasks* e concluíram que quando um padrão de máscara for menor do que 32 *bits*, é necessário armazenar informações relacionadas à posição inicial, onde a máscara deve ser pouco aplicada. Nas análises foram considerados os padrões de máscara em diferentes tamanhos (de 1 *bit* até 32 *bits*). A Tabela 3.1 mostra o resumo dos estudos das *bitmasks* realizado em [54, 55], onde cada linha representa o número de alterações permitidas e cada coluna representa o tamanho do padrão da máscara, e uma entrada em branco é deixada quando não é possível fazer as combinações.

Tabela 3.1 – Custo de várias combinações de *bitmasks* [54, 55]

Bit de mudança	Tamanho do padrão da máscara					
	1 bit	2 bits	4 bits	8 bits	16 bits	32 bits
32 bits	165	100	59	42	35	32
16 bits	84	51	30	21	17	
8 bits	43	26	15	10		
4 bits	22	13	7			
2 bits	11	6				
1 bit	5					

O algoritmo de compressão proposto em [54, 55] aceita código original composto por vetores de 32 *bits*. O algoritmo primeiramente cria a distribuição da frequência dos vetores, considerando dois tipos de informações para calcular a frequência, que são: as sequências repetitivas e as sequências possíveis de correspondência por *bitmasks*. Depois, o algoritmo escolhe o menor tamanho possível para o dicionário, sem afetar significativamente a taxa de compressão. É importante salientar que os autores afirmaram que um dicionário maior aumenta o tempo de acesso e, consequentemente, reduz a eficiência da descompressão. Logo em seguida, o algoritmo converte cada vetor de 32 *bits* em códigos comprimidos (quando possível), utilizando o formato mostrado na Figura 3.2; o código comprimido junto com o código não comprimido é composto em série para gerar o código final do programa. O

algoritmo encerra resolvendo o problema das instruções de desvios ajustando os desvios alvos.

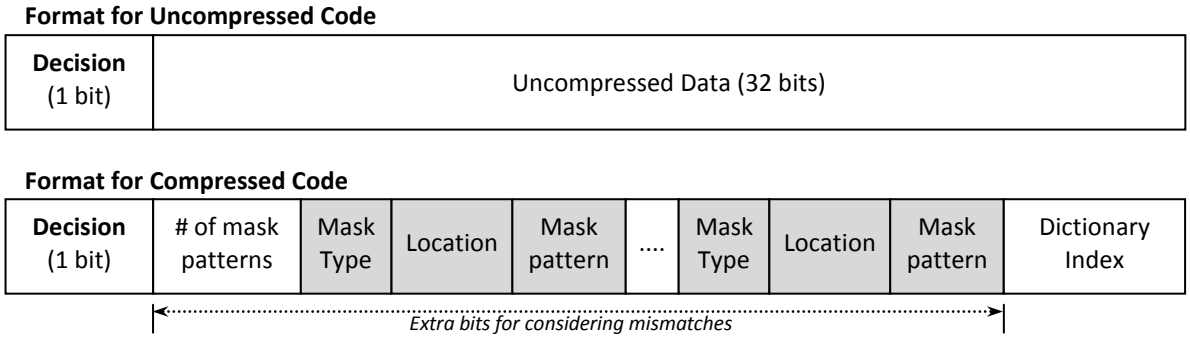


Figura 3.2 – Formato da codificação do método *Bitmasks Compression* [54, 55]

A Figura 3.3 mostra o projeto da unidade de descompressão baseado em *bitmask* que é chamado de *Decompression Engine for Bitmask Encoding* (DCE) [54, 55]. Para agilizar o processo de descompressão, o DCE foi otimizado para a eficiência, dependendo da escolha dos *bits* das máscaras usadas. A vantagem deste projeto foi que a geração de uma máscara com comprimento de instrução pode ser feito em paralelo com o acesso ao dicionário. Na geração de uma máscara de 32 *bits* não são acrescentadas quaisquer penalidades adicionais para o DCE, assim, o DCE pode descomprimir múltiplas instruções por ciclo.

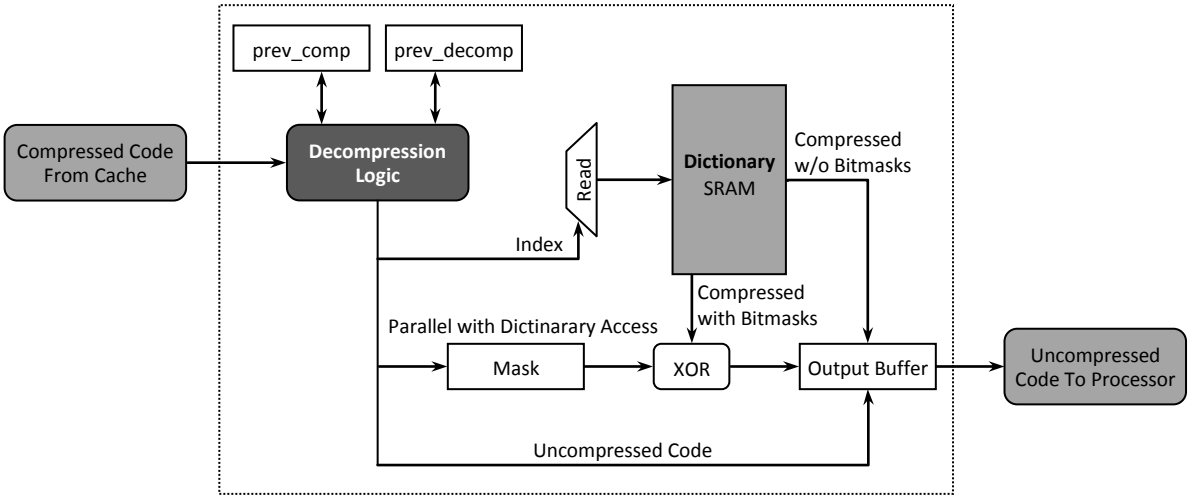


Figura 3.3 – Máquina para descompressão do código de *bitmask* [54, 55]

Os experimentos realizados em [54, 55] usaram as plataformas TMS320C6x, MIPS e SPARC, juntamente com os *benchmarks MediaBench* e *MiBench*. Foram utilizados três formatos personalizados de *bitmasks* sendo: (i) codificação 1 - uma máscara de 8 *bit*; (ii) codificação 2 - duas máscaras de 4 *bits* e (iii) codificação 3 - máscaras de 4 *bits* e 8 *bits*.

Conforme SEONG & MISHRA [55] os experimentos comprovaram que o método proposto superou os métodos existentes baseados em dicionário em uma média de 15%, obtendo assim uma razão de compressão de 65%, 60% e 55% para os processadores TMS320C6x, MIPS e SPARC respectivamente e um desempenho de 95%.

3.2 Profiling Agregado

BRORSSON & COLLIN [13] desenvolveram um método de compressão de código baseado em *profiling* agregado, onde é permitido que dois ou mais programas compartilhem o mesmo conteúdo do dicionário criado, tornando-se assim uma característica inovadora na redução dos códigos de instruções proposto neste método. O algoritmo de compressão permite gerar três níveis distintos de detalhamento do *profiling*, sendo:

- **Funcionalidade *funct***: o *profile* é baseado em todas as instruções executadas quando uma funcionalidade específica de um aplicativo é executada;
- **Aplicação *appli***: o *profile* é baseado na execução das funcionalidades de um aplicativo;
- **MediaBench *bench***: o *profile* é uma agregação dos padrões das instruções executadas com maior frequência ao longo de todas as diferentes aplicações executadas.

O algoritmo realiza a compressão dos códigos das instruções executadas com maior frequência, a fim de reduzir o tráfego para a *cache* de instrução e, assim, alcançar um melhor desempenho para *caches* menores juntamente com uma melhora energética. A Figura 3.4 mostra a ideia central do esquema de compressão proposto em [13].

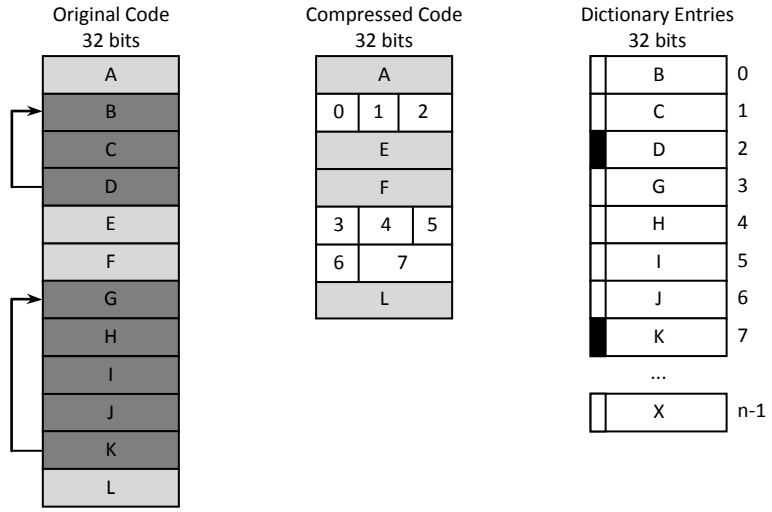


Figura 3.4 – Dicionário do esquema de compressão baseado no *Profiling* Agregado [13]

As sequências de instruções, {B, C, D} e {G, H, I, J, K} constituem na construção de *loops*, possivelmente iterados várias vezes. Durante a execução, as instruções dentro dos *loops* são possivelmente as instruções executadas com maior frequência e são localizadas nos trechos sequenciais e não-iterativos. Quando é aplicada a compressão sobre as instruções executadas com maior frequência, o número de *bytes* obtidos a fim de executar a sequência de código é reduzido à medida que mais informações são obtidas em cada busca. No entanto, a densidade do código tende a melhorar afetando assim a interface da memória e do processador, melhorando, presumivelmente tanto o desempenho quanto o consumo de energia, segundo [13]. Conforme mostrado na Figura 3.4, depois do *profile*, o *bit* padrão das oito instruções nos dois circuitos estão cada um atribuído e colocado em uma entrada do dicionário. As instruções de desvio são marcadas com um *bit* extra de codificação no dicionário, indicado pelo marcador preto na Figura 3.4.

A Figura 3.5 mostra o fluxo básico do processo de compressão de código proposto em [13], com base em um *profile* na utilização da instrução. O compilador contém um mecanismo de compressão, que trabalha com os blocos básicos, um de cada vez e realiza a compressão em *codewords*, e é terminado com o alinhamento de modo que cada bloco básico começa e termina sempre em um limite da palavra. Isto significa que uma única instrução nunca é comprimida em um bloco base, a menos que, assim, alcance uma redução no número de unidades de busca do bloco básico.

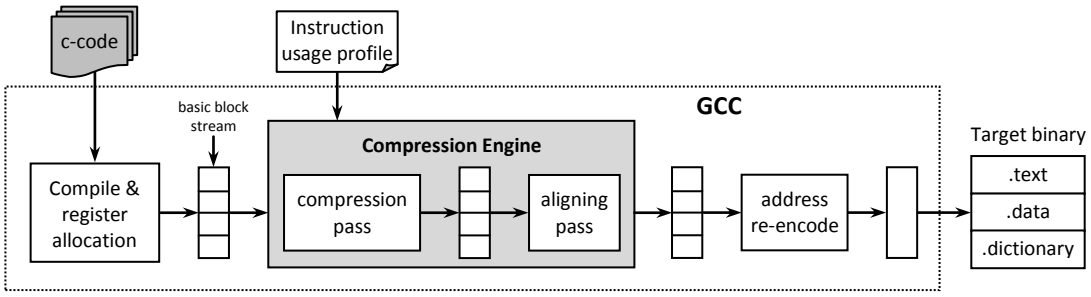


Figura 3.5 – Fluxo básico do processo da compressão de código [13]

Para os experimentos, os autores usaram um dicionário que consistiu por palavras de 32 *bits* com o tamanho de 256 linhas de entradas. A plataforma usada foi o MIPS IV, juntamente com o *benchmark MediaBench*. Conforme descrito em [13] o sistema realiza uma compressão estática e dinâmica satisfatória, quando os *profiles* de funcionalidades específicas são usados para determinar o conteúdo do dicionário. Os autores obtiveram uma razão de compressão de 75% e um aumento de 8 a 12% no desempenho do sistema.

3.3 DISE (*Dynamic Instruction Stream Editing*)

O método de compressão/descompressão de código desenvolvido por CORLISS *et al* [17] é baseado no uso de DISE com a arquitetura PDC. O DISE é a única forma de compressão que suporta a descompressão parametrizada; e tem uma interface de programação que permite dicionários específicos do programa, e utiliza hardware que não é específico de descompressão. A Figura 3.6 mostra o hardware do DISE que é composto por dois blocos básicos, sendo:

- ***DISE Engine***: realiza toda a expansão correspondente ao fluxo de busca de um aplicativo;
- ***DISE Controller***: é um co-processador que só é ativado quando o *DISE Engine* está configurado para o uso.

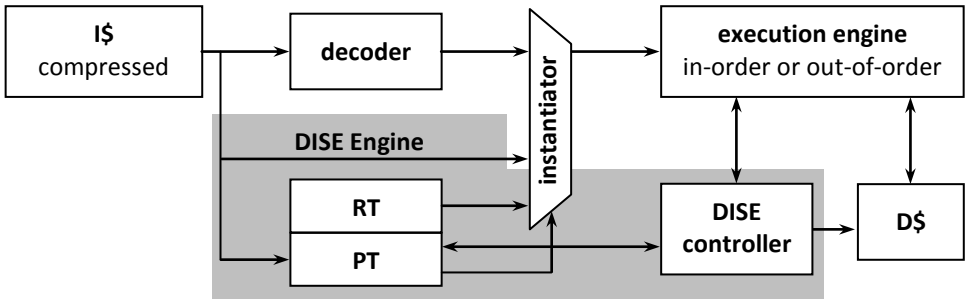


Figura 3.6 – Hardware do DISE no *pipeline* do processador [17]

O algoritmo de compressão de código usando DISE, que foi desenvolvido em [17], é composto por três etapas. Primeiro, um *profile* de compressão é obtido em uma ou mais aplicações. Em seguida, o algoritmo iterativo utiliza o *profile* de compressão para construir um dicionário de descompressão. E por fim, o executável estático é comprimido usando o dicionário (em sentido inverso) para substituir as sequências compressíveis (ou seja, aquelas que correspondem às entradas no dicionário), com os *codewords* DISE adequados.

Conforme CORLISS *et al* [17], com o uso do DISE, um modelo único de código original (sem compressão) pode render sequências descomprimidas com nomes diferentes ou valores de registradores imediatos quando instanciado com diferentes "argumentos" em diferentes locais no código estático comprimido. Desta forma, a parametrização pode ser usada para fazer o uso mais eficiente do espaço do dicionário. O uso e benefícios da descompressão parametrizada são mostrados na Figura 3.7.

Na Figura 3.7, a parte (a), mostra códigos estáticos originais e cada uma das duas caixas destacadas possui três instruções, que são as candidatas à compressão. Na parte (b) são

mostrados o código estático e o dicionário (RT – *Replacement Table*) para a compressão não parametrizada. Uma vez que as sequências de instruções diferem ligeiramente, elas exigem distintas entradas no dicionário. Com a descompressão por parâmetros, na parte (c), as duas sequências podem compartilhar uma única entrada no dicionário com parâmetros. A entrada utiliza dois parâmetros (em negrito): **P1** parametriza na primeira instrução a entrada e saída do registrador e na segunda instrução o registrador de entrada; **P2** parametriza o operando imediato na primeira instrução. Para recuperar a sequência original sem a compressão, a *codeword* da primeira sequência usa **a2** e **8** como valores para os dois parâmetros, enquanto para a segunda sequência se usa **a3** e **-8**, respectivamente.

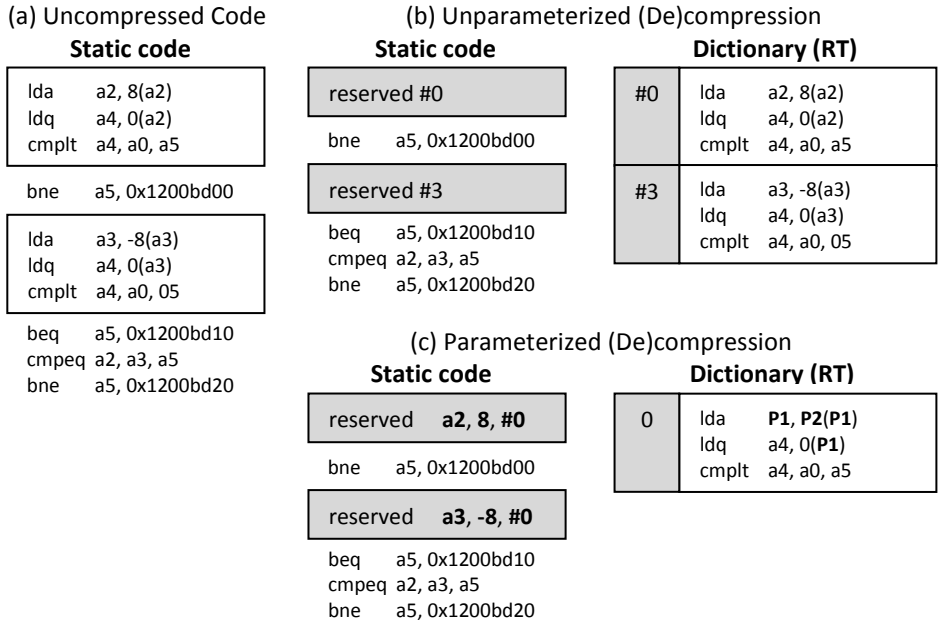


Figura 3.7 – Exemplos de (des)compressão do método DISE [17]

CORLISS *et al* [17] utilizaram a plataforma Alpha EV6 (Compaq) para realizarem seus experimentos com o simulador *SimpleScalar*. Os *benchmarks SPEC'2000* e *MediaBench* foram usados como sendo a carga de entrada para os experimentos. Os autores obtiveram uma razão de compressão entre 65% a 75%; uma melhora no desempenho do sistema de 5% a 20% e uma redução no consumo de energia de 10% com o método de compressão/descompressão usando DISE.

3.4 Hybrid Compression

HAIDER & NAZHANDALI [25] desenvolveram um novo método de codificação híbrido, combinando a tradicional codificação baseada em *bitmask* e a codificação baseada em

prefixo de *Huffman*, gerando assim um novo dicionário e uma nova técnica de seleção baseada no algoritmo *non-greedy*.

A compressão baseada em *bitmask* gera a compressão baseada nas frequências e aproveita a curta distância de *hamming* entre algumas instruções. No método desenvolvido em [25] existem dois tipos distintos de instruções comprimidas, sendo: (i) as instruções de entrada no dicionário (DE) e (ii) as instruções de *children* no dicionário (DC).

A Figura 3.8 mostra as características físicas dos diferentes tipos de instruções, bem como um exemplo de compressão com o *bitmask*. Em uma instrução **DE** o seu código de instrução original é armazenado no dicionário. Quando comprimidas, essas instruções só precisam de referências como índice de entrada correspondente no dicionário. Já uma instrução **DC** depende de uma instrução diferente, nomeadamente uma instrução DE, a ser codificada. Quando comprimida, ela contém uma referência para o índice da instrução DE, juntamente com alguns *bits* que especificam a máscara de *bits* necessários para recuperar a instrução de seus DE associados.

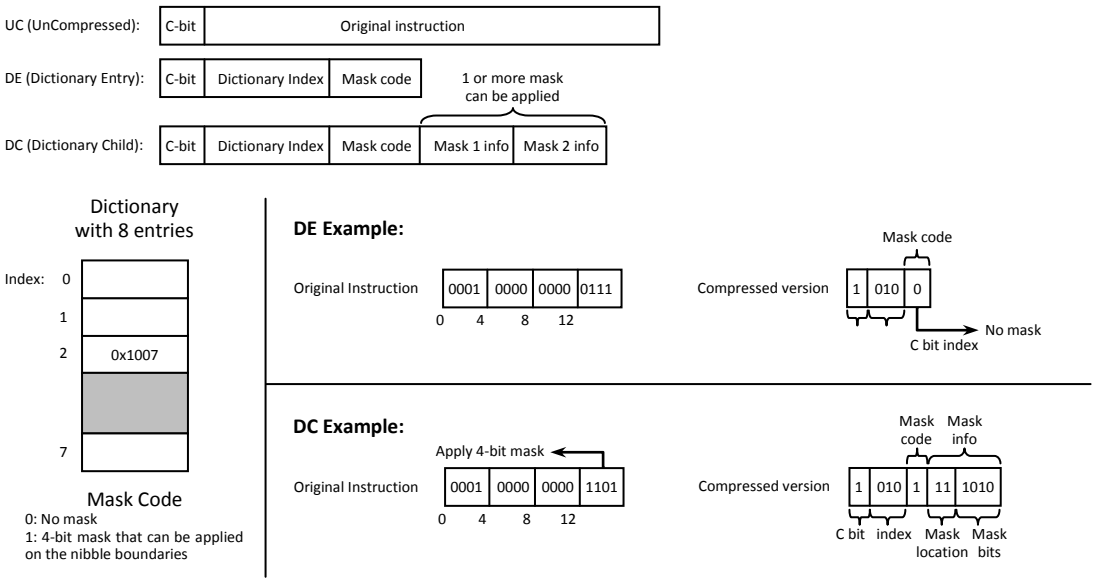


Figura 3.8 – Tipos de instruções e exemplo da compressão com *bitmask* [25]

O *flag C-bit* na Figura 3.8 especifica se uma instrução é comprimida ou não. No entanto, o *flag C-bit* é seguido por *bits* que representam o índice no dicionário com as instruções comprimidas. O número desses *bits* depende do tamanho do dicionário. O código da máscara decide o número e o tipo de máscaras que devem ser usadas para processar a entrada no dicionário, a fim de recuperar a instrução original. São as instruções DC que contêm as informações sobre o número e o tipo das máscaras necessárias. Contudo, a *bitmask*

contém informações mais detalhadas sobre as máscaras incluindo a sua localização e a alternância necessária entre os *bits* reais, conforme se explica em [25].

Quando o sistema de descompressão encontra uma DE, ele acessa o dicionário e recupera a instrução desejada especificada pelo índice do dicionário. A DC também tem um valor de índice, depois que o sistema de descompressão recupera o conteúdo de uma instrução no dicionário que então passa a alternar os *bits*, conforme especificado pelos dados da máscara no DC (realizado por uma operação de XOR). Para uma instrução descomprimida a máquina de descompressão apenas retira o primeiro pedaço da UC.

Segundo HAIDER & NAZHANDALI [25] com a codificação dos prefixos de *Huffman*, o código da máscara é uma maneira simples para melhorar a taxa de compressão. O código da máscara para uma instrução DE fica reduzido a 1 *bit*. Algumas instruções do DC implicam em um custo adicional, pois seus códigos de máscaras tornam-se maiores, mas a economia em DE é superada com os custos. Este método híbrido que combina a codificação de prefixo do código de máscara com a codificação regular dos índices do dicionário é capaz de aproveitar as vantagens da codificação de *Huffman*. Porém, a codificação de *Huffman* é mais eficiente quando há apenas alguns símbolos para codificar porque os códigos são curtos e a decodificação é simples.

Os experimentos realizados em [25], obtiveram uma razão de compressão de 91% para os dicionários com 4.096 e 8.092 entradas (instruções) e 80% para os dicionários com 512 e 1.024 entradas (instruções) aplicando o novo método de compressão de código híbrido. A plataforma de teste foi um ARM usando o *benchmark MiBench*.

3.5 Merge Bitmask and DISE

THURESSON & STENSTROM [58] propuseram um algoritmo de compressão de código no qual combinaram dois métodos já propostos anteriormente (os métodos de *bitmask* e DISE). Os autores [58] afirmaram que um dos pontos mais impactantes na pesquisa realizada foi mostrar que o número de parâmetros dos operandos tem um efeito significativo sobre a compressibilidade. Além de que o novo esquema proposto reduziu o tamanho do dicionário e o número de *codewords* de forma significativa, o que permitiu que a implementação fosse mais eficiente com as técnicas de compressão de código baseada em dicionário.

Na Figura 3.9(a) e (b), é mostrada a combinação de ambos os esquemas de compressão de duas sequências de códigos. A parte (a) da Figura 3.9 mostra um trecho do código original

da aplicação *pgp* (disponível no *benchmark MediaBench*). Já a parte (b) da Figura 3.9 mostra a entrada no dicionário que pode representar todas as instruções contidas na parte (a). As *codewords* resultantes são mostradas no final da parte (b) dessa figura. Conforme o algoritmo proposto em [58], a primeira *codeword* é expandida com o parâmetro P1 e está operando *.LL19* com *bitmask* e a instrução "*mov*" e cancela na entrada do dicionário. A segunda *codeword* atribui o parâmetro *.LL55* e a *bitmask* cancela a primeira e a última instrução.

Programa Descomprimido

add	%o0	-1	%o0
cmp	%o0	0	
bge	.LL19		
st	%o0	[%i1]	
..			
cmp	%o0	0	
bge	.LL55		
mov	%o0	%10	

(a) Código Original

Dicionário

DE1	add	%o0	-1	%o0
	cmp	%o0	0	
	bge	**P1**		
	mov	%o0	%10	
	st	%o0	[%i1]	

Programa Comprimido

(DE1	.LL19	11101)
..		
(DE1	.LL55	01110)

(b) Código Comprimido

Figura 3.9 – Exemplo da compressão de código para o programa *pgp* do *MediaBench* usando as técnicas de compressão baseada em dicionário estendido [58]

A Figura 3.10 mostra a arquitetura DISE e como ela foi usada para a compressão de código estático em *codeword* e são substituídas por sequências de instruções no dicionário. Na arquitetura do DISE o módulo *Codeword Logic* simplesmente detecta se uma instrução é uma *codeword* e então a força no dicionário para gerar a sequência de instruções adequadas.

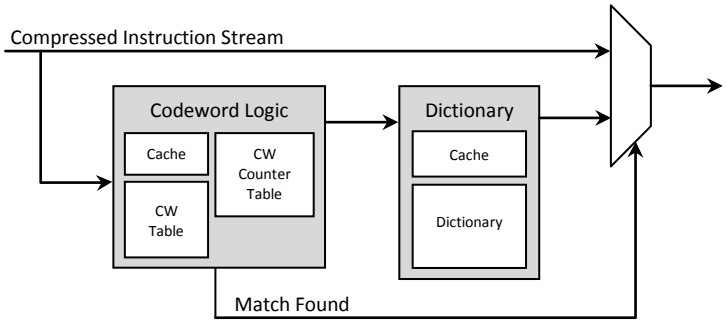


Figura 3.10 – Arquitetura DISE para a descompressão das *codewords* usando o dicionário de *cache* [58]

Uma nova ideia no DISE, proposto pelos autores [58], é que ele permite parâmetros na operação das *codewords*. Desta forma, o DISE consegue usar a mesma *codeword* na entrada de dicionários semelhantes, mas não idêntico às seções de instruções. Para avaliar como os diversos parâmetros afetam a taxa de compressão, o número de *codeword* e o tamanho do

dicionário foram usados os aplicativos do *benchmark MediaBench*. A plataforma de teste usada foi a SPARC.

Devido à elevada complexidade computacional do algoritmo proposto considerou-se apenas mil instruções de cada aplicação como entradas nos experimentos, exceto para a aplicação *adpcm* que contém apenas 365 instruções. Assim, os autores obtiveram como resultados dos experimentos uma redução de aproximadamente 20% no tamanho final dos códigos das instruções.

3.6 CCTP (*Code Compressed THUMB Processor*)

XU *et al* [59] definiram uma nova arquitetura que é independente do algoritmo de compressão e descompressão de código. Os autores não tiveram a intenção de se aprofundar em desenvolvimento ou melhoria de um novo algoritmo de compressão de código, mais sim de analisar os algoritmos já conceituados na área da pesquisa e usá-los na nova arquitetura, chamada de *Code Compressed THUMB Processor* (CCTP) (ver Figura 3.11).

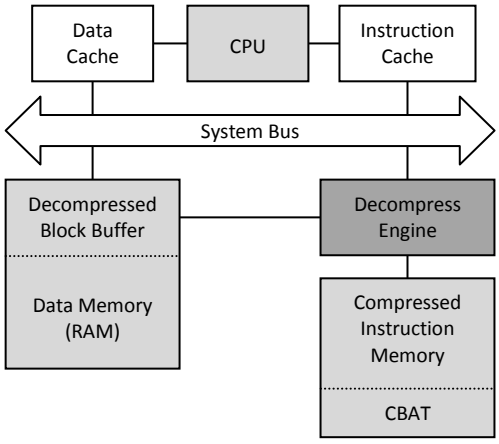


Figura 3.11 – Arquitetura CCTP [59]

Os componentes da arquitetura CCTP são: *Caches* de dados e instruções, CPU, Memória RAM, Barramento, *Buffer* do Bloco Descomprimido (DBB) que armazena temporariamente os blocos das instruções descomprimidas, Memória de instruções comprimidas, Tabela de Endereço dos Blocos Comprimidos (CBAT) que é usada para converter o espaço de endereço comprimido a uma tabela de descompressão e uma Máquina de Descompressão (DE) que realiza todo o processo de descompressão, e gerencia os *hits* na DBB.

Os algoritmos de compressão de código podem aparecer na forma de software ou hardware. Porém, os autores preferiram usar um descompressor em *hardware* por ser mais rápido do que o correspondente descompressor em software.

A Figura 3.12 mostra o modelo da unidade da máquina de descompressão proposta por XU *et al* [59]. Dois *buffers* são integrados para manter o bloco atual descomprimido e o bloco corrente comprimido. A máquina de descompressão realiza quatro tarefas, a saber: (i) cacheia todas as *misses caches*; (ii) gerencia a DBB como *cache* L2; (iii) envia a linha de *cache* necessária e (iv) lança a execução do descompressor.

Os autores escolheram o ISA (*Instruction Set Architecture*) da plataforma ARM como base, porque desenvolveram uma arquitetura que comprimiu com eficiência o código ARM/THUMB. No entanto, a arquitetura CCTP também é capaz de trabalhar com outros ISA.

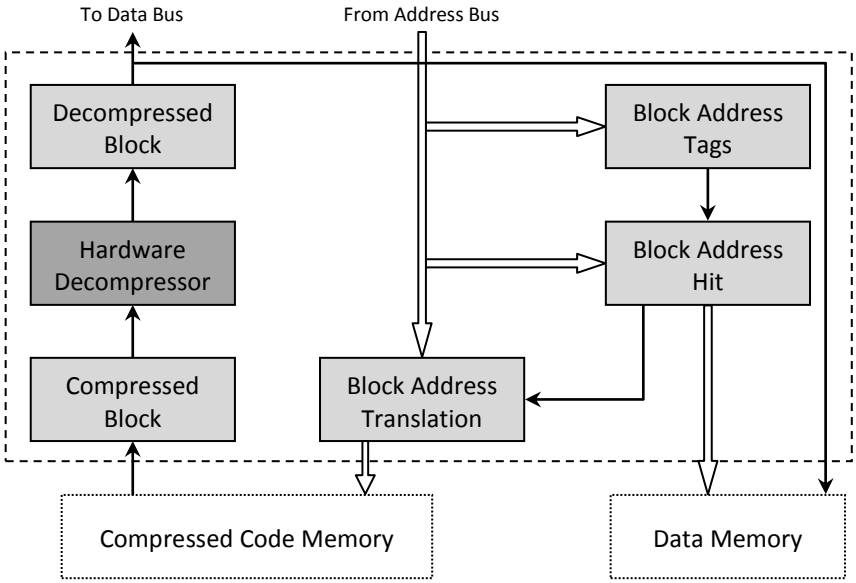


Figura 3.12 – Máquina de descompressão proposta por XU *et al* [59]

Para os experimentos foi usada a ferramenta *SimpleScalar* definida como plataforma de simulação para a arquitetura CCTP em software. A compressão de código trabalhou com o núcleo do microprocessador ARM9. Como pacote de entrada para os experimentos, usaram-se os aplicativos do *benchmark MiBench*. Segundo XU *et al* [59] na arquitetura CCTP a memória de compressão proposta mostrou uma redução de 15 a 20% no tamanho do código das instruções do processador ARM/THUMB e um aumento de 5% no desempenho final do sistema.

3.7 Tripla Otimizações do I-Cache

RAWLINS & GORDON-ROSS [51] desenvolveram uma pesquisa explorando a interação de três tipos de otimizações para a *cache* de instruções, sendo: *cache loop*, configuração de *cache* e compressão de código.

O *cache loop* são pequenos dispositivos que fornecem um método eficaz para diminuir o consumo de energia na hierarquia da memória, armazenando o código frequentemente executado (regiões críticas), em uma estrutura mais eficiente energeticamente do que a de um *cache* de nível L1 [51]. O principal objetivo do *cache loop* é fornecer ao processador o máximo de instruções possíveis. No *cache loop* o *Preloaded Loop Cache* (PLC) exige projeto aplicado com pré-análise estática para armazenar as regiões de código complexo (código com saltos) e o *Adaptive Loop Cache* (ALC) realiza essa análise durante a execução e não requer nenhum esforço de projeto.

Os microprocessadores disponíveis usam uma única configuração de *cache* que funciona de forma adequada para a maioria dos programas de um determinado conjunto de aplicativos. No entanto, essa configuração média raramente é a configuração ideal para o pedido de diversas aplicações que apresentam comportamentos diferentes em tempo de execução. As instruções da configuração de *cache* analisam o fluxo da instrução e configura a *cache* para o menor consumo de energia (ou mais alto desempenho) ao configurar o tamanho do *cache*, o tamanho dos blocos e a associatividade.

Já as técnicas de compressão de código foram desenvolvidas inicialmente para reduzir o tamanho do código estático nos sistemas embarcados. No entanto, pesquisas recentes de compressão código analisaram seus efeitos na energia consumida na busca das instruções nos sistemas embarcados. Nestes sistemas, a energia é conservada por armazenar instruções comprimidas no *cache* de instrução de nível L1 e descomprime estas instruções (durante a execução) com um baixo consumo de energia e um alto desempenho. Dessa forma, este trabalho [51] encontra-se na classificação PDC (*Processor Decompressor Cache*).

Os autores usaram o *SimpleScalar* para modelar e analisar os três tipos de otimizações para a *cache* de instruções, porém foi modificado o código da *SimpleScalar* para inserir a unidade de descompressão, a LAT e o *loop cache*. Os programas do *benchmark EEMBC*, *MiBench* e *Powerstone* foram usados como carga de entrada para os experimentos realizados na simulação supondo uma plataforma SPARC v9.

Os resultados alcançados mostraram que o *loop cache* aumentou a redução de energia em até 26% em relação à configuração de *cache* sozinha e reduziu a sobrecarga de energia na

descompressão em 73%. No entanto, para explorar plenamente o *loop cache*, a configuração de *cache* combinada e a compressão de código, um algoritmo de compressão/descompressão com menor sobrecarga do que a técnica de codificação de *Huffman* é necessária.

3.8 Multiple Bitstream Compression

SEONG & MISHRA [50] desenvolveram uma nova técnica de colocação de *bitstream* comprimida para apoiar a descompressão paralela, sem prejudicar a eficiência da compressão. A técnica proposta permite a divisão de uma única *bitstream* (instrução binária) obtida a partir da memória em múltiplos *bitstreams*, que são inseridos em diferentes decodificadores. Como resultado, múltiplos decodificadores podem trabalhar simultaneamente para produzir o efeito da alta largura de banda na decodificação.

Os autores [50] afirmam que existe um obstáculo que impede decodificar simultaneamente todos os campos da mesma instrução porque o início de cada campo da instrução comprimida é desconhecido, a menos que descomprima todos os campos anteriores. No entanto, uma forma intuitiva que os autores tiveram para resolver este problema foi separar todo o código em duas partes, comprimir cada uma delas separadamente e depois colocá-las separadas. Usando tal alternativa, as diferentes partes da mesma instrução podem simultaneamente ser decodificadas usando dois ponteiros. No entanto, se uma parte do código (parte B) é comprimida de forma mais eficaz do que a outra parte (parte A), então o espaço restante e não utilizado na parte B será desperdiçado (conforme se observa na Figura 3.13).

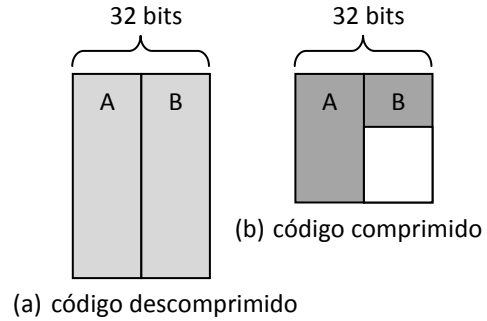


Figura 3.13 – Descompressão paralela [50]

Segundo SEONG & MISHRA [50] os métodos de codificação de comprimento fixo são apropriados para descompressão paralela, mas prejudica a eficiência da compressão devido à codificação fixa. Porém, o foco dos autores, nesse trabalho, foi de permitir a descompressão paralela para os binários comprimidos com os métodos de codificação de comprimento variável.

A Figura 3.14 mostra o diagrama de blocos do *framework* de compressão e descompressão proposto pelos autores em [50]. O diagrama é composto de quatro partes principais: compressão (*encoder*), *bitstream merge logic*, *bitstream split logic* e descompressão (*decoder*).

Conforme SEONG & MISHRA [50], durante a compressão (ver Figura 3.14(a)), primeiramente são quebrados todos os blocos de armazenamento de entrada, que contém um ou mais instruções em vários campos e depois são aplicados os codificadores específicos para cada um deles. O resultado da compressão dos *streams* é combinado por uma unidade de *Bitstream Merge Logic* com base em um Algoritmo de Posicionamento de *Bitstream* (BPA) cuidadosamente projetado. Na descompressão (ver Figura 3.14(b)) cada palavra trazida da *cache* é primeiramente dividida em várias partes, cada uma das quais pertence a um *bitstream* comprimido produzido pelo codificador. Em seguida, a unidade de *Bitstream Split Logic* despachá-as para os *buffers* de decodificadores corretos de acordo com o BPA. Estes decodificadores devem decodificar cada *bitstream* e gerar os campos de instruções descomprimidas. Depois de combinar estes campos em conjunto, o resultado final da descompressão deve ser idêntico ao original do armazenamento de bloco de entrada correspondente.

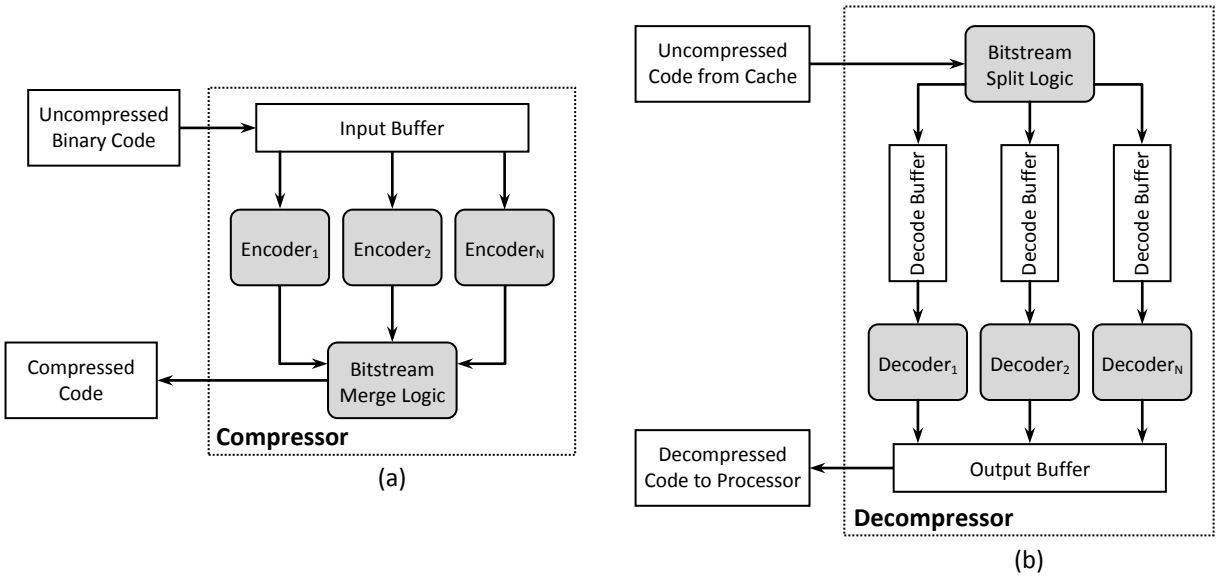


Figura 3.14 – Proposta do *framework* de compressão de instrução: (a) técnica de compressão (b) mecanismo de descompressão [50]

A compressão de instrução e os experimentos da descompressão paralela usando o *framework* descrito em [50] foram realizados com os aplicativos dos *benchmarks MediaBench* e *MiBench*. Também foram utilizadas como arquitetura alvo as plataformas TI TMS320C6x, PowerPC, SPARC e MIPS. O *hardware* descompressor foi implementado em Verilog HDL e

realizaram uma análise e avaliação da sobrecarga do *hardware* introduzido pelo mecanismo de descompressão paralelo.

SEONG & MISHRA [50] concluíram que a abordagem proposta pode ser vista como uma aproximação do esquema de colocação teórico ideal, que minimiza o número de falhas durante a descompressão. Além disso, o método de colocação proposto não se restringe a um algoritmo de compressão específico, uma vez que pode ser empregado para acelerar qualquer algoritmo de compressão com perda insignificante na razão de compressão. Os resultados experimentais demonstraram que a abordagem baseada em *Multiple Bitstream* melhorou a largura da banda na descompressão em até quatro vezes com um impacto menor que 3% sobre a eficiência da compressão.

3.9 Dual Code Compression

SHRIVASTAVA & MISHRA [57] propuseram um novo esquema de compressão de código dual que procurou otimizar simultaneamente a melhoria no desempenho e a redução no tamanho do código. A diferença entre os métodos de compressão existentes e a abordagem proposta pelos autores é que a compressão e a descompressão são feitas duas vezes, sendo a primeira para a melhoria do desempenho e a segunda para a redução do tamanho dos códigos, com base nas frequências dinâmicas e estáticas, respectivamente.

De acordo com os autores [57], para alcançar um aumento na velocidade de processamento da aplicação é necessário reduzir a taxa de *miss cache,* o que é possível com a inserção de código comprimido no *cache*. Assim, armazenando as instruções executadas com mais frequência na forma comprimida aumenta-se o uso do *cache* e, consequentemente melhora-se o desempenho do sistema com a redução da taxa de *miss cache*.

A Figura 3.15 mostra uma visão geral do método de compressão de código dual. Conforme SHRIVASTAVA & MISHRA [57], a primeira etapa no processo de compressão é a criação do *perfil*, que envolve a identificação de todos os blocos básicos de código do programa e as frequências relativas com que são buscadas, em seguida é criado um dicionário com base nos blocos buscados com maior frequência incluindo todos os destinos de salto. A segunda etapa de forma eficiente comprime o código de uma maneira que melhor explora a localidade das instruções buscadas com mais frequência no bloco básico (ver Figura 3.16).

Os autores definiram que quando as palavras não contêm nenhuma informação sobre se ela é comprimida ou não, uma tabela com o *Bloco Básico de Mapeamento* (BBM) é necessário [57]. Cada entrada na tabela é composta de informações sobre um bloco básico,

como: endereço da primeira instrução do bloco, endereço da última instrução e seu endereço mapeado para o formato comprimido. A tabela BBM elimina a necessidade de *flag* de *bits*/*bytes* que indica se a instrução é comprimida ou não. O tamanho da tabela BBM é pequena, pois só contém informações sobre as instruções mais buscadas nos blocos básicos.

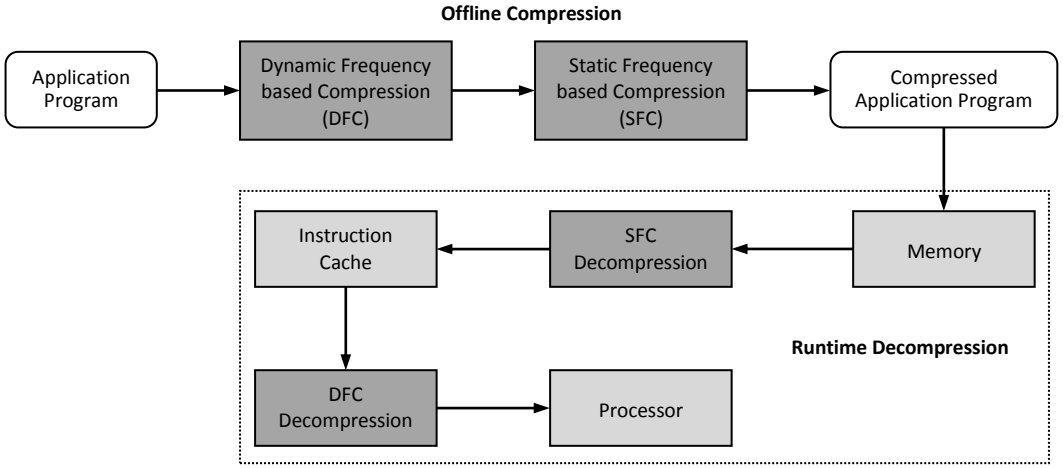


Figura 3.15 – Visão geral do método de compressão de código dual [57]

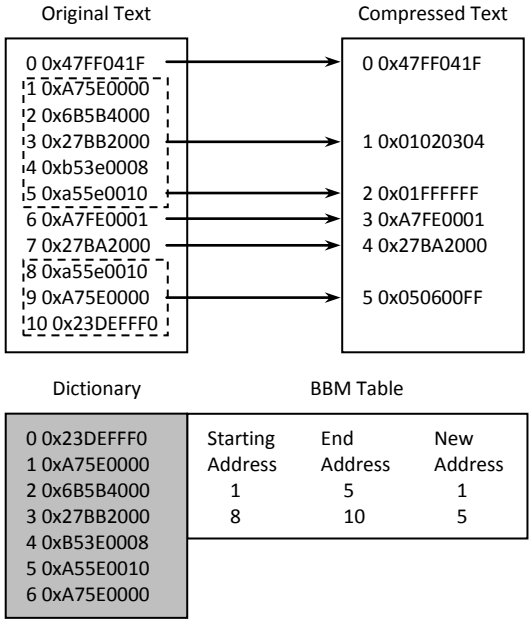


Figura 3.16 – Mecanismo de compressão baseada na DFC do *Dual Code Compression* [57]

De acordo com SHRIVASTAVA & MISHRA [57], o fato do descompressor não estar no caminho crítico de execução, ou seja, o descompressor não é chamado para cada busca pelo processador; permite usar mecanismos de compressão eficientes. A compressão SFC (baseada em frequência estática) é realizada no código comprimido DFC (baseada em frequência dinâmica). No entanto, é útil considerar um tamanho maior para o dicionário

quando o atual tamanho do dicionário não pode acomodar todas as instruções com o valor de frequência acima de determinado limite. O tamanho do dicionário na compressão SFC foi de 2.048 (entradas). Ainda é importante salientar que um dicionário maior aumenta o tempo de acesso e, consequentemente, reduz a eficiência na descompressão.

Conforme se observa na Figura 3.15, o DFC deve ser concluído antes do início da compressão SFC (*offline*). Por isso, a descompressão SFC precisa ser feita antes da descompressão DFC (*on-the-fly*).

Os experimentos foram realizados com o simulador *SimpleScalar* usando a plataforma MIPS juntamente com programas dos *benchmarks MediaBench* e *MiBench*. Foi inserido no código fonte do simulador o código da implementação dos descompressores DFC e SFC. Os resultados experimentais obtidos com essa técnica mostraram que simultaneamente foi possível atingir o melhor da compressão dinâmica e estática, ou seja, obteve-se uma redução de aproximadamente 40% (razão de compressão variando entre 60% a 65%) e uma melhora de 50% no desempenho do sistema. Os autores [57] concluem afirmando que os sistemas embarcados geralmente usam *caches* pequenas então a técnica proposta é benéfica em tais sistemas.

3.10 Two-Level Dictionary Code Compression

COLLIN & BRORSSON [16] desenvolveram um novo método de compressão que reduz o tamanho do código da instrução. No projeto foram usados dois dicionários, que são capazes de lidar com a compressão das instruções individuais e as sequências de instruções no código. Os dois dicionários estão em estágio diferente do *pipeline* e trabalham juntos para descomprimir as instruções e as sequências de instruções. O impacto sobre o tamanho do armazenamento para os dicionários é pequeno, porém, as sequências no dicionário são armazenadas como instruções individuais em vez de instruções normais [16]. Conforme os autores, o método proposto consiste em dois tipos de *codewords*: (i) *Codeword* de Instrução (ICW) e (ii) *Codeword* de Sequência (SCW), conforme se observa na Figura 3.17.

De acordo com COLLIN & BRORSSON [16], sempre que o *cache* de instrução é acessado, assume-se que uma busca de palavra (PF) de 32 *bits* é obtida. Dependendo do código comprimido, três casos podem acontecer. Uma palavra buscada pode conter até quatro palavras de sequência (caso I na Figura 3.17), até três palavras de instrução de código (caso II) ou uma instrução descomprimida (caso III).

- **Caso I**: os dados buscados da memória correspondem a quatro SCWs, que exigem uma descompressão completa. Uma por uma as SCWs individuais são usadas para indexar a sequência no dicionário e recuperar uma sequência de palavras (SW) que contém uma sequência de ICW. No exemplo da Figura 3.17, a SW recuperada contém quatro ICWs e cada uma é usada para indexar a instrução no dicionário, finalmente, a instrução original é recuperada;
- **Caso II**: o segundo formato é um formato comprimido de ICW, o que corresponde a 2 ou 3 instruções comprimidas e representadas como uma ICW. Uma vez que nenhuma palavra de ordem é envolvida, pode-se ignorar a sequência do dicionário e ir diretamente ao índice da instrução no dicionário usando os ICWs;
- **Caso III**: a busca da palavra contém apenas uma instrução descomprimida. Uma vez que a instrução não é comprimida, a unidade de busca da instrução pode ignorar os dicionários.

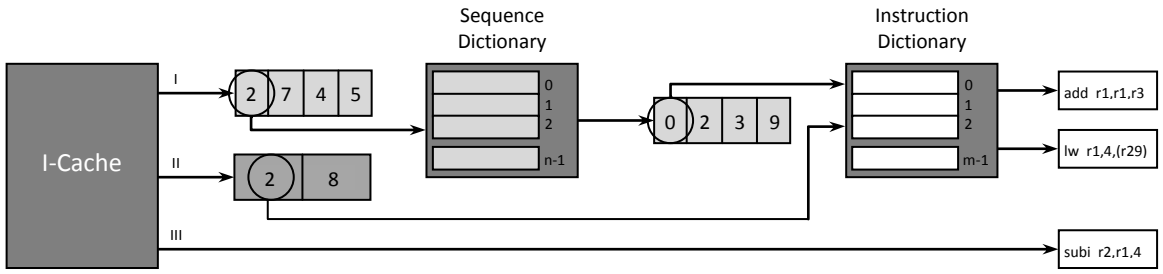


Figura 3.17 – Processo de geração de código na compilação e os dois níveis de buscas [16]

Conforme os autores [16] para a arquitetura do compressor de código foi definida a CCA (Arquitetura da Compressão de Código) que é a forma de codificar instruções descomprimidas que pode coexistir com as ICWs e SCWs. É importante que as instruções descomprimidas possam coexistir com as instruções comprimidas como apenas uma pequena fração de todas as instruções que constituem o programa inteiro. A Figura 3.18 mostra uma visão geral da arquitetura proposta em [16].

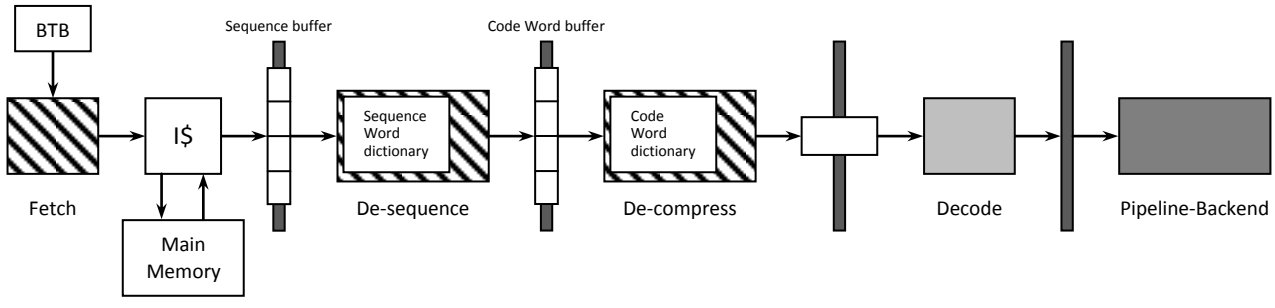


Figura 3.18 – Visão geral da arquitetura com *pipeline* ampliado [16]

Foram realizadas alterações na ferramenta *SimpleScalar* e usou-se o conjunto de instruções ISA da plataforma MIPS IV juntamente com programas do *benchmark MediaBench* para avaliar a eficácia desse novo sistema de compressão. Os resultados obtidos nos experimentos, com essa técnica, mostraram uma razão de compressão média de 77% na compressão dinâmica e uma redução de 2% a 21% no consumo de energia da arquitetura.

3.11 Resumo Comparativo dos Métodos Baseados em Dicionário

A Tabela 3.2 apresenta uma comparação dos métodos de compressão de código baseados em dicionários descritos neste capítulo. Nessa comparação considera-se: o método de compressão/descompressão; a plataforma base; a suíte de *benchmark* usada; a razão de compressão obtida; o desempenho final do sistema e o consumo de energia do método. Na comparação considera-se como 100% a razão de compressão e o tempo de execução do programa original (descomprimido).

Conforme se observa na Tabela 3.2 as plataformas Alpha EV6, e PowerPC são únicas nos trabalhos analisados neste capítulo, porém os métodos que usaram essas plataformas não podem ser comparados com outros métodos. Mesmo assim destacamos que o método DISE (Seção 3.3) obteve métricas aceitáveis para a implementação em sistemas embarcados.

Tabela 3.2 – Sumarização dos métodos de compressão de código baseados em dicionário

Método	Plataforma	Benchmark	Razão de Compressão	Desempenho	Consumo de Energia
Bitmasks Compression [54, 55]	TMS320C6x	MediaBench, MiBench	65%	95%	---
	MIPS		60%		
	SPARC		55%		
Profiling Agregado [13]	MIPS IV	MediaBench	75%	108% a 112%	---
DISE [17]	Alpha EV6	SPEC'2000, MediaBench	65% a 75%	80% a 95%	90%
Hybrid Compression [25]	ARM	MiBench	80% DicSmall	---	---
			91% DicLarge		
Merge Bitmask and DISE [58]	SPARC	MediaBench	80%	---	---
CCTP [59]	ARM9	MiBench	80% a 85%	105%	---
Tripla Otimizações do I-cache [51]	SPARC	EEMBC, MiBench, Powerstone	80%	---	73%
Multiple Bitstream Compression [50]	TMS320C6x	MediaBench, MiBench	---	---	---
	PowerPC				
	SPARC				
	MIPS				
Dual Code Compression [57]	MIPS	MediaBench, MiBench	60% a 65%	50%	---
Two-Level Dictionary Code Compression [16]	MIPS IV	MediaBench	77%	---	79% a 98%

O método Multiple Bitstream Compression (Seção 3.8) não possui valores devido o autor não ter apresentado os dados obtidos para a razão de compressão, desempenho e consumo de energia.

Dos trabalhos que utilizaram a plataforma MIPS, o método Bitmasks Compression (Seção 3.1) apresentou maiores resultados na razão de compressão (55% a 65%), embora a métrica de desempenho tenha sido a favor do método Dual Code Compression (Seção 3.9) que alcançou uma razão de compressão entre 60 a 65%. Já no uso da plataforma MIPS IV, pelos métodos Profiling Agregado (Seção 3.2) e Two-Level Dictionary Code Compression (Seção 3.10), os resultados obtidos foram próximos, apresentando uma pequena vantagem de 2% para o primeiro método (*Profiling* Agregado), mas deve-se lembrar que esse método teve uma degradação de 8% a 12% no desempenho final do sistema.

Os métodos Hybrid Compression (Seção 3.4) e CCTP (Seção 3.6) usaram a plataforma ARM e os maiores resultados alcançados por esses métodos foram idênticos, ou seja, 80% para a métrica razão de compressão. Ainda destacamos que HAIDER & NAZHANDALI [25] no método *Hybrid Compression* definiram dois tamanhos diferentes de dicionários, sendo um para 512 a 1.024 instruções e outro para 4.096 a 8.092 instruções.

Para a plataforma SPARC o método Bitmasks Compression apresentou uma razão de compressão maior (55% a 65%) do que a dos métodos Merge Bitmask and DISE (Seção 3.5) e Tripla Otimizações do I-cache (Seção 3.7) que foi de 80% para ambos os métodos.

3.12 Considerações Finais do Capítulo

Nesse capítulo, estudamos de forma detalhada dez novos métodos para redução do tamanho do código de instruções, publicados entre os anos de 2003 a 2011. A escolha desses artigos deu-se devido ao fato dos métodos implementados serem baseados em dicionário e também usarem a ferramenta de simulação arquitetural *SimpleScalar*.

Ainda destacamos que em muitos casos, os autores dos artigos não fornecem informações suficientes sobre os métodos, as métricas (razão de compressão, desempenho e consumo de energia) e, quando o fazem, muitas vezes os dados não são diretamente comparáveis devido às diferenças de plataformas e *benchmarks* utilizadas para os experimentos.

A média da razão de compressão dos métodos estudados e apresentados nos Capítulos 2 e 3 dessa tese ficaram em aproximadamente 70%, o desempenho final do sistema em 90% e o consumo de energia em 75%.

No próximo capítulo descreve-se sobre as diferentes técnicas de avaliação mais usadas (simulação, prototipação e modelagem) em arquiteturas de computadores, em seguida apresenta-se a ferramenta de simulação arquitetural *SimpleScalar* e o *benchmark MiBench* para medição de desempenho. Também se explica as principais características dos processadores embarcados, enfatizando o processador ARM, que é um dos processadores foco dessa tese.

CAPÍTULO 4

AMBIENTE EXPERIMENTAL DE SIMULAÇÕES

Inicialmente, são apresentadas as características básicas das técnicas de avaliações (para arquiteturas de computadores) mais usadas e discutidas suas vantagens e desvantagens. Para execução dessa tese será utilizada a metodologia de simulação com a ferramenta *SimpleScalar*. São ressaltadas neste capítulo as características dos programas do *benchmark MiBench* que são utilizados para realizar as medições de desempenho da compressão/descompressão dos códigos de instruções; em seguida apresenta-se uma caracterização dos processadores embarcados enfatizando o processador ARM, e por fim, a infraestrutura utilizada pelos métodos desenvolvidos nessa tese.

4.1 Metodologias de Avaliação

Em geral, o estudo e avaliação de um sistema computacional são baseados na descrição da arquitetura e na definição dos programas aplicativos que estimulam o sistema. As técnicas de avaliação existentes são [49]: técnica de simulação, técnica de prototipação e técnica de modelagem, as quais são descritas brevemente.

4.1.1 Técnicas de Simulação

De acordo com [49], um simulador emula o comportamento dos programas aplicativos num ambiente específico, levando em consideração as características do sistema que determinam como ele se comportará quando submetido a um determinado programa. Características da rede de interconexão, da arquitetura do processador, do sistema de memória e até do software do sistema podem ser incluídas no ambiente de simulação. Segundo

ORDONEZ [49], as técnicas de simulação mais usadas são: Simulação Comandada por Rastro (SCR - *Trace Driven Simulation*), Simulação Comandada por Execução (SCE - *Execution Driven Simulation*) e Simulação Comandada por Programa (SCP - *Program Driven Simulation*). Esta tese utiliza a técnica de simulação comandada por execução que é comentada a seguir.

O simulador controla a execução das tarefas, fazendo com que a intercalação (*interleaving*) das referências à memória seja similar à da máquina simulada (máquina real). Além disso, não requer o armazenamento de rastros. No caso do *SimpleScalar* [14], uma ferramenta que permite a simulação comandada por execução, só se faz necessário o pré-processamento e a recompilação da aplicação depois de cada mudança em qualquer parâmetro da arquitetura.

Geralmente um simulador comandado por execução pode ser visto como uma ferramenta composta por duas partes principais: um gerador de eventos e um simulador do sistema alvo, conforme ORDONEZ [49]. O gerador de eventos modela a execução de um aplicativo com o uso do processador. Quando o programa realiza uma operação de interesse (exemplo uma referência à memória), o gerador transmite o evento ao simulador do sistema alvo.

O simulador do sistema alvo modela a rede de interconexão, os módulos de memória do sistema e todos os recursos de hardware ou software que oferecem contenções ou sobrecargas [49]. De modo geral, são considerados todos os módulos de interesse a serem estudados e avaliados, determinando quais processos podem continuar em execução com base no momento em que cada evento se completa. A Figura 4.1 mostra esse processo. Outro componente não visível nessa figura são as bibliotecas de simulação que gerenciam os eventos e os processos do programa em execução.

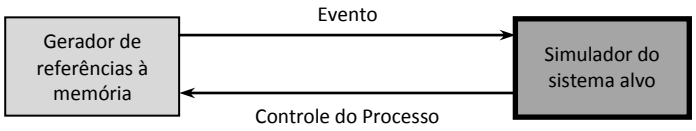


Figura 4.1 – Simulação Comandada por Execução (SCE) [49]

4.1.2 Técnica de Prototipação

De acordo com [49], a construção de protótipos é outro mecanismo possível para avaliar o desempenho de novas arquiteturas. A sua principal vantagem é o fornecimento de dados sobre o custo de construção do sistema e detalhes do hardware.

Porém, muitas vezes sistemas digitais devem ser prototipados o mais rápido possível, seja para atender ao mercado ou para verificação das especificações iniciais, conforme citado em CARRO [15]. Um protótipo antecipado do hardware é importante para não ocorrer os atrasos na elaboração e desenvolvimento dos softwares. Quando for necessária a utilização de um determinado circuito com alta frequência de operação ou uma interface rápida e assíncrona com o mundo real, apenas simulação em ferramentas CAD (*Computer-Aided Design*) não é o suficiente, já que estes fatores não são simuláveis. Portanto, faz-se necessário a prototipação do hardware em um meio físico como FPGAs.

Do ponto de vista tecnológico, existe uma vasta gama de opções para a implementação em hardware. As soluções mais interessantes para hardware dedicados são os *Gate Array* e os FPGAs [18]. Estes tipos de componentes proporcionam a integração de lógica e memória em um único circuito, acelerando a integração e a prototipação dos sistemas.

4.1.3 Técnica de Modelagem

A técnica de modelagem também conhecida como modelo analítico, descreve as operações do sistema e os programas aplicativos em termos matemáticos. As estimativas de desempenho são obtidas mediante uma solução analítica ou numérica do modelo matemático resultante. Esses modelos podem ser determinísticos ou probabilísticos. Os determinísticos são usados para estimar a ordem de magnitude ou limites de determinados índices de desempenho, mas não são muito empregados já que não representam a variabilidade dos programas aplicativos. Por outro lado, os modelos probabilísticos permitem a representação de certos aspectos da variabilidade, conforme ORDONEZ [49].

Assim, nesta tese a metodologia de avaliação aplicada é a Técnica de Simulação, especificamente usando-se a ferramenta de simulação arquitetural *SimpleScalar*.

4.2 Simulador SimpleScalar

A ferramenta *SimpleScalar Tool Set* [14] foi desenvolvida na Universidade de *Wisconsin* (*Madison*) como parte do projeto *MultiScalar*. O *SimpleScalar* é um ambiente de simulação que agrupa um conjunto de ferramentas (simuladores, compiladores, *assemblers* e *linkers* para a arquitetura simulada) que suportam simulações detalhadas para algumas características encontradas nos processadores modernos.

A *SimpleScalar* é uma das principais ferramentas de simulação utilizada no meio acadêmico e profissional [14], as simulações das arquiteturas de processadores são focadas

principalmente nas características de arquiteturas escalares e super-escalares. Esses simuladores são amplamente utilizados para a pesquisa de novos mecanismos e comportamento de arquiteturas de processadores devido à flexibilidade e possibilidade de extensão do seu código, bem como o detalhamento nos resultados.

A arquitetura implementada pelos simuladores é muito similar à dos processadores embarcados: MIPS (PISA - *Portable ISA*), ARM (StrongARM SA-11xx) e ALPHA (Compaq EV6), a qual é amplamente usada no ensino de arquiteturas de computadores [26], com versões *big-endian* e *little-endian*. Seis simuladores orientados à execução são incluídos. Além disso, a ferramenta fornece códigos binários pré-compilados (incluindo programas do *benchmark* científico *SPEC'95*) e uma versão do compilador GCC (GNU *Compiler* C) modificado que compila códigos fontes FORTRAN ou C, gerando códigos binários executáveis para a arquitetura específica do simulador. Um depurador e um visualizador de *pipeline* (em modo texto) também estão disponíveis. A Figura 4.2 mostra a arquitetura geral do simulador *SimpleScalar* definido por BURGER & AUSTIN [14].

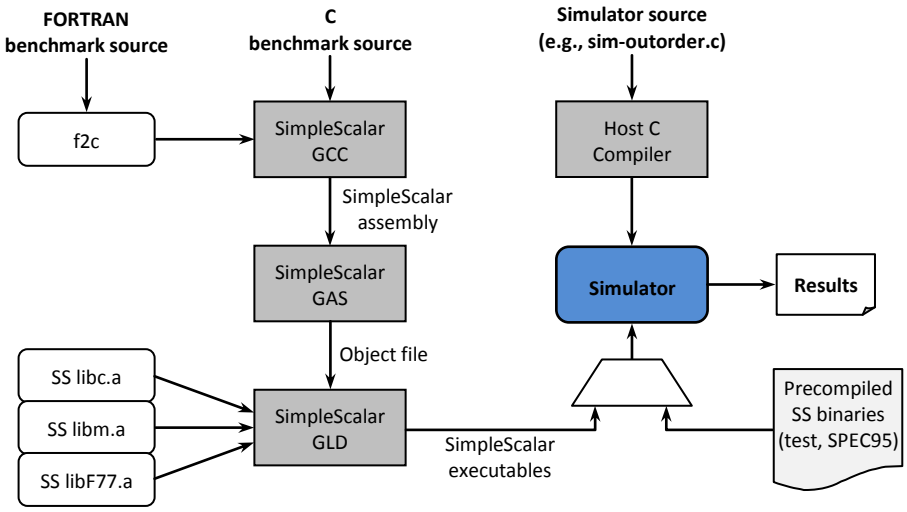


Figura 4.2 – Arquitetura do simulador *SimpleScalar* definida por BURGER & AUSTIN [14]

As chamadas de sistema feitas pelos binários executados em simulação são interceptadas por um tratador que decodifica a chamada, copia os argumentos e faz a chamada correspondente ao sistema operacional da máquina hospedeira, e copia os resultados da chamada para a memória do programa simulado. O *SimpleScalar* utiliza espaço de endereçamento de 31 *bits*. Os seis simuladores orientados à execução inclusos na ferramenta *SimpleScalar* são [14]:

- **Sim-fast**: cada instrução é executada sequencialmente;

- **Sim-safe**: cada instrução é executada sequencialmente e é verificado o alinhamento e a permissão de acesso a cada referência de memória, o que diminui o seu desempenho;
- **Sim-cache**: faz a simulação de hierarquia de memória *cache*;
- **Sim-cheetah**: faz a simulação de hierarquia de memória juntamente com uma nova política de substituição;
- **Sim-profile**: gera o *profile* de cada classe de instruções (desvio, cálculo inteiro/ponto-flutuante, acesso à memória);
- **Sim-outorder**: modela um processador completo, incluindo simulação de ciclos e suporta execução fora de ordem, baseada na RUU (*Register Update Unit*) e em uma fila de *loads* e *stores*. Este simulador implementa um *pipeline* de seis estágios: busca, despacho, escalonamento, execução, escrita de resultados e *commit*.

4.3 Benchmarks para Medição de Desempenho

Atualmente existe uma grande variedade de *benchmarks*, tais como: *Dhrystone*, *Linpack*, *HPL*, *SPEC*, *Splash-2* e *MediaBench*, sendo, em sua maioria, desenvolvidos para uma área específica da computação. Dentre os *benchmarks* existentes, um dos mais utilizados é o SPEC (*Standard Performance Evaluation Corporation*), que atualmente está na versão SPEC2010 [69].

O SPEC é um *benchmark* para computadores de uso geral, que agrupa um conjunto de programas e dados divididos em duas categorias: inteiro e ponto flutuante [69]. Esse *benchmark* ganhou grande popularidade no mundo comercial e acadêmico, por ser um ótimo medidor de desempenho. Essa popularidade tem influenciado o desenvolvimento de microprocessadores de uso geral, empregados em computadores pessoais e servidores, de modo que esse processadores são projetados para ter alta performance no SPEC, segundo GUTHAUS *et al* [24].

De acordo com [24], apesar de o SPEC ser uma referência para processadores de propósito geral, não é uma boa opção para avaliar desempenho de microprocessadores dedicados. A principal característica dessa classe de processadores é a diversidade de domínios de aplicação, e, além disso, não existe uma padronização para o conjunto de instruções das arquiteturas, como acontece nos processadores de uso geral, o que torna difícil a utilização da maioria dos *benchmarks*.

Então, surgiram algumas pesquisas para desenvolver *benchmarks* para processadores especializados. Umas das principais é a suíte desenvolvida e conhecida como EEMBC (*EDN Embedded Microprocessor Benchmark Consortium*) [67]. Este *benchmark* reúne vários domínios de aplicação e disponibiliza um conjunto de *benchmarks* para avaliar cinco domínios de processadores embarcados.

Neste contexto, GUTHAUS *et al* [24] apresentam o *MiBench*, um *benchmark* desenvolvido para microprocessadores de uso específico, que possui trinta e cinco aplicações divididas em seis domínios de aplicações (categorias): *Automotive and Industrial Control*, *Consumer Devices*, *Office Automation*, *Networking*, *Security* e *Telecommunications*.

O *MiBench* é um *benchmark* livre, sendo basicamente implementado utilizando a linguagem de programação C. Todos os programas que compõem o *MiBench* estão disponíveis em código fonte, sendo possível executar em qualquer arquitetura de microprocessador que possua um compilador C.

No *MiBench* são consideradas as classes de instrução: controle, aritmética inteira, aritmética em ponto flutuante e acesso à memória. Nas categorias *Telecommunications* e *Security* todas as aplicações têm mais de 50% de operações inteiras na ULA (*Arithmetic Logic Unit*). A categoria *Consumer Devices* tem mais operações de acesso à memória por causa do processamento de grandes arquivos de imagem e áudio. A categoria *Office Automation* faz uso intenso de operações de controle e acesso à memória. Os programas dessa categoria usam muitas chamadas de funções para bibliotecas de *string* para manipular os textos, o que causa o uso de várias instruções de desvio. Além disso, os textos ocupam bastante espaço na memória, o que requer muitas instruções de acesso à memória. A categoria *Automotive and Industrial Control* faz uso intenso de todas as classes de instruções devido a sua aplicabilidade e por fim, a categoria *Networking* faz muito uso de operações aritméticas de inteiros, pontos flutuantes e acesso à memória, devido às tabelas de IPs para roteamento em rede de computadores, ordenação de dados e outros. Em comparação ao *benchmark* SPEC apresenta, aproximadamente, a mesma distribuição de operações para todas as categorias.

A Tabela 4.1 apresenta o número de instruções executadas por cada aplicação contida no *MiBench*, para cada conjunto de dados. Os conjuntos de dados padrão são divididos em duas classes para cada categoria do *benchmark*: conjunto de dados reduzido (*small*) e conjunto de dados expandido (*large*). O primeiro representa o uso moderado da aplicação do *benchmark*, enquanto que o segundo representa a sobrecarga da aplicação.

Tabela 4.1 – Número de instruções executadas para cada aplicação contida no *MiBench* [24]

Benchmark	Small Instruction Count	Large Instruction Count	Benchmark	Small Instruction Count	Large Instruction Count
basicmath	65.459.080	1.000.000.000	ispell	8.378.832	640.420.106
bitcount	49.671.043	384.803.644	rsynth	57.872.434	85.005.687
qsort	43.604.903	595.400.120	stringsearch	158.646	85.005.687
susan.corners	1.062.891	586.076.156	blowfish.decode	52.400.008	737.920.623
susan.edges	1.836.965	732.517.639	blowfish.encode	42.407.674	246.770.499
susan.smoothing	24.897.492	1.000.000.000	pgp.decode	85.006.293	259.293.845
jpeg.decode	6.677.595	990.912.065	pgp.encode	38.960.650	824.946.344
jpeg.encode	28.108.471	543.976.667	rijndael.decode	23.706.832	140.889.705
lame	175.190.457	544.057.733	rijndael.encode	3.679.378	24.910.267
mad	25.501.771	272.657.564	sha	13.541.298	20.652.916
tiff2bw	34.003.565	697.493.266	CRC32	52.839.894	61.659.073
tiff2rgba	36.948.939	1.000.000.000	FFT.inverse	65.667.015	377.253.252
tiffdither	273.926.642	1.000.000.000	FFT	52.625.918	143.263.412
tiffmedian	141.333.005	817.729.663	adpcm.decode	30.159.188	151.699.690
typeset	23.395.912	84.170.256	adpcm.encode	37.692.050	832.956.169
dijkstra	64.927.863	272.657.564	gsm.decode	23.868.371	548.023.092
patricia	103.923.656	1.000.000.000	gsm.encode	55.361.308	472.171.446
ghostscript	286.770.117	673.391.179			

As seis categorias do *MiBench* oferecem estruturas de programas que permitem explorar as funcionalidades do *benchmark* e do compilador para se ajustarem melhor a um determinado microprocessador. Portanto, os programas utilizados nas simulações nesta tese são do pacote *MiBench*, por serem específicos para sistemas embarcados e apresentarem categorias de aplicações diferentes [24].

4.4 Processadores Embarcados

Para avaliar os novos métodos de compressão de código propostos nesta tese e descritos nos Capítulos 5 e 6, são usadas quatro das principais arquiteturas de processadores embarcados, sendo elas: ARM [22, 53], MIPS [59a], PowerPC [42b] e SPARC [58a]. Estes quatro processadores são do tipo RISC de 32 *bits*. Uma breve introdução do processador ARM é dada na subseção seguinte, uma vez que este processador apresenta irregularidade no seu conjunto de instruções (ver Figura 4.5) e que também motivou ao desenvolvimento de um método de compressão de código específico para este processador.

4.4.1 Processador ARM

A arquitetura ARM (*Advanced RISC Machine*) é um processador RISC de 32 *bits* desenvolvido pela ARM *Limited* no início dos anos 80 [22, 53]. Devido às suas características de economia de energia e alto poder de processamento (características primordiais para o desenvolvimento de sistemas embarcados), as CPUs ARM dominaram rapidamente o mercado de eletroeletrônicos. Segundo FITZPATRICK [21], atualmente os processadores ARM são aproximadamente 92% dos processadores embarcados RISC de 32 *bits* usados no mercado de eletroeletrônicos.

Conforme YIU [63], diversas variações das CPUs ARM estão disponíveis, diferenciadas pela versão da arquitetura e pela família à qual pertence. Atualmente, a arquitetura ARM está na sétima versão, mas somente as versões 4 em diante continuam em uso. A seguir apresenta-se uma breve descrição das principais características implementadas de cada versão:

- **v1**: a primeira versão da CPU ARM incluí um conjunto de 16 registradores da CPU, instruções básicas de *load* e *store*, suportando dados de 8, 32 ou múltiplos de 32 *bits*, instruções gerais de processamento de dados, um barramento de dados de 32 *bits* e um barramento de endereços de 26 *bits*. Essa versão não foi utilizada em nenhum produto comercial;
- **v2**: a segunda versão da CPU ARM incluía instruções de multiplicação, multiplicação e acúmulo (utilizadas em operações de DSP), suporte a co-processador e bancos de registradores adicionais para interrupções rápidas (FIQ);
- **v3**: a terceira versão da CPU ARM expandiu o barramento de endereços para 32 *bits* (o que permitiu o endereçamento de até 4 *GBytes*), incluiu novos registradores de estado da CPU, instruções adicionais para manipulação desses registradores, novos modos de processamento de exceção da CPU (*Data*/*Prefetch abort* e instrução indefinida) com novos registradores para suporte a estes e um conjunto alternativo de instruções de 16 *bits* (*Thumb*) que aumenta a densidade de código com um mínimo impacto na velocidade de execução. As CPUs que incluem o modo *Thumb* são caracterizadas pelo sufixo T, por exemplo, v3T;
- **v4**: a quarta versão da arquitetura ARM adicionou as instruções para manipulação de 16 *bits* e instruções de carga de 8 e 16 *bits* com extensão de sinal;
- **v5**: a quinta geração da arquitetura ARM adicionou novas instruções, como a contagem de zeros à esquerda, *breakpoint* por software e etc. Melhorou a

interoperabilidade entre os modos ARM e *Thumb* e incluiu dois módulos de suporte à CPU: instruções avançadas de processamento digital de sinais (v5TE) e módulo de acelerador Java (*Jazelle*) (v5TEJ);

- **v6**: a sexta geração da arquitetura ARM abrange diversos melhoramentos no suporte à memória, multiprocessamento e gerenciamento de exceções. Inclui os módulos DSP e Java lançados na versão 5. Também foram inseridas instruções especiais para processamento de áudio e vídeo com a filosofia SIMD (*Single Instruction Multiple Data*) e introduzida a nova versão do modo *Thumb* chamada de *Thumb*-2, que aumenta a funcionalidade do conjunto de instruções *Thumb* de 16 *bits*, incluindo diversas novas instruções, inclusive de 32 *bits*;
- **v7**: a geração mais recente da arquitetura ARM é totalmente baseada na tecnologia *Thumb*-2, e garante performances muito mais elevadas. Existem três variantes da versão 7 sendo: A - para alta performance; R - para aplicações de tempo real e M - para pequenas aplicações de baixo custo.

As diferentes versões das CPUs ARM resultaram na criação de diversas famílias ARM composta por variações, em que cada uma é evolução da família anterior, normalmente visando o aumento da performance [63], podemos citar algumas famílias como: ARM7, ARM9, ARM9E, ARM10E, ARM11, Cortex®, StrongARM, XScale e outras.

Conforme FURBER [22] e SEAL [53], a arquitetura ARM é composta por 37 registradores, todos com largura de 32 *bits*. Os registradores são organizados em vários bancos. A quantidade de registradores disponíveis depende do estado da CPU. No modo ARM estão disponíveis 17 registradores (16 de uso geral e um de estado do programa). Já no modo *Thumb* há 12 registradores (8 de uso geral e 4 de uso interno da CPU).

Cada instrução no ARM é codificada com palavra de 32 *bits*. Todas as instruções do processador ARM são condicionalmente executadas, o que significa dizer que a sua execução pode ou não ocorrer dependendo dos valores dos *flags* (matemáticos e controle do sistema) **N**, **Z**, **C** e **V** do registrador CPSR (*Current Program Status Register*). A Figura 4.3 mostra o registrado CPSR.

Figura 4.3 – Registrador CPSR do processador ARM [22, 53]

Os *bits* 8 até 27 da Figura 4.3 são reservados e não são utilizados nas versões inferiores à 5 da arquitetura ARM, por isso não devem ser alterados pelo programa do usuário. Os *bits* **I** (controle das interrupções IRQ), **F** (controle das interrupções FIQ), **T** (controle de estado da CPU ARM/*Thumb*) e **M4** até **M0** são usados para o controle do funcionamento do processador, conforme se indica na Tabela 4.2.

Tabela 4.2 – *Bits* de seleção do modo de processamento [22, 53]

M4	M3	M2	M1	M0	Mode
1	0	0	0	0	User
1	1	1	1	1	System
1	0	0	0	1	FIQ
1	0	0	1	0	IRQ
1	0	0	1	1	Supervisor
1	0	1	1	1	Abort
1	1	0	1	1	Undefined

O formato da codificação básica das instruções ARM, são as instruções de: Carga (*load*), Armazenamento (*store*), Mover (*move*), Aritmética (*arithmetic*) e Lógica (*logic*), conforme é mostrado na Figura 4.4. Neste formato, as instruções contém um campo de execução condicional (*condition*) ou sufixo da instrução, o campo de opcode (*opcode*), dois ou três campos de registradores (*Rn*, *Rd* e *Rm*), e o campo para algumas informações necessárias (*other info*). A Figura 4.5 mostra um resumo do conjunto de instruções do processador ARM.

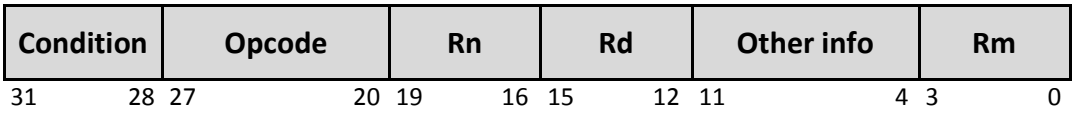


Figura 4.4 – Formato da codificação básica das instruções ARM [22, 53]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Cond				0	0	I	Opcode				S	Rn				Rd				Operand 2												**Data Processing / PSR Transfer**
Cond				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				**Multiply**
Cond				0	0	0	0	1	U	A	S	RdHi				RdLo				Rn				1	0	0	1	Rm				**Multiply Long**
Cond				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				**Single Data Swap**
Cond				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				**Branch and Exchange**
Cond				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				**Halfword Data Transfer: register offset**
Cond				0	0	0	P	U	1	W	L	Rn				Rd				Offset				1	S	H	1	Offset				**Halfword Data Transfer:**

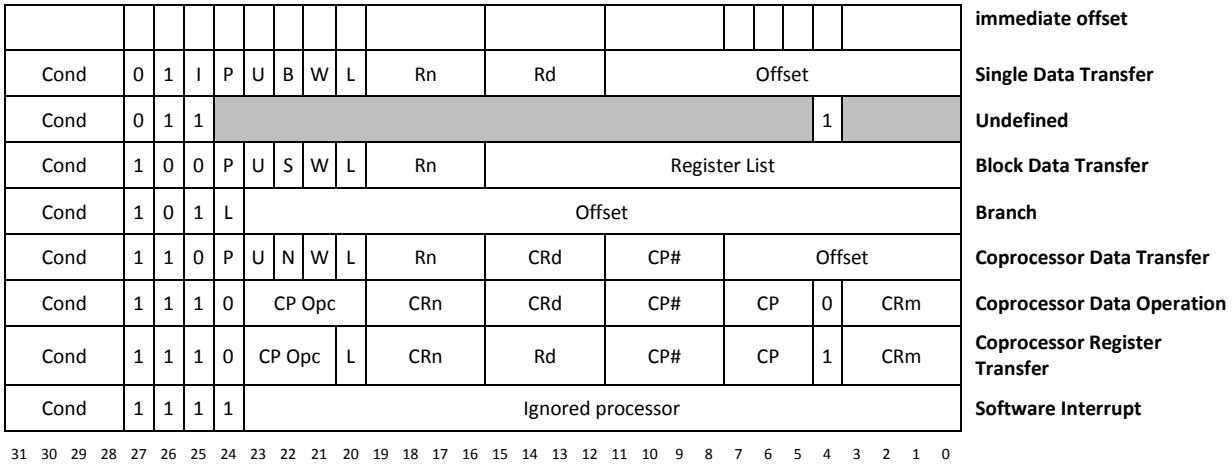


Figura 4.5 – Resumo do conjunto de instruções do processador ARM [22, 53]

4.5 Infraestrutura Utilizada

Todas as simulações desta tese foram montadas sobre uma infraestrutura de software e hardware, que envolvem mais de 8.700 linhas de código implementadas nas Linguagens C [30a] e VHDL [50a], sendo essas linhas dos métodos CPB-ARM, HDPB, CCHPB e CC-MLD que são apresentados nos Capítulos 5 e 6. Ainda foi necessário alterar o código fonte dos simuladores *SimpleScalar* e *Sim-Panalyzer* para gerar alguns relatórios da execução dos programas do *MiBench* e também para a inclusão do hardware descompressor dos métodos desenvolvidos nesta tese. A Figura 4.6 mostra os principais componentes da infraestrutura utilizada.

A partir do código fonte (.c) dos *MiBenchs*, usa-se o compilador GCC específico para cada um dos processadores embarcados apresentados na Seção 4.4, para gerar o código executável no formato ELF (*Executable and Linking Format*). Em seguida, aplica-se o arquivo executável na ferramenta Extrator de ELF que retira do executável apenas a seção do bloco destinado a compressão, ou seja, apenas a seção .text que serve como entrada para o compressor. O mesmo arquivo executável também é usado como entrada para o simulador (*SimpleScalar* ou *Sim-Panalyzer*) e quando executado são gerados relatórios (estatísticos, arquiteturais e trace de execução). Então, o arquivo executável, a seção .text e os dados dos relatórios estatísticos são usados como parâmetros de entrada para o compressor.

Após o compressor ser executado um novo arquivo com código comprimido é gerado, sendo este usado para realimentar a *SimpleScalar* e a memória da FPGA. Então, com a execução deste novo código comprimido são obtidas as medidas de desempenho do hardware descompressor implementado tanto em software quanto em hardware.

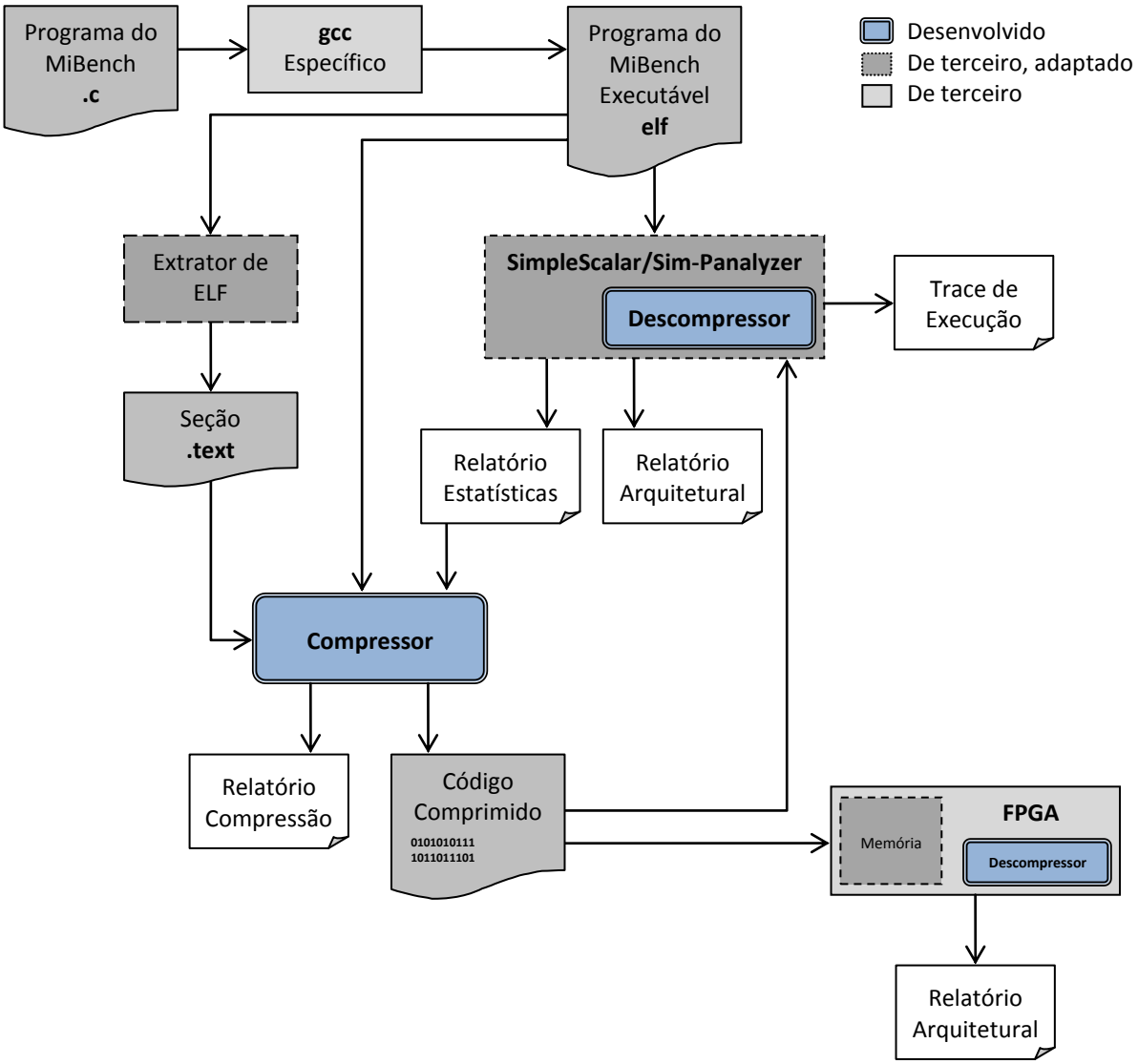


Figura 4.6 – Componentes da Infraestrutura utilizada

4.6 Considerações Finais do Capítulo

Este capítulo descreveu as diferentes técnicas de avaliação mais usadas (simulação, prototipação e modelagem), em seguida apresentou a ferramenta de simulação arquitetural *SimpleScalar* e os *benchmarks* para medição de desempenho, destacando o *MiBench*, pois seus programas estão sendo usados nessa tese, visto que são amplamente usados pela comunidade científica quando o foco é sistemas embarcados. Em seguida, explicamos as principais características dos processadores embarcados, destacando o processador ARM, pois a escolha desse processador entre os diversos processadores para sistemas embarcados (XScale, FreeScale, DSP e outros) presentes no mercado deu-se devido sua forte aceitação, pois segundo [21] o ARM representa cerca de 92% dos processadores usados em eletroeletrônicos atualmente, e também por ser uma arquitetura com alto poder de

processamento e baixo consumo de energia. Concluímos este capítulo apresentando a infraestrutura utilizada pelos novos métodos de compressão de código propostos e desenvolvidos nesta tese, que serão apresentados de forma detalhada nos dois capítulos seguintes.

CAPÍTULO 5

COMPRESSÃO BASEADA EM PADRÕES DE BLOCOS DE INSTRUÇÕES USANDO DICIONÁRIO

Este capítulo detalha o método CPB-ARM (*ComPatternBlocks-ARM*), que é uma das contribuições desta tese. Este e os demais métodos que são apresentados no Capítulo 6, surgiram após verificarmos que em sua maioria, os métodos de compressão de código existentes baseados em dicionários (já estudados nos Capítulos 2 e 3), possuem dicionários compostos apenas por instruções unitárias. Uma abordagem diferente é formar padrões de blocos para serem comprimidos. Assim, desenvolvemos este método de compressão de código que explora a presença dos padrões de blocos. No entanto, o método CPB-ARM é otimizado apenas para o conjunto de instruções do processador ARM.

O restante deste capítulo está organizado da seguinte forma, na Seção 5.1 apresentamos a redundância das instruções no código dos programas do *MiBench* com o intuito de justificarmos o uso dos padrões de blocos para serem comprimidos; logo em seguida, na Seção 5.2 realizamos a descrição e análise detalhada do método CPB-ARM, mostrando as suas características e limitações, as análises dos resultados obtidos nas simulações e a arquitetura do hardware descompressor. E por fim, uma conclusão do capítulo.

5.1 Redundância das Instruções

O pequeno conjunto de instruções das arquiteturas RISC aliado à regularidade do código gerado pelos compiladores faz com que os programas possuam certa quantidade de instruções e/ou padrões de instruções que se repetem com alguma frequência nos códigos. Seguindo essa premissa, esta seção apresenta resultados prévios das análises realizadas que

exemplificam e justificam o uso dos padrões de blocos pelos métodos de compressão de código propostos e desenvolvidos nesta tese.

A escolha dos tipos de instruções (unitária ou padrão de bloco) que compõem o dicionário usado por cada um dos métodos foi obtida após análises realizadas baseadas em informações estáticas dos códigos dos programas (*profile* estático). Por outro lado, fez-se necessário realizarmos alterações no código fonte do simulador arquitetural *SimpleScalar* (mais específico o simulador orientado à execução *Sim-outorder*) para assim obtermos tais informações. Usamos os programas do *benchmark MiBench* como entrada para as simulações e geramos relatórios com dados que justificam o uso dos padrões de blocos no processo de compressão dos códigos. A Tabela 5.1 mostra a contagem total das instruções, das instruções únicas e as repetições para cada programa do *MiBench*.

Tabela 5.1 – Número e porcentagem das instruções no código dos programas

	Total de Instruções				**Instruções Únicas (%)**				**Instruções Repetidas (%)**			
MiBench	ARM	MIPS	PowerPC	SPARC	ARM	MIPS	PowerPC	SPARC	ARM	MIPS	PowerPC	SPARC
bitcount	10.132	11.013	11.143	9.384	3.060 30,2%	3.072 27,9%	3.299 29,6%	2.895 30,9%	7.072 69,8%	7.941 72,1%	7.844 70,4%	6.489 69,1%
susan	18.036	19.794	18.680	17.212	5.456 30,3%	6.389 32,3%	6.417 34,4%	6.242 36,3%	12.580 69,7%	13.405 67,7%	12.263 65,6%	10.970 63,7%
JPEG: comprime	26.540	36.043	35.388	32.008	6.171 23,3%	9.686 26,9%	8.925 25,2%	8.082 25,3%	20.369 76,7%	26.357 73,1%	26.463 74,8%	23.926 74,7%
descomprime	31.480	38.042	36.681	33.116	7.284 23,1%	11.280 29,7%	10.225 27,9%	9.444 28,5%	24.196 76,9%	26.762 70,3%	26.456 72,1%	23.672 71,5%
lame	52.274	61.149	58.123	52.220	13.736 26,3%	15.503 25,4%	17.728 30,5%	16.281 31,2%	38.538 73,7%	45.646 74,6%	40.395 69,5%	35.939 68,8%
dijkstra: small	12.890	13.251	13.753	12.100	3.738 29%	3.346 25,3%	3.752 27,3%	3.385 28%	9.152 71%	9.905 74,7%	10.001 72,7%	8.715 72%
large	12.890	13.251	13.753	12.100	3.738 29%	3.346 25,3%	3.752 27,3%	3.385 28%	9.152 71%	9.905 74,7%	10.001 72,7%	8.715 72%
patricia	13.754	14.108	14.510	13.028	4.081 29,7%	3.581 25,4%	3.989 27,5%	3.693 28,4%	9.673 70,3%	10.527 74,6%	10.521 72,5%	9.335 71,6%
stringsearch: small	10.251	10.191	10.251	8.724	3.220 31,4%	2.986 29,3%	3.220 31,4%	2.845 32,6%	7.031 68,6%	7.205 70,7%	7.031 68,6%	5.879 67,4%
large	10.245	10.187	10.245	8.736	3.214 31,4%	2.982 29,3%	3.214 31,4%	2.857 32,7%	7.031 68,6%	7.205 70,7%	7.031 68,6%	5.879 67,3%
rijndael: encriptar	12.670	13.712	13.307	12.020	3.592 28,4%	4.287 31,3%	4.365 32,8%	3.387 28,2%	9.078 71,6%	9.425 68,7%	8.942 67,2%	8.633 71,8%
decriptar	12.670	13.712	13.307	12.020	3.592 28,4%	4.287 31,3%	4.365 32,8%	3.387 28,2%	9.078 71,6%	9.425 68,7%	8.942 67,2%	8.633 71,8%
sha	9.822	10.454	10.402	8.872	3.063 31,2%	2.949 28,2%	3.159 30,4%	2.782 31,4%	6.760 68,8%	7.505 71,8%	7.243 69,6%	6.090 68,6%
crc32:small	10.022	10.526	10.503	9.156	3.151 31,4%	2.875 27,3%	3.122 29,7%	2.887 31,5%	6.871 68,6%	7.651 72,7%	7.381 70,3%	6.269 68,5%
large	10.022	10.526	10.503	9.156	3.151 31,4%	2.875 27,3%	3.122 29,7%	2.887 31,5%	6.871 68,6%	7.651 72,7%	7.381 70,3%	6.269 68,5%
fft	24.406	13.174	13.113	11.200	0 0%	3.499 26,6%	3.976 30,3%	3.465 30,9%	24.406 100%	9.675 73,4%	9.137 69,7%	7.735 69,1%
Média	**17.381**	**18.696**	**18.354**	**16.316**	**4.390 27,1%**	**5.184 28%**	**5.414 29,9%**	**4.869 30,2%**	**12.991 72,9%**	**13.512 72%**	**12.940 70,1%**	**11.447 69,8%**

Analisando os valores da Tabela 5.1 destaca-se que em média (para as quatro arquiteturas de processador embarcado), apenas 28,8% das instruções dos códigos são únicas, o que indica a possibilidade de existir padrões de blocos no código dos programas, nos levando ao desenvolvimento de novos métodos que exploram tal característica dos programas. Assim, o impacto na taxa de compressão com a substituição de um padrão de bloco por uma instrução de tamanho menor (*codeword*) provavelmente torna-se maior do que a substituição de uma única instrução por uma *codeword* no processo da compressão de código.

Ainda vale ressaltar que os métodos de compressão que utilizam a abordagem de *profile* estático na criação do dicionário, em geral, obtêm maiores taxas de compressão, em relação aos métodos que utilizam a abordagem de *profile* dinâmico, os quais apresentam maior ganho de desempenho e/ou menor consumo de energia, conforme [45, 46].

Portanto, o que se espera de um algoritmo de compressão de código é que ele possa fazer uso destas redundâncias das instruções nos programas para diminuir o tamanho do código e, por conseguinte, aumentar o desempenho e reduzir o consumo de energia nos sistemas embarcados.

5.2 Descrição e Análise do Método CPB-ARM

O método *ComPatternBlocks-ARM* (CPB-ARM) é um algoritmo de compressão que foi desenvolvido com o objetivo principal de reduzir a quantidade de acessos à memória principal. Este método de compressão de código foi publicado em [22].

Uma nova codificação usando *codeword* foi proposta no método CPB-ARM substituindo as instruções que formam os padrões de blocos no código original. As *codewords* são codificadas de forma que elas sejam escritas no código comprimido em uma única palavra de memória, e, por conseguinte elimina acessos múltiplos à memória para a descompressão de várias instruções.

O método CPB-ARM foi otimizado para as características específicas do conjunto de instruções do processador embarcado ARM (modelo ARMv5, versão SA-1110) [1, 53], pois este processador é altamente usado no mercado atualmente [21], principalmente em eletroeletrônicos tais como: *tablets*, PDAs, *smartphones*, celulares, tocadores de música digital, consoles portáteis de jogos, calculadoras, periféricos como discos rígidos, roteadores e outros, além de ser um processador com alto poder de processamento e baixo consumo de energia.

Conforme simulações realizadas usando a *SimpleScalar*, constatou-se algumas características predominantes em determinadas classes do conjunto de instruções do processador embarcado ARM. Na Tabela 5.2 mostra-se a representatividade média da quantidade de instruções pertencente a cada uma das classes de instruções (ver Figura 4.5 do Capítulo 4) do processador ARM presente no código dos programas do *MiBench*, sendo estes valores obtidos de forma estática e dinâmica.

Tabela 5.2 – Representatividade média da quantidade de instruções estáticas e dinâmicas por cada classe de instrução do processador embarcado ARM

Classe	**Instrução Estática**		**Instrução Dinâmica**	
	Quantidade	(%)	Quantidade	(%)
Branch and Exchange	0	0%	0	0%
Branch and Branch with Link	5.003	23,9%	4.160.066	7,1%
Data Processing	6.000	38,2%	30.915.113	45,5%
PSR Transfer	2	0,01%	2	0%
Multiply and Multiply-accumulate	126	0,8%	2.525	0,2%
Multiply and Multiply-accumulate Long	0	0%	199.675	0,3%
Single Data Transfer	5.050	32%	12.876.665	22,3%
Halfword and Signed Data Transfer	21	0,1%	5.851.689	11%
Block Data Transfer	836	5,5%	7.033.283	12%
Single Data Swap	0	0%	0	0%
Software Interrupt	114	0,4%	120	0%
Coprocessor Data Operations	19	0,09%	207.426	0,3%
Coprocessor Data Transfers	19	0,09%	618.596	0,9%
Coprocessor Register Transfers	10	0,04%	306.980	0,4%
Undefined Instruction	181	1%	0	0%

Observando os valores na Tabela 5.2 pode-se concluir que aproximadamente 39% de todas as instruções estáticas e 46% das instruções dinâmicas dos programas do *MiBench* pertencem à classe *Data Processing*, pois esta classe realiza todas as operações de processamento dos dados. Já a classe *Single Data Transfer* também possui uma representatividade de 32% e 22,3% de todas as instruções estáticas e dinâmicas, respectivamente. Vale destacar que apenas estas duas classes de instruções do processador ARM, representam juntas aproximadamente 70,2% e 67,8% de todas as instruções (estáticas e dinâmicas respectivamente) dos programas do *MiBench*.

Por esse motivo, o método CPB-ARM foi implementado para realizar a compressão apenas nas instruções pertencentes a estas duas classes de instruções e em padrões de blocos de instruções idênticas repetidas sequencialmente que também são encontrados nos códigos compilados para o processador ARM em virtude da redundância das instruções (ver Tabela

5.1). No entanto, não desenvolvemos uma técnica específica para a classe *Branch and Branch with Link* (que representa, respectivamente, 23,9% e 7,1% das instruções estáticas e dinâmicas) por a mesma representar as instruções de saltos, que não são comprimidas pelo método CPB-ARM.

O método CPB-ARM aplica três técnicas distintas na tentativa de obter uma maior taxa de compressão dos programas, a saber: (i) compressão em padrão de bloco de instruções idênticas repetidas sequencialmente; (ii) compressão em padrão de bloco da classe *Data Processing* e (iii) compressão em padrão de bloco da classe *Single Data Transfer*. Essas técnicas são detalhadas a seguir.

5.2.1 Técnica-1 – Compressão em Padrão de Bloco de Instruções Idênticas Repetidas Sequencialmente (PB-IIRS)

Conforme valores apresentados na Tabela 5.1, esta técnica faz uma análise estática no código do programa em busca de blocos de instruções idênticas e que tenham uma frequência de repetição consecutiva superior a duas instruções. Quando uma sequência de instruções deste padrão de bloco é encontrada no código que está sendo submetido à compressão o mesmo é substituído por uma *codeword* formada por 32 *bits* e uma representação deste padrão de bloco é inserida no dicionário. Ainda, se ressalta que esta técnica não apresenta nenhum outro tipo de restrição quanto: às classes do ISA do processador, os sufixos das instruções ou ainda a quantidade de *bits* dos operandos da instrução, conforme apresentado na Subseção 4.4.1 do Capítulo 4.

O código que for comprimido por esta técnica usa uma *codeword* no lugar das instruções que compõem os padrões de blocos. A Figura 5.1 mostra a *codeword#1* usada nesta técnica. No entanto, com o uso desta *codeword* é possível identificar a técnica de compressão aplicada pelo método CPB-ARM, o tamanho do padrão de bloco e a sua localização dentro do dicionário.

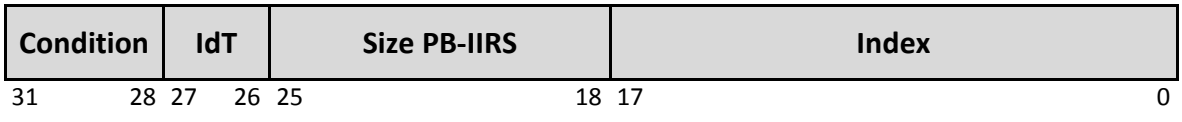


Figura 5.1 – Codeword#1 – Técnica-1 PB-IIRS

Onde,

- **Condition**: é o sufixo de execução condicional da instrução (ver Figura 4.4 do Capítulo 4), sendo que *Condition* = 1111_2 identifica uma instrução comprimida pelo método CPB-ARM;
- **IdT**: é o identificador do tipo da técnica de compressão usada pelo método CPB-ARM, sendo que para a técnica-1 o IdT = 00_2;
- **Size PB-IIRS**: é o tamanho do bloco de instruções idênticas repetidas sequencialmente, ou seja, a quantidade de instruções idênticas repetidas a serem comprimidas, podendo esta técnica comprimir um bloco formado por até 256 instruções idênticas repetidas sequencialmente;
- **Index**: é o índice da posição, ou seja, o endereço do padrão de bloco no dicionário.

O campo *Index* na *codeword#1* é formado por 18 *bits*, então, é possível indexar um dicionário com até 262.143 posições entradas; No entanto, dicionários desse tamanho não são usados em projetos de compressão de código e muito menos para sistemas embarcados. Assim, usa-se apenas uma faixa de *bits* mais alta do campo *Index* sendo os demais *bits* não usados preenchidos com zero.

A Figura 5.2 mostra um exemplo do trecho de código do programa *lame* pertencente ao *MiBench*, categoria dispositivos de consumo. Analisando a Figura 5.2 é possível verificar que as linhas 16.744 até 16.750 (instrução "`andeq r13,r2,#200704 (196 >>> 11)`") apresentam uma sequência de instruções idênticas. Então, no código comprimido pode-se substituir estas instruções pela *codeword#1*, e uma representação desta instrução é inserida no dicionário. Neste exemplo já se elimina cinco instruções repetidas no código comprimido, ou seja, uma compressão de aproximadamente 38,5%. A Figura 5.3 mostra o exemplo deste mesmo código após o processo de compressão pela técnica-1 do método CPB-ARM e a Figura 5.4 mostra os *bits* resultantes da *codeword#1* gerada para o exemplo da Figura 5.2.

```
16.741 @ 0x0202daec:   [e5963000]   ldr  r3,[r6,#0]
16.742 @ 0x0202daf0:   [e50b301c]   str  r3,[r11,-#28]
16.743 @ 0x0202daf4:   [e4d32001]   ldrb  r2,[r3],#1
16.744 @ 0x0202daf8:   [0202dbc4]   andeq  r13,r2,#200704 (196 >>> 11)
16.745 @ 0x0202dafc:   [0202dbc4]   andeq  r13,r2,#200704 (196 >>> 11)
16.746 @ 0x0202db00:   [0202dbc4]   andeq  r13,r2,#200704 (196 >>> 11)
16.747 @ 0x0202db04:   [0202dbc4]   andeq  r13,r2,#200704 (196 >>> 11)
16.748 @ 0x0202db08:   [0202dbc4]   andeq  r13,r2,#200704 (196 >>> 11)
16.749 @ 0x0202db0c:   [0202dbc4]   andeq  r13,r2,#200704 (196 >>> 11)
16.750 @ 0x0202db10:   [0202dbc4]   andeq  r13,r2,#200704 (196 >>> 11)
16.751 @ 0x0202db14:   [e24b001c]   sub  r0,r11,#28 (28 >>> 0)
16.752 @ 0x0202db18:   [e1a02005]   mov  r2,r5
16.753 @ 0x0202db1c:   [ebffff49]   bl  0x0202d8bc
```

Figura 5.2 – Exemplo de trecho do código original do programa *lame* obtido pela ferramenta

SimpleScalar

```
16.741 @ 0x0202daec:   [e5963000]   ldr  r3,[r6,#0]
16.742 @ 0x0202daf0:   [e50b301c]   str  r3,[r11,-#28]
16.743 @ 0x0202daf4:   [e4d32001]   ldrb  r2,[r3],#1
16.744 @ 0x0202daf8:   [f01c0001]   codeword#1
16.745 @ 0x0202dafc:   [e24b001c]   sub  r0,r11,#28 (28 >>> 0)
16.746 @ 0x0202db00:   [e1a02005]   mov  r2,r5
16.747 @ 0x0202db04:   [ebffff49]   bl  0x0202d8a4
```

Figura 5.3 – Exemplo de trecho do código comprimido pela técnica-1

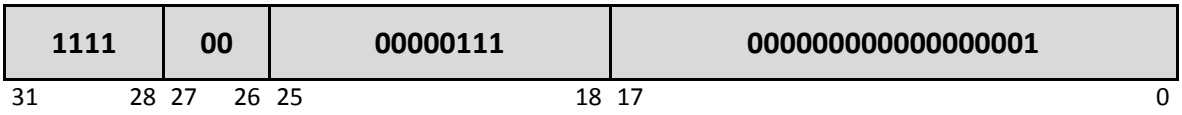


Figura 5.4 – Codeword#1 gerada para o exemplo da Figura 5.2

Portanto, destaca-se que a técnica-1 explorou a redundância das instruções encontradas no trecho do código do programa. No caso do nosso exemplo, mostramos o programa *lame* e como se conseguiu reduzir o tamanho final do código.

5.2.2 Técnica-2 – Compressão em Padrão de Bloco da Classe *Data Processing* (PB-CDP)

Esta técnica encontra os padrões de blocos analisando os *opcodes* de cada uma das instruções que pertencem a esta classe. Esses *opcodes* podem ser iguais (divergindo apenas o formato das instruções ou os seus registradores) ou diferentes (no formato das instruções e seus registradores). Nesta técnica o método CPB-ARM foi otimizado para a classe das instruções *Data Processing* com o sufixo *always*[1]. É importante lembrar que esta classe realiza as operações de processamento dos dados, tais como: operações lógicas, adições, subtrações, movimentações de dados, comparações dos registradores (*Rn*, *Rd* e *Operand2*) entre outras.

Conforme apresentado na Tabela 5.2, a classe *Data Processing* obteve as maiores médias na quantidade de instruções presentes nos programas, sendo 38,2% estáticas e 45,5% dinâmicas, no total geral da média de todas as instruções dos programas do *MiBench*. Isso justifica a importância do uso dessa classe no desenvolvimento de uma técnica específica para o método CPB-ARM. A Tabela 5.3 mostra as instruções pertencentes à classe *Data*

[1] As instruções que possuem o sufixo *always* significam que a instrução sempre será executada independentemente dos valores de codificação do registrador CPSR (conforme Figura 4.3 do Capítulo 4) do processador ARM.

Processing do processador ARM, bem como suas operações e na Figura 5.5 o formato básico destas instruções.

Tabela 5.3 – Instruções da classe *Data Processing* em arquiteturas ARM [1]

Mnemônico	Opcode	Operação
and	0000_2	Rd := Rn and Op2
eor	0001_2	Rd := (Rn and not Op2) or (Op2 and not Rn)
sub	0010_2	Rd := Rn - Op2
rsb	0011_2	Rd := Op2 – Rn
add	0100_2	Rd := Rn + Op2
adc	0101_2	Rd := Rn + Op2 + Carry
sbc	0110_2	Rd := Rn - Op2 - 1 + Carry
rsc	0111_2	Rd := Op2 - Rn - 1 + Carry
tst	1000_2	CPSR *flags* := Rn and Op2
teq	1001_2	CPSR *flags* := Rn eor Op2
cmp	1010_2	CPSR *flags* := Rn - Op2
cmn	1011_2	CPSR *flags* := Rn + Op2
orr	1100_2	Rd := Rn or Op2
mov	1101_2	Rd : = Op2
bic	1110_2	Rd := Rn and not Op2
mvn	1111_2	Rd := 0xFFFFFFFF eor Op2

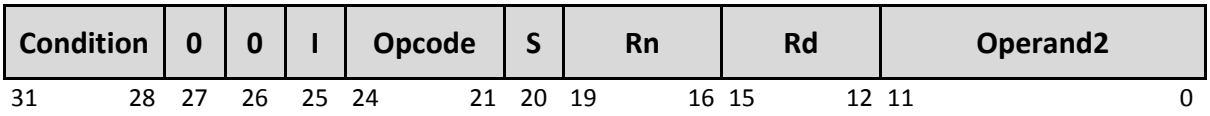


Figura 5.5 – Formato básico das instruções da classe *Data Processing* [1]

Onde,

- **Condition**: são os *bits* de verificação condicional, se a condição for verdadeira a instrução é executada;
- **00**: são *bits default* da classe *Data Processing*;
- **I**: é o *bit* que identifica se na instrução o *Operand2* possui ou não um valor de imediato;
- **Opcode**: é o *opcode* da instrução (ver Tabela 5.3);
- **S**: é um *bit* de controle interno da classe *Data Processing*;
- **Rn**: é o registrador do operando da instrução;
- **Rd**: é o registrador de destino da instrução;
- **Operand2**: é um registrador da instrução.

Para comprimir as instruções da classe *Data Processing* com o sufixo *always* usando o método CPB-ARM, primeiramente foi necessário identificar os formatos das instruções pertencentes a esta classe e assim atribuir um identificador binário diferente a cada um desses formatos, conforme mostrado na Tabela 5.4.

Tabela 5.4 – Formatos das instruções da classe *Data Processing* [1]

Identificador do Formato	Formato da Instrução	Exemplo de Instrução
000_2	opcode reg1, reg2	mov r3, r1
001_2	opcode reg1, reg2, reg3	and r12, r14, r7
010_2	opcode reg1, #v1 (v2 >>> v3)	tst r3, #192 (2 >>> 10)
011_2	opcode reg1, reg2, #v1 (v2 >>> v3)	bic r3, r3, #256 (1 >>> 12)
100_2	opcode reg1, reg2, ShiftType reg3	cmp r8, r2, lsl r5
101_2	opcode reg1, reg2, reg3, ShiftType reg4	orr r3, r3, r4, lsr r6
110_2	opcode reg1, reg2, ShiftType #v1	mov r1, r14, ror #27
111_2	opcode reg1, reg2, reg3, ShiftType #v1	add r8, r4, r3, asr #35

Observando a Tabela 5.4, constata-se que existem oito tipos de formatos diferentes para as instruções da classe *Data Processing*. Porém, as instruções com *mnemônico* **mov**, **mvn**, **cmp**, **cmn**, **teq** e **tst** (ver Tabela 5.3) só podem assumir os formatos das instruções caso o identificador binário seja 000_2, 010_2, 100_2 e 110_2. Ainda assim, o valor atribuído ao registrador **reg1** desses formatos pode variar de acordo com o *mnemônico* da instrução, sendo que, para as instruções que fazem a movimentação de dados (mov e mvn – instruções de operando único) o valor de **reg1** receberá o valor do registrador *Rd* (conforme Figura 5.5), e para as instruções que fazem apenas as comparações e testes nos valores dos registradores (cmp, cmn, teq e tst – instruções que não produzem um resultado) o valor de **reg1** receberá o valor do registrador *Rn* (ver Figura 5.5). A Tabela 5.5 apresenta detalhadamente os formatos das instruções que possuem os *mnemônicos* citados acima.

Tabela 5.5 – Formatos das instruções da classe *Data Processing* com *mnemônico* mov, mvn, cmp, cmn, teq e tst [1]

Identificador do Formato	Formato da Instrução	Sintaxe do Formato	Tamanho dos Registradores
000_2	opcode reg1, reg2	cond opcode <Rd|Rn>, Operand2	8 *bits*
010_2	opcode reg1, #v1 (v2 >>> v3)	cond opcode <Rd|Rn>, #Operand2 (Operand2 [7:0] >>> Operand2 [11:8])	16 *bits*
100_2	opcode reg1, reg2, ShiftType reg3	cond opcode <Rd|Rn>, Operand2 [3:0], ShiftType [6:5] Operand2 [11:8]	16 *bits*
110_2	opcode reg1, reg2, ShiftType #v1	cond opcode <Rd|Rn>, Operand2 [3:0],	16 *bits*

		ShiftType [6:5] #Operand2 [11:7]	

Já para as instruções que possuem os *mnemônicos* **and**, **eor**, **sub**, **rsb**, **add**, **adc**, **sbc**, **rsc**, **orr** e **bic** (ver Tabela 5.3) só podem assumir os outros quatro formatos das instruções, ou seja, os formatos com identificador binário 001_2, 011_2, 101_2 e 111_2. Assim, a Tabela 5.6 apresenta em detalhes os formatos das instruções com esses *mnemônicos*.

Tabela 5.6 – Formatos das instruções da classe *Data Processing* com *mnemônico* and, eor, sub, rsb, add, adc, sbc, rsc, orr e bic [1]

Identificador do Formato	Formato da Instrução	Sintaxe do Formato	Tamanho dos Registradores
001_2	opcode reg1, reg2, reg3	cond opcode Rd, Rn, Operand2	12 *bits*
011_2	opcode reg1, reg2, #v1 (v2 >>> v3)	cond opcode Rd, Rn, #Operand2 (Operand2 [7:0] >>> Operand2 [11:8])	20 *bits*
101_2	opcode reg1, reg2, reg3, ShiftType reg4	cond opcode Rd, Rn, Operand2 [3:0], ShiftType [6:5] Operand2 [11:8]	20 *bits*
111_2	opcode reg1, reg2, reg3, ShiftType #v1	cond opcode Rd, Rn, Operand2 [3:0], ShiftType [6:5] #Operand2 [11:7]	20 *bits*

Devido o processador ARM ser uma arquitetura RISC de 32 *bits*, onde todas as instruções são compostas por *bits* de: verificação condicional, *opcode*, registradores e *flags* de informações conforme mostrado na Figura 4.4 do Capítulo 4, não há nenhum modo de carregar os valores grandes (em quantidade de *bits*) de imediatos no registrador em uma instrução. Este problema pode ser resolvido com o uso do *ShiftType*. Então, quando o registrador *Operand2* da instrução é especificado para ser deslocado de um determinado registrador a outro, a operação de deslocamento é controlada pelo *ShiftType* (para os formatos das instruções da Tabela 5.4 que possuem os identificadores binários iguais a 100_2, 101_2, 110_2 e 111_2), ou seja, este campo da instrução indica o tipo de deslocamento ou rotacionamento a ser realizado, conforme é mostrado na Tabela 5.7.

Tabela 5.7 – Tipo de deslocamento ou rotacionamento [1]

Identificador do ShiftType	Tipo de ShiftType	
00_2	lsl (*logical shift left*)	deslocamento lógico para à esquerda
01_2	lsr (*logical shift right*)	deslocamento lógico para à direita
10_2	asr (*arithmetic shift right*)	deslocamento aritmético para à direita
11_2	ror (*rotate right*)	rotacionamento para à direita

No método CPB-ARM a técnica-2 ainda pode formar blocos contendo instruções com os *opcodes* diferentes ou iguais, sendo chamados de "bloco misto" e "bloco igual", respectivamente. A Tabela 5.8 exemplifica a formação desses padrões de blocos. Logo a seguir temos a descrição desses tipos de blocos:

- **Bloco Misto** (valor 0_2 para a *flag* **mi** das *codewords*): os blocos mistos podem conter instruções com *opcodes* e/ou formato das instruções diferentes (ver Tabela 5.4);
- **Bloco Igual** (valor 1_2 para *flag* **mi** das *codewords*): esses blocos possuem apenas instruções com o mesmo *opcode*, podendo o formato das instruções ser diferentes.

Tabela 5.8 – Formação de padrões de blocos misto ou igual

Flag mi da Codeword	Instruções
mi = 0_2	mov r5, r3
	sub r8, r4
	add r12, r2
	tst r1, r7
mi = 1_2	and r3, r5
	and r2, #32 (5 >>> 0)
	and r1, r6

Portanto, os padrões de blocos propriamente ditos para a técnica-2 são as possíveis combinações dos formatos das instruções, desde que o tamanho de seus registradores somados não ultrapasse a 32 *bits* (ver Tabelas 5.5 e 5.6). Assim, o tamanho dos padrões de blocos da classe *Data Processing* formados para a compressão, sempre terão 24, 28 ou 32 *bits*. No entanto, esses padrões de blocos conterão os valores dos registradores (*Rn*, *Rd* e *Operand2*) das instruções comprimidas, e os mesmos serão inseridos no dicionário do método CPB-ARM.

Já no código comprimido será usada uma *codeword* no lugar das instruções pertencentes ao padrão de bloco, onde será possível identificar o tamanho do bloco, os formatos das instruções, os *opcodes*, a técnica de compressão usada e a localização do padrão de bloco no dicionário.

Então, para que a compressão do código usando a técnica-2 apresente ganhos na taxa de compressão, é necessário que os padrões de blocos formados sejam sempre compostos por três ou quatro instruções. Sendo que, para os padrões formados por três instruções o ganho é de aproximadamente 33%, já para os padrões com quatro instruções este valor aumenta para 50%. Estes ganhos são justificados pelas trocas de três ou quatro instruções de 32 *bits* cada

por duas novas instruções de 32 *bits*, sendo que uma delas é inserida no código comprimido (como uma *codeword*) e a outra no dicionário (como um padrão de bloco). Conclui-se então, que para a formação de padrões de blocos eficientes para a técnica-2, não é vantajoso o uso de apenas duas instruções, pois a taxa de compressão não apresentará nenhum ganho e sim uma redução no desempenho do sistema devido às operações do processo de descompressão dos blocos.

Ainda vale ressaltar que para os padrões de blocos formados com cinco ou mais instruções será necessário mais de uma linha de posicionamento no dicionário, ou seja, mais de uma busca no dicionário, e ainda assim a *codeword* terá mais de 32 *bits* causando o desalinhamento das instruções no código dos programas. A Tabela 5.9 mostra as possíveis combinações nos tamanhos (em *bits*) dos formatos das instruções para a criação dos padrões de blocos da técnica-2.

Tabela 5.9 – Padrões de blocos da técnica-2

Identificador do Padrão de Bloco (IdPB)	Identificador do Formato	Soma do Tamanho dos Registradores = Tamanho do Padrão de Bloco
0000_2	$000_2 + 000_2 + 000_2$	8 + 8 + 8 = 24 *bits*
0001_2	$000_2 + 000_2 + 001_2$	8 + 8 + 12 = 28 *bits*
0010_2	$000_2 + 001_2 + 000_2$	8 + 12 + 8 = 28 *bits*
0011_2	$001_2 + 000_2 + 000_2$	12 + 8 + 8 = 28 *bits*
0100_2	$000_2 + 000_2 + 010_2$	8 + 8 + 16 = 32 *bits*
0101_2	$000_2 + 000_2 + 100_2$	8 + 8 + 16 = 32 *bits*
0110_2	$000_2 + 000_2 + 110_2$	8 + 8 + 16 = 32 *bits*
0111_2	$000_2 + 010_2 + 000_2$	8 + 16 + 8 = 32 *bits*
1000_2	$000_2 + 100_2 + 000_2$	8 + 16 + 8 = 32 *bits*
1001_2	$000_2 + 110_2 + 000_2$	8 + 16 + 8 = 32 *bits*
1010_2	$010_2 + 000_2 + 000_2$	16 + 8 + 8 = 32 *bits*
1011_2	$100_2 + 000_2 + 000_2$	16 + 8 + 8 = 32 *bits*
1100_2	$110_2 + 000_2 + 000_2$	16 + 8 + 8 = 32 *bits*
1101_2	$000_2 + 001_2 + 001_2$	8 + 12 + 12 = 32 *bits*
1110_2	$001_2 + 000_2 + 001_2$	12 + 8 + 12 = 32 *bits*
1111_2	$001_2 + 001_2 + 000_2$	12 + 12 + 8 = 32 *bits*

Observa-se que na Tabela 5.9 não foram usados os formatos das instruções que possuem o tamanho dos registradores igual a 20 *bits*, conforme apresentado na Tabela 5.6. Esta é uma restrição imposta por essa técnica-2. Pois, devido às permutações dos formatos

com esse tamanho juntamente com os demais formatos de tamanhos (dos registradores) diferentes nunca obter um padrão de bloco que seja formado com três ou quatro instruções e com tamanho dos registradores de no máximo 32 *bits*. Assim, deste ponto em diante desconsidera-se na formação dos padrões de blocos para a técnica-2, o uso dos formatos das instruções que possuem o valor binário do identificador igual a 011_2, 101_2 e 111_2 (conforme mostrado na Tabela 5.6).

Então, duas novas *codewords* foram criadas para essa técnica, sendo que a *codeword#2* é usada no lugar dos padrões de blocos formados por três instruções e a *codeword#3* no lugar dos padrões formados por quatro instruções. As Figuras 5.6 e 5.7 apresentam as *codewords#2* e *#3*, respectivamente. Sendo que, para a *codeword#3* não é preciso usar o identificador do padrão de bloco (IdPB), pois todos os tamanhos dos registradores do formato das instruções possuem 8 *bits* cada.

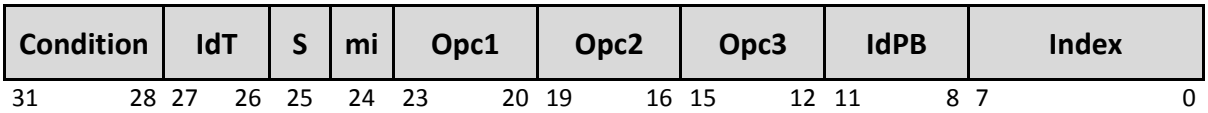


Figura 5.6 – Codeword#2 – Técnica-2 PB-CDP

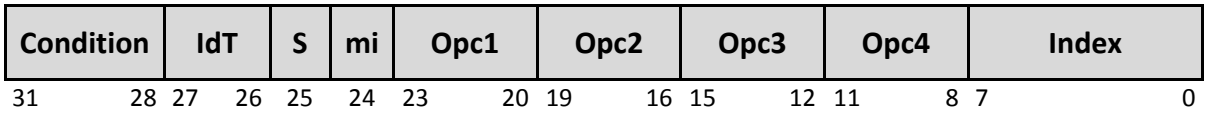


Figura 5.7 – Codeword#3 – Técnica-2 PB-CDP

Onde,

- **Condition**: é o sufixo de execução condicional da instrução, sendo que *Condition* = 1111_2 identifica uma instrução comprimida pelo método CPB-ARM;
- **IdT**: é o identificador do tipo da técnica de compressão usada pelo método CPB-ARM, sendo que para a técnica-2 o IdT = 01_2;
- **S**: identifica se o padrão de bloco é formado por três (S = 0_2) ou quatro (S = 1_2) instruções;
- **mi**: é o identificador de bloco misto (mi = 0_2) ou bloco igual (mi = 1_2);
- **Opc1**, **Opc2**, **Opc3** e **Opc4**: são os *opcodes* das instruções do bloco (ver Tabela 5.3);
- **IdPB** (*Index Block Pattern*): é o identificador do padrão de bloco, usado apenas pela *codeword#2*;
- **Index**: é o endereço do padrão de bloco no dicionário, podendo a técnica-2 indexar um dicionário com até 256 posições de entradas.

A Figura 5.8 mostra um exemplo do trecho de código original do programa *susan*, onde pode-se constatar que nas linhas 12.854 até 12.856 e 12.864 até 12.867 são identificados dois padrões de blocos diferentes, nos quais todas as instruções pertencem à classe *Data Processing* do processador ARM e que é possível comprimi-las usando a técnica-2 do método CPB-ARM. Já a Figura 5.9 mostra o exemplo deste mesmo código após a compressão e as Figuras 5.10 e 5.11 mostram os *bits* das *codewords* e os registradores dos padrões de blocos (que é armazenado no dicionário) gerados para o exemplo da Figura 5.8.

```
12.850 @ 0x02000e38:  [e51b307c]  ldr  r3,[r11,-#124]
12.851 @ 0x02000e3c:  [e793c002]  ldr  r12,[r3,r2]
12.852 @ 0x02000e40:  [e24b3044]  sub  r3,r11,#68 (68 >>> 0)
12.853 @ 0x02000e44:  [e7932002]  ldr  r2,[r3,r2]
12.854 @ 0x02000e48:  [e1a0000e]  mov  r0,r14
12.855 @ 0x02000e4c:  [e081100c]  add  r1,r1,r12
12.856 @ 0x02000e50:  [e06c2002]  rsb  r2,r12,r2
12.857 @ 0x02000e54:  [0202dbc4]  andeq  r13,r2,#200704 (196 >>> 11)
12.858 @ 0x02000e58:  [ebffd7cb]  bl  0x020074fc
12.859 @ 0x02000e5c:  [e59f1030]  ldr  r1,[r15,#48]
12.860 @ 0x02000e60:  [e1a0e000]  mov  r14,r0
12.861 @ 0x02000e64:  [e24b001c]  sub  r0,r11,#28 (28 >>> 0)
12.862 @ 0x02000e68:  [e5d73000]  ldrb  r3,[r7,#0]
12.863 @ 0x02000e6c:  [ebffff49]  bl  0x0202d8bc
12.864 @ 0x02000e70:  [e1a02004]  mov  r2,r4
12.865 @ 0x02000e74:  [e1a0e00f]  mov  r14,r15
12.866 @ 0x02000e78:  [e1a0f003]  mov  r15,r3
12.867 @ 0x02000e7c:  [e1a00004]  mov  r0,r4
12.868 @ 0x02000e80:  [eb003ab9]  bl  0x0200f990
12.869 @ 0x02000e84:  [e24b0028]  sub  r0,r11,#40 (40 >>> 0)
12.870 @ 0x02000e88:  [e3530000]  cmp  r3,#0 (0 >>> 0)
```

Figura 5.8 – Exemplo de trecho do código original do programa *susan* obtido pela ferramenta *SimpleScalar*

```
12.850 @ 0x02000e38:  [e51b307c]  ldr  r3,[r11,-#124]
12.851 @ 0x02000e3c:  [e793c002]  ldr  r12,[r3,r2]
12.852 @ 0x02000e40:  [e24b3044]  sub  r3,r11,#68 (68 >>> 0)
12.853 @ 0x02000e44:  [e7932002]  ldr  r2,[r3,r2]
12.854 @ 0x02000e48:  [f4d43d02]  codeword#2
12.855 @ 0x02000e4c:  [0202dbc4]  andeq  r13,r2,#200704 (196 >>> 11)
12.856 @ 0x02000e50:  [ebffd7cb]  bl  0x020074f4
12.857 @ 0x02000e54:  [e59f1030]  ldr  r1,[r15,#48]
12.858 @ 0x02000e58:  [e1a0e000]  mov  r14,r0
12.859 @ 0x02000e5c:  [e24b001c]  sub  r0,r11,#28 (28 >>> 0)
12.860 @ 0x02000e60:  [e5d73000]  ldrb  r3,[r7,#0]
12.861 @ 0x02000e64:  [ebffff49]  bl  0x0202d8b4
12.862 @ 0x02000e68:  [f7dddd03]  codeword#3
12.863 @ 0x02000e6c:  [eb003ab9]  bl  0x0200f97c
12.864 @ 0x02000e70:  [e24b0028]  sub  r0,r11,#40 (40 >>> 0)
12.865 @ 0x02000e74:  [e3530000]  cmp  r3,#0 (0 >>> 0)
```

Figura 5.9 – Exemplo de trecho do código comprimido pela técnica-2

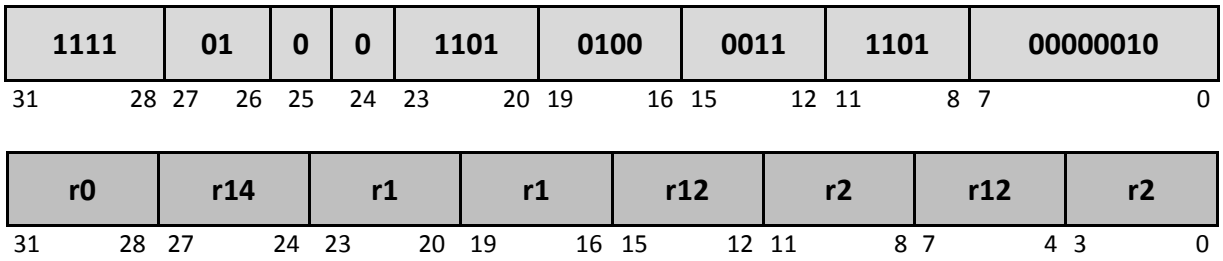


Figura 5.10 – Codeword#2 e padrão de bloco gerado para o exemplo da Figura 5.8

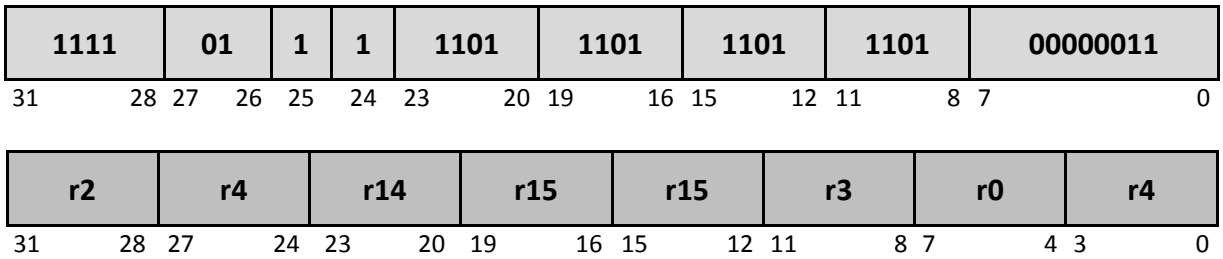


Figura 5.11 – Codeword#3 e padrão de bloco gerado para o exemplo da Figura 5.8

5.2.3 Técnica-3 – Compressão em Padrão de Bloco da Classe *Single Data Transfer* (PB-CSDT)

Assim como as outras duas técnicas do método CPB-ARM apresentadas anteriormente, esta técnica também tenta localizar padrões de blocos para realizar a compressão. Porém, conforme resultados apresentados na Tabela 5.2 esta classe possui 32% de todas as instruções dos programas do *MiBench* obtidas de forma estática e mais de 22% de forma dinâmica. Por isso desenvolvemos uma técnica de compressão específica para esta classe de instruções, que por sinal é composta apenas pelas instruções **ldr** (*load*) e **str** (*store*). A instrução "*ldr*" realiza operações de carregar *bytes* ou palavras de dados da memória para os registradores; Já a instrução "*str*" realiza operações inversas, ou seja, armazena *bytes* ou palavras de dados na memória. A Figura 5.12 mostra o formato básico destas instruções.

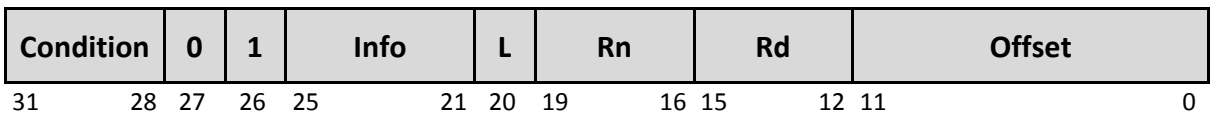


Figura 5.12 – Formato básico das instruções da classe *Single Data Transfer* [1]

Onde,

- **Condition**: são os *bits* de verificação condicional, se a condição for verdadeira a instrução é executada;

- **01**: são *bits default* da classe *Single Data Transfer*;
- **Info**: são *bits* de controle interno da instrução;
- **L**: é o *opcode* da instrução, sendo L = 0_2 para instrução *str* e L = 1_2 para instrução *ldr*;
- **Rn**: é o registrador do operando da instrução;
- **Rd**: é o registrador de destino da instrução;
- **Offset**: é o índice de deslocamento da instrução.

Para comprimir as instruções usando a técnica-3, também foi necessário identificar os formatos das instruções desta classe, conforme mostrado na Tabela 5.10.

Tabela 5.10 – Formatos das instruções da classe *Single Data Transfer* [1]

Identificador do Formato	Formato da Instrução	Tamanho	Exemplo de Instrução
000_2	opcode reg1, [reg2, #v1]	12 *bits*	ldr r9, [r15, #48]
001_2	opcode reg1, [reg2, #v1]!	13 *bits*	ldr r1, [r7, #254]!
010_2	opcode reg1, [reg2, reg3]	12 *bits*	ldr r2, [r3, r2]
011_2	opcode reg1, [reg2, reg3, ShiftType #v1]	18 *bits*	ldr r3, [r14, r12, lsl #2]
100_2	opcode reg1, [reg2], #v1	12 *bits*	ldr r3, [r2], #2

Assim como mostrado na Subseção 5.2.2, o campo *ShiftType* do formato da instrução com identificador 011_2 apresenta a mesma funcionalidade descrita na Tabela 5.7.

Conforme simulações realizadas com a *SimpleScalar* usando os programas do *MiBench*, constatou-se que os formatos das instruções com identificadores 000_2 e 010_2 apresentados na Tabela 5.10 representam juntos cerca de 98% das instruções pertencentes a esta classe. Portanto, restringiu-se esta técnica para realizar a compressão apenas nos padrões de blocos compostos por instruções destes dois formatos.

A *codeword* desenvolvida para esta técnica possui as mesmas características das demais *codewords* já apresentadas nas Subseções 5.2.1 e 5.2.2. A Figura 5.13 mostra a *codeword#4*, vale destacar que se o padrão de bloco formado para a compressão tiver três instruções, então as *flags* **S** e **Opc4** da *codeword#4* receberão 0_2 e os *bits* 9 até 11 da *flag* **IdP** receberá 000_2 sendo assim, o mesmo não será levado em consideração no momento da descompressão.

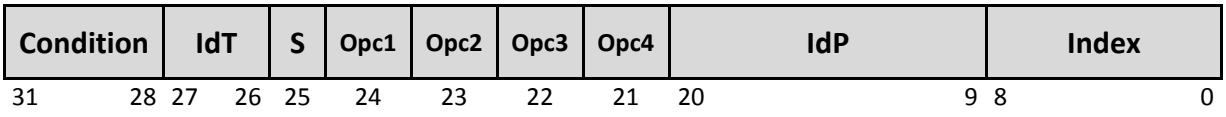


Figura 5.13 – Codeword#4 – Técnica-3 PB-CSDT

Onde,

- **Condition**: é o sufixo de execução condicional da instrução, sendo que *Condition* = 1111_2 identifica uma instrução comprimida pelo método CPB-ARM;
- **IdT**: é o identificador do tipo da técnica de compressão usada pelo método CPB-ARM, sendo que para a técnica-3 o IdT = 10_2;
- **S**: identifica se o padrão de bloco é formado por três (S = 0_2) ou quatro (S = 1_2) instruções;
- **Opc1**, **Opc2**, **Opc3** e **Opc4**: são os *opcodes* das instruções do bloco, sendo 0_2 para instrução *str* e 1_2 para instrução *ldr*;
- **IdP** (*IdPattern*): é o identificador do padrão de bloco usado pela *codeword#4*;
- **Index**: é o endereço do padrão de bloco no dicionário, podendo a técnica-3 indexar um dicionário com até 512 posições de entradas.

A Figura 5.14 mostra um exemplo do trecho de código não comprimido do programa *sha*. Observando a figura constata-se que nas linhas 3.530 até 3.533 é formado um padrão de bloco que é possível ser comprimido pela técnica-3 do método CPB-ARM. Já na Figura 5.15 mostra-se o mesmo exemplo após a compressão e na Figura 5.16 são mostrados os *bits* gerados pela *codeword#4* bem como os registradores do padrão de bloco para o exemplo da Figura 5.14.

```
3.524 @ 0x02011818:   [e51b505c]   ldr  r5,[r11,-#92]
3.525 @ 0x0201181c:   [e3550000]   cmp  r5,#0 (0 >>> 0)
3.526 @ 0x02011820:   [0a000019]   beq  0x201188c
3.527 @ 0x02011824:   [e5952010]   ldr  r2,[r5,#16]
3.528 @ 0x02011828:   [e3520000]   cmp  r2,#0 (0 >>> 0)
3.529 @ 0x0201182c:   [0202dbc4]   andeq  r13,r2,#200704 (196 >>> 11)
3.530 @ 0x02011830:   [e5923034]   ldr  r3,[r2,#52]
3.531 @ 0x02011834:   [e585300c]   str  r3,[r5,#12]
3.532 @ 0x02011838:   [e5923038]   ldr  r3,[r2,#56]
3.533 @ 0x0201183c:   [e5943034]   ldr  r3,[r4,#52]
3.534 @ 0x02011840:   [e3550000]   cmp  r5,#0 (0 >>> 0)
3.535 @ 0x02011844:   [e0811003]   add  r1,r1,r3
3.536 @ 0x02011848:   [e5d73000]   ldrb  r3,[r7,#0]
```

Figura 5.14 – Exemplo de trecho do código original do programa *sha* obtido pela ferramenta *SimpleScalar*

```
3.524 @ 0x02011818:   [e51b505c]   ldr  r5,[r11,-#92]
3.525 @ 0x0201181c:   [e3550000]   cmp  r5,#0 (0 >>> 0)
3.526 @ 0x02011820:   [0a000019]   beq  0x201188c
3.527 @ 0x02011824:   [e5952010]   ldr  r2,[r5,#16]
3.528 @ 0x02011828:   [e3520000]   cmp  r2,#0 (0 >>> 0)
3.529 @ 0x0201182c:   [0202dbc4]   andeq  r13,r2,#200704 (196 >>> 11)
3.530 @ 0x02011830:   [fb619407]   codeword#4
3.531 @ 0x02011834:   [e3550000]   cmp  r5,#0 (0 >>> 0)
3.532 @ 0x02011838:   [e0811003]   add  r1,r1,r3
```

Figura 5.15 – Exemplo de trecho do código comprimido pela técnica-3

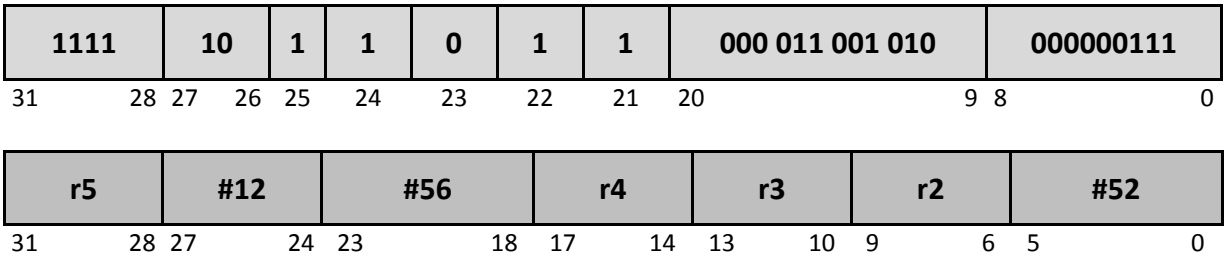


Figura 5.16 – Codeword#4 e padrão de bloco gerado para o exemplo da Figura 5.14

E por fim, a Figura 5.17 mostra um pseudo-código do algoritmo de compressão do método CPB-ARM.

```
CódigoComprimido COMPRESS (CódigoOriginal)
{
  1 Dicionário ← {Conjunto Vazio}
  2 Buffer ← {Conjunto Vazio}
  3 Enquanto não for o fim do CódigoOriginal ou o Dicionário não estiver
completo
  4 Lê uma instrução do CódigoOriginal
    - Insere a instrução no Buffer
    4.1 Se a instrução lida for da classe Data Processing e sufixo always
        - Tenta aplicar a codeword#3
        - Tenta aplicar a codeword#2
    4.2 Senão, se a instrução lida for da classe Single Data Transfer
        - Tenta aplicar a codeword#4
    4.3 Senão, se a instrução lida for igual à 1ª instrução do Buffer
        - Incrementa o contador do Buffer
  5 Se uma das codewords for aplicada, então
    - Insere a codeword no CódigoComprimido
    - Insere o padrão de bloco no dicionário
    - Salta a leitura no CódigoOriginal para a posição abaixo do padrão de
bloco formado e aplicado
    - Volta ao passo 3
  6 Senão, se o contador do Buffer for > 2, e a instrução lida for
diferente da 1ª instrução do Buffer, então
    - Insere a codeword#1 no CódigoComprimido
    - Insere a 1ª instrução do Buffer no dicionário
    - Salta a leitura no CódigoOriginal o tamanho do contador do Buffer
    - Zera o contador do Buffer
    - Limpa o Buffer
    - Volta ao passo 3
  7 Fim do Enquanto
  8 Insere o Dicionário no CódigoComprimido
  9 Atualiza os endereços das Instruções de Saltos (Branchs)
  10 Retorna o CódigoComprimido
```

```
}
```

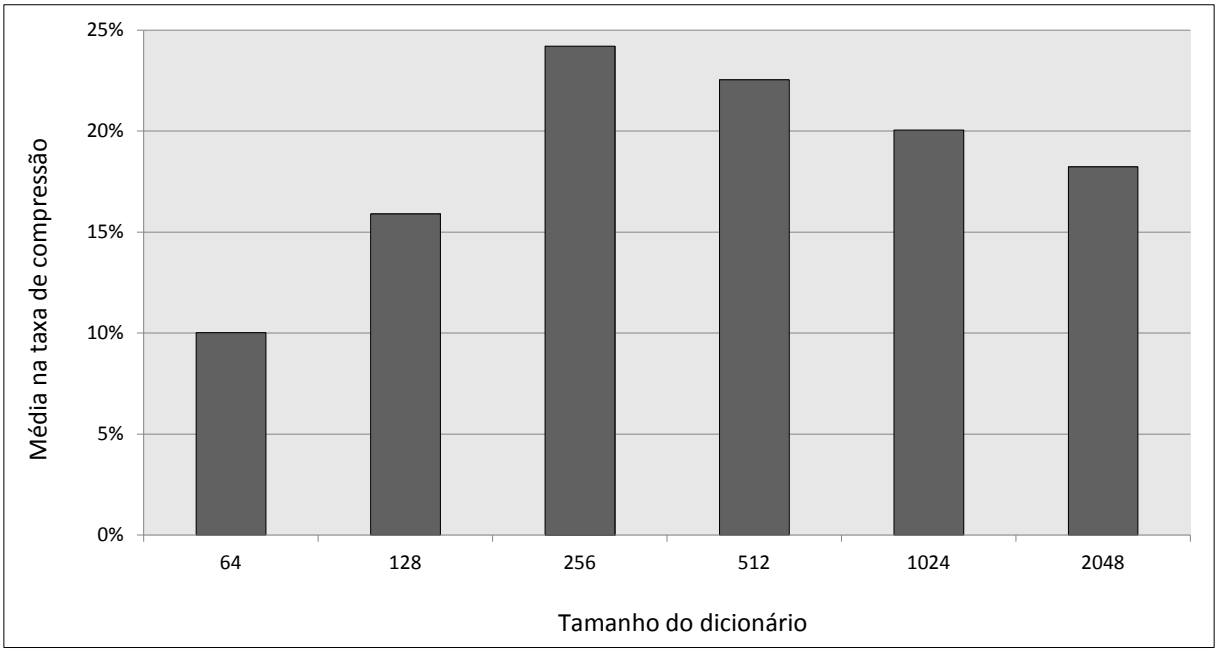
Figura 5.17 – Pseudo-código do algoritmo de compressão do método CPB-ARM

5.3 Simulações e Análises do Método CPB-ARM

O método CPB-ARM foi implementado para ser usado em arquitetura CDM (ver Seção 2.4 do Capítulo 2), pois segundo NETTO [44] este tipo de arquitetura normalmente obtém maiores taxas de compressão e não necessita de intrusões no núcleo do processador. Conforme já apresentado anteriormente usou-se o processador embarcado ARM (modelo ARMv5, versão SA-1110) [1, 53] para a otimização do método CPB-ARM. As simulações foram realizadas com os programas das seis categorias do *MiBench* (ver Seção 4.3 do Capítulo 4) para a análise e validação desse método.

Primeiramente, foram definidos tamanhos diferentes para o dicionário, variando de 64 até 2.048 posições de entradas, assim conforme mostrado no Gráfico 5.1 verifica-se que o dicionário com tamanho 256 obteve a maior média na taxa de compressão dos programas do *MiBench* pelo método CPB-ARM. Este tamanho de dicionário é comumente usado por outros métodos apresentados nos Capítulos 2 e 3 e também foi adotado para este método.

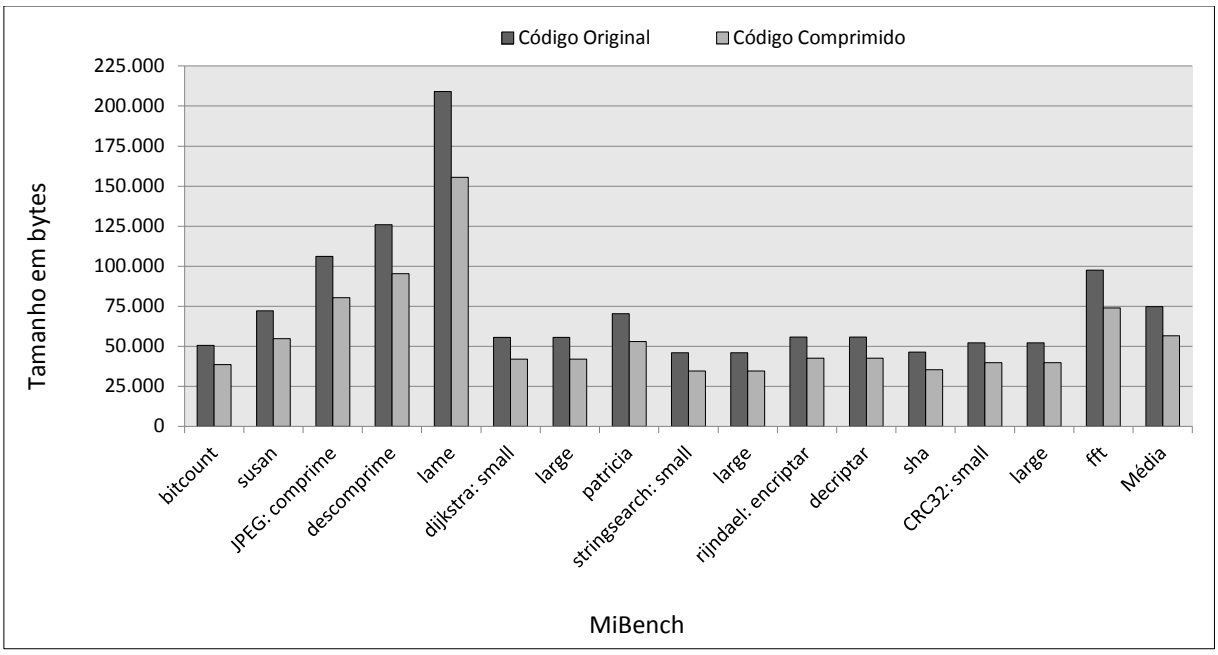
Gráfico 5.1 – Média na taxa de compressão obtida pelo método CPB-ARM



O Gráfico 5.2 mostra o tamanho dos programas (em *bytes*) antes e depois da compressão; uma análise mais detalhada pode ser vista no Gráfico 5.3: a taxa de compressão

obtida por cada um dos programas do *MiBench* e também a representatividade (em %) de cada uma das três técnicas do método CPB-ARM na taxa de compressão.

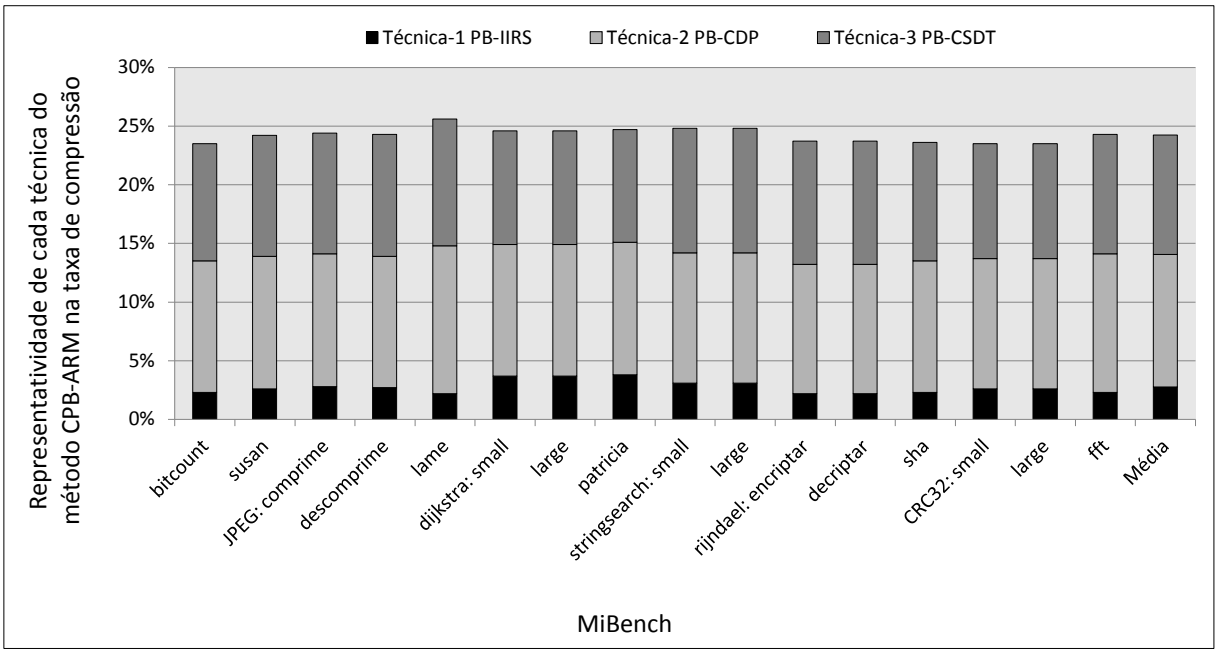
Gráfico 5.2 – Tamanho dos programas do *MiBench* em *bytes*



Observando o Gráfico 5.2, constata-se que em todos os programas simulados houve uma redução (em *bytes*) expressiva em seu tamanho quando aplicada a compressão pelo método CPB-ARM, em média 24,2%. Comprovando esta afirmação observa-se no Gráfico 5.3 que o programa *lame* apresentou a maior taxa de compressão, ou seja, 25,6% quando comparado com o seu tamanho original e os programas *bitcount* e *crc32* (*small* e *large*) obtiveram uma taxa de compressão de 23,5%, inferior a média geral, porém tal diferença equivale a menos de 1%.

Assim como já esperado, pode-se constatar no Gráfico 5.3 que a técnica-2 (PB-CDP) apresentou uma média de 11,3% na representatividade das instruções comprimidas. Ressalta-se que 38,2% de todas as instruções dos programas do *MiBench* pertencem a classe das instruções comprimidas pela técnica-2. Já a técnica-3 (PB-CSDT) representou 10,2% das instruções comprimidas pelo método CPB-ARM. Vale lembrar que as instruções da classe comprimida pela técnica-3 possuem 32% de todas as instruções dos programas do *MiBench* compilados para o processador ARM. E por fim, a técnica-1 (PB-IIRS) representou 2,8% na taxa de compressão dos programas usados nas simulações.

Gráfico 5.3 – Taxas de compressão obtida pelo método CPB-ARM



Portanto, levando em consideração que o método CPB-ARM aplica a compressão em padrões de blocos que são otimizados para apenas duas classes específicas de instruções do processador ARM, sendo que tais classes juntas só possuem 18 tipos de instruções em sua maioria condicionadas ao sufixo *always*, e também à compressão em padrões de blocos constituídos por instruções idênticas repetidas sequencialmente, destacamos que o método CPB-ARM obteve uma taxa de compressão compatível com as taxas dos métodos estudados nos Capítulos 2 e 3 desta tese. Assim, concluímos que o método CPB-ARM mostrou-se eficiente para o uso em sistemas embarcados.

5.4 Hardware Descompressor do Método CPB-ARM

Por mais simples que seja o hardware descompressor, ele sempre influencia diretamente na área ocupada no sistema, e dependendo da eficiência da técnica desenvolvida, pode aumentar o tempo de ciclo do processador e consequentemente o consumo de energia no sistema [71].

A Figura 5.18 apresenta uma visão geral da arquitetura do hardware descompressor do método CPB-ARM e as Figuras 5.19, 5.20 e 5.21 mostram detalhadamente as arquiteturas internas do descompressor para cada uma das técnicas implementadas e usadas por esse método. Mais detalhes são apresentados no último parágrafo da Subseção 6.4.2 do Capítulo 6 dessa tese.

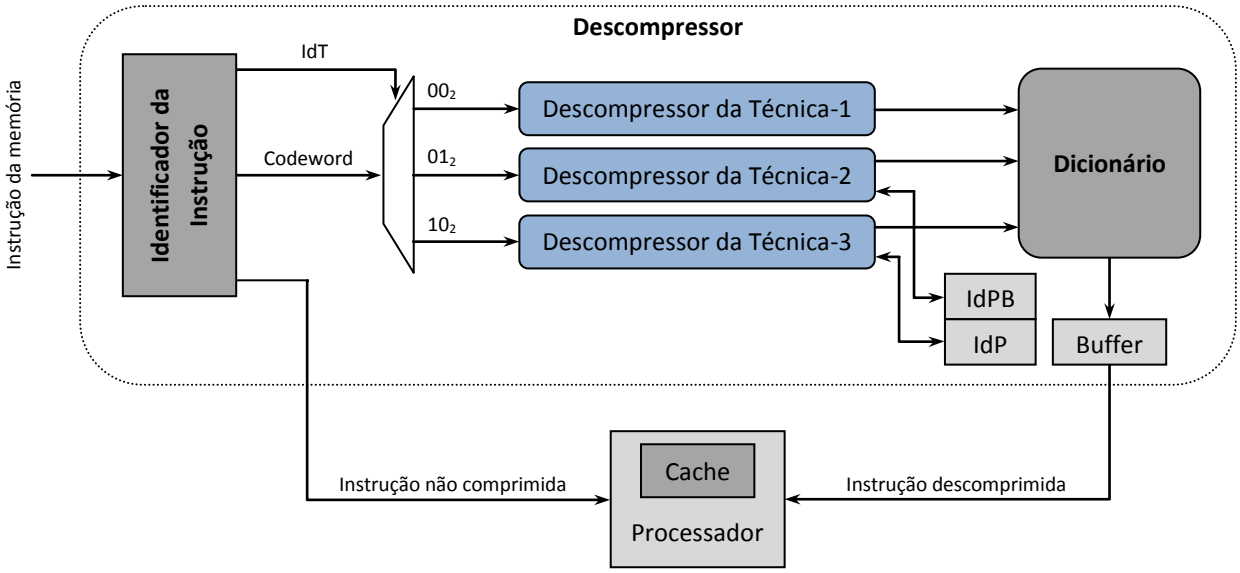


Figura 5.18 – Arquitetura do hardware descompressor do método CPB-ARM

A instrução solicitada pelo processador e encontrada na memória principal do sistema, primeiramente passa pelo componente "Identificador da Instrução" para que seja verificado através do sufixo da execução condicional (*bits* 28 a 31) da instrução do processador ARM se a mesma está ou não comprimida (conforme já apresentado anteriormente). Se o valor contido no sufixo da instrução for diferente de 1111_2, então significa que a instrução está descomprimida e a mesma é imediatamente repassada para o processador e para a memória *cache*. Se não, a instrução é uma *codeword* e então um dos três descompressores específicos do método CPB-ARM é acionado (conforme valor da *flag* IdT presente na *codeword*) e assim a instrução é descomprimida e salva no componente "Buffer" que se responsabiliza em entregá-la ao processador e à memória *cache*.

O hardware descompressor também utiliza dois componentes (tipo tabela com valores já pré-definidos) chamados de "IdPB" e "IdP" que são usados pelas técnicas 2 e 3, respectivamente. Ambos os componentes possuem os índices que identificam o formato das instruções (conforme Tabelas 5.9 e 5.10). Assim, o hardware descompressor consegue identificar nos padrões de blocos vindo do dicionário os valores dos registradores *Rn*, *Rd* e *Operand2* de cada uma das instruções pertencentes à *codeword*.

Estima-se que os descompressores específicos do método CPB-ARM quando implementados em hardware executarão em apenas um *clock* na técnica-1 e no máximo em quatro *clocks* nas técnicas 2 e 3, tendo em vista que as mesmas descomprimem padrões de blocos formados por três ou quatro instruções. Possivelmente desta forma ocorrerá pouco *overhead* no sistema quando executado o hardware descompressor desse método.

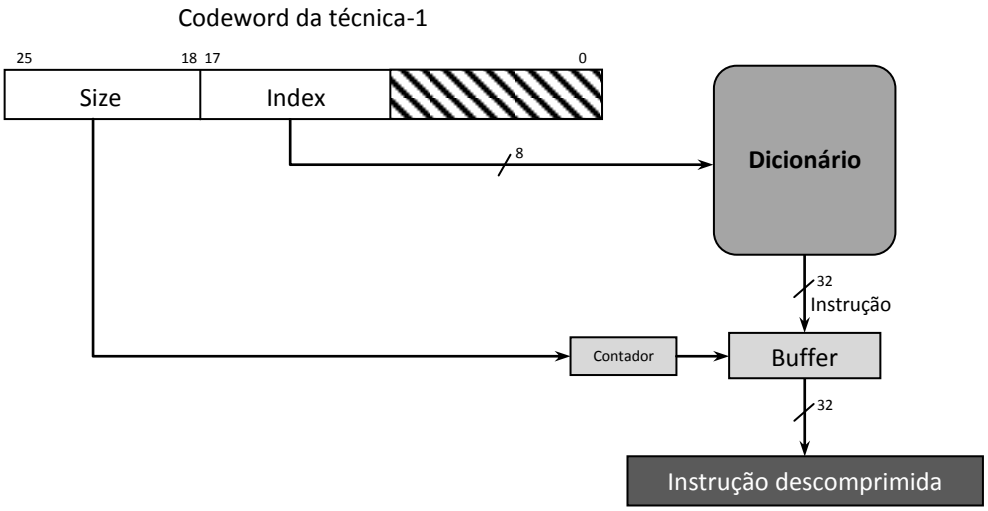


Figura 5.19 – Arquitetura interna do descompressor da técnica-1

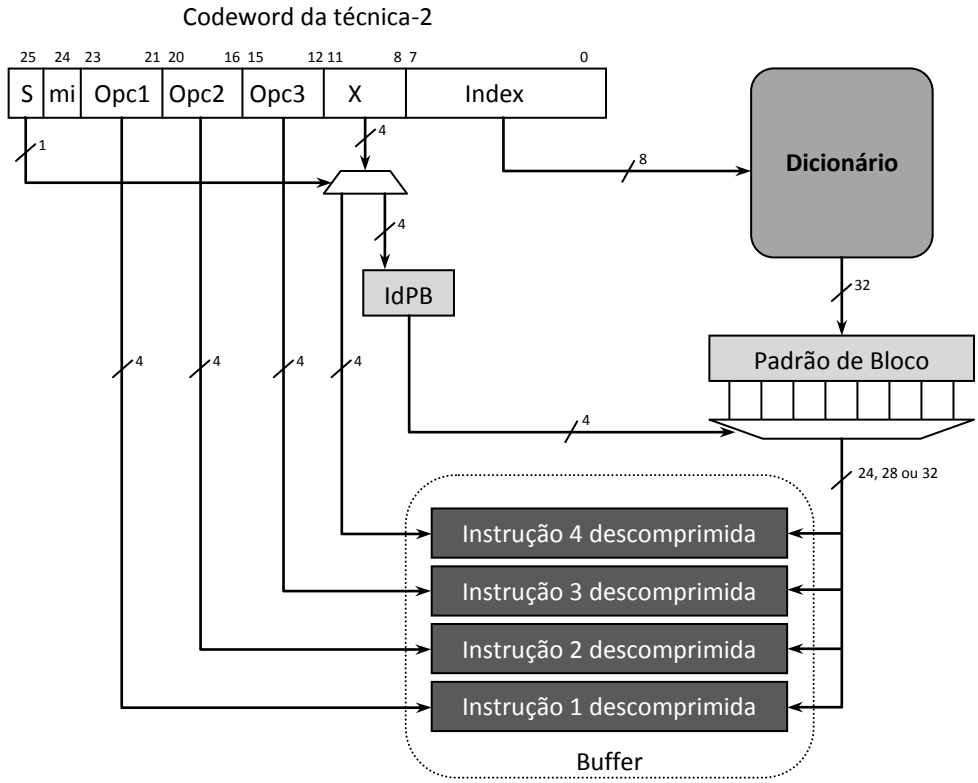


Figura 5.20 – Arquitetura interna do descompressor da técnica-2

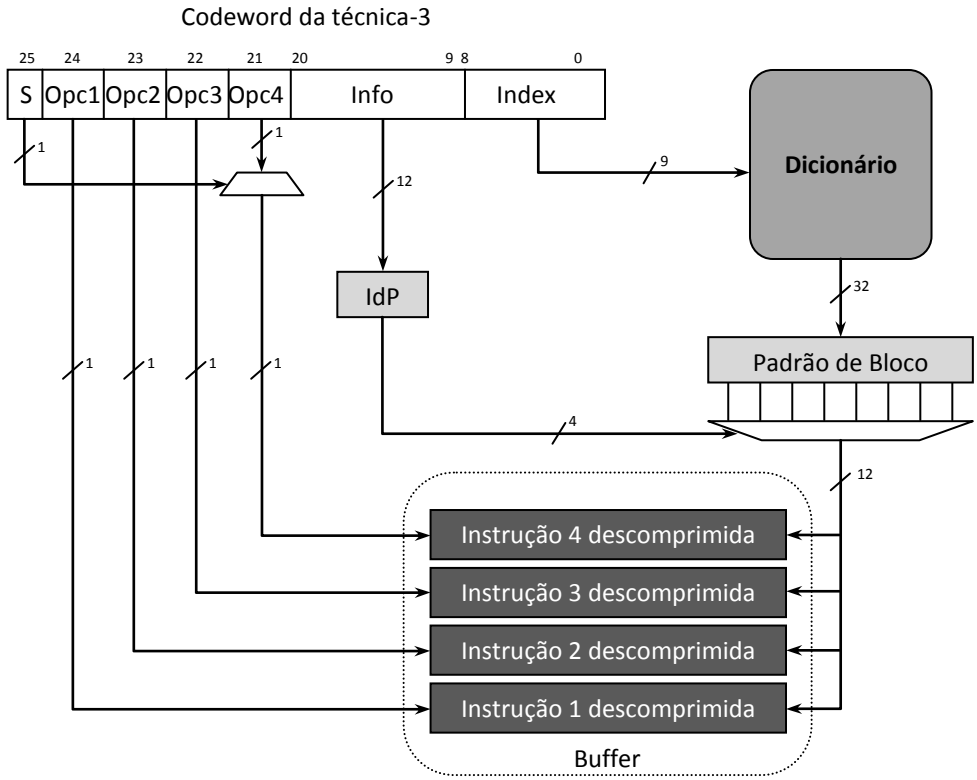


Figura 5.21 – Arquitetura interna do descompressor da técnica-3

5.5 Considerações Finais do Capítulo

Nesse capítulo apresentamos o método CPB-ARM (*ComPatternBlocks-ARM*) que é uma das contribuições dessa tese. Primeiramente, foram analisadas as redundâncias das instruções que mostraram a viabilidade no uso dos padrões de blocos para serem comprimidos, pois aproximadamente 72% de todas as instruções estáticas pertencentes aos programas do *MiBench* são instruções repetidas. Logo em seguida, foi apresentado detalhadamente o método CPB-ARM, especificando suas características e limitações, os resultados gerados nas simulações bem como suas análises, e por fim, apresentou-se uma proposta da arquitetura do hardware descompressor. Ainda destacamos que o método CPB-ARM obteve uma taxa de compressão média de 24,2%. Portanto, concluímos que esse método mostrou-se eficiente para o uso em sistemas embarcados.

No próximo capítulo, apresentamos novos métodos desenvolvidos nessa tese e que usam um novo tipo de dicionário também proposto nessa tese.

CAPÍTULO 6

COMPRESSÃO USANDO MÚLTIPLOS DICIONÁRIOS

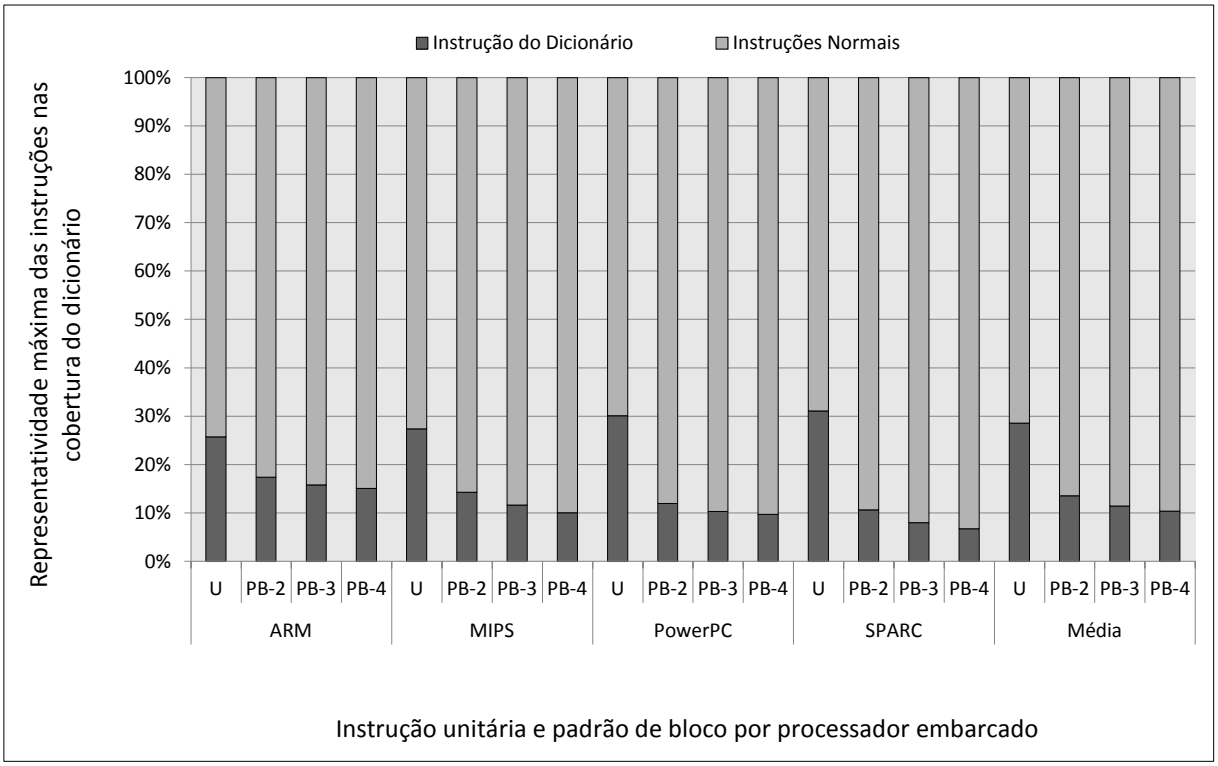
Este capítulo apresenta os métodos HDPB (*Code Compression using Huffman and Dictionary-Based Pattern Blocks*), CCHPB (*Compressed Code using Huffman + Pattern Blocks and Multi-Level Dictionary*) e CC-MLD (*Compressed Code using Huffman-Based Multi-Level Dictionary*), que são contribuições dessa tese. Esses métodos usam os padrões de blocos presentes no código dos programas para serem comprimidos e um novo tipo de dicionário múltiplo chamado de *Dicionário Multi-Nível* onde é possível armazenar instruções unitárias e padrões de blocos em níveis diferentes no mesmo dicionário. Esses métodos são independentes das características específicas do conjunto de instruções, podendo ser usados por quaisquer processadores embarcados.

O restante do capítulo está organizado da seguinte forma, na Seção 6.1 apresentamos o dicionário multi-nível e a ideia central dos passos executados pelo compressor de código; na Seção 6.2 detalhamos o método HDPB e suas características, também apresentamos as análises dos resultados obtidos nas simulações; na Seção 6.3 mostramos o método CCHPB bem como as simulações e análises desse método; na Seção 6.4 realizamos a descrição e análise detalhada do método CC-MLD, mostrando as suas características e limitações, as análises dos resultados obtidos nas simulações, o hardware descompressor e uma análise das métricas arquiteturais do processo de descompressão. Já na Seção 6.5 fazemos uma análise comparativa entre os métodos propostos e desenvolvidos nesta tese e os principais métodos de compressão de código apresentados nos Capítulos 2 e 3. E por fim, uma conclusão do capítulo.

6.1 Dicionário Multi-Nível e o Processo da Compressão de Código

Conforme resultados obtidos em simulações prévias, mostra-se no Gráfico 6.1 a representatividade das instruções pertencentes ao dicionário em relação à quantidade total de instruções dos programas apresentada na Tabela 5.1 do Capítulo 5. As "instruções unitárias" são chamadas do Gráfico 6.1 em diante apenas de U; e os "padrões de blocos" formados com duas, três e quatro instruções, de PB-2, PB-3 e PB-4, respectivamente.

Gráfico 6.1 – Representatividade das instruções na cobertura do dicionário



Nota-se no Gráfico 6.1 que a representatividade média para as instruções unitárias foi de 28,6%, já para os padrões de blocos formados com duas, três e quatros instruções foi de 13,6%, 11,5% e 10,4%, respectivamente, considerando essas médias para os programas do *MiBench* apresentados na Tabela 5.1. Assim, pode-se inferir que o uso de padrões de blocos em técnicas de compressão de código baseadas no algoritmo de *Huffman* mostra-se promissor, ou seja, mesmo com menor probabilidade de ser encontrado no código quando comparado com instruções unitárias e com a necessidade de um espaço maior para o dicionário, o uso dos padrões de blocos pode ser compensado pela substituição de uma sequência maior de *bits* por uma sequência menor de *bits*, mais detalhes são apresentados nas seções seguintes.

Os resultados do Gráfico 6.1 nos induzem a pensar na possibilidade de termos múltiplos dicionários na compressão de código. Assim, nessa tese desenvolvemos um novo tipo de

dicionário (chamado de *dicionário multi-nível*) composto por vários níveis, onde em cada nível do dicionário é possível armazenar tipos diferentes de instruções comprimidas no código dos programas. O dicionário multi-nível é usado pelos métodos HDPB, CCHPB e CC-MLD. A Figura 6.1 exemplifica um dicionário multi-nível (composto por dois níveis), sendo que o nível 1 armazena padrões de blocos formados por duas instruções e o nível 2 apenas instruções unitárias.

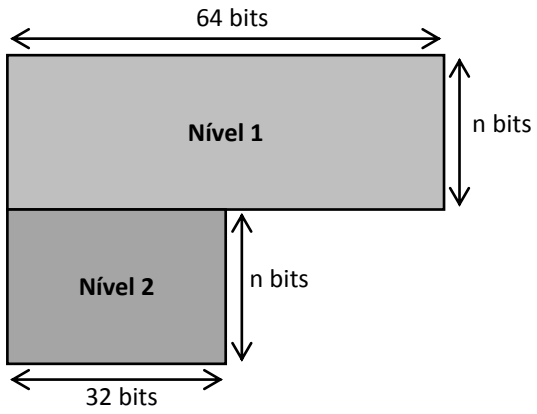


Figura 6.1 – Exemplo do dicionário multi-nível

É importante ressaltar que os tamanhos do comprimento e da largura dos níveis do dicionário multi-nível são flexíveis, ou seja, podem variar o seu tamanho de acordo com o tipo da técnica de compressão usada, então se o processo de compressão for realizado em instruções unitárias a largura do nível do dicionário terá 32 *bits* considerando uma arquitetura do tipo RISC, porém se a compressão for realizada em padrões de blocos formado por duas instruções a largura do nível do dicionário será de 64 *bits*, ou ainda, se a técnica realizar a compressão em *codewords* com tamanho de 8 *bits* então esta será a largura do nível do dicionário e assim por diante. Já o tamanho do comprimento para cada um dos níveis do dicionário é abordado na Subseção 6.2.4.

Os métodos que são apresentados nas Seções 6.2, 6.3 e 6.4 utilizam a mesma estrutura no processo de compressão. A Figura 6.2 mostra detalhadamente os passos executados pelo compressor de código desses métodos.

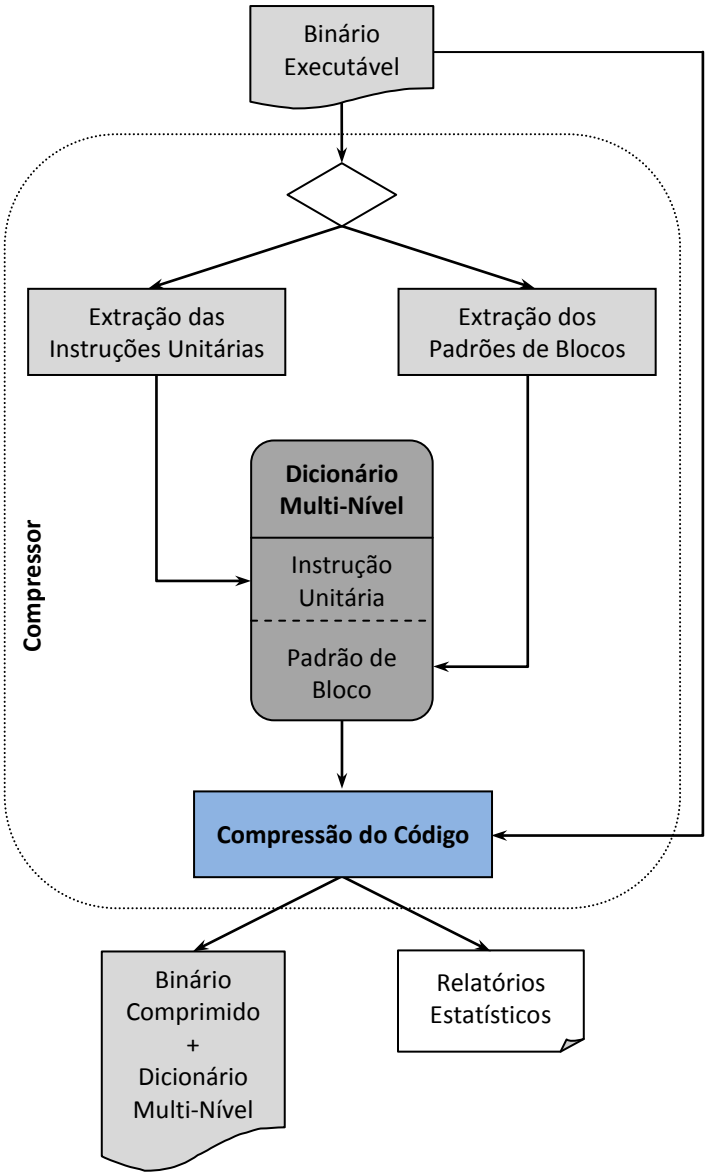


Figura 6.2 – Passos executados pelo compressor de código dos métodos HDPB, CCHPB e CC-MLD

- **Binário Executável**: o arquivo binário original no formato ELF [21a] é lido e analisado. Um arquivo ELF é composto por várias partes, chamadas de seções. Cada uma das seções pode conter código, dados, informações de depuração ou controle da carga do programa na memória. Porém, todas as seções de código (.ini, .fini, .text e outras) são verificadas e a seção correspondente ao bloco destinado a compressão (.text) tem suas instruções analisadas;
- **Extração das Instruções Unitárias**: as instruções unitárias são localizadas e ordenadas de forma decrescente pela frequência de ocorrências no código dos programas. Porém, certa quantidade destas instruções (definida pelo tamanho do

comprimento do nível do dicionário) será comprimida e armazenada no seu respectivo nível do dicionário multi-nível. O Gráfico 6.2 mostra a taxa de compressão das instruções unitárias para os programas do *MiBench* usando diversos tamanhos para o dicionário multi-nível;

- **Extração dos Padrões de Blocos**: as instruções lidas e analisadas são agrupadas em blocos compostos por duas ou mais instruções, em seguida são ordenadas de forma decrescente pela sua frequência de ocorrências. O Gráfico 6.3 mostra a taxa de compressão obtida para os três tamanhos de padrão de bloco em diferentes tamanhos de dicionário usando os programas do *MiBench*. Conforme resultados apresentados no Gráfico 6.3 constata-se que o padrão de bloco formado por duas instruções obteve a maior taxa de compressão;
- **Dicionário Multi-Nível**: cada linha do dicionário multi-nível corresponde a uma instrução unitária ou a um padrão de bloco das instruções que foram comprimidas. O dicionário multi-nível é consultado durante o processo de descompressão;
- **Compressão do Código**: a cada instrução unitária ou padrão de bloco (definido pela técnica de compressão do método usado) pertencente ao dicionário multi-nível que é encontrado no código original, é atribuído um identificador e uma *codeword* correspondente. O código original é reescrito substituindo estas instruções pelo par [id, *codeword*], onde *id* significa identificador da instrução (normal ou comprimida);
- **Binário Comprimido**: o arquivo com o binário comprimido é gerado logo após o processo de compressão, nele é inserido uma nova seção (.dic) contendo o dicionário multi-nível e que será usado pelo hardware descompressor. Este binário comprimido será executado nos sistemas embarcados que usam técnicas de compressão de código.

Gráfico 6.2 – Taxa de compressão das instruções unitárias em tamanhos diferentes do dicionário

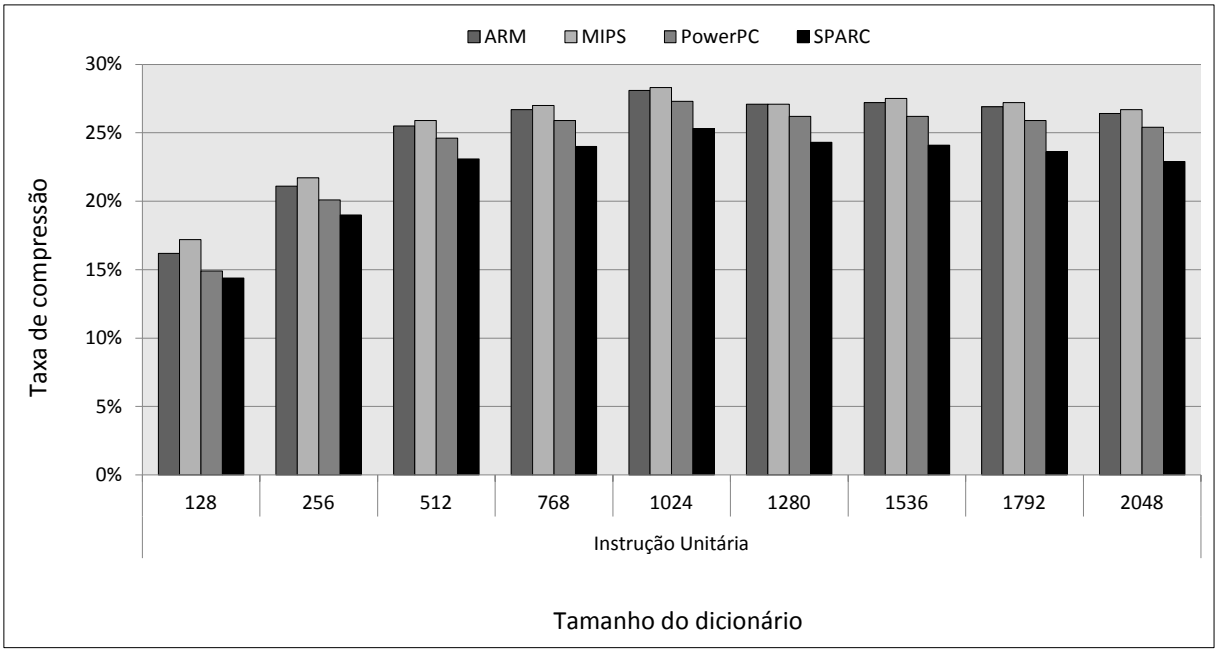
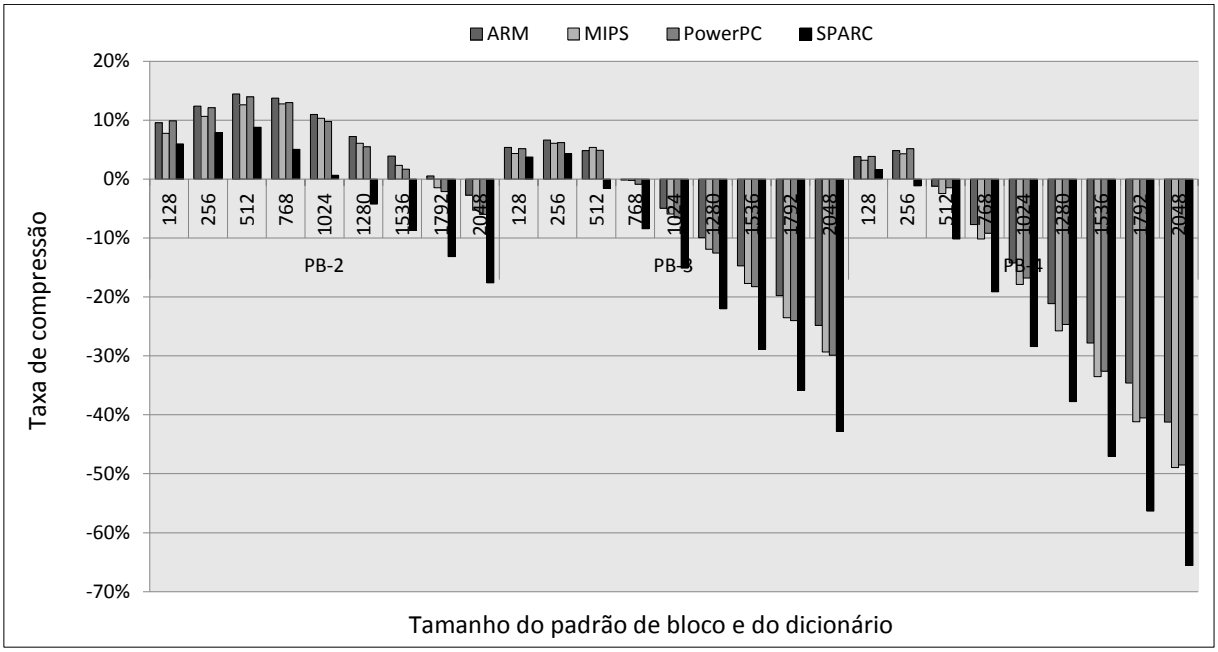


Gráfico 6.3 – Taxa de compressão dos padrões de blocos em tamanhos diferentes do dicionário



Define-se que os métodos HDPB, CCHPB e CC-MLD são algoritmos de compressão de código implementados em software que tentam localizar as instruções que se repetem com maior frequência no código dos programas (tanto unitárias quanto padrões de blocos) e atribuem uma *codeword* de menor tamanho a essas instruções, utilizando um dicionário multi-nível para armazenar as instruções que são comprimidas. Igualmente usadas por Lekatsas [37], as *codewords* geradas pelo algoritmo de *Huffman* implementado pelos métodos HDPB,

CCHPB e CC-MLD também possuem comprimento fixo, simplificando assim a lógica da descompressão em acessar o dicionário multi-nível e possivelmente reduzindo a latência do processo da descompressão.

Uma das características destes métodos é alterar o mínimo possível a forma como o projeto de um sistema embarcado é desenvolvido (como alterações na arquitetura do processador), então, mediante a isto também se escolheu desenvolver os métodos que são apresentados neste capítulo para a arquitetura CDM (já citada na Seção 2.4 do Capítulo 2), assim como o método CPB-ARM, uma vez que é possível obter melhores resultados na taxa de compressão usando a arquitetura CDM.

Conforme apresentado anteriormente usou-se os processadores embarcados: ARM (modelo ARMv5, versão SA-1110) [53]; MIPS (modelo MIPS-I, versão R3000) [59a]; PowerPC (modelo PPCe300, versão MPC83xx) [42b] e SPARC (modelo SPARCv8, versão Leon-3) [58a], como sendo os processadores alvos do estudo. E para análise e validação desses métodos, foram realizadas simulações com os programas das seis categorias do *MiBench* (apresentados na Seção 4.3 do Capítulo 4) que são específicos para sistemas embarcados.

Nas Seções 6.2, 6.3 e 6.4 apresentam-se as particularidades, os resultados obtidos nas simulações e as análises dos métodos HDPB, CCHPB e CC-MLD.

6.2 Descrição e Análise do Método HDPB

Com o objetivo de alcançar uma maior taxa na compressão do código dos programas, o método chamado de *Code Compression using Huffman and Dictionary-Based Pattern Blocks* (HDPB), implementa três técnicas semelhantes usadas sequencialmente, a saber: (i) compressão usando *Huffman* em instruções unitárias, que usa o nível 1 do dicionário; (ii) compressão usando *Huffman* em padrões de blocos nas *codewords* geradas pela técnica-1, que usa o segundo nível do dicionário e por fim, (iii) compressão usando *Huffman* em padrões de blocos (formados por duas instruções) em instruções que não foram comprimidas pelas técnicas anteriores, usando o terceiro nível do dicionário. Este método de compressão de código foi publicado em [20].

Nas Subseções 6.2.1, 6.2.2 e 6.2.3 são descritos e exemplificados o funcionamento de cada uma dessas técnicas implementadas para este método.

6.2.1 Técnica-1 – Compressão de Huffman em Instruções Unitárias

A ideia básica desta técnica é atribuir códigos de *bits* menores para instruções unitárias que se repetem com maior frequência no código dos programas. O processo de compressão primeiramente gera uma tabela com as instruções unitárias e suas respectivas frequências de repetições. Esta tabela é ordenada de forma decrescente pelo campo frequência. Em seguida, o primeiro nível do dicionário é criado.

O código original do programa é lido linha a linha, e a cada instrução unitária encontrada neste código e que também esteja inserida no dicionário nível 1 é realizada a troca pela *codeword* correspondente. Este processo se repete para todas as instruções do programa original. Assim, ao final do processo um novo código parcialmente comprimido (pré-comprimido) é obtido juntamente com o primeiro nível do dicionário multi-nível.

6.2.2 Técnica-2 – Compressão de Huffman em Padrões de Blocos nas Codewords

O processo de compressão nesta técnica é semelhante ao processo realizado pela técnica-1, usa-se como entrada para análise da compressão dos códigos o arquivo pré-comprimido que foi gerado pela técnica-1. O algoritmo nesta técnica procura apenas padrões de blocos nas *codewords* originadas pela técnica-1. Novamente uma tabela ordenada de forma decrescente é gerada para guardar todos os padrões de blocos das *codewords* e suas respectivas frequências de repetições no código. Assim, o dicionário nível 2 é formado e o processo de compressão inicia executando novamente a análise linha a linha. Ao término do processo de compressão um novo código pré-comprimido é obtido juntamente com o segundo nível do dicionário multi-nível.

6.2.3 Técnica-3 – Compressão de Huffman em Padrões de Blocos

Com base nas evidências expostas na Seção 5.1 do Capítulo 5, o objetivo desta técnica é localizar os padrões de blocos de instruções que mais se repetem no código do programa e que ainda não foram comprimidas pela técnica-1 e assim gerar uma nova instrução de tamanho reduzido (*codeword*). E assim como nas técnicas 1 e 2, é criado um novo nível do dicionário (nível 3) contendo as instruções dos padrões de blocos encontradas no código pré-comprimido.

A cada duas ou mais instruções sequenciais ainda não comprimidas no código pré-comprimido é formado um novo padrão de bloco que é armazenado em um *buffer* e em seguida é consultado se o mesmo está ou não contido no dicionário; se estiver as instruções que formaram o padrão de bloco são substituídas pela *codeword* correspondente, se não

permanecem como estão, ou seja, descomprimidas. Este processo se repete para todas as instruções do programa pré-comprimido. Ao término desse processo, o código final comprimido é obtido juntamente com o dicionário multi-nível.

Para todas as instruções do código é preciso identificar se houve ou não a compressão da mesma, assim, usa-se um identificador (Id) de tamanho fixo para esta identificação, sendo:

- $\mathbf{00}_2$: instruções que não foram comprimidas no código;
- $\mathbf{01}_2$: instruções comprimidas pela técnica-1;
- $\mathbf{10}_2$: instruções comprimidas pela técnica-2;
- $\mathbf{11}_2$: instruções comprimidas pela técnica-3.

Um exemplo da geração do código comprimido é especificado na Tabela 6.1 e pode ser visto na Figura 6.3. Os identificadores são inseridos antes de cada instrução não comprimida e da *codeword*.

Tabela 6.1 – Exemplo do código a ser comprimido

Endereço	Instrução	Instrução não comprimida e Codeword	Identificador
0x0212db7c	sub r0,r11,#28 (28 >>> 0)	$11100010010010110000000000011100_2$	00_2
0x0212db80	mov r2,r5	1100110111_2	01_2
0x0212db84	cmp r5,#0 (0 >>> 0)	1111111100_2	01_2
0x0212db88	str r3,[r11,-#28]	11100010_2	11_2
0x0212db8c	ldr r3,[r6,#0]		
0x0212db90	andeq r13,r2,#200704 (196 >>> 11)	1001010_2	10_2
0x0212db94	add r1,r1,r12	$11100000100000010001000000001100_2$	00_2
0x0212db98	beq 0x0201188c	$00001010000000000000000000011001_2$	00_2

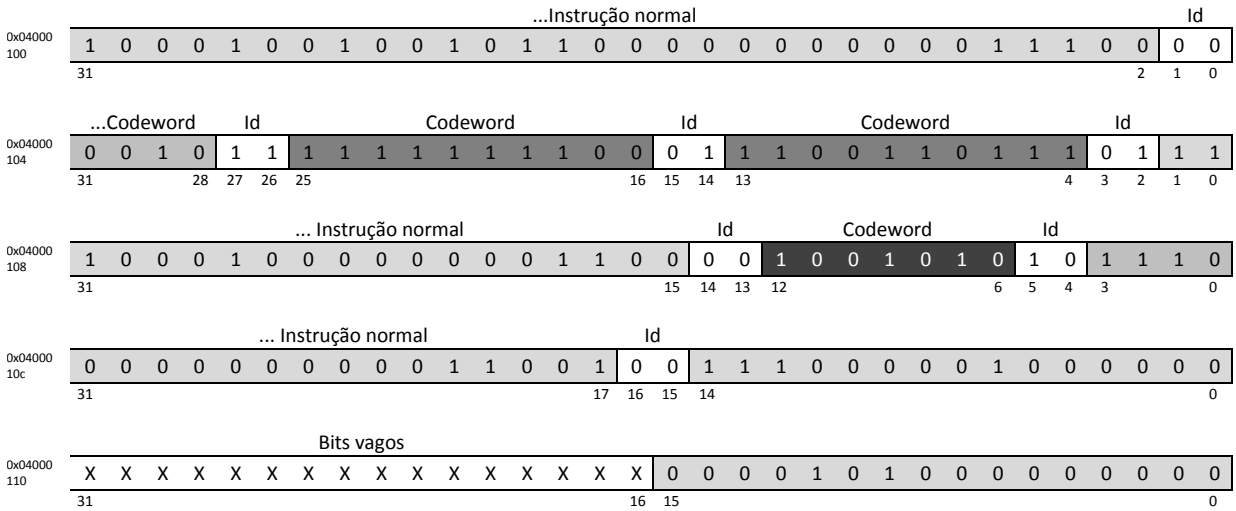


Figura 6.3 – Conteúdo da memória para um trecho do código comprimido pelo método CC-MLD

A Figura 6.4 mostra o pseudo-código do algoritmo de compressão do método HDPB.

```
Dicionário FunctionCreatDicML (Código, NívelDic, TamanhoDic)
{
  1 Dicionário(NívelDic) [TamanhoDic] ← {Conjunto Vazio}
  2 Tabela de Frequência ← {Conjunto Vazio}
  3 Enquanto não for o fim do Código
  4 Lê uma instrução do Código
    4.1 Se a instrução lida estiver na Tabela de Frequência, então
        - Incrementa o contador de frequências
    4.2 Senão,
        - Insere a instrução na Tabela de Frequência
        - Atribui o valor 1 ao contador de frequência
  5 Fim do Enquanto
  6 Ordena de forma decrescente a Tabela de Frequência pelo campo
frequência
  7 Salva a Tabela de Frequência no Dicionário(NívelDic) sem o campo
frequência
  8 Retorna o Dicionário
}

CódigoComprimido FunctionCompU (Código, Dicionário, NívelDic)
{
  1 Enquanto não for o fim do Código
  2 Lê uma instrução do Código
    2.1 Se a instrução lida estiver no Dicionário(NívelDic), então
        - Insere o identificador adequado no CódigoComprimido
        - Insere a codeword adequado no CódigoComprimido
    2.2 Senão,
        - Insere o identificador adequado no CódigoComprimido
        - Insere a instrução lida no CódigoComprimido
  3 Fim do Enquanto
  4 Retorna CódigoComprimido
}

CódigoComprimido FunctionCompPB (Código, Dicionário, NívelDic)
{
  1 Buffer ← {Conjunto Vazio}
  2 Enquanto não for o fim do Código
  3 Lê uma instrução do Código
    3.1 Salva a instrução lida no Buffer
    3.2 Se o Buffer estiver completo, então
        3.2.1 Se o Buffer estiver no Dicionário(NívelDic), então
              - Insere o identificador adequado no CódigoComprimido
              - Insere a codeword adequado no CódigoComprimido
              - Limpa o Buffer
        3.2.2 Senão,
              - Insere o identificador adequado no CódigoComprimido
              - Insere a 1ª instrução do Buffer no CódigoComprimido
              - Reorganiza o Buffer
  4 Fim do Enquanto
  5 Retorna CódigoComprimido
}

CódigoComprimido COMPRESS (CódigoOriginal)
{
  /* Técnica-1 – Compressão de Huffman em Instruções Unitárias */
  1 Chama a FunctionCreatDicML (CódigoOriginal, 1, TamanhoDic)
  2 Chama a FunctionCompU (CódigoOriginal, Dicionário, 1)
```

```
  /* Técnica-2 – Compressão de Huffman em Padrões de Blocos nas Codewords*/
  3 Chama a FunctionCreatDicML (CódigoPréComprimido, 2, TamanhoDic)
  4 Chama a FunctionCompPB (CódigoPréComprimido, Dicionário, 2)

  /* Técnica-3 – Compressão de Huffman em Padrões de Blocos */
  5 Chama a FunctionCreatDicML (CódigoPréComprimido, 3, TamanhoDic)
  6 Chama a FunctionCompPB (CódigoPréComprimido, Dicionário, 3)

  7 Insere o Dicionário Multi-Nível no CódigoComprimido
  8 Retorna o CódigoComprimido
}
```

Figura 6.4 – Pseudo-código do algoritmo de compressão do método HDPB

6.2.4 Simulações e Análises do Método HDPB

Para a escolha do tamanho do dicionário multi-nível que obtém a maior taxa de compressão, foram realizadas simulações (variando os tamanhos para cada nível entre 128 até 2.048 posições de entradas) para encontrar qual o melhor tamanho para cada um dos níveis do dicionário.

A Tabela 6.2 e o Gráfico 6.4 apresentam resultados das simulações da compressão de código usando instruções unitárias e padrões de blocos formados com duas, três e quatro instruções, e a Tabela 6.3 mostra uma sumarização das médias nas taxas de compressão apresentadas na Tabela 6.2.

Tabela 6.2 – Média na taxa de compressão usando *Huffman* em instruções unitárias e nos padrões de blocos

		Tamanho do nível do dicionário (número de entradas)								
	Processador	**128**	**256**	**512**	**768**	**1.024**	**1.280**	**1.536**	**1.792**	**2.048**
Instrução Unitária	ARM	16,2%	21,1%	25,5%	26,7%	**28,1%**	27,1%	27,2%	26,9%	26,4%
	MIPS	17,2%	21,7%	25,9%	27%	**28,3%**	27,1%	27,5%	27,2%	26,7%
	PowerPC	14,9%	20,1%	24,6%	25,9%	**27,3%**	26,2%	26,2%	25,9%	25,4%
	SPARC	14,4%	19%	23,1%	24%	**25,3%**	24,3%	24,1%	23,6%	22,9%
PB-2	ARM	9,6%	12,4%	**14,4%**	13,7%	11%	7,2%	3,9%	0,5%	-2,7%
	MIPS	7,8%	10,6%	12,6%	**12,8%**	10,3%	6,1%	2,3%	-1,5%	-5,3%
	PowerPC	9,9%	12,1%	**14%**	13%	9,8%	5,5%	1,7%	-2,1%	-6%
	SPARC	6%	7,9%	**8,8%**	5%	0,6%	-4,2%	-8,7%	-13,2%	-17,6%
PB-3	ARM	5,4%	**6,6%**	4,8%	-0,2%	-4,9%	-10%	-14,7%	-19,8%	-24,8%
	MIPS	4,4%	**6,1%**	5,4%	-0,2%	-5,9%	-11,9%	-17,7%	-23,5%	-29,3%
	PowerPC	5,1%	**6,2%**	4,9%	-0,9%	-6,6%	-12,6%	-18,3%	-24%	-29,9%
	SPARC	3,7%	**4,3%**	-1,6%	-8,4%	-15,1%	-22%	-28,9%	-35,9%	-42,8%
PB-4	ARM	3,8%	**4,8%**	-1,2%	-7,7%	-14,3%	-21,1%	-27,8%	-34,6%	-41,2%
	MIPS	3,2%	**4,3%**	-2,4%	-10,2%	-17,9%	-25,7%	-33,5%	-41,2%	-49%
	PowerPC	3,9%	**5,2%**	-1,5%	-9,2%	-16,8%	-24,6%	-32,6%	-40,5%	-48,5%
	SPARC	**1,6%**	-1,1%	-10,1%	-19,1%	-28,4%	-37,8%	-47%	-56,3%	-65,6%

Tabela 6.3 – Média geral na taxa de compressão usando *Huffman* em instruções unitárias e nos padrões de blocos

	Tamanho do nível do dicionário (número de entradas)								
	128	**256**	**512**	**768**	**1.024**	**1.280**	**1.536**	**1.792**	**2.048**
Instrução Unitária	15,7%	20,5%	24,7%	25,9%	**27,2%**	26,1%	26,2%	25,9%	25,4%
PB-2	8,3%	10,8%	**12,5%**	11,1%	7,9%	3,6%	-0,2%	-4,1%	-7,9%
PB-3	4,6%	**5,8%**	3,4%	-2,4%	-8,1%	-14,1%	-19,9%	-25,8%	-31,7%
PB-4	3,1%	**3,3%**	-3,8%	-11,6%	-19,3%	-27,3%	-35,2%	-43,2%	-51,1%

Observando o Gráfico 6.4 conclui-se que os padrões de blocos formados com três e quatro instruções não apresentaram resultados satisfatórios quando comparados com os resultados das instruções unitárias e dos padrões de blocos formados com duas instruções, sendo que na maioria das vezes as taxas de compressão e crescimento foram até negativas. Portanto, concluímos que o uso de padrões de blocos formados com três ou quatro instruções não são adequados para serem comprimidos e armazenados em dicionário multi-nível. Sendo assim, deste ponto em diante nessa tese são usados e analisados para as demais simulações apenas os padrões de blocos formados com duas instruções.

Para a escolha do tamanho ideal do primeiro nível do dicionário usado pelo método HDPB, pode-se observar no Gráfico 6.4 que os tamanhos 512 e 1.024 apresentaram as melhores combinações entre as taxas de compressão e crescimento para as instruções unitárias. Obtendo uma média de 26% na taxa de compressão e de 2,8% na taxa de crescimento para os quatro processadores embarcados.

Já o Gráfico 6.5 também apresenta resultados das taxas de compressão e crescimento obtidas usando a segunda técnica do método HDPB. Foram pré-definidos os tamanhos (512 e 1.024) para o nível 1 do dicionário e variou-se os tamanhos para o segundo nível do dicionário. Conforme se observa no Gráfico 6.5, os tamanhos 128 e 256 também apresentaram os maiores resultados nas taxas de compressão e crescimento para o dicionário nível 2.

E por fim, quando permutados os tamanhos pré-definidos anteriormente para os níveis 1 e 2 do dicionário multi-nível, observa-se no Gráfico 6.6 que os tamanhos 256 e 512 alcançaram as maiores taxas de compressão e crescimento para o terceiro nível do dicionário. Assim, a Tabela 6.4 apresenta uma sumarização das médias na taxa de compressão do método HDPB usando os quatro processadores embarcados para os programas do *MiBench*.

Gráfico 6.4 – Taxas de compressão e crescimento usando *Huffman* em instruções unitárias e padrões de blocos

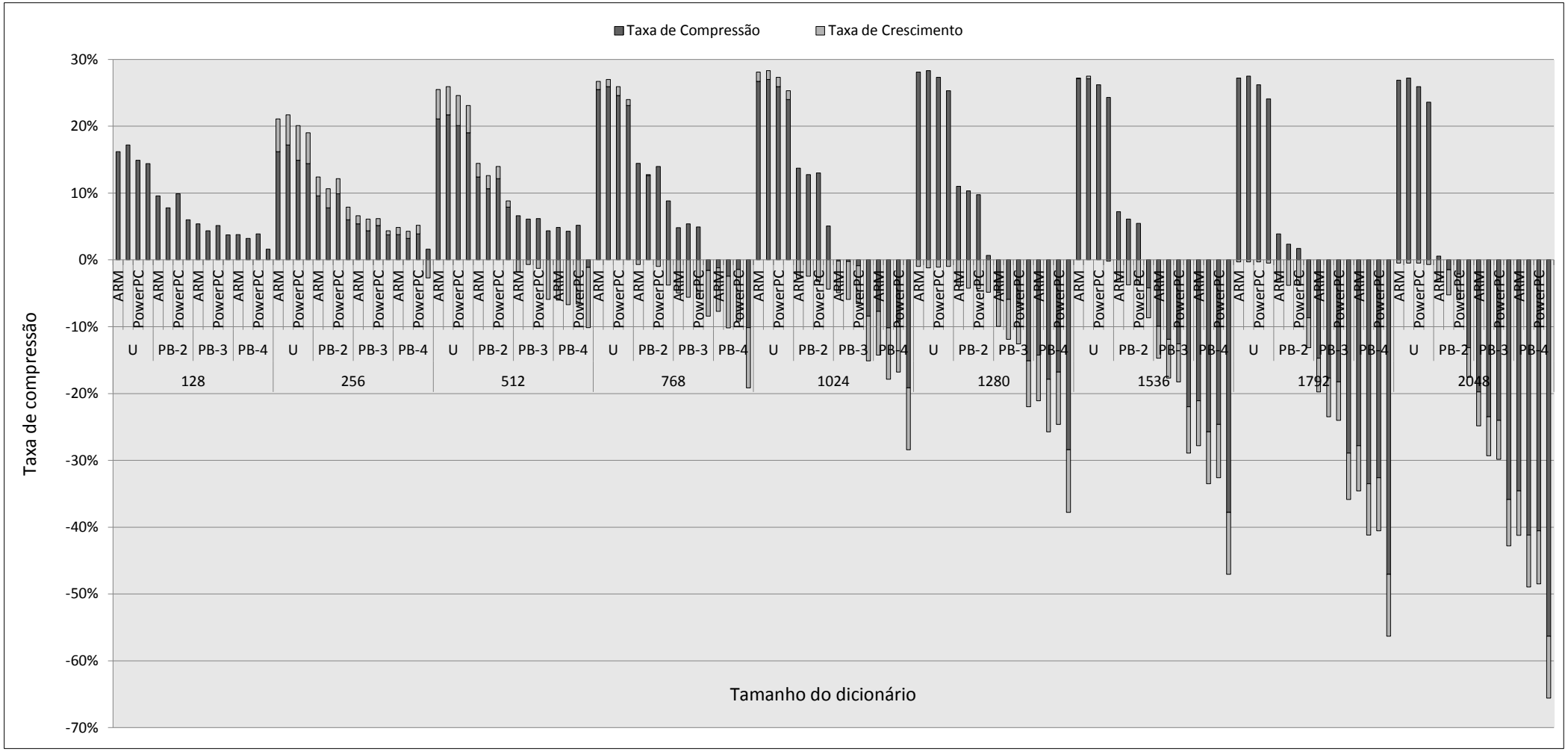


Gráfico 6.5 – Taxa de compressão dos padrões de blocos das *codewords* em diferentes tamanhos do dicionário nível 2

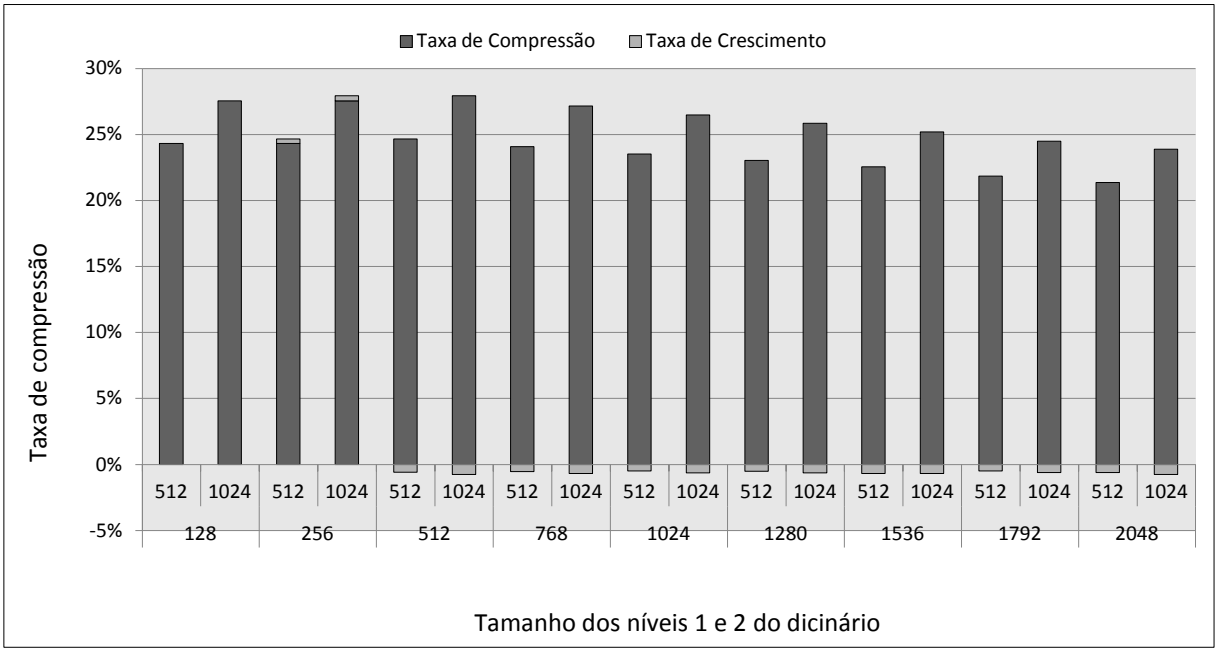
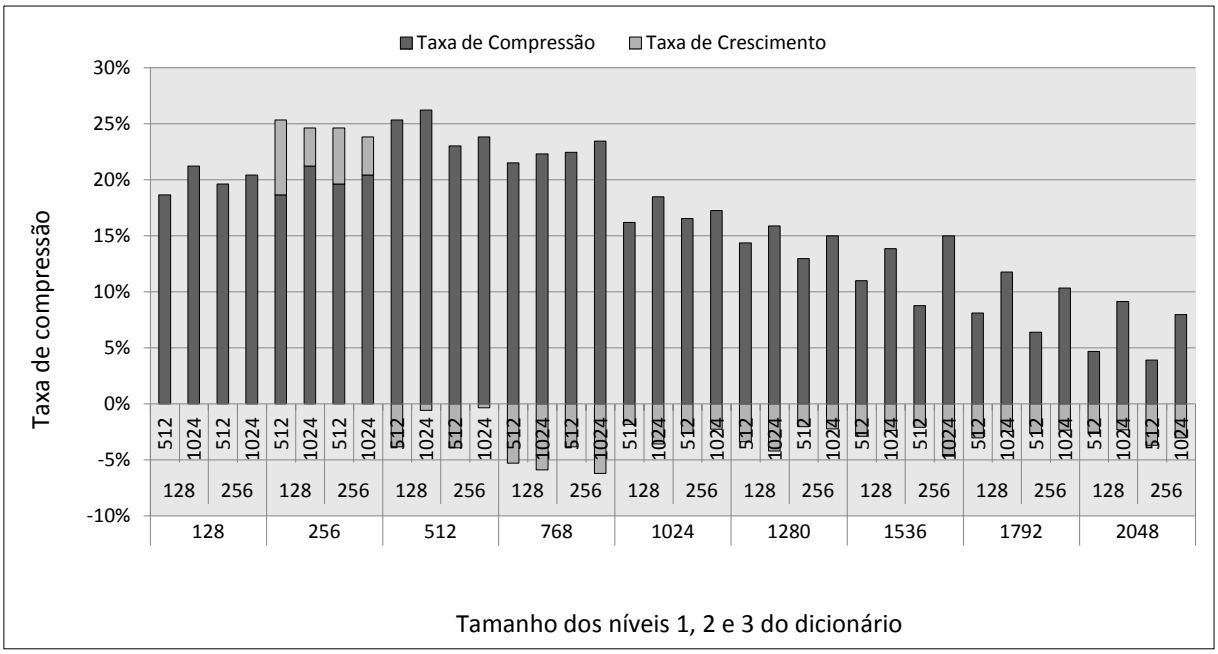


Gráfico 6.6 – Taxa de compressão dos padrões de blocos em diferentes tamanhos do dicionário nível 3

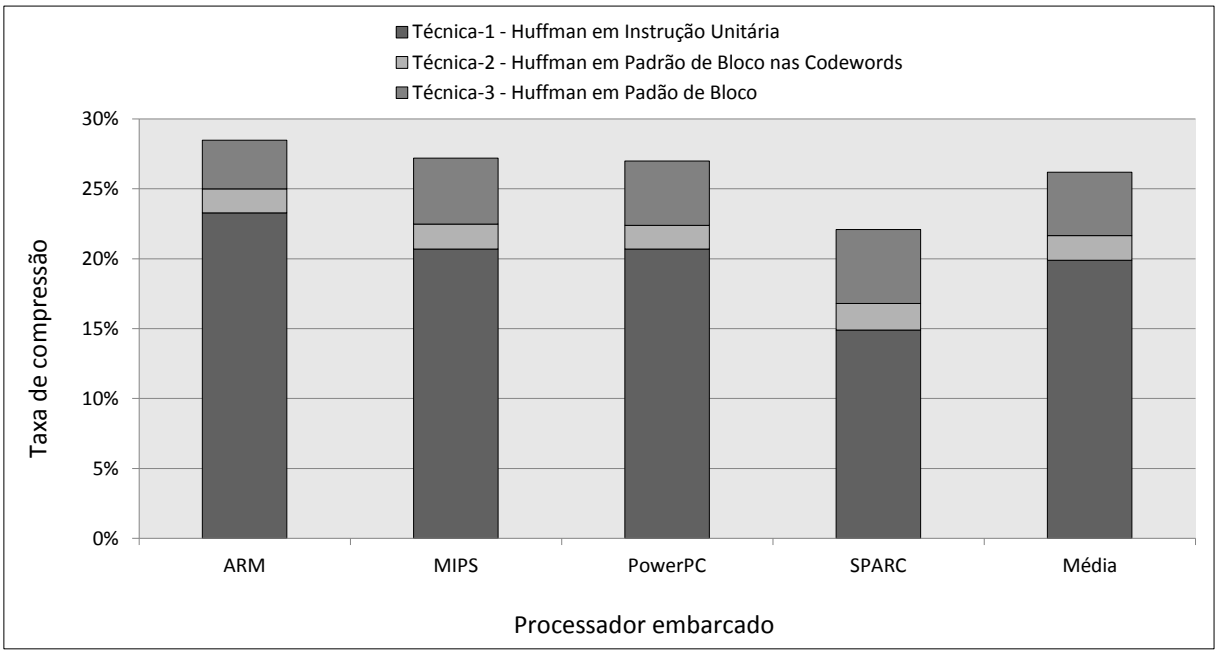


Observa-se na Tabela 6.4 que a melhor configuração para o dicionário multi-nível usando esse método é quando os níveis 1, 2 e 3 do dicionário possuem tamanhos de 1.024, 128 e 256 posições de entradas respectivamente. O Gráfico 6.7 apresenta detalhadamente a taxa de compressão obtida pelo método HDPB.

Tabela 6.4 – Média geral na taxa de compressão usando o método HDPB

		Nível 1	
Nível 2	**Nível 3**	**512**	**1024**
128	**256**	25,3%	**26,2%**
	512	21,5%	22,3%
256	**256**	23%	23,8%
	512	22,4%	23,4%

Gráfico 6.7 – Taxa de compressão obtida pelo método HDPB



Constata-se no Gráfico 6.7 que a técnica-1 foi responsável por 19,9% na representatividade da taxa de compressão, a técnica-2 por apenas 1,8% e os outros 4,5% restantes pela técnica-3, alcançando assim uma taxa média de compressão de 26,2% para os programas do *MiBench*.

No entanto, analisando e comparando os valores das Tabelas 6.3 e 6.4 (apenas as instruções unitárias), constatamos que o método HDPB não apresentou nenhuma configuração que obtivesse uma taxa de compressão superior às taxas de compressão apresentadas na Tabela 6.3, dando origem ao método CCHPB que é apresentado a seguir.

6.3 Descrição e Análise do Método CCHPB

O método chamado de *Compressed Code using Huffman + Pattern Blocks and Multi-Level Dictionary* (CCHPB) é basicamente idêntico ao método HDPB apresentado na Seção

6.2. Após algumas análises verificou-se que alterando a ordem sequencial da execução das técnicas, o resultado alcançado na taxa de compressão poderia ser melhorado. Então, o método CCHPB primeiramente executa a técnica de compressão usando *Huffman* em padrões de blocos (formados com duas instruções), que usa o nível 1 do dicionário, em seguida executa-se a técnica de compressão usando *Huffman* em instruções unitárias, que usa o segundo nível do dicionário e por último a técnica de compressão usando *Huffman* em padrões de blocos nas *codewords* geradas pela nova técnica-2, que usa o terceiro nível do dicionário. Este método foi publicado em [21, 23].

O pseudo-código do algoritmo de compressão do método CCHPB é quase idêntico ao apresentado na Figura 6.4, alterando apenas a ordem na chamada de execução das técnicas na função **`COMPRESS`**, passando agora a ser chamada de acordo com a sequência exposta no parágrafo anterior.

6.3.1 Simulações e Análises do Método CCHPB

As mesmas análises feitas na Subseção 6.2.4 para escolha do tamanho ideal dos níveis do dicionário multi-nível também são mantidas para este método. Analisando os resultados apresentados no Gráfico 6.4, destacam-se quatro importantes pontos que justificam a escolha dos tamanhos 256, 512 e 768 posições de entradas para nível 1 do dicionário usado pelo método CCHPB. O primeiro ponto mostra que as maiores taxas de compressão foram alcançadas com dicionários destes tamanhos, sendo em média (para os quatro processadores) 10,8% para o dicionário com tamanho 256; 12,5% e 11,1% para os dicionários com tamanhos 512 e 768, respectivamente.

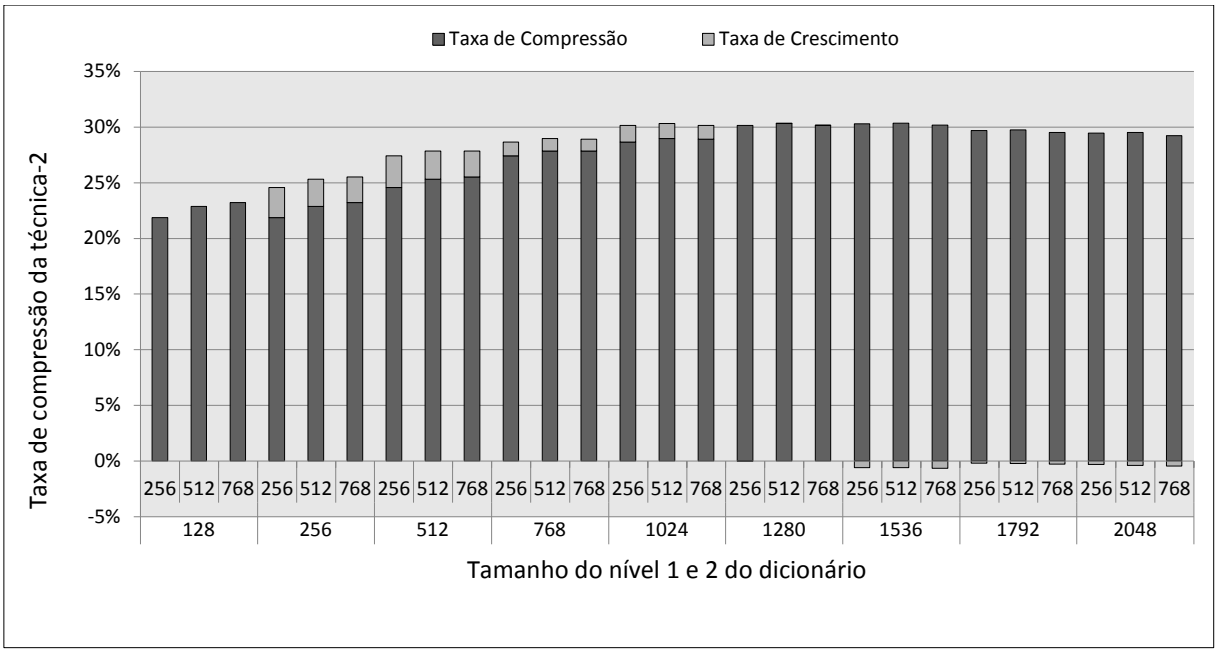
O segundo ponto destaca que os dicionários com estes tamanhos foram os que apresentaram as maiores taxas de crescimento quando comparados com seus dicionários de tamanho antecessor, ou seja, 512 vs. 256, 756 vs. 512 e assim por diante, sendo em média um crescimento de aproximadamente 3% para o dicionário com tamanho 256; 1,7% para o dicionário com tamanho 512 e -1,3% quando o dicionário tiver o tamanho 768. Essas médias também foram obtidas analisando os quatro processadores embarcados.

O terceiro ponto mostra que a taxa de compressão para os dicionários acima de 768 posições de entradas torna-se cada vez menor, conforme se observa na Tabela 6.2 e no Gráfico 6.4, chegando um instante em que a taxa de compressão tenderá a um valor negativo (exemplo: PB-2, tamanho do dicionário 1.280, processador SPARC) e assim realizará um efeito inverso, ou seja, ao invés da taxa de compressão aumentar ela diminuirá em virtude do tamanho do dicionário em relação à quantidade de instruções comprimidas. E por fim, o

quarto ponto destaca que quanto menor o tamanho do dicionário nível 1, menor é a diferença entre a taxa de compressão obtida para as instruções unitárias e os padrões de blocos (ver Tabela 6.3).

O Gráfico 6.8 apresenta os resultados das taxas de compressão obtidas logo após a execução da técnica-2 do método CCHPB. Os tamanhos determinados anteriormente para o nível 1 do dicionário foram fixados e em seguida variados os tamanhos para o segundo nível do dicionário. Então, conforme se observa no Gráfico 6.8 os tamanhos 512, 768 e 1.024 também obtiveram os maiores resultados nas taxas de compressão e crescimento.

Gráfico 6.8 – Taxas de compressão e crescimento usando a técnica-2 na busca do tamanho ideal para o segundo nível do dicionário multi-nível



E por último, permutou-se os tamanhos pré definidos anteriormente para os níveis 1 e 2 do dicionário multi-nível e constatou-se que os tamanhos 128, 256 e 512, foram os que apresentaram as maiores taxas de compressão para o terceiro nível do dicionário, conforme pode ser observado no Gráfico 6.9. Assim, a Tabela 6.5 apresenta uma sumarização das médias nas taxas de compressão do método CCHPB usando os quatro processadores embarcados para os programas do *MiBench*.

Conforme os valores apresentados na Tabela 6.5, constata-se que a melhor configuração para o dicionário multi-nível usando o método CCHPB é quando os níveis 1, 2 e 3 do dicionário possuem os tamanhos 512, 1.024 e 128 posições de entradas respectivamente,

obtendo uma taxa de compressão média de 32% para os programas do *MiBench*. O Gráfico 6.10 mostra a taxa de compressão obtida por cada uma das técnicas do método CCHPB.

Gráfico 6.9 – Taxas de compressão e crescimento usando a técnica-3 na busca do tamanho ideal para o terceiro nível do dicionário multi-nível

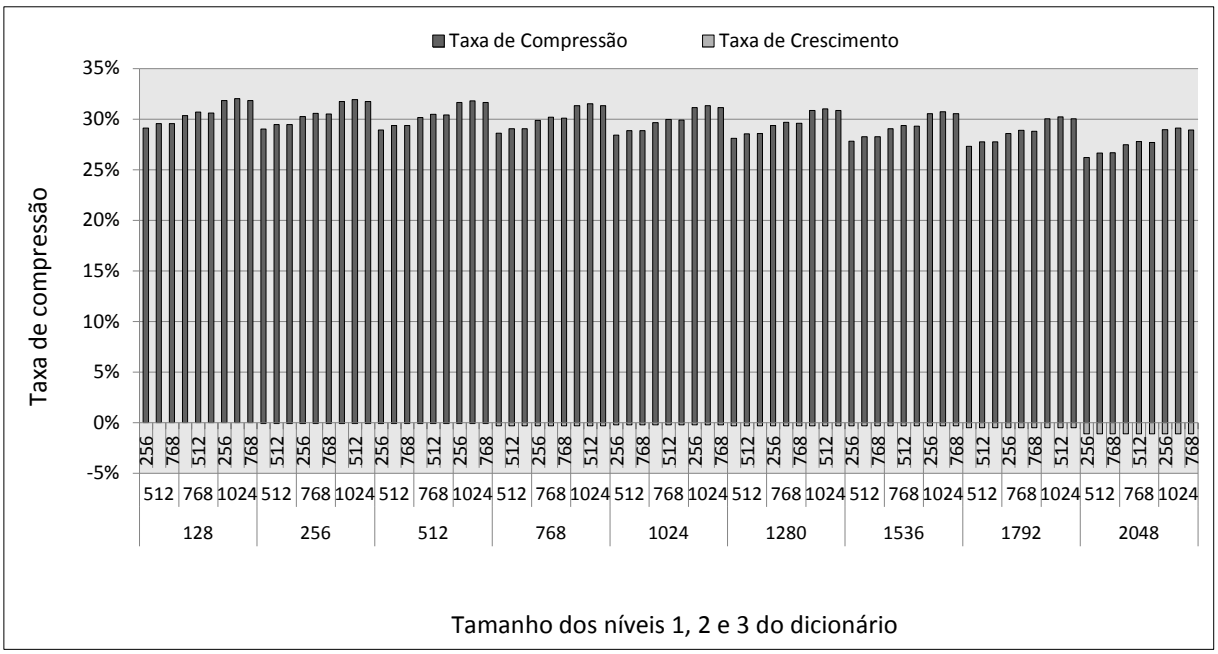
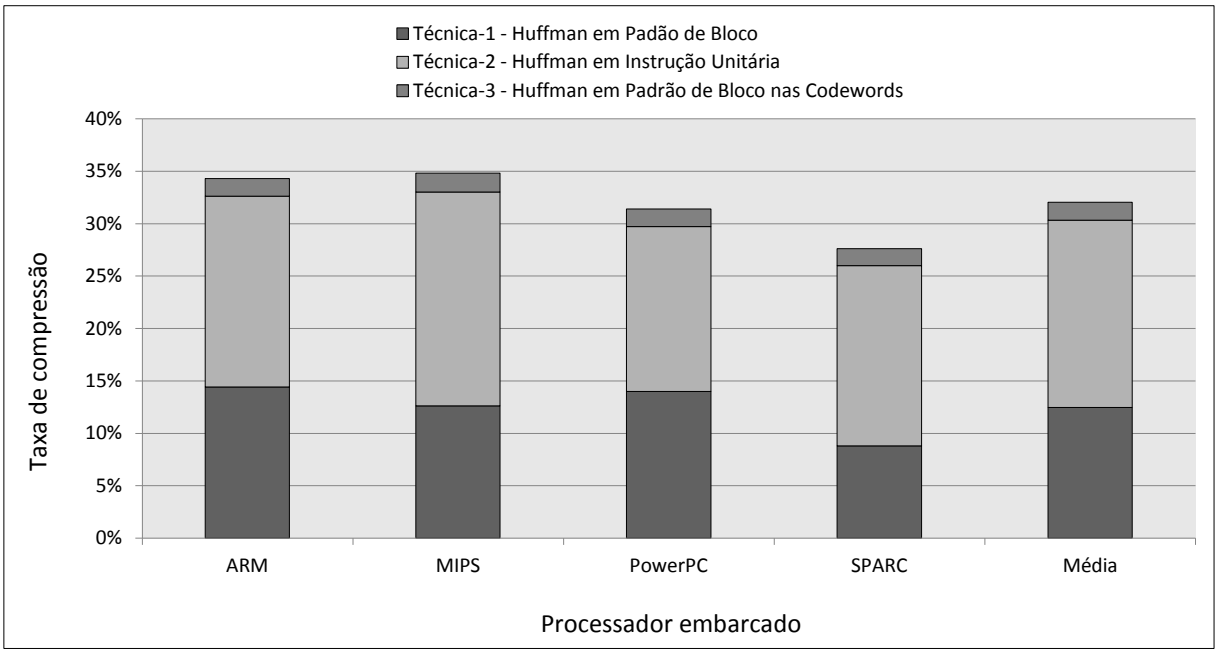


Tabela 6.5 – Média geral na taxa de compressão usando o método CCHPB

		Nível 1		
Nível 2	**Nível 3**	**256**	**512**	**768**
512	**128**	29,1%	29,5%	29,6%
	256	29%	29,4%	29,5%
	512	28,9%	29,3%	29,4%
768	**128**	30,4%	30,7%	30,6%
	256	30,3%	30,6%	30,5%
	512	30,2%	30,5%	30,4%
1.024	**128**	31,8%	**32%**	31,8%
	256	31,7%	31,9%	31,7%
	512	31,6%	31,8%	31,6%

Observando os valores no Gráfico 6.10 verifica-se que a técnica-1 do método CCHPB foi responsável por 12,4% da taxa de compressão, a técnica-2 por 17,9% e a técnica-3 por apenas 1,7%.

Gráfico 6.10 – Taxa de compressão obtida pelo método CCHPB



Concluí-se verificando os valores das Tabelas 6.4 e 6.5 que apenas fazendo uma reordenação na sequência de execução das técnicas desenvolvidas para o método HDPB foi possível obtermos uma melhora na taxa de compressão, ou seja, a melhor configuração (1024, 128 e 256 para os níveis 1, 2 e 3) do dicionário multi-nível para o método HDPB obteve uma média na taxa de compressão de 26,2%, comparando (com o mesmo tamanho de dicionário) ao método CCHPB que obteve uma média de 31,8% esta diferença foi de 5,6% a favor do método CCHPB. Este valor foi ainda mais significativo quando a comparação foi feita de forma inversa, sendo agora que a melhor configuração (512, 1024 e 128 para os níveis 1, 2 e 3) para o dicionário multi-nível do método CCHPB obteve uma taxa média de compressão de 32% e para o método HDPB esta taxa média foi de 22,3%, atingindo assim uma diferença de 9,7% a mais para o método CCHPB.

6.4 Descrição e Análise do Método CC-MLD

Analisando os resultados apresentados no Gráfico 6.10 constata-se que a técnica-3 do método CCHPB contribuiu apenas com 1,7% na taxa final da compressão do código dos programas do *MiBench*, sendo este um valor quase insignificante e o que nos induziu a pensar que esta técnica-3 produz apenas mais complexidade ao método proposto, gerando assim uma possível perda de desempenho do sistema devido ao processo da descompressão das *codewords* comprimidas. Então, mediante essa observação, uma forma de otimizar o método,

é excluir a técnica-3 do CCHPB e assim renomeá-lo para o método chamado de CC-MLD (*Compressed Code using Huffman-Based Multi-Level Dictionary*).

O método CC-MLD executa apenas as técnicas de compressão usando *Huffman* em padrões de blocos e em instruções unitárias, nessa ordem. A Figura 6.5 mostra o pseudo-código do algoritmo de compressão do método CC-MLD.

```
Dicionário FunctionCreatDicML (Código, NívelDic, TamanhoDic)
{
  1 Dicionário(NívelDic) [TamanhoDic] ← {Conjunto Vazio}
  2 Tabela de Frequência ← {Conjunto Vazio}
  3 Enquanto não for o fim do Código
  4 Lê uma instrução do Código
    4.1 Se a instrução lida estiver na Tabela de Frequência, então
        - Incrementa o contador de frequências
    4.2 Senão,
        - Insere a instrução na Tabela de Frequência
        - Atribui o valor 1 ao contador de frequência
  5 Fim do Enquanto
  6 Ordena de forma decrescente a Tabela de Frequência pelo campo
frequência
  7 Salva a Tabela de Frequência no Dicionário(NívelDic) sem o campo
frequência
  8 Retorna o Dicionário
}

CódigoComprimido FunctionCompU (Código, Dicionário, NívelDic)
{
  1 Enquanto não for o fim do Código
  2 Lê uma instrução do Código
    2.1 Se a instrução lida estiver no Dicionário(NívelDic), então
        - Insere o identificador adequado no CódigoComprimido
        - Insere a codeword adequado no CódigoComprimido
    2.2 Senão,
        - Insere o identificador adequado no CódigoComprimido
        - Insere a instrução lida no CódigoComprimido
  3 Fim do Enquanto
  4 Retorna CódigoComprimido
}

CódigoComprimido FunctionCompPB (Código, Dicionário, NívelDic)
{
  1 Buffer ← {Conjunto Vazio}
  2 Enquanto não for o fim do Código
  3 Lê uma instrução do Código
    3.1 Salva a instrução lida no Buffer
    3.2 Se o Buffer estiver completo, então
        3.2.1 Se o Buffer estiver no Dicionário(NívelDic), então
              - Insere o identificador adequado no CódigoComprimido
              - Insere a codeword adequado no CódigoComprimido
              - Limpa o Buffer
        3.2.2 Senão,
              - Insere o identificador adequado no CódigoComprimido
              - Insere a 1ª instrução do Buffer no CódigoComprimido
              - Reorganiza o Buffer
  4 Fim do Enquanto
  5 Retorna CódigoComprimido
}
```

```
CódigoComprimido COMPRESS (CódigoOriginal)
{
   /* Técnica-1 – Compressão de Huffman em Padrões de Blocos */
   1 Chama a FunctionCreatDicML (CódigoOriginal, 1, TamanhoDic)
   2 Chama a FunctionCompPB (CódigoOriginal, Dicionário, 1)

   /* Técnica-2 – Compressão de Huffman em Instruções Unitárias */
   3 Chama a FunctionCreatDicML (CódigoPréComprimido, 2, TamanhoDic)
   4 Chama a FunctionCompU (CódigoPréComprimido, Dicionário, 2)

   5 Insere o Dicionário Multi-Nível no CódigoComprimido
   6 Retorna o CódigoComprimido
}
```

Figura 6.5 – Pseudo-código do algoritmo de compressão do método CC-MLD

6.4.1 Simulações e Análises do Método CC-MLD

As simulações para o método CC-MLD são basicamente as mesmas simulações que foram realizadas para os métodos HDPB e CCHPB. É definido um dicionário com apenas dois níveis para o método CC-MLD, sendo o primeiro para guardar os padrões de blocos formados com duas instruções e o segundo para guardar as instruções unitárias. Os melhores tamanhos encontrados para o primeiro nível do dicionário foram 256, 512 e 768 e para o segundo nível 512, 1.024 e 1.280 posições de entradas. A Tabela 6.6 mostra uma sumarização das médias nas taxas de compressão, comprimindo os programas do *MiBench* com o método CC-MLD para cada um dos quatro processadores embarcados e a Tabela 6.7 mostra a média geral para os resultados apresentados na Tabelas 6.6.

Tabela 6.6 – Média na taxa de compressão para cada processador embarcado usando o método CC-MLD

		Nível 1		
Nível 2	**Processador**	**256**	**512**	**768**
512	ARM	29,5%	28,7%	26,5%
	MIPS	29%	28,2%	26,2%
	PowerPC	28,5%	27,7%	25,3%
	SPARC	25,7%	23,6%	19,2%
1.024	ARM	**31,3%**	30%	27,3%
	MIPS	**30,7%**	29,3%	27,1%
	PowerPC	**30,3%**	28,9%	26,1%
	SPARC	29,9%	24,7%	20,3%
1.280	ARM	30,5%	28,8%	26%
	MIPS	30%	28,3%	25,8%
	PowerPC	29,5%	27,7%	24,7%
	SPARC	**30,5%**	23,2%	18,8%

Tabela 6.7 – Média geral na taxa de compressão usando o método CC-MLD

	Nível 1		
Nível 2	**256**	**512**	**768**
512	28,2%	27%	24,3%
1.024	**30,6%**	28,3%	25,2%
1.280	30,1%	27%	23,8%

Observando os valores das Tabelas 6.3 e 6.7 destaca-se que, se fixarmos os tamanhos para o primeiro e segundo nível do dicionário (Tabela 6.7) e compararmos os valores das taxas de compressão com os valores dos mesmos tamanhos de dicionários apresentados na Tabela 6.3, nota-se que na maioria das vezes o método CC-MLD conseguiu obter uma taxa de compressão maior quando comparado com a compressão tradicional de *Huffman*, seja comprimindo instruções unitárias ou padrões de blocos. Então, fixando o tamanho dos níveis 1 e 2 do dicionário multi-nível em 256 e 512 (768 posições de entradas) foi alcançado um aumento de 2,3% na compressão do código comparada com a taxa das instruções unitárias; Já para os tamanhos 256 e 1.024 esta diferença foi de 4,5%; O mesmo ocorreu para os tamanhos 256 e 1.280 atingindo 3,9% de crescimento na taxa de compressão. No entanto, para o tamanho 768 e 1.280 (2.048 posições de entrada) o método CC-MLD obteve uma taxa de compressão menor (-1.6%) do que o método tradicional de *Huffman*.

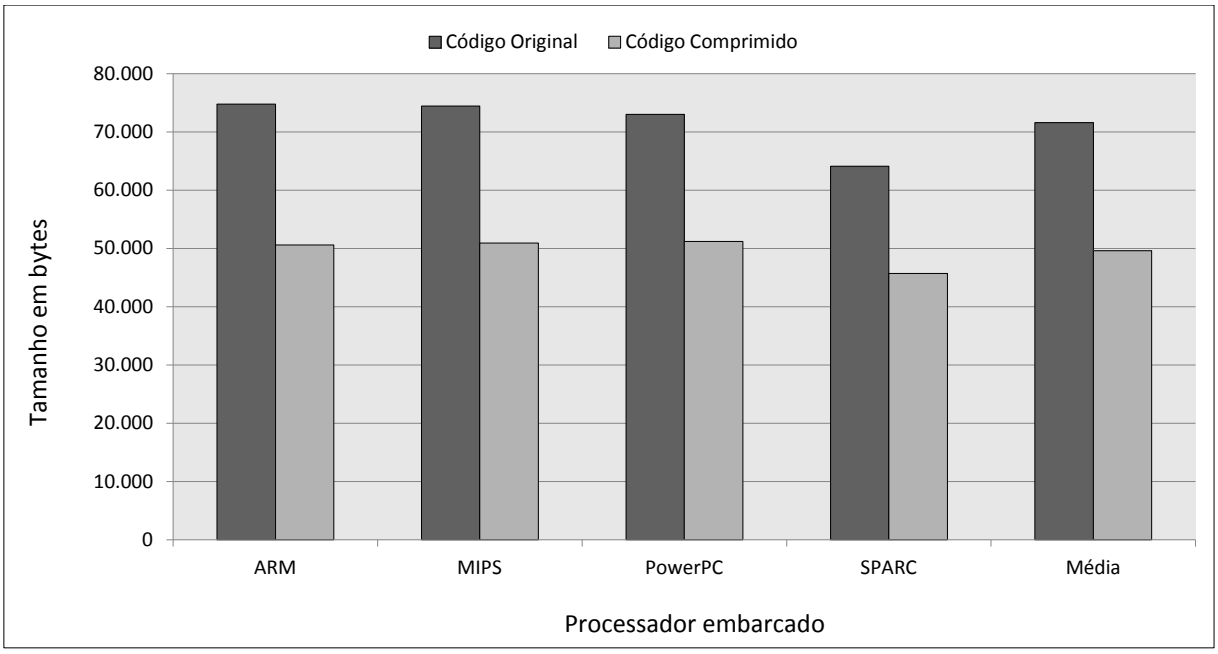
Destaca-se ainda ao observar a Tabela 6.7, que na medida em que o tamanho do nível 1 do dicionário começa a crescer, a taxa de compressão realiza um efeito inverso, ou seja, começa a diminuir. Então, conclui-se que não é vantajoso no método CC-MLD o uso de um dicionário multi-nível com o tamanho de níveis maiores, conforme provado pelos resultados expressos nas Tabelas 6.3 e 6.6 para os tamanhos dos níveis 1 e 2 igual a 768 e 512; 768 e 1.024; 768 e 1.280 do dicionário multi-nível. Portanto, ressalta-se que para projetos de sistemas embarcados onde o espaço físico é um requisito de alta prioridade o uso de um dicionário multi-nível de tamanho menor provavelmente terá um efeito mais significativo do que um dicionário onde os tamanhos dos níveis são maiores.

Um dos principais focos dos métodos apresentados nesse capítulo é mostrar que existem outras formas de aumentar a taxa de compressão utilizando o algoritmo de *Huffman*. Sendo que, para os métodos HDPB, CCHPB e CC-MLD desenvolveu-se uma nova abordagem que é a compressão usando *Huffman* em padrões de blocos formados por duas ou mais instruções. No entanto, ao analisar as Tabelas 6.3, 6.4, 6.5 e 6.7 conclui-se que mesmo sendo necessário espaço adicional para um novo dicionário multi-nível os métodos apresentados nesse capítulo conseguiram ser tão eficiente na compressão dos códigos quanto os resultados apresentados

por outros métodos estudados nos Capítulos 2 e 3 dessa tese e que utilizaram como base o algoritmo de *Huffman* e dicionários.

Também ressalta-se que a melhor configuração do dicionário multi-nível para a compressão dos programas do *MiBench* usando o método CC-MLD nas quatro arquiteturas dos processadores embarcados é quando os níveis 1 e 2 do dicionário possuem 256 e 1.024 posições de entradas respectivamente (ver Tabela 6.7). No Gráfico 6.11 mostra-se uma média do tamanho dos programas (em *bytes*, originais e comprimidos) para cada um dos quatro processadores embarcados.

Gráfico 6.11 – Média do tamanho dos programas (originais e comprimidos) para cada processador embarcado

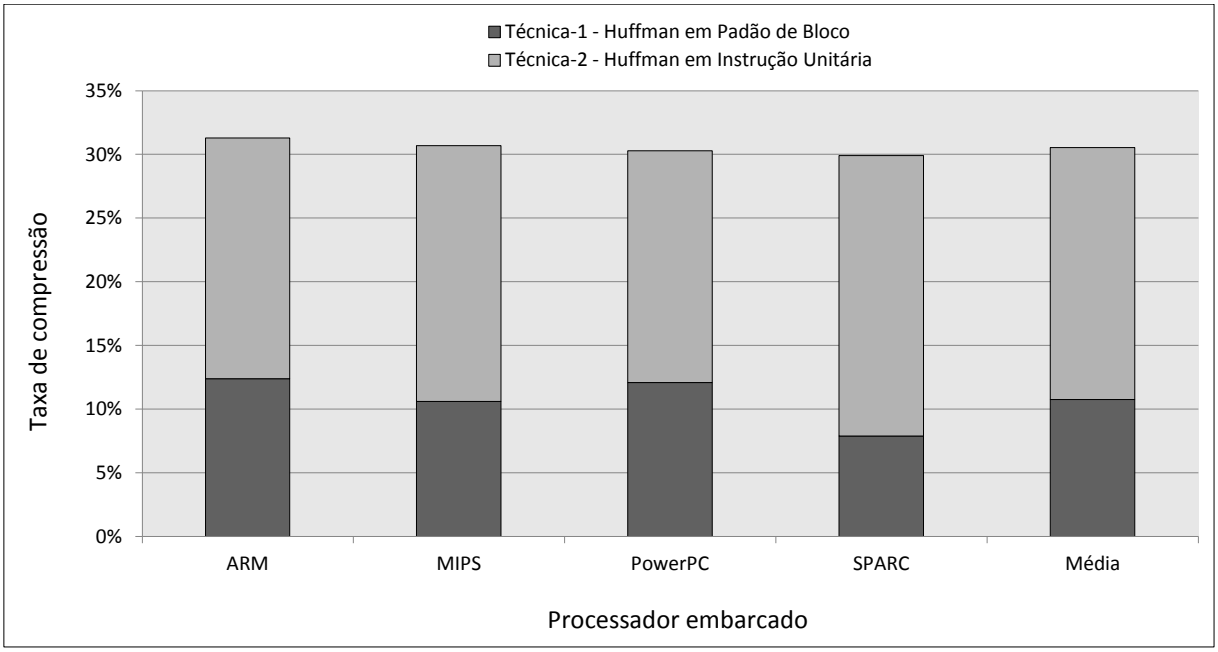


Observa-se no Gráfico 6.11 que em todas as arquiteturas dos processadores embarcados houve uma redução (em *bytes*) significativa no tamanho dos programas do *MiBench* quando aplicados ao processo de compressão, em média 30,6%, ou seja, houve variação do tamanho médio original de 71.589 *bytes* para 49.678 *bytes* comprimido. Comprovando esta afirmação mostra-se detalhadamente no Gráfico 6.12 a taxa de compressão obtida por cada uma das técnicas do método CC-MLD.

Observando o Gráfico 6.12 constata-se que a técnica-1 foi responsável por 10,8% da taxa de compressão obtida pelo método CC-MLD e a técnica-2 por 19,8%. A diferença no tamanho das taxas de compressão obtidas entre uma técnica e a outra foi de apenas 9%, sendo este valor menor do que o valor obtido nas simulações prévias apresentadas no Gráfico 6.1 no

qual apontava indícios para uma possível diferença média de 15%, ou seja, o método CC-MLD mostrou-se ainda mais promissor para o uso na compressão de códigos em sistemas embarcados.

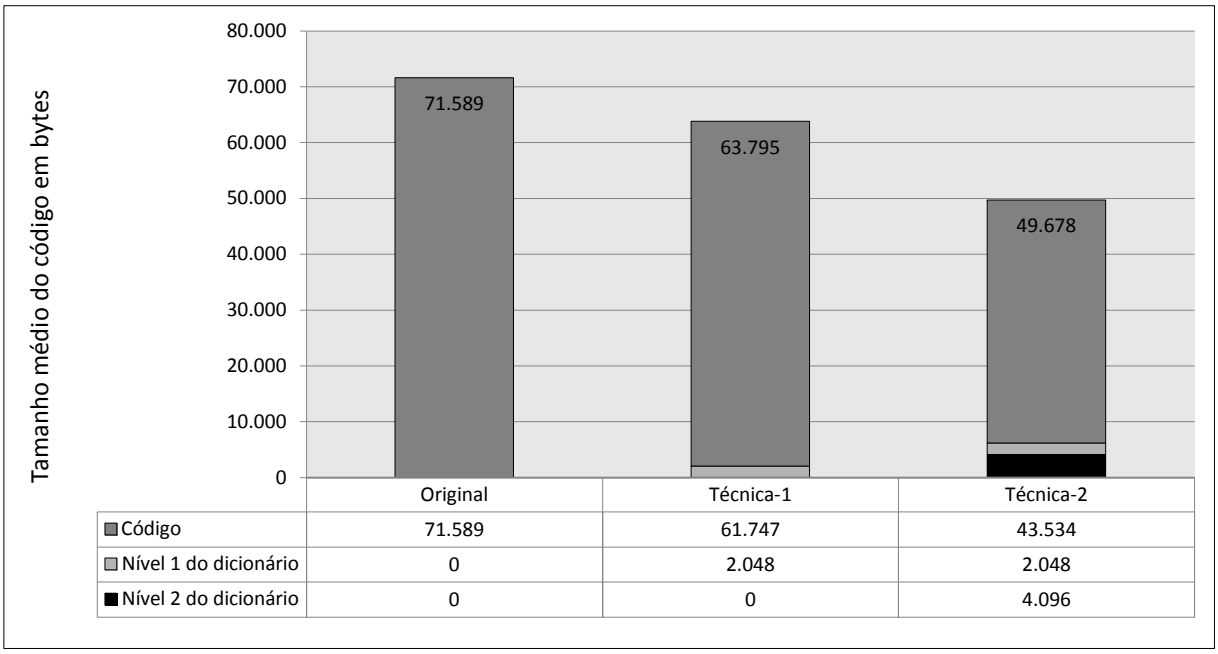
Gráfico 6.12 – Taxas de compressão obtida pelo método CC-MLD



E por fim, o Gráfico 6.13 apresenta um comparativo (para a média dos quatro processadores embarcados) no processo de compressão mostrando o tamanho do código original e também do código comprimido para cada uma das técnicas do método CC-MLD, confirmando assim que a compressão nas duas técnicas obteve resultados satisfatórios.

Analisando o Gráfico 6.13 destaca-se que para o melhor tamanho do dicionário multi-nível usado pelo método CC-MLD, o espaço ocupado pelo dicionário não influencia muito no tamanho final do código comprimido, ou seja, apenas 12,3% no total, sendo que o nível 1 do dicionário corresponde por somente 4,1% e os outros 8,2% pelo segundo nível do dicionário. Já quando comparado com o tamanho do código original o dicionário multi-nível influencia em apenas 8,5%. Portanto, conclui-se que o uso do dicionário multi-nível pelo método CC-MLD trouxe ainda mais benefícios na compressão do código dos programas do *MiBench*.

Gráfico 6.13 – Comparativo das técnicas de compressão do método CC-MLD



	Original	Técnica-1	Técnica-2
Código	71.589	61.747	43.534
Nível 1 do dicionário	0	2.048	2.048
Nível 2 do dicionário	0	0	4.096

6.4.2 Hardware Descompressor do Método CC-MLD

O tempo gasto na descompressão de uma instrução comprimida é considerado crítico, em virtude deste processo ocorrer em tempo de execução (ver Figura 1.1 do Capítulo 1). Então, o hardware descompressor deve ser capaz de fornecer uma instrução descomprimida a cada ciclo do processador para evitar que ocorra *overhead* na execução de um código comprimido. Assim, o projeto do hardware descompressor deve ter sua implementação baseada na execução de apenas um único ciclo.

O hardware descompressor do método CC-MLD foi projetado para atuar como um módulo independente das memórias principal e *cache*. Uma vez que a arquitetura escolhida para ser utilizada pelo método foi a CDM. A Figura 6.6 mostra a arquitetura do hardware descompressor desenvolvida para o método CC-MLD e a Figura 6.7 apresenta a estrutura básica do componente "Lógica do Descompressor".

O processo da descompressão de uma instrução unitária ou de um padrão de bloco é executado da seguinte forma: supondo que a instrução buscada na memória principal do sistema possua 34 *bits* (32 do tamanho da instrução do processador e 2 *bits* de controle para a compressão, sendo que os dois primeiros *bits* são o identificador da instrução (Id), conforme já apresentado nas Seções 6.1 e 6.2). Então, primeiramente a instrução passa pelo componente "Lógica do Descompressor", pois este componente é responsável em identificar se a instrução está ou não comprimida (conforme Figura 6.7).

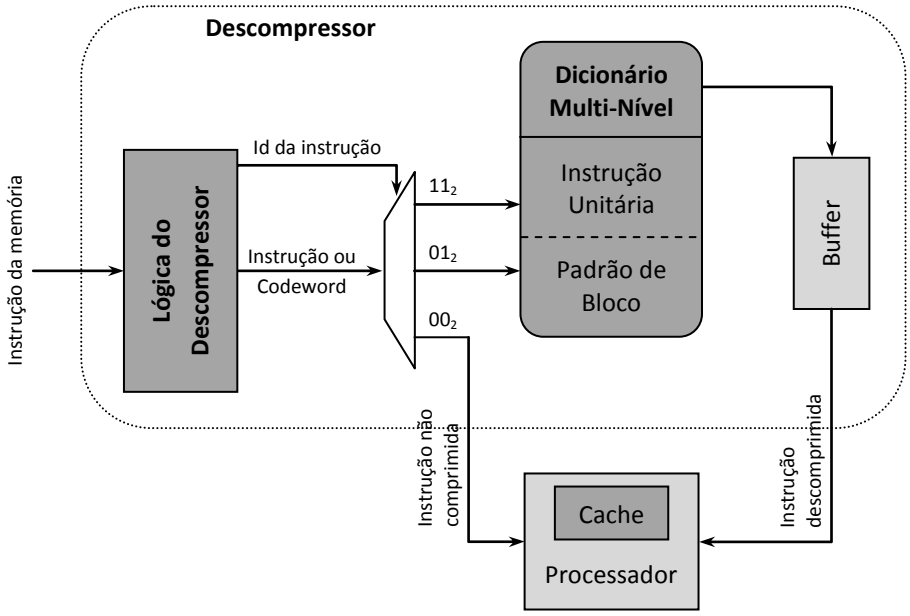


Figura 6.6 – Arquitetura do hardware descompressor do método CC-MLD

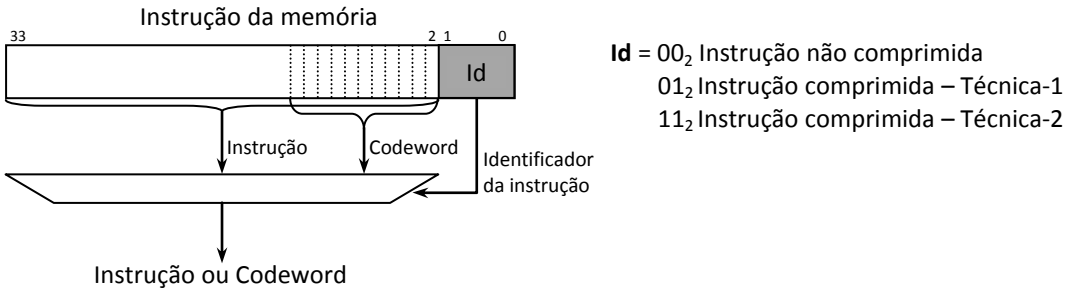


Figura 6.7 – Estrutura básica do componente Lógica do Descompressor

Caso a instrução não esteja comprimida então imediatamente ela é repassada para o processador e a memória *cache*. Agora se a instrução estiver comprimida então o nível correto do "Dicionário Multi-Nível" é acessado diretamente na posição indicada pela *codeword* e assim a instrução unitária ou o padrão de bloco contido nesta posição é repassado para o componente "Buffer" que se encarrega de entregar a instrução para o processador e a memória *cache*. Quando um padrão de bloco é carregado no Buffer, somente após a execução de todas as instruções pertencentes ao padrão de bloco é que um novo acesso à memória principal precisa ser feito (exceto no caso das instruções de desvio). Aqui, é possível visualizar que quanto mais padrões de blocos forem encontrados durante a execução do código, menor será a necessidade de acessos à memória principal do sistema levando assim a um possível aumento no desempenho do sistema e a uma possível redução no consumo de energia. Também é importante destacar que o componente Buffer na arquitetura do hardware descompressor é um elemento fundamental, porque ele evita acessos desnecessários à

memória principal do sistema. A Figura 6.8 mostra o pseudo-código do algoritmo do hardware descompressor do método CC-MLD.

```
InstruçãoDescomprimida DECOMPRESS (InstruçãoDaMemória)
{
  1 Id ← {Conjunto Vazio}
  2 RegInstrução ← {Conjunto Vazio}
  3 Buffer ← {Conjunto Vazio}
  4 Lê a InstruçãoDaMemória, e
    - Salva os dois primeiros bits em Id
    - Salva os outros bits em RegInstrução

  /* Instrução não comprimida */
  5 Se Id for igual a 00₂, então
    - Retorna RegInstrução

  /* Padrão de Bloco comprimido */
  6 Senão, se Id for igual a 01₂, então
    - Extrai a codeword da RegInstrução
    - Busca no dicionário nível 1 a instrução contida na posição codeword
    - Carrega as instruções do padrão de bloco no Buffer
    - Retorna a 1ª e depois a 2ª instrução do Buffer

  /* Instrução Unitária comprimida */
  7 Senão, se Id for igual a 11₂, então
    - Extrai a codeword da RegInstrução
    - Busca no dicionário nível 2 a instrução contida na posição codeword
    - Carrega a instrução unitária no Buffer
    - Retorna a 1ª instrução do Buffer
}
```

Figura 6.8 – Pseudo-código do algoritmo do hardware descompressor do método CC-MLD

Foi implementada uma versão do hardware descompressor do método CC-MLD em software (usando a linguagem C) e inserida no código fonte do simulador *SimpleScalar*. Assim, foi possível obtermos dados arquiteturais dos processadores embarcados tanto na execução de códigos originais quanto comprimidos, o que nos permitiu fazer uma análise comparativa e apontar os benefícios alcançados com o uso da compressão de código em sistemas embarcados. O Gráfico 6.14 mostra o desempenho final do sistema usando o método CC-MLD, onde pode-se constatar que a execução dos códigos comprimidos dos programas do *MiBench* tiveram um *overhead* médio de 4% (para os quatro processadores embarcados). A Tabela 6.8 mostra a média na quantidade de ciclos obtidos na execução dos códigos originais e comprimidos, onde se constata que para os códigos originais é de 70.612.984 ciclos e para os códigos comprimidos 73.884.805 ciclos.

Gráfico 6.14 – Desempenho do sistema usando o método CC-MLD

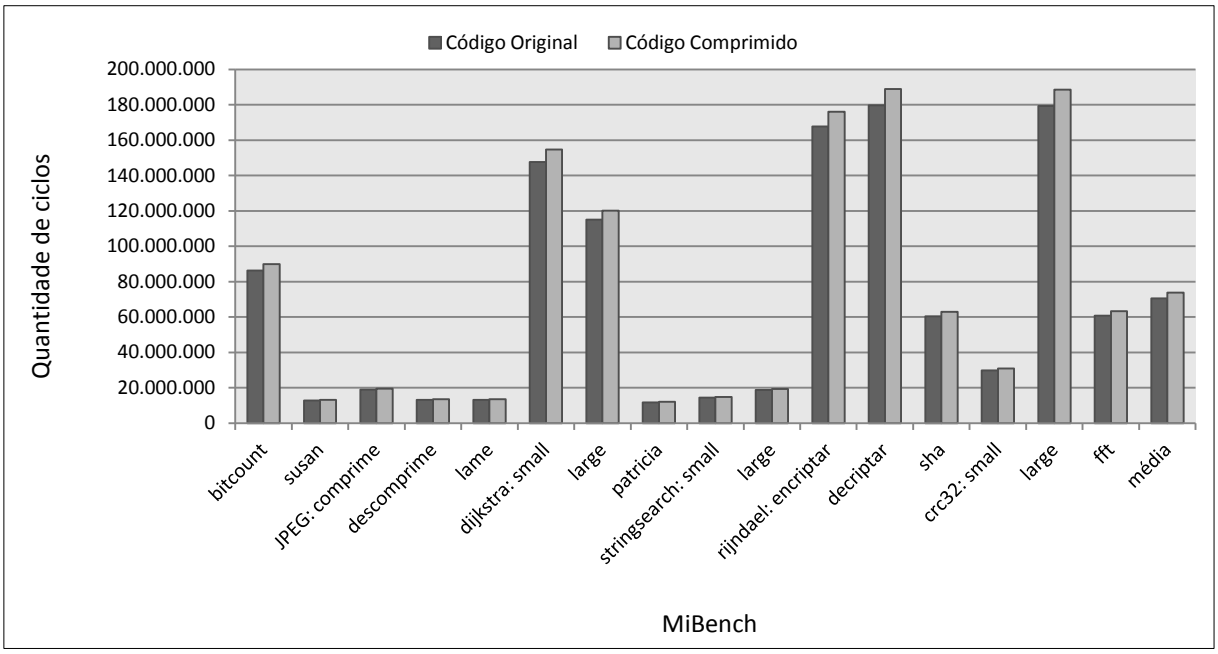


Tabela 6.8 – Quantidade média de ciclos nos programas do MiBench originais e comprimidos

	Quantidade de ciclos		
MiBench	**Original**	**Comprimido**	**Overhead**
bitcount	86.354.874	89.982.781	4,2%
susan	12.777.179	13.187.048	3,2%
JPEG: comprime	18.943.081	19.606.088	3,5%
descomprime	13.151.704	13.586.710	3,3%
lame	13.178.215	13.613.096	3,3%
dijkstra: small	147.605.571	154.691.630	4,8%
large	115.115.882	120.181.980	4,4%
patricia	11.692.354	12.067.515	3,2%
stringsearch: small	14.403.882	14.894.680	3,4%
large	18.730.935	19.387.580	3,5%
rijndael: encriptar	167.724.195	176.111.464	5%
decriptar	179.655.031	188.997.092	5,2%
sha	60.515.333	62.935.961	4%
crc32: small	29.871.448	31.007.563	3,8%
large	179.275.036	188.598.337	5,2%
fft	60.813.024	63.307.357	4,1%
Média	**70.612.984**	**73.884.805**	**4%**

Uma versão do hardware descompressor também foi implementada em VHDL e prototipada em uma FPGA da família *Cyclone-II*, modelo EP2C35F672C6, usou-se a ferramenta *Quartus-II Web Edition* da Altera® [66] para sintetizar o hardware descompressor

do método CC-MLD. A Figura 6.9 mostra os resultados do processo da execução do hardware descompressor.

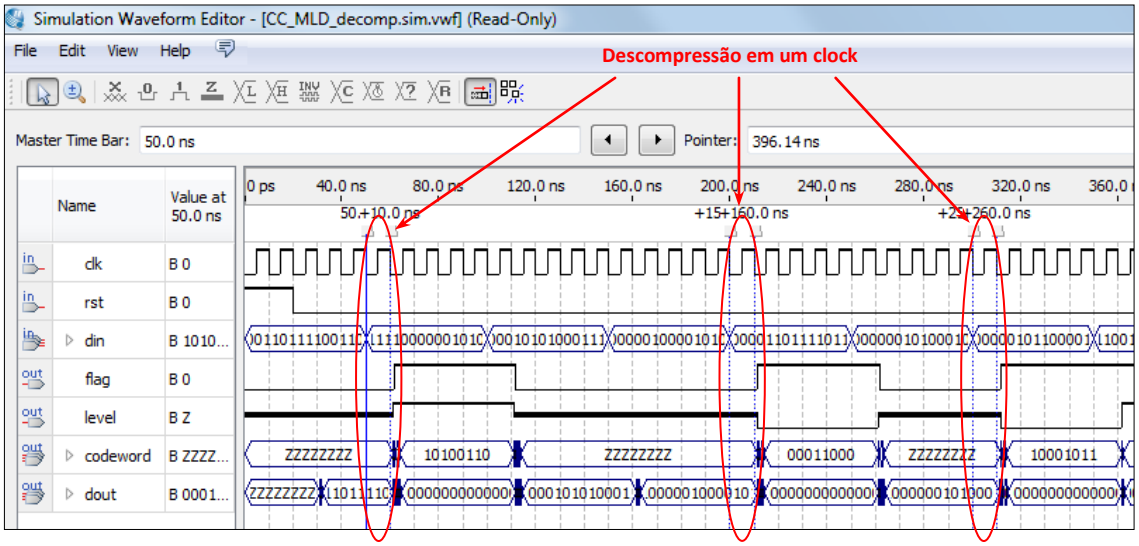


Figura 6.9 – Processo de execução do hardware descompressão do método CC-MLD

Observa-se na Figura 6.9 que o processo de descompressão é executado em apenas um *clock* levando em consideração o tempo do *clock* em 10 *ns*. Comparando este resultado com os resultados apresentados em [51], onde o autor desenvolveu uma ferramenta de comparação e avaliação para diferentes métodos de compressão de código. Em [51] o autor usou vários parâmetros para analisar a compensação entre espaço-tempo-custo e assim mostrou resultados obtidos na execução do hardware descompressor para sete métodos baseados em dicionário (dicionário com tamanho fixo, dicionário usando LAT com tamanho fixo, distância de *Hamming*, múltiplos dicionários para os processadores TI, ARM e MIPS e o método de *Huffman*), quando implementados usando HDL e prototipado em uma FPGA é preciso vários *clocks* para o processo da descompressão. Especificamente, os métodos utilizaram entre 10 a 119 *clocks*. Assim, o hardware descompressor do método CC-MLD, usando como base para a compressão o algoritmo de *Huffman* (em instruções unitárias e padrões de blocos) mostrou-se melhor. O descompressor do método CC-MLD utilizou apenas 1% do total geral de elementos lógicos e registradores da FPGA para diversos tamanhos do dicionário multi-nível, já a quantidade de pinos variou entre 16% e 17% do total presente na FPGA, como mostrado na Tabela 6.9 e para efeito de comparação, a Tabela 6.10 mostra os mesmos recursos ulizados na FPGA usando um dicionário normal pelo método de *Huffman*.

Tabela 6.9 – Recursos da FPGA utilizados pelo método CC-MLD

	Tamanho do dicionário multi-nível								
Componentes da FPGA	**128**	**256**	**512**	**768**	**1024**	**1280**	**1536**	**1792**	**2048**
Total de elementos lógicos	87 < 1%	95 < 1%	97 < 1%	110 < 1%	99 < 1%	113 < 1%	117 < 1%	131 < 1%	101 < 1%
Funções combinacionais	69 < 1%	76 < 1%	76 < 1%	84 < 1%	76 < 1%	87 < 1%	90 < 1%	102 < 1%	76 < 1%
Registradores lógicos dedicados	81 < 1%	83 < 1%	85 < 1%	87 < 1%	87 < 1%	89 < 1%	89 < 1%	89 < 1%	89 < 1%
Total de registradores	81 < 1%	83 < 1%	85 < 1%	87 < 1%	87 < 1%	89 < 1%	89 < 1%	89 < 1%	89 < 1%
Total de pinos	77 < 16%	78 < 16%	79 < 17%	80 < 17%	80 < 17%	81 < 17%	81 < 17%	81 < 17%	81 < 17%

Tabela 6.10 – Recursos da FPGA utilizados pelo método de *Huffman*

	Tamanho do dicionário normal (apenas instruções unitárias)								
Componentes da FPGA	**128**	**256**	**512**	**768**	**1024**	**1280**	**1536**	**1792**	**2048**
Total de elementos lógicos	86 < 1%	94 < 1%	96 < 1%	109 < 1%	98 < 1%	112 < 1%	116 < 1%	130 < 1%	100 < 1%
Funções combinacionais	69 < 1%	76 < 1%	76 < 1%	84 < 1%	76 < 1%	84 < 1%	90 < 1%	102 < 1%	76 < 1%
Registradores lógicos dedicados	81 < 1%	83 < 1%	85 < 1%	87 < 1%	87 < 1%	89 < 1%	89 < 1%	89 < 1%	89 < 1%
Total de registradores	81 < 1%	83 < 1%	85 < 1%	87 < 1%	87 < 1%	89 < 1%	89 < 1%	89 < 1%	89 < 1%
Total de pinos	76 < 16%	77 < 16%	78 < 16%	79 < 17%	79 < 17%	80 < 17%	80 < 17%	80 < 17%	80 < 17%

Observa-se nas Tabelas 6.9 e 6.10 que quando comparamos os recursos da FPGA utilizados pelos métodos CC-MLD e *Huffman,* houve variação apenas nos componentes "elementos lógicos" e "pinos". Conforme já constatado e apresentado no Gráfico 6.13, notamos que os recursos utilizados na FPGA pelo método CC-MLD são praticamente idênticos aos recursos utilizados pelo método de *Huffman*, comprovando assim as vantagens apresentadas pelo uso do dicionário multi-nível.

Conforme resultados obtidos nas simulações usando a ferramenta *PowerPlay Power Analyzer* que está embutida na ferramenta *Quartus-II,* constatamos que o hardware descompressor do método CC-MLD quando executado consome 114.04 *mW* (micro *Watts*) de energia da FPGA e o hardware do método de *Huffman* consome 112.02 *mW*, uma diferença de 1,8%, ou seja, 2.02 *mW*. Assim, verifica-se que em termos de energia o consumo em utilizar um dicionário multi-nível é praticamente igual ao consumo de se usar um dicionário normal. Esse tipo de medida não foi apresentado por outros trabalhos estudados nos Capítulos 2 e 3.

Desta forma, pode-se considerar que o tempo da descompressão de uma instrução unitária ou de um padrão de bloco e o consumo de energia gasto pelo hardware descompressor são quase insignificantes e não geram muito *overhead* para o método CC-MLD (ver Tabela 6.8).

A Figura 6.10 mostra a arquitetura do hardware descompressor do método CC-MLD obtida através da RTL gerada pela ferramenta *Quartus-II*. Portanto, é possível verificarmos e concluirmos com base na Figura 6.10 e com os valores expressos nas Tabelas 6.8 e 6.9 que o hardware descompressor implementado para esse método é simples, econômico (em termos de energia e também dos recursos espaciais usados da FPGA) e ao mesmo tempo eficiente.

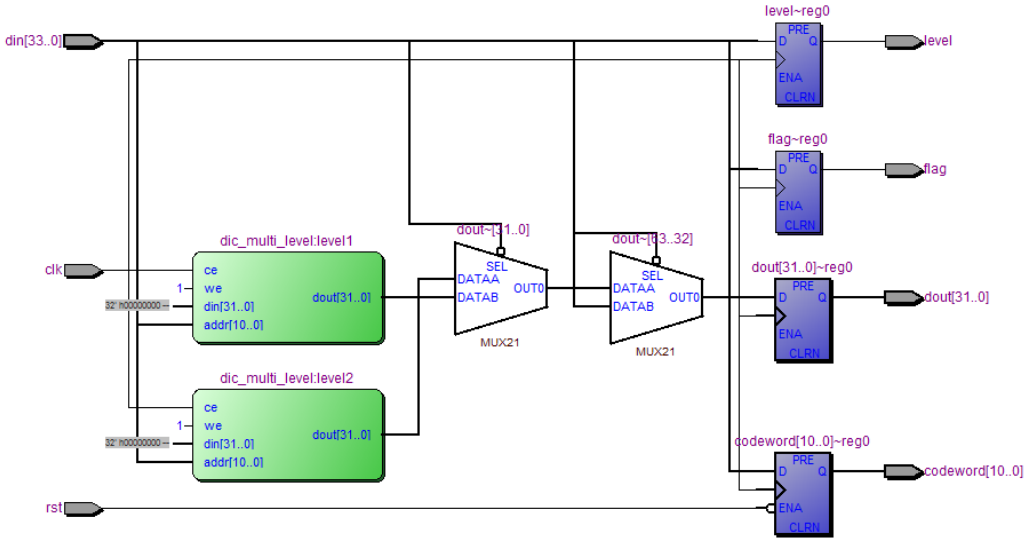


Figura 6.10 – RTL da arquitetura do hardware descompressor do método CC-MLD

6.5 Resumo Comparativo com os Trabalhos Relacionados

Uma das tarefas mais complicadas quando propomos avaliar métodos de compressão de código é a comparação entre os mesmos, uma vez que cada trabalho apresenta um método inovador, usam diversas plataformas e *benchmarks*, além de usarem métricas diferentes.

A Tabela 6.11 mostra uma sumarização comparativa das Tabelas 2.1 e 3.2 após a inclusão dos métodos CPB-ARM, HDPB, CCHPB e CC-MLD. Para comparar a taxa de compressão resultante de cada um dos métodos, é necessário avaliar cuidadosamente como estas foram calculadas, principalmente no que diz respeito aos fatores apresentados na Seção 2.7 do Capítulo 2. Sendo assim, as comparações nessa seção para o método CPB-ARM se restringirá apenas aos métodos que utilizam a arquitetura do processador embarcado ARM e os programas do *MiBench*, e as comparações para os métodos HDPB, CCHPB e CC-MLD são realizadas somente com os métodos que utilizam os processadores embarcados ARM,

MIPS, PowerPC e SPARC aplicando o algoritmo de *Huffman* no processo de compressão, o uso de dicionários e do *benchmark MiBench*.

Tabela 6.11 – Sumarização comparativa entre os métodos CPB-ARM, HDPB, CCHPB, CC-MLD e os principais métodos de compressão de código

	Método	Plataforma	Benchmark	Razão de Compressão	Taxa de Compressão	Desempenho	Consumo de Energia
CAPÍTULO 2	CCRP [62]	MIPS	lex, pswarp, yacc, eightq, matrix25, spim, lloop01, xlisp, espresso	73%	27%	---	---
	PCACHE [32]	SPARC	gcc, go	60%	40%	111%	---
	CodePack [28]	PowerPC	MediaBench, SPEC95	60%	40%	90% a 110%	---
	IBC [2]	MIPS	SPECint95	53,6%	46,4%	88% a 184%	---
		SPARC		63,9%	36,1%	105,9%	
	Instruction Splitting [11]	ARM	MiBench	59%	41%	---	---
		MIPS		60%	40%		
		PowerPC		62%	38%		
	Agrupamento por Características Próprias [37]	SPARC	compress, diesel, i3d, key, mpeg, smo, trick	65%	35%	75%	72%
		Xtensa-1040					---
	Compressão por Linhas de Cache [4]	DLX	Ptolomy	72%	28%	---	70%
	Compressão por Codeword [34]	PowerPC	SPECint95	61%	39%	---	---
		ARM		66%	34%		
		i386		74%	26%		
	PDC-ComPacket [44]	SPARC	MediaBench, MiBench	72% a 88%	12% a 28%	55%	54%
CAPÍTULO 3	Bitmasks Compression [54, 55]	TMS320C6x	MediaBench, MiBench	65%	35%	95%	---
		MIPS		60%	40%		
		SPARC		55%	45%		
	Profiling Agregado [13]	MIPS IV	MediaBench	75%	25%	108% a 112%	---
	DISE [17]	Alpha EV6	SPEC'2000, MediaBench	65% a 75%	25% a 35%	80% a 95%	90%
	Hybrid Compression [25]	ARM	MiBench	80% DicSmall	20% DicSmall	---	---
				91% DicLarge	9% DicLarge		
	Merge Bitmask and DISE [58]	SPARC	MediaBench	80%	20%	---	---
	CCTP [59]	ARM9	MiBench	80% a 85%	15% a 20%	105%	---
	Tripla Otimizações do I-cache [51]	SPARC	EEMBC, MiBench, Powerstone	80%	20%	---	73%
	Multiple Bitstream Compression [50]	TMS320C6x	MediaBench, MiBench	---	---	---	---
		PowerPC					
		SPARC					
		MIPS					
	Dual Code Compression [57]	MIPS	MediaBench, MiBench	60% a 65%	35% a 40%	50%	---
	Two-Level Dictionary Code Compression [16]	MIPS IV	MediaBench	77%	23%	---	79% a 98%
CAPÍTULOS 5 e 6	**CPB-ARM [22]**	**ARM**	**MiBench**	**75,8%**	**24,2%**	**---**	**---**
	HDPB [20]	**ARM**	**MiBench**	**71,5%**	**28,5%**	**---**	**---**
		MIPS		**72,8%**	**27,2%**	**---**	**---**
		PowerPC		**73%**	**27%**	**---**	**---**
		SPARC		**77,9%**	**22,1%**	**---**	**---**
	CCHPB [21, 23]	**ARM**	**MiBench**	**65,7%**	**34,3%**	**---**	**---**
		MIPS		**65,2%**	**34,8%**	**---**	**---**
		PowerPC		**68,2%**	**31,4%**	**---**	**---**
		SPARC		**72,4%**	**27,6%**	**---**	**---**
	CC-MLD	**ARM**	**MiBench**	**68,7%**	**31,3%**	**103,8%**	**---**
		MIPS		**69,3%**	**30,7%**	**104%**	**---**

PowerPC	69,7%	30,3%	103,9%	---
SPARC	70,1%	29,9%	104,3%	---

Primeiramente destacamos que não faremos comparações paralelas entre os métodos apresentados nos Capítulos 2 e 3, uma vez que isso já foi feito no final dos respectivos capítulos. Então, as comparações serão diretamente com os métodos desenvolvidos e apresentados nos Capítulos 5 e 6 dessa tese.

Analisando individualmente o método CPB-ARM, destaca-se que o mesmo apresentou uma taxa de compressão superior a dos métodos *Hybrid Compression* [25] e *CCTP* [59], ficando em média uma diferença de 4,2% a mais para os melhores casos. No entanto, a taxa de compressão obtida pelo método CPB-ARM foi inferior ao do método *Instruction Splitting* [11]. É importante relembrar que o método CPB-ARM é otimizado para comprimir apenas as instruções idênticas repetidas sequencialmente e os padrões de blocos formados pelas instruções das classes *Data Processing* e *Single Data Transfer*, tendo ainda algumas restrições de sufixo. Estas características limitantes não são impostas ao método *Instruction Splitting* que faz uma análise completa em todas as instruções do código e assim comprime sem nenhuma restrição de classe, sufixo e outras, e consequentemente obteve uma maior taxa de compressão.

O sufixo de execução condicional presente nas instruções do processador ARM (ver Figura 4.4), proporcionou ao método CPB-ARM a possibilidade de identificar os padrões de blocos comprimidos sem a necessidade de inserir *bits* extras. Assim, quando o sufixo é checado e constata-se que o seu valor é diferente de 1111_2, imediatamente a instrução é repassada ao processador e a memória *cache* sem a necessidade da descompressão, já que se trata de uma instrução descomprimida, então, é possível diminuir o *overhead* imposto pelo descompressor quando implementado em hardware visto que o mesmo só será ativado quando efetivamente for necessário.

Devido os métodos HDPB, CCHPB e CC-MLD serem praticamente iguais (inclusive com o uso dos mesmos processadores embarcados), ou seja, serem apenas variações na sequência de execução das técnicas, sendo a única diferença mais significativa a ausência da técnica que comprime as *codewords* no método CC-MLD. Então, os mesmos tipos de análises comparativas feitas com os trabalhos relacionados são validas para esses três métodos. Mesmo assim, destaca-se que o método CCHPB alcançou a maior média na taxa de compressão sendo 32% vindo em seguida o método CC-MLD com 30,6% e depois o método HDPB com 26,2%. Assim, comparamos apenas os resultados obtidos pelo método CC-MLD

com os demais trabalhos relacionados, uma vez que o método CC-MLD pode ser considerado como uma otimização dos métodos HDPB e CCHPB.

Então, comparando o método CC-MLD constatam-se alguns pontos importantes. Observando a taxa de compressão obtida para o processador ARM destacamos que a mesma só não foi mais eficiente do que a do método *Instruction Splitting* [11], sendo para os demais métodos uma diferença de até quase 12% para os melhores casos. Já os processadores MIPS e PowerPC apresentaram uma taxa de compressão inferior do que a maioria dos outros métodos. Porém, o que pode ser afirmado é que os métodos *Bitmasks Compression* [54, 55] e *Dual Code Compression* [57] também usaram nas simulações os programas do *MediaBench* que são maiores e por isto podem apresentar uma taxa de compressão melhor. Ainda assim, em ambos os métodos o dicionário usado tinha 2.048 posições de entradas, possibilitando comprimir mais instruções. A diferença média na taxa de compressão foi de 7,6% e 7,7% a menos para os processadores MIPS e PowerPC, respectivamente, usando o método CC-MLD.

Agora quando se analisa os resultados obtidos para o processador SPARC constata-se uma diferença de até quase 18% a mais (para o método CC-MLD) contra a taxa de compressão do método *PDC-ComPacket* [44], já o mesmo não ocorre para o desempenho e o consumo de energia. Comparando com o método *Bitmasks Compression* [54, 55] constata-se que esse usou dicionário e *benchmark* maiores. Como já dito, para *benchmarks* maiores é factível encontrar uma taxa de compressão melhor, assim justificamos o porquê da taxa de compressão maior obtida por esse método.

Baseando-se no processador ARM, verifica-se que o desempenho com a execução do código comprimido obtido pelo método CC-MLD é melhor do que o desempenho alcançado pelo método *CCTP* [59], uma vez que conforme constatado na Figura 3.12 a máquina de descompressão do método *CCTP* é mais complexa do que o descompressor desenvolvido para o método CC-MLD, e por consequência reduz o desempenho do sistema.

Já o desempenho para os processadores MIPS e SPARC quando comparandos com os seus pares mais próximos mostrou-se inferior aos desempenhos alcançados pelos métodos *PDC-ComPacket* [44], *Bitmasks Compression* [54, 55] e *Dual Code Compression* [57], seguindo assim a mesma tendência apresentada com a taxa de compressão. Destacamos que o método *PDC-ComPacket* foi implementado usando a arquitetura PDC e conforme já estudado, esta arquitetura costuma-se apresentar melhores desempenhos em relação aos métodos implementados em arquitetura CDM. O método *Bitmask Compression* [54, 55] utiliza um dicionário com 256 posições de entradas, é importante salientar que quanto menor o tamanho do dicionário usado, maior é a eficiência no desempenho do método. Assim,

analisando de forma geral, ou seja, não levando em consideração a plataforma, o *benchmark* e o tipo de arquitetura usada, destacamos que o método CC-MLD obteve uma média no desempenho melhor do que alguns métodos, tais como: *PCACHE* [32], *CodePack* [28], *IBC* [2], *Profiling Agregado* [13], *CCTP* [59]. Portanto, o baixo *overhead* ocasionado pelo processo de descompressão é compensado pela economia de espaço físico ocupado pelo código comprimido do programa, podendo assim, o sistema embarcado carregar possivelmente em sua memória mais aplicativos simultaneamente.

Também é importante destacar que o dicionário multi-nível (com 1.280 posições de entradas) usado pelo método CC-MLD representou apenas 12,3% do total do código comprimido, e 8,5% quando comparado com o código original. Analisando o método *IBC* [2] constatamos que as médias foram de 24% e 13,5% para os códigos comprimido e original, respectivamente, considerando um dicionário com 1.024 posições de entrada.

Fazendo um confronto direto entre os métodos CC-MLD e *Instruction Splitting* [11], destaca-se que, conforme BONNY & HENKEL [11], a maneira para alcançarem uma taxa de compressão tão expressiva foi através de uma análise realizada exaustivamente pelo algoritmo de compressão nos *bits* individuais de cada uma das instruções do código dos programas para encontrar os padrões repetitivos, e ainda assim usaram uma variação do algoritmo do *Huffman*, ou seja, a Codificação *Canônica* de *Huffman* para comprimir os padrões repetitivos encontrados no código. O método CC-MLD apresentou uma abordagem mais simples e que em média foi inferior em menos de 9% na taxa de compressão para os processadores embarcados ARM, MIPS e PowerPC. Então, ao analisarmos cuidadosamente o método *Instruction Splitting* [11] verifica-se um elevado grau de complexidade do hardware descompressor (ver Figura 2.10 do Capítulo 2) em virtude da técnica desenvolvida, podendo assim muitas vezes até inviabilizar o uso deste método em projeto de sistemas embarcados que empregam técnicas de compressão de código, uma vez que os resultados de desempenho e consumo de energia não foram apresentados pelos autores em [11]. Já o mesmo não ocorre com o método CC-MLD, que apresentou taxas de compressão, desempenho e consumo de energia, relativamente iguais e em muitos casos até melhor do que alguns métodos estudados nos Capítulos 2 e 3, e ainda assim utilizou um hardware descompressor que é simples, economico e eficiente.

Agora quando comparado apenas os resultados obtidos pelo método CC-MLD para os quatro processadores embarcados usados nas simulações, o fato do processador ARM apresentar uma maior taxa de compressão do que as taxas dos processadores MIPS, PowerPC e SPARC são atribuídas a dois importantes aspectos. Primeiro, o dicionário multi-nível tem

um tamanho fixo (1.280 posições de entradas) igual para todos os processadores, então a quantidade de instruções presente no dicionário será sempre igual para todos os processadores, só que a diferença está no fato de que os processadores MIPS e PowerPC possuem mais instruções estáticas no código dos programas do *MiBench* comparado com o processador ARM (ver Tabela 5.1), gerando consequentemente uma taxa de compressão menor. O segundo aspecto é explicado em virtude do processador SPARC conter um grau elevado de instruções do tipo NOP (*No Operation*) [58a] o que reduziu a possibilidade de uma maior frequência de repetição para outros tipos de instruções. Portanto, destaca-se que a diferença média na taxa de compressão para os quatro processadores embarcados usados nas simulações do método CC-MLD ficou entre 0,6% a 1,4% (ver Tabela 6.6, nível 1 igual 256 e nível 2 igual 1.024), mostrando-se assim uma característica relevante de consistência do método CC-MLD na compressão dos códigos dos programas.

Confrontando os métodos CPB-ARM e CC-MLD destaca-se que ambos foram desenvolvidos com intuito de explorar as mesmas características dos programas, ou seja, a presença dos padrões de blocos gerados pela redundância das instruções (conforme já apresentado na Seção 5.1). Embora, a otimização do método CPB-ARM ser apenas para o conjunto de instruções do processador ARM e o método CC-MLD ser totalmente independente das características específicas do conjunto de instruções de qualquer processador, os dois métodos apresentaram resultados equivalentes ao dos métodos estudados nos Capítulos 2 e 3. O método CC-MLD conseguiu uma taxa de compressão (analisando apenas o processador ARM) maior do que o método CPB-ARM, sendo esta uma diferença de 7,1%. A justificativa para esta diferença é que o método CC-MLD não possui quaisquer restrições na análise das instruções a serem comprimidas, o que já não é seguido pelo método CPB-ARM, ou seja, apenas 18 tipos de instruções e ainda condicionadas ao sufixo *always* são analisadas e comprimidas por esse método, causando uma diminuição significativa na probabilidade da quantidade de possíveis instruções candidatas a formação de padrões de blocos para serem comprimidos. Outro ponto que também deve ser levado em consideração é o tamanho dos dicionários usados por cada um desses métodos, sendo que para o método CPB-ARM o dicionário tinha apenas 256 posições de entradas e para o método CC-MLD o dicionário foi de 1.280 posições de entradas.

Observando os métodos dos Capítulos 2 e 3, onde verificou-se uma média geral na taxa de compressão de cada uma das quatro plataformas bases usadas nas simulações dessa tese, independente do tipo de *benchmark* usado, assim, chegamos a tais resultados: (i) para o processador ARM a média geral foi de 28,7%, o que mostra ser superior a taxa obtida pelo

método CPB-ARM, mas ao mesmo tempo inferior a taxa alcançada pelo método CC-MLD; (ii) para o processador MIPS a média foi de 33,7%; (iii) para o processador PowerPC esta média foi de 39% e por último (iv) para o processador SPARC esta taxa foi de 34%. Conclui-se mediante a estes valores expostos que os métodos propostos nessa tese (CPB-ARM, HDPB, CCHPB e CC-MLD) apresentaram resultados compatíveis com os demais resultados dos trabalhos estudados.

No geral, podemos chegar a algumas conclusões para os métodos CPB-ARM, HDPB, CCHPB e CC-MLD, as quais estão descritas a seguir:

- Todos os métodos conseguiram detectar e explorar as redundâncias das instruções nos códigos dos programas e assim formar os padrões de blocos para serem comprimidos;
- Com o uso da arquitetura CDM não é preciso alterar o núcleo do processador, ocasionando assim menos intrusão no desenvolvimento dos projetos de sistemas embarcados;
- Os métodos desenvolvidos nessa tese apresentaram uma simplicidade na complexidade de suas técnicas, bem como no hardware descompressor, podendo ser facilmente desenvolvidas e usadas;
- O uso do dicionário multi-nível pelos métodos HDPB, CCHPB e CC-MLD proporcionou simultaneamente a possibilidade de comprimir e armazenar tanto as instruções unitárias quanto os padrões de blocos;
- Os métodos apresentados no Capítulo 6 mostraram que é possível aumentar a taxa de compressão, mesmo utilizando o algoritmo de *Huffman*, sendo este aplicado em padrões de blocos formado por duas ou mais instruções;
- Os quatro métodos apresentados nessa tese obtiveram taxa de compressão, desempenho e consumo de energia que em alguns casos até suplantaram ao dos demais trabalhos estudados nos Capítulos 2 e 3, provando assim ser métodos tão eficientes quanto aos demais métodos;
- As *codewords* de tamanho fixo usadas pelos métodos desenvolvidos nessa tese, proporcionaram uma simplicidade na lógica do descompressor, fato que não ocorreu nos métodos *CodePack* [28, 29] e *Instruction Splitting* [11].
- Devido aos bons resultados apresentados (via simulação) pelos métodos CPB-ARM, HDPB, CCHPB e CC-MLD, provavelmente será possível obter benefícios reais

quando os mesmos forem empregados em projetos de sistemas embarcados que utilizarem técnicas de compressão de código.

6.6 Considerações Finais do Capítulo

Nesse capítulo apresentamos os métodos HDPB (*Code Compression using Huffman and Dictionary-Based Pattern Blocks*), CCHPB (*Compressed Code using Huffman + Pattern Blocks and Multi-Level Dictionary*) e CC-MLD (*Compressed Code using Huffman-Based Multi-Level Dictionary*) que também foram as contribuições dessa tese. Primeiramente foi apresentado o novo tipo de dicionário múltiplo proposto e desenvolvido nessa tese, chamado de *Dicionário Multi-Nível*, em seguida mostrou-se na Figura 6.2 os passos executados pelos compressores de códigos dos métodos HDPB, CCHPB e CC-MLD. Posteriormente, foram apresentados os detalhes de cada um dos três novos métodos propostos nesse capítulo, bem como suas características, limitações, resultados e as análises das simulações. Ainda destacamos a simplicidade, a economia e a eficiência do hardware descompressor implementado em VHDL e prototipado em uma FPGA para o método CC-MLD.

E por fim, realizou-se uma análise comparativa entre os métodos propostos e desenvolvidos nessa tese e os principais métodos de compressão de código estudados nos Capítulos 2 e 3. Então, concluímos que os métodos HDPB, CCHPB e CC-MLD apresentaram um dos melhores resultados (*circa* maio 2013) quando comparados com os seus pares mais próximos. Obtendo médias nas taxas de compressão de 26,2%, 32% e 30,6% para os métodos HDPB, CCHPB e CC-MLD, respectivamente, mostrando-se serem métodos promissores para sistemas embarcados.

CAPÍTULO 7

CONCLUSÕES E TRABALHOS FUTUROS

O resultado central dessa tese foi o desenvolvimento de quatro novos métodos de compressão de código chamados de CPB-ARM (*ComPatternBlocks-ARM*), HDPB (*Code Compression using Huffman and Dictionary-Based Pattern Blocks*), CCHPB (*Compressed Code using Huffman + Pattern Blocks and Multi-Level Dictionary*) e CC-MLD (*Compressed Code using Huffman-Based Multi-Level Dictionary*), que são direcionados para arquiteturas de processadores RISC, sendo que o método CPB-ARM é dependente do conjunto de instruções do processador ARM e os demais métodos são independentes de quaisquer características específicas do conjunto de instrução dos processadores embarcados, o que os tornaram métodos ortogonais.

O método CPB-ARM foi otimizado para comprimir apenas as instruções idênticas repetidas sequencialmente e os padrões de blocos formados pelas instruções das classes *Data Processing* (com sufixo *always*) e *Single Data Transfer*. Esse método apresentou uma taxa de compressão média de 24,2% para alguns programas do *benchmark MiBench* (ver Tabela 5.1).

Já os métodos HDPB, CCHPB e CC-MLD usaram um novo tipo de dicionário múltiplo chamado de *Dicionário Multi-Nível* que também foi proposto e desenvolvido nessa tese. Com o dicionário multi-nível foi possível armazenar em níveis diferentes do mesmo dicionário as instruções unitárias e as instruções dos padrões de blocos encontradas no código dos programas. Os métodos HDPB, CCHPB e CC-MLD usaram como base comparativa os processadores embarcados ARM, MIPS, PowerPC e SPARC, e as técnicas de compressão desenvolvidas para esses métodos basearam-se no algoritmo de *Huffman*.

Os resultados obtidos nas simulações desses três métodos (HDPB, CCHPB e CC-MLD) usando o processador ARM alcançaram uma das melhores taxas de compressão entre os

métodos estudados nos Capítulos 2 e 3 e os resultados para os demais processadores encontrasse concorrente com taxas de compressão próximas ao dos demais trabalhos correlatos (ver Tabela 6.11).

O melhor tamanho para o dicionário multi-nível usado pelo método HDPB foi quando os níveis 1, 2 e 3 do dicionário tiveram 1.024, 128 e 256 posições de entradas, respectivamente, alcançando 26,2% como média na taxa de compressão. Já para o método CCHPB o melhor tamanho para os níveis 1, 2 e 3 teve 512, 1.024 e 128 posições de entradas, respectivamente, obtendo uma média de 32% na taxa de compressão. E para o método CC-MLD, o dicionário com os níveis 1 e 2 igual a 256 e 1.024 posições de entradas, respectivamente, obteve a melhor taxa de compressão, sendo 30,6%. Todas essas médias foram alcançadas usando os programas do *MiBench* apresentados na Tabela 5.1 do Capítulo 5.

Conforme comprovado nas simulações realizadas nessa tese, o uso das técnicas de compressão de código em sistemas embarcados não apenas podem melhorar o requisito de espaço físico exigido pelos programas, mas também o desempenho do sistema que consequentemente reduz o consumo de energia.

Assim, o desempenho médio alcançado pelo método CC-MLD foi de 104%, gerando um *overhead* de 4% na execução do hardware descompressor implantado no simulador *SimpleScalar*. Quanto ao consumo de energia foi possível reduzir/aumentar em yy% quando executado o código comprimido. O hardware descompressor que foi prototipado em FPGA mostrou-se simples, econômico e eficiente (ver Figuras 6.6 e 6.10). Desta forma, concluímos que é viavel o uso desse método em sistemas embarcados.

Com a utilização das técnicas de compressão de código, a arquitetura RISC consegue minimizar um de seus maiores problemas, que é a quantidade de memória necessária para armazenar os programas. Então, futuramente estas técnicas podem-se tornar um requisito necessário e primordial nos projetos de sistemas embarcados.

Os métodos desenvolvidos nessa tese basearam em uma infraestrutura de software e hardware implementadas nas linguagens C e VHDL que juntam envolveram quase 9.000 linhas de código, sem contabilizar as alterações feitas no simulador *SimpleScalar*.

Pela primeira vez foi apresentado um estudo da quantidade de instruções únicas e repetidas (obtidas de forma estática) dos programas do *MiBench* compilados para os processadores ARM, MIPS, PowerPC e SPARC. Além de, uma análise mais detalhada da quantidade de instruções estáticas e dinâmicas (presentes nos programas do *MiBench*) pertencentes a cada classe do conjunto de instruções do processador ARM.

Ainda destacamos que o grande desafio de um bom método de compressão de código sempre foi ter um mecanismo de compressão eficiente o bastante para reduzir ao máximo o tamanho do código em memória e ao mesmo tempo, um descompressor simples a ponto de influenciar o mínimo possível na área, no tempo de execução do programa e no consumo de energia. Entretanto, observamos na Tabela 6.11 que algumas das propostas estudadas nos Capítulos 2 e 3 tentaram melhorar a taxa de compressão, mas degradaram o desempenho do processador e/ou ainda aumentaram o consumo de energia do sistema ao realizar a descompressão, conforme resultados dos métodos *pcache* [32], *IBC* [2], *profiling agregado* [13] e outros. Portanto, concluímos que o nosso método CC-MLD conseguiu mostrar-se efiente computacionalmente obtendo bons resultados na tríade compressão-desempenho-consumo.

7.1 Publicações

DIAS, W. R. A.; MORENO, E. D. Multi-Level Dictionary Used in Code Compression for Embedded Systems. In Proceedings of the Conference on Data Compression (DCC 2013), Snowbird, Utah, USA, page 487, March 2013.

DIAS, W. R. A.; MORENO, E. D. Code Compression using Multi-Level Dictionary. In Proceedings of 4th IEEE Latin American Symposium on Circuits and Systems (LASCAS 2013), Cusco, Peru, pages 1-4, February 2013.

DIAS, W. R. A.; MORENO, E. D. Code Compression in ARM Embedded Systems using Multiple Dictionaries. In Proceedings of 15th IEEE International Conference on Computational Science and Engineering (CSE 2012), Paphos, Cyprus, pages 209-214, December 2012.

DIAS, W. R. A.; MORENO, E. D. CPB-ARM - A New Code Compression Method for Embedded Systems. In Proceedings of 13th Symposium on High Performance Computing Systems (WSCAD-SSC 2012), Petropolis, Rio de Janeiro, Brazil, pages 25-32, October 2012.

7.2 Trabalhos Futuros

Embora os métodos CPB-ARM, HDPB, CCHPB e CC-MLD já obtenham uma das maiores taxas de compressão para os processadores ARM, MIPS, PowerPC e SPARC (*circa* maio 2013) quando comparados com os seus pares próximos. Alguns aspectos ainda devem ser mais bem estudados, podendo vir a serem melhorados futuramente, a saber:

- A implementação do hardware descompressor dos métodos CPB-ARM, HDPB e CCHPB nas Linguagens C e VHDL;

- A implantação e simulação dos descompressores (implementados em C) no simulador *SimpleScalar*, bem como a prototipação (implementado em VHDL) em FPGA;
- Para o método CPB-ARM desenvolver novas técnicas de compressão especifica para as demais classes do processador ARM;
- Testar a compressão e descompressão de códigos usando outros pacotes de programas de *benchmark* para os métodos desenvolvidos nessa tese;
- Analisar o comportamento dos métodos CPB-ARM, HDPB, CCHPB e CC-MLD usando a arquitetura PDC (*Processor Decompressor Cache*);
- Melhorar a forma de identificação das instruções normais ou comprimidas pelos métodos desenvolvidos no Capítulo 6, onde acreditamos que é possível conseguir um aumento em aproximadamente 3,5% na taxa de compressão, uma vez que as instruções passaram a ter 34 *bits*. Uma alternativa é usar os *bits* não utilizados nas instruções;
- Uma forma melhor de codificação para as instruções e/ou *codewords* geradas a partir dos métodos HDPB, CCHPB e CC-MLD, de forma que fiquem armazenadas alinhadamente nas memórias principal e *cache*;
- Analisar a tríade compressão-desempenho-consumo dos métodos CPB-ARM, HDPB, CCHPB e CC-MLD usando instruções obtidas de forma dinâmica;
- Chegar a uma implementação em ASIC do hardware descmpressor dos métodos desenvolvidos nessa tese, de modo que o trabalho transcendesse o âmbito acadêmico, servindo como uma contribuição, também, para o meio industrial.

REFERÊNCIAS

[1] ARM. ARM-7TDMI Data Sheet. Advanced RISC Machines (ARM) Ltd., August 1995.

[2] AZEVEDO, R. J. de. Uma Arquitetura para Código Comprimido em Sistemas Dedicados. Tese de Doutorado, Instituto de Computação, Universidade Estadual de Campinas, Junho de 2002.

[3] BELL, T. C.; CLEARY, J. G.; WITTEN, I. H. Text Compression. – Englewood Cliffs, New Jersey, USA: Prentice-Hall, 1990. 318p.

[4] BENINI, L.; MACII, A.; NANNARELLI, A. Cached-Code Compression for Energy Minimization in Embedded Processor. In Proceedings of the International Symposium on Low-Power, Electronics and Design (ISLPED'01), Huntington Beach, California, USA, pages 322-327, August 2001.

[5] BENINI, L.; MACII, A.; NANNARELLI, A. Code Compression Architecture for Cache Energy Minimization in Embedded Systems. IEEE Proceedings on Computers and Digital Techniques, 149(4):157-163, July 2002.

[6] BENTLEY, J. L.; MCGEOCH, C. Amortized Analyses of Self-Organizing Sequential Search Heuristics. Communications of the ACM, 28(4):404-411, April 1985.

[7] BESZÉDES, Á.; FERENC, R.; GYIMÓTHY, T. Survey of Code-Size Reduction Methods. ACM Computing Surveys (CSUR), 35(3):223-267, September 2003.

[8] BISHOP, B.; KELLIHER, T. P.; IRWIN, M. J. A Detailed Analysis of MediaBench. In Proceeding of Workshop on Signal Processing Systems (SIPS'99), Taipei, Taiwan, pages 448-455, October 1999.

[9] BODDEN, E.; CLASEN, M.; KNEIS, J. Arithmetic Coding Revealed - A Guided tour from Theory to Praxis. Sable Research Group, Technical Report Nº 2007-5, School of Computer Science, McGill University, Montréal, Québec, Canada, May 25, 2007.

[10] BONNY, T.; HENKEL, J. Efficient Code Density Through Look-up Table Compression. In Proceeding Design Automation and Test in Europe Conference (DATE'07), Nice Acropolis, France, pages 809-814, April 2007.

[11] BONNY, T.; HENKEL, J. Instruction Splitting for Efficient Code Compression. In Proceedings of the 44th Annual Design Automation Conference (DAC'07), San Diego, California, USA, pages 646-651, June 2007.

[12] BONNY, T.; HENKEL, J. Using Lin-Kernighan Algorithm for Look-up Table Compression to Improve Code Density. In Proceedings of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI'06), Philadelphia, Pennsylvania, USA, pages 259-265, April 2006.

[13] BRORSSON, M.; COLLIN, M. Adaptive and Flexible Dictionary Code Compression for Embedded Applications. In Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06), Seoul, Korea, pages 113-124, October 2006.

[14] BURGER, D; AUSTIN, T. M. The Simplescalar Tool Set, version 2.0. ACM SIGARCH Computer Architecture News, 25(3):13-25, June 1997.

[15] CARRO, L. Projeto e Prototipação de Sistemas Digitais. – Porto Alegre, Brasil: Editora Universidade/UFRGS, 2001, 171p.

[16] COLLIN, M.; BRORSSON, M. Two-Level Dictionary Code Compression: a New Scheme to Improve Instruction Code Density of Embedded Applications. In Proceedings of 17th International Symposium on Code Generation and Optimization (CGO'09), Seattle, Washington, USA, pages 231-242, March 2009.

[17] CORLISS, M. L.; LEWIS, E. C.; ROTH, A. A DISE Implementation of Dynamic Code Decompression. In Proceedings of the ACM SIGPLAN Conference on Language, Compiler and Tool for Embedded Systems (LCTES'06), San Diego, California, USA, pages 232-243, June 2003.

[18] COSTA, C. da. Projetando Controladores Digitais com FPGA. – São Paulo, Brasil: Novatec Editora, 2006, 159p.

[19] DAVIS II, J.; GOEL, M.; HYLANDS, C.; KIENHUIS, B.; LEE, E. A.; LIU, J.; LIU, X.; MULIADI, L.; NEUENDORFFER, S.; REEKIE, J.; SMYTH, N.; TSAY, J.; XIONG, Y. Overview of the Ptolemy Project, ERL Technical Memorandum UCB/ERL, Technical Report Nº M-99/37, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California, USA, July 6, 1999.

[20] DIAS, W. R. A.; MORENO, E. D. Code Compression in ARM Embedded Systems using Multiple Dictionaries. In Proceedings of 15th IEEE International Conference on Computational Science and Engineering (CSE 2012), Paphos, Cyprus, pages 209-214, December 2012.

[21] DIAS, W. R. A.; MORENO, E. D. Code Compression using Multi-Level Dictionary. In Proceedings of 4th IEEE Latin American Symposium on Circuits and Systems (LASCAS 2013), Cusco, Peru, pages 1-4, February 2013.

[22] DIAS, W. R. A.; MORENO, E. D. CPB-ARM - A New Code Compression Method for Embedded Systems. In Proceedings of 13th Symposium on High Performance

Computing Systems (WSCAD-SSC 2012), Petropolis, Rio de Janeiro, Brazil, pages 25-32, October 2012.

[23] DIAS, W. R. A.; MORENO, E. D. Multi-Level Dictionary Used in Code Compression for Embedded Systems. In Proceedings of the Conference on Data Compression (DCC 2013), Snowbird, Utah, USA, page 487, March 2013.

[24] ELIAS, P. Universal Codeword Sets and Representation of Integers. IEEE Transaction on Information Theory, 21(2):194-203, March 1975.

[25] ELF. Tool Interface Standard - Executable and Linking Format Specification version 1.2. TIS Committee, May 1995.

[26] FITZPATRICK, J. An Interview with Steve Furber. Communications of the ACM, 54(5):34-39, May 2011.

[27] FREY, B. PowerPC Architecture Book, Version 2.02. PowerPC Architect, IBM, 2005, http://www.ibm.com/developerworks/eserver/library/es-archguide-v2.html.

[28] FURBER, S. ARM System-on-Chip Architecture. Addison-Wesley Professional, 2nd edition, 2000, 432p.

[29] GUTHAUS, M.; RINGENBERG, J.; ERNST, D.; AUSTIN, T.; MUDGE, T.; BROWN, R. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In Proceedings of the IEEE 4th Annual Workshop on Workload Characterization (WWC-4), Austin, Texas, USA, pages 3-14, December 2001.

[30] HAIDER, S. I.; NAZHANDALI, L. A Hybrid Code Compression Technique using Bitmask and Prefix Encoding with Enhanced Dictionary Selection. In Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'07), Salzburg, Austria, pages 58-62, September 2007.

[31] HENNESSY, J. L.; PATTERSON, D. A. Computer Architecture: A Quantitative Approach. – San Francisco, California, USA: Morgan Kaufmann Publishers, 4th edition, 2006, 704p.

[32] HUFFMAN, D. A. A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the Institute of Radio Engineers (IRE), 40(9):1098-1101, September 1952.

[33] IBM. CodePack: PowerPC Code Compression Utility User's Manual, version 4.1. International Business Machines (IBM) Corporation, March 2001.

[34] KEMP, T. M.; MONTOYE, R.; HARPER, J.; PALMER, J.; AUERBACH, D. A Decompression Core for PowerPC. IBM Journal of Research and Development, 42(6):807-812, September 1998.

[35] KERNIGHAN, B. W.; RITCHIE, D. M. The C Programming Language. – Englewood Cliffs, New Jersey: Prentice-Hall, 2nd edition, 1988, 274p.

[36] KLEIN, S. Space and Time-Efficient Decoding with Canonical Huffman Trees. In Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM 1997), Aarhus, Denmark, pages 65-75, June 1997.

[37] KILLIAN, E.; WARTHMAN, F. Xtensa® Instruction Set Architecture (ISA) Reference Manual. Tensilica, Inc., 2001.

[38] KIROVSKI, D.; KIN, J.; MANGIONE-SMITH, W. Procedure Based Program Compression. In Proceedings of 30th Annual International Symposium on Microarchitecture (MICRO 30), Research Triangle Park, North Carolina, USA, pages 457-466, December 1997.

[39] KOZUCH, M.; WOLFE, A. Compression of Embedded System Programs. In Proceedings of the IEEE International Conference on Computer Design (ICCS'94), VLSI in Computer & Processors, Cambridge, Massachusetts, USA, pages 270-277, October 1994.

[40] LEFURGY, C.; BIRD, P.; CHEN, I-C.; MUDGE, T. Improving Code Density Using Compression Techniques. In Proceedings of 30th Annual International Symposium on Microarchitecture (MICRO 30), Research Triangle Park, North Carolina, USA, pages 194-203, December 1997.

[41] LEFURGY, C.; PICCININNI, E.; MUDGE, T. Evaluation of a High Performance Code Compression Method. In Proceedings of 32nd Annual International Symposium on Microarchitecture (MICRO 32), Haifa, Israel, pages 93-102, November 1999.

[42] LEKATSAS, H.; HENKEL, J.; JAKKULA, V. Design of One-Cycle Decompression Hardware for Performance Increase in Embedded Systems. In Proceedings of the 39th Annual Design Automation Conference (DAC'02), New Orleans, Louisiana, USA, pages 34-39, June 2002.

[43] LEKATSAS, H.; HENKEL, J.; WOLF, W. Code Compression for Low Power Embedded System Design. In Proceedings of the 37th Annual Design Automation Conference (DAC'00), Los Angeles, California, USA, pages 294-299, June 2000.

[44] LEKATSAS, H.; HENKEL, J.; WOLF, W. Design and Simulation of a Pipelined Decompression Architecture for Embedded Systems. In Proceedings of the 14th International Symposium on Systems Synthesis (ISSS'01), Montréal, Quebec, Canada, pages 63-68, October 2001.

[45] LEKATSAS, H.; WOLF, W. Code Compression for Embedded Systems. In Proceedings of the 35th Annual Design Automation Conference (DAC'98), San Francisco, California, USA, pages 516-521, June 1998.

[46] LELEWER, D. A.; HIRSCHBERG, D. S. Data Compression. ACM Computing Surveys (CSUR), 19(3):261-296, September 1987.

[47] LIAO, S.; DEVADAS, S.; KEUTZER, K. Code Density Optimizations for Embedded DSP Processors using Data Compression Techniques. In Proceedings of the 16th Conference on Advanced Research in VLSI (ARVLSI'95), pages 272-285, March 1995.

[48] MARWEDEL, P. Embedded System Design. – New York, USA: Springer-Verlag, 2nd edition, 2006, 241p.

[49] MAY, C.; SIKHA, E.; SIMPSON, R. The PowerPC Architecture: A Specification for a New Family of RISC Processors. – San Francisco, California, USA: Morgan Kaufmann Publishers, 2nd edition, 2004, 518p.

[50] MICHELI, G. de. Computer-Aided Hardware-Software Codesign. IEEE Micro, 14(4):10-16, August 1994.

[51] MENON, S. K. An Efficient Tool-Chain for Analyzing Tradeoffs of Code Compression Schemes in Embedded Processors. In Proceedings of 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2012), Seoul, Korea, pages 192-201, August 2012.

[52] NEKRITCH, Y. Decoding of Canonical Huffman Codes with Look-up Tables. In Proceedings of the Conference on Data Compression (DCC'00), Snowbird, Utah, USA, page 566, March 2000.

[53] NETTO, E. B. W. Compressão de Código Baseada em Multi-Profile. Tese de Doutorado, Instituto de Computação, Universidade Estadual de Campinas, Maio de 2004.

[54] NETTO, E. B. W.; AZEVEDO, R.; CENTODUCATTE, P.; ARAÚJO, G. Mixed Static/Dynamic Profiling for Dictionary Based Code Compression. In Proceedings of the International Symposium on System-on-Chip (SoC'03), Tampere, Finland, pages 159-163, November 2003.

[55] NETTO, E. B. W.; AZEVEDO, R.; CENTODUCATTE, P.; ARAÚJO, G. Multi-Profile Based Code Compression. In Proceedings of the 41th Annual Design Automation Conference (DAC'04), San Diego, California, USA, pages 244-249, June 2004.

[56] NETTO, E. B. W.; OLIVEIRA, R. S. de; AZEVEDO, R.; CENTODUCATTE, P. Compressão de Código em Sistemas Embarcados. HOLOS CEFET-RN. Natal, Ano 19, páginas 23-28, Dezembro, 2003. 94p.

[57] OLIVEIRA, A. S. de; ANDRADE, F. S. de. Sistemas Embarcados - Hardware e Firmware na Prática. – São Paulo, Brasil: Editora Érica, 2006, 316p.

[58] ORDOÑEZ, E. D. M. Caches Remotos e Prefetching em Sistemas Multiprocessadores de Alto Desempenho - Considerações Arquiteturais. Tese de Doutorado, Departamento de Engenharia Eletrônica, Escola Politécnica da Universidade de São Paulo, Agosto de 1998.

[59] PERRY, D. L. VHDL - Programming by Example. McGraw-Hill Professional, 4th edition, 2002, 476p.

[60] QIN, X.; MISHRA P. A Universal Placement Technique of Compressed Instructions for Efficient Parallel Decompression. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 28(8):1224-1236, August 2009.

[61] RAWLINS, M.; GORDON-ROSS, A. On the Interplay of Loop Caching, Code Compression, and Cache Configuration. In Proceedings 16th Asia and South Pacific - Design Automation Conference (ASP-DAC'11), Yokohama, Japan, pages 243-248, January 2011.

[62] SAYOOD, K. Introduction to Data Compression. – San Francisco, California, USA: Morgan Kaufmann Publishers, 3rd edition, 2005, 704p.

[63] SEAL, D. ARM Architecture Reference Manual. Addison-Wesley Professional, 2nd edition, 2001, 816p.

[64] SEONG, S-W.; MISHRA, P. Bitmask-based Code Compression Technique for Embedded Systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 27(4):673-685, April 2008.

[65] SEONG, S-W.; MISHRA, P. A Bitmask-based Code Compression Technique for Embedded Systems. In Proceedings of the International Conference on Computer-Aided Design (ICCAD'06), San Jose, California, USA, pages 251-254, November 2006.

[66] SHANNON, C. E. A Mathematical Theory of Communication. The Bell System Technical Journal, 27(1):379-423, 623-656, July and October 1948.

[67] SHRIVASTAVA, K.; MISHRA, P. Dual Code Compression for Embedded Systems. In Proceedings of 24th International Conference on VLSI Design (VLSID), IIT Madras, Chennai, India, pages 177-182, January 2011.

[68] SPARC. The SPARC Architecture Manual - version 8. SPARC International, Inc., December 1992.

[69] SWEETMAN, D. See MIPS Run. – San Francisco, California, USA: Morgan Kaufmann Publishers, 2nd edition, 2006, 512p.

[70] THURESSON, M.; STENSTROM, P. Evaluation of Extended Dictionary-Based Static Code Compression Schemes. In Proceedings of 2nd Conference on Computing Frontiers (CF'05), Ischia, Italy, pages 77-86, May 2005.

[71] XIE, Y.; WOLF, W.; LEKATSAS, H. Compression Ratio and Decompression Overhead Tradeoffs in Code Compression for VLIW Architectures. In Proceedings of the 4th International Conference on ASIC (ASICON 2001), Shanghai, China, pages 337-341, October 2001.

[72] XU, X. H.; CLARKE, C. T.; JONES, S. R. High Performance Code Compression Architecture for the Embedded ARM/THUMB Processor. In Proceedings of the First Conference on Computing Frontiers (CF'04), Ischia, Italy, pages 451-456, April 2004.

[73] WILNER, W. T. B1700 Memory Utilization. In Proceeding Fall Joint Computer Conference (AFIPS'72), Part I, Anaheim, California, USA, pages 579-586, December 1972.

[74] WOLF, W. H. Hardware-Software Co-Design of Embedded Systems. Proceedings of the IEEE, 82(7):967-989, July 1994.

[75] WOLFE, A.; CHANIN, A. Executing Compressed Programs on an Embedded RISC Architecture. In Proceedings of 25th Annual International Symposium on Microarchitecture (MICRO 25), Portland, Oregon, USA, pages 81-91, December 1992.

[76] YIU, J. The Definitive Guide to the ARM Cortex-M3. Newnes, 2nd edition, 2009, 479p.

[77] ZIV, J.; LEMPEL, A. A Universal Algorithm for Sequential Data Compression. IEEE Transaction on Information Theory, 23(3):337-343, May 1977.

[78] ZIV, J.; LEMPEL, A. On the Complexity of Finite Sequences. IEEE Transaction on Information Theory, 22(1):75-81, January 1976.

[79] ALTERA® Corporation. Disponível em: http://www.altera.com. Acessado em 01 de dezembro de 2010.

[80] EEMBC® EDN Embedded Microprocessor Benchmark Consortium. Disponível em: http://www.eembc.org. Acessado em 02 de Fevereiro de 2011.

[81] MIPS® Technologies. Disponível em: http://www.mips.com. Acessado em 30 de novembro de 2010.

[82] SPEC© Corporation. Disponível em http://www.spec.org. Acessado em 03 de abril de 2011.

[83] SIMPLESCALAR© LLC. Disponível em http://www.simplescalar.com. Acessado em 30 de novembro de 2010.

[84] XILINX©, Inc. Disponível em: http://www.xilinx.com. Acessado em 30 de novembro de 2010.