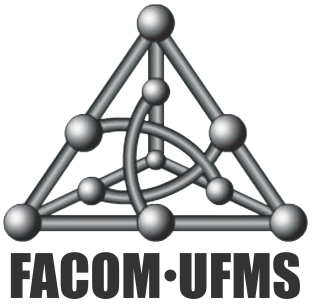
Dissertação de Mestrado

# PBIW-SPARC: Uma Estratégia de Codificação de Instruções para Programas SPARC

Renato Fernando dos Santos

Orientação:Prof. Dr. Ricardo Ribeiro dos Santos

Área de Concentração: Sistemas Computacionais




Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
21 de junho de 2013.

Pesquisa desenvolvida com auxílio financeiro da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) e Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) por meio de bolsa de mestrado.

PBIW-SPARC: Uma Estratégia de
Codificação de Instruções para
Programas SPARC

Campo Grande, 21 de junho de 2013.

Banca Examinadora:

- Prof. Dr. Ricardo Ribeiro dos Santos (FACOM/UFMS) - orientador
- Prof. Dr. Edward David Moreno Ordonez (DCOMP/UFS)
- Prof. Dr. Milton Ernesto Romero Romero (FAENG/UFMS)
- Prof. Dr. Luciano Gonda (FACOM/UFMS)

# Agradecimentos

A Deus, pelas oportunidades que me deu na vida.

Ao meu orientador, Profº. Dr. Ricardo Ribeiro dos Santos, pela confiança depositada ao longo desses dois anos, em que mesmo nos momentos mais difíceis, não deixou de acreditar em mim. Agradeço pelo exemplo de postura acadêmica como professor e como orientador, certamente levarei esse exemplo para minha carreira e vida.

Aos meus colegas e amigos do LSCAD, Richard Stéffano, Lucas Silva, Felipe Yonehara, Felipe Oliveira, Marcel Grassi, Márcio Grassi e principalmente ao grande parceiro Renan Marks, que proporcionaram um ambiente harmônico de trabalho, no qual todos sempre foram muito solícitos, aprendi muito com todos vocês.

Aos professores que fizeram parte de minha formação, pois sem eles não seria possível ter chegado até aqui.

Aos meus pais Raimundo e Magali, por terem paciência comigo nos momentos de dificuldade e pelo apoio incondicional, pela criação que me deram, pois foi fundamental na trajetória.

A minha noiva Ingrid, por suportar a distância, a ausência, me apoiar nos momentos mais dificeis e por amenizar os muitos momentos de solidão.

A minha avó Rosa Gasparoto (in memoriam) e meu avô Belino Giroto (in memoriam).

Ao IFMS Câmpus Coxim e colegas, por compreender a necessidade de me ausentar para comparecer as aulas e reuniões do mestrado.

Finalmente, agradeço à CAPES e CNPq pelo apoio financeiro.

# Resumo

Este trabalho apresenta o projeto e implementação da técnica de codificação de instruções PBIW (*Pattern Based Instruction Word*), baseada em fatoração de padrões, sobre o conjunto de instruções SPARCv8. A técnica PBIW é implementada sobre uma infraestrutura de codificação de instruções que mapeia o código de saída de um compilador para o esquema de codificação PBIW em um processador alvo. Na codificação PBIW-SPARC, as instruções passam a ter o tamanho de 16 bits e novos padrões de instruções possuem tamanho de 24 bits. Mesmo com a sobrecarga dos padrões, pode-se notar que o tamanho dos programas diminui significativamente, gerando, inclusive, impactos no desempenho final dos programas codificados. A fim de possibilitar o endereçamento a uma quantidade maior de padrões, extendeu-se a codificação PBIW-SPARC para instruções de 24 bits. Essa extensão possibilitou codificar programas dos benchmarks MediaBench e MiBench. Os experimentos realizados visam caracterizar detalhadamente a técnica de codificação PBIW-SPARC sobre os efeitos gerados sobre o código do programa e sobre o processador alvo. Os resultados demonstram que a técnica PBIW-SPARC oferece ganhos significativos tanto na compressão do tamanho programa (resultados estáticos) - 42% de redução do código - quanto no desempenho do código final (resultados dinâmicos) - *speedup* de 2,06 sobre o código original SPARCv8. Os resultados também permitem notar que a utilização da técnica PBIW-SPARC oferece oportunidades interessantes para exploração do espaço de projeto de decodificadores de código junto à via de dados e controle do processador alvo.

**Palavras-chave:** Codificação de Instruções, PBIW, Decodificadores de Instrução, SPARCv8.

# Abstract

This work presents the design and implementation of the PBIW (Pattern Based Instruction Word), based on pattern factorization, instruction encoding technique on the SPARCv8 instruction set architecture. The PBIW encoding technique is implemented on the top of an instruction encoding software infrastructure that maps the output generated code from a compiler into the PBIW encoding scheme designed for a target processor. PBIW-SPARC encoded instructions have 16 bits size and encoding patterns have 24 bits size. Even with the pattern size overhead, encoding programs have a significant size reduction. The size reduction provides impacts on the final performance of encoded programs. In order to address more patterns, the PBIW-SPARC instruction encoding has been extended to 24 bits (16 bits of pattern index). The 24 bits encoding instructions encode programs of MediaBench and MiBench benchmarks. Static and dynamic experiments have been performed in order to characterize the effects of the PBIW-SPARC technique on the generated code and on the target processor. The results show that PBIW provides significant gains of compression ratio, 40% of program size reduction, and performance, 2.06 speedup over SPARCv8 programs, on the final generated program. The results also show that PBIW-SPARC offers opportunities to explore the design space for instruction decoders on data and control path of a target processor.

**Keywords:** Instruction Encoding, PBIW, Instruction Decoders, SPARCv8.

# Sumário

# Siglas

**AHB** *Advanced High-performance Bus*.

**AMBA** *Advanced Microcontroller Bus Architecture*.

**ARM** *Advanced RISC Machine*.

**CWP** *Current Window Pointer*.

**ELF** *Executable and Linkable Format*.

**EPIC** *Explicitly Parallel Instruction Computing*.

**FPGA** *Field-Programmable Gate Array*.

**GNU GPL** *GNU General Public License*.

**I-Cache** *Instruction cache*.

**ISA** *Instruction Set Architecture*.

**LRR** *Least Recently Replaced*.

**LRU** *Least Recently Used*.

**MIPS** *Microprocessor without Interlocked Pipeline Stages*.

**MMU** *Memory Manegement Unit*.

**nPC** *next Program Counter*.

**PBIW** *Pattern Based Instruction Word*.

**PC** *Program Counter*.

**P-Cache** *Pattern cache*.

**PSR** *Processor State Register*.

**RISC** *Reduced Instruction Set Computer*.

**SPARC** *Scalable Processor Architecture*.

**SPARCv8** *Scalable Processor Architecture version 8*.

**UF** Unidade Funcional.

**UI** Unidade de Inteiro.

**UPF** Unidade de Ponto Flutuante.

**VEX** *VLIW Example*.

**VHDL** *VHSIC hardware description language*.

**VLIW** *Very Long Instruction Word*.

**WIM** *Window Invalid Mask*.

# Lista de Figuras

# Lista de Tabelas

# Capítulo 1

# Introdução

Nas últimas décadas tem-se observado um aumento consideravel no número de propostas com foco em soluções para o problema de *Memory Wall* [15–17]. O *Memory Wall* diz respeito à diferença crescente entre o desempenho do processador e o tempo de acesso à memória. Paralelamente, a partir das restrições de espaço em memória e desempenho impostas pelos sistemas embarcados, novas técnicas têm sido apresentadas com intuito de reduzir o tamanho dos programas, de forma a também reduzir o impacto dos mesmos nos acessos à memória e, consequentemente, aumentar o desempenho final do código.

Muitas dessas técnicas de redução de programas focaram em novos esquemas de codificação de instruções enquanto outras utilizavam técnicas para compressão de instruções. Enquanto as técnicas de codificação procuram manter a semântica das instruções a fim de flexibilizar o processo de decodificação, as técnicas de compressão objetivam retirar redundância das instruções para reduzir o tamanho final do código. O objetivo primordial dessas técnicas vai diretamente ao encontro das demandas de soluções para o *Memory Wall* e sistemas embarcados: reduzir a quantidade de dados armazenados na memória principal e, consequentemente, diminuir os *cache misses* e o volume de dados transferidos entre memória e processador. Historicamente, tais técnicas têm sido propostas desde a década de 90 e são aplicadas tanto em arquiteturas de alto desempenho [18, 19], de propósito geral [20, 21], quanto em arquiteturas voltadas para domínios específicos de aplicações [22–27].

Os principais benefícios de um código menor são o menor consumo de energia, maior velocidade de execução e menor uso de memória. Tais características são especialmente desejáveis em sistemas embarcados, devido ao consumo de energia restrito, limitada capacidade de processamento (em comparação com computadores desktop e servidores) e baixa capacidade de memória. No entanto, há de se considerar também a necessidade de balancear tais ganhos com os impactos gerados na microarquitetura do processador alvo, devido, inevitavelmente, à inclusão de hardware específico para decodificar, em tempo de execução, o programa que foi codificado em tempo de compilação.

Mesmo diante de várias propostas existentes na literatura da área, nota-se ainda algumas lacunas ainda existentes no tocante ao projeto de algoritmos e sistemas codificadores/decodificadores de programas:

- Ausência de estudos científicos que demonstrassem limites teóricos e práticos de esquemas de codificação e impactos no processador alvo: a maior parte dos trabalhos que propõem esquemas de codificação apresentam a taxa de compressão como resultado principal para validação e avaliação da proposta.
- Ausência de técnicas de codificação que possam ser adaptáveis para diferentes conjuntos de instruções: a maior parte das propostas de codificação são *adhoc* e dependentes do conjunto de instruções da máquina alvo.
- Ausência de ferramentas de compilação e simulação com possibilidades de algoritmos para codificação de programas: o compilador GCC oferece suporte à codificação de programas para processadores *Microprocessor without Interlocked Pipeline Stages* (MIPS) e *Advanced RISC Machine* (ARM) mas não há ferramentas que possibilitam simular e avaliar os impactos da geração de programas codificados. Essa lacuna também engloba ferramentas que poderiam auxiliar no projeto de técnicas de codificação.

A técnica de codificação de instruções PBIW [28] foi projetada, inicialmente, para arquiteturas que buscam instruções longas na memória [1, 10, 29] como processadores baseados em arquiteturas *Explicitly Parallel Instruction Computing* (EPIC) [30, 31], *Very Long Instruction Word* (VLIW) [32], arquiteturas reconfiguráveis com várias unidades funcionais e arquiteturas de alto desempenho baseadas em matrizes de unidades funcionais. A técnica é composta por um algoritmo que realiza a fatoração de operandos [33–35] e por uma memória cache chamada de *Pattern cache* (P-Cache).

Dada a limitação existente em comparar técnicas de codificação e compressão de código, este trabalho de mestrado volta-se para o projeto e implementação de um codificador de instruções, baseado na técnica PBIW, para o conjunto de instruções SPARC revisão 8 (SPARCv8) [11]. Especificamente, o presente trabalho estende os trabalhos anteriores de [28] e [1] pois utiliza uma infraestrutura de codificação (*framework de software*) para aplicar o algoritmo PBIW sobre o conjunto de instruções SPARCv8. A aplicação de PBIW sobre programas SPARC será doravante denominada técnica PBIW-SPARC. Este trabalho também discute sobre a complexidade de tempo e espaço gerada pela codificação PBIW-SPARC. Além da proposta de uma nova codificação para processadores baseados na *Instruction Set Architecture* (ISA) SPARC, este trabalho explora o espaço de projeto de decodificadores de instruções junto à via de dados do processador *soft-core* Leon 3 [36]. O projeto do decodificador de instruções PBIW-SPARC permite avaliar os impactos da adição de um circuito de decodificação junto à microarquitetura do processador alvo no que diz respeito ao aumento de área provocado pelo circuito. Adicionalmente, este trabalho apresenta um conjunto de experimentos visando validar e avaliar a técnica comparando com programas gerados para o conjunto de instruções SPARCv8. Alguns resultados são comparados com outra técnica de compressão de código, denominada SPARC16 [9], também voltada para codificação de programas SPARCv8.

O texto dessa desta dissertação está organizado da seguinte forma:

- no Capítulo 2 será realizada uma revisão da literatura de compressão e codificação de código. Devido à existência de uma vasta gama de esquemas de compressão e compactação de código disponíveis, a abordagem de todos não será descrita neste documento (para evitar torná-lo extenso). No entanto, referências bibliográficas são

fornecidas para o leitor interessado em conhecer, com mais detalhes, outras técnicas de compressão e codificação.

- no Capítulo 3 será apresentada a técnica de codificação de instruções PBIW, a infraestrutura de codificação PBIW independente de conjunto de instruções e uma discussão sobre a complexidade de tempo e espaço de codificações baseadas em PBIW. Neste capítulo, tem-se como objetivo principal a compreensão dos conceitos e princípios de projeto para adoção da técnica PBIW para uma ISA específica.

- O Capítulo 4 descreve as decisões de projeto para adoção da técnica PBIW sobre o conjunto de instruções SPARCv8. Além de apresentar formatos de instruções e padrões codificados, este capítulo também apresenta e discute o projeto e implementação do decodificador de instruções PBIW-SPARC junto à microarquitetura do processador Leon 3.

- O Capítulo 5 apresenta e discute todos os experimentos realizados e resultados obtidos visando à validação e avaliação da técnica de codificação PBIW-SPARC usando a infraestrutura de codificação descrita no Capítulo 3. Este capítulo abrange desde experimentos estáticos voltados para avaliar a capacidade de codificação da técnica PBIW-SPARC, até comparações de taxa de compressão e desempenho com a técnica de compressão SPARC16.

- As conclusões deste trabalho assim como proposições para desenvolvimentos de trabalhos futuros são apresentadas no Capítulo 6.

# Capítulo 2

# Fundamentação Teórica

A área de Compressão de Código tem provado que a redução do tamanho do código não é apenas uma questão de reduzir a área de memória usada por um programa. Também é uma forma efetiva de melhorar o desempenho de programas por meio da redução dos efeitos do *Memory Wall* [15–17], bem como a redução do consumo de energia total [9].

Há basicamente duas abordagens para reduzir tamanho do código. Na primeira, compressão de instruções, as instruções ou blocos de instruções são representados de uma maneira particular, não mantendo significado semântico original. O hardware descompressor é implementado fora do processador, geralmente sendo localizado entre a memória principal e a memória cache ou entre o processador e o primeiro nível de memória. Na segunda abordagem as instruções são codificadas mantendo significado semântico de forma bem definida. Um hardware decodificador é implementado na via de dados do processador realizando a decodificação em paralelo a outras tarefas de processamento com impacto mínimo no desempenho dos programas [8]. Em ambos os casos, há geralmente recursos embutidos nas instruções para mudar o modo de operação: de uma instrução codificada/comprimida para original.

Neste capítulo são apresentadas técnicas e métodos de redução do tamanho de instruções, estando organizado em três seções: Métodos Clássicos de Compressão de Código (seção 2.1); Técnicas de Codificação (seção 2.2) e; Considerações Finais (seção 2.3).

## 2.1 Métodos Clássicos de Compressão de Código

Nesta seção são apresentadas técnicas de compressão que tomam como base a frequência ou a probabilidade com que as instruções (ou bloco de instruções) ocorrem em um programa. São elas: compressão de Huffman [37](subseção 2.1.1), compressão aritmética [2](subseção 2.1.2) e compressão com base em dicionários [3](subseção 2.1.3).

A utilização de métricas para calcular a qualidade da redução de um programa devido à utilização de uma técnica de compressão/codificação pode variar de autor para autor. Neste trabalho considera-se a taxa de compressão, equação 2.1, conforme utilizado em [9].

$$\text{Taxa de Compressão} = \frac{\text{Tamanho Comprimido} + \text{Overhead}}{\text{Tamanho Original}} \tag{2.1}$$

Na equação 2.1 o *Tamanho Comprimido* é o tamanho do programa (em Bytes) após a compressão. O termo *Overhead* é o código adicional que pode ser gerado durante a compressão. O *Tamanho Original* é o tamanho do programa (em Bytes) não comprimido. A *Taxa de Compressão* é a razão do tamanho do código comprimido (mais *overhead*) em relação ao tamanho do código original.

### 2.1.1 Compressão de Huffman

Compressão de Huffman é uma das principais técnicas de compressão de código, devido a sua simplicidade e efetividade. A técnica proposta por Huffman em 1952 [37], tem como ideia principal encontrar a frequência com que cada símbolo aparece na cadeia de dados a ser comprimida, sendo que símbolos que ocorrem com maior frequência são representados com menos bits, quando comprimidos, do que símbolos que ocorrem com menor frequência. Um símbolo comprimido tem tamanho variável, pois sua frequência é proporcional aos demais símbolos que ocorrem na cadeia de dados, podendo variar na quantidade de símbolos e de ocorrências para cada cadeia de dados distinta.

O primeiro passo da técnica consiste em encontrar a frequência dos símbolos que devem ser codificados. Em seguida, deve-se atribuir os códigos gerados aos respectivos símbolos. Por fim, deve-se utilizar uma estrutura de dados eficiente, que seja pequena no processo de armazenamento e rápida quando empregada no processo de descompressão. Para realização das duas últimas etapas, Huffman encontrou uma solução com base em árvore binária. Na estrutura da árvore, cada nó folha representa um símbolo e sua frequência, nós internos (inclusive a raíz), representam a soma das frequências dos símbolos de suas subárvores e as arestas possuem valor 0 quando ligam um nó ao filho esquerdo e valor 1 quando ligam ao filho direito. O código binário que representa um símbolo comprimido é obtido pela sequência de bits no caminho da raíz até a folha que contém esse símbolo.

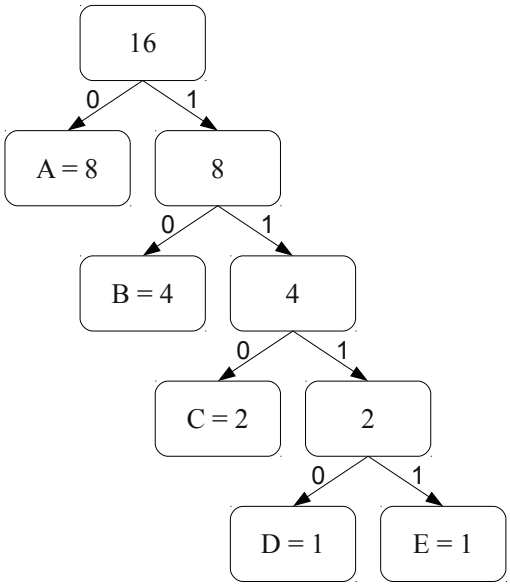



Figura 2.1: Exemplo de árvore de Huffman para a cadeia AAAAAAAABBBBCCDE [1].

No exemplo da Figura 2.1, considere a cadeia AAAAAAAABBBBCCDE na qual cada símbolo pode ser representado por 3 bits (A = 000, B = 001, C = 010, D = 011, E = 100). Antes da compressão, essa cadeia é representada por 48 bits: 000000000000000000000000000010 010010010010011100. Após a compressão de Huffman, a mesma cadeia é representada como 000000001010101011011011101111, utilizando apenas 30 bits. A compressão reduziu 18 bits no tamanho da cadeia de dados gerando uma taxa de compressão de 62,5%.

A compressão de Huffman tem desempenho ótimo quando as frequências dos símbolos são potências negativas de dois $(2^{-1}, 2^{-2}, ..., 2^{-n})$. Devido a cada símbolo possuir um código de tamanho variável, é necessário uma análise prévia para resolver problemas de endereçamento de desvios no fluxo de execução. A utilização desse esquema de codificação diretamente em instruções de máquina pode não ser trivial.

### 2.1.2 Compressão Aritmética

Diferente da compressão de Huffman que utiliza frequências, o algoritmo de compressão aritmética [2] utiliza probabilidades, que são representadas por números reais, sendo usadas para dividir proporcionalmente um intervalo (normalmente $[0, 1)$) em sub-intervalos, onde respectivamente cada símbolo é atribuído. O processo de compressão é responsável pela atribuição dos símbolos a esses sub-intervalos.

A compressão aritmética consiste em representar uma mensagem em intervalos de números reais. Quanto maior a quantidade de símbolos distintos, maior será a quantidade de intervalos e menor será cada intervalo, sendo necessário uma quantidade maior de bits para representá-los. De maneira oposta, menos símbolos significa intervalos maiores e em menores quantidades, utilizando menos bits para representá-los.

Para exemplificar, supõe-se a compressão da mensagem *eaii!* composta pelo alfabeto $\Sigma = \{a, e, i, o, u, !\}$. Usando um modelo fixo — conta a frequência de cada item do alfabeto e calcula suas probabilidades — têm-se as probabilidades vistas na Tabela 2.1. Esta tabela contém valores aleatórios com o objetivo de auxiliar na compreensão do exemplo.

| **Símbolo** | **Probabilidade** | **Intervalo** |
|---|---|---|
| $a$ | 0.2 | $[0, 0.2)$ |
| $e$ | 0.3 | $[0.2, 0.5)$ |
| $i$ | 0.1 | $[0.5, 0.6)$ |
| $o$ | 0.2 | $[0.6, 0.8)$ |
| $u$ | 0.1 | $[0.8, 0.9)$ |
| $!$ | 0.1 | $[0.9, 1.0)$ |

Tabela 2.1: Exemplo de modelo fixo para o alfabeto $\{a, e, i, o, u, !\}$ [2].

Inicialmente sabe-se que o intervalo padrão é $[0, 1)$. Depois de ler o primeiro símbolo $e$, o compressor restringe o intervalo de compressão para $[0.2, 0.5)$, o espaço de probabilidade alocado para o símbolo $e$. O segundo símbolo $a$ restringirá esse novo intervalo — $[0.2, 0.5)$, inicialmente alocado para comprimir o símbolo $e$ — para o primeiro $1/5$ dele próprio, o intervalo $[0.2, 0.26)$.

| | | | |
|---|---|---|---|
| Inicialmente: | | [0, | 1) |
| Depois de ver: | $e$ | [0.2, | 0.5) |
| | $a$ | [0.2, | 0.26) |
| | $i$ | [0.23, | 0.236) |
| | $i$ | [0.233, | 0.2336) |
| | *!* | [0.23354, | 0.2336) |

Figura 2.2: Exemplo de compressão aritmética [2].

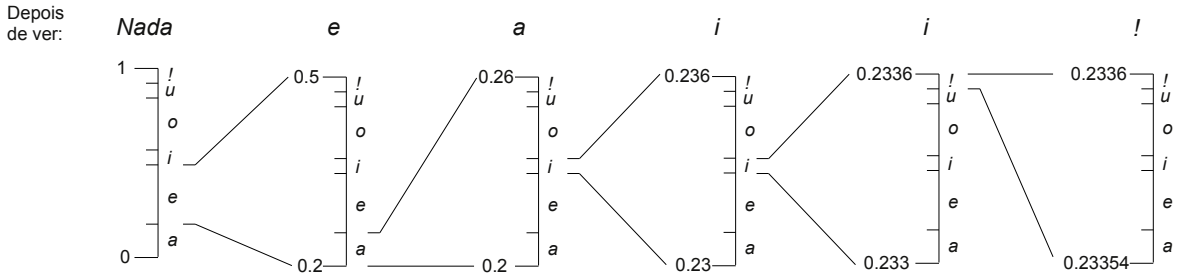



Figura 2.3: Processo da compressão passo a passo com cada estágio de restrição do intevalo sendo ampliado [2].

O próximo símbolo, $i$, é alocado no intervalo $[0.5, 0.6)$ que aplicado proporcionalmente ao intervalo $[0.2, 0.26)$ obtém-se o intervalo mais estreito $[0.23, 0.236)$. Procedendo desta maneira até o final, a mensagem comprimida pode ser vista nas Figuras 2.2 e 2.3, nas quais é possível notar que a compressão se dá principalmente por sub-divisões de cada intervalo ao longo do processamento da mensagem.

A descompressão é realizada através dos mesmos passos realizados na compressão. Para a descompressão não há necessidade de ser dado um intervalo, basta que seja um número contido nele. Por exemplo, no intervalo $[0.23354, 0.2336)$ os números 0.23355, 0.23354, 0.23357 ou até mesmo 0.23354321 representam a mesma informação comprimida.

Para exemplificar a descompressão, supõe-se o número real 0.23355. Inicialmente, o descompressor busca em qual intervalo o número 0.23355 está contido. Começando com o maior intervalo disponível, que no caso é $[0, 1)$, ele restringe o intervalo que contém o número real dado, que é o intervalo $[0.2, 0.5)$. Neste ponto, o descompressor já tem a informação que o primeiro símbolo comprimido é o símbolo $e$. Com este novo intervalo, escolhe-se novamente o sub-intervalo que contenha o número real dado, que neste caso é o sub-intervalo $[0.2, 0.26)$. Ao fazer esta escolha, o descompressor tem uma nova informação a respeito do segundo símbolo comprimido, que é o símbolo $a$. A descompressão acontece realizando esses passos sucessivamente até encontrar o símbolo especial de finalização de mensagem, que neste caso é o símbolo !.

### 2.1.3 Compressão Baseada em Dicionários

Sang-Joon Nam *et al.* propuseram em [3] uma técnica para redução de tamanho do código em processadores VLIW. Esta técnica foi embasada na compressão de código utilizando

dicionários, que teve seu conceito estendido para suportar o isomorfismo entre instruções. Instruções isomorfas são aquelas que têm o mesmo opcode com pequenas diferenças no conjunto de operandos, ou o mesmo conjunto de operandos com opcodes diferentes. Nessa abordagem, apenas as instruções mais utilizadas são comprimidas.

Tanto instruções isomorfas quanto instruções idênticas, desde que usadas frequentemente no programa, terão seus opcodes e operações armazenados em dois dicionários. Cada instrução ao ser comprimida é substituída por dois códigos, um que aponta para uma entrada do dicionário de opcodes e outra que aponta para uma entrada no dicionário de operandos.

Instruções usadas frequentemente podem ser classificadas em dois grupos: instruções idênticas e instruções isomorfas. Instruções idênticas normalmente ocorrem com pouca frequência em processadores VLIW, devido ao fato de que tais instruções são compostas por várias operações. A Figura 2.4 exemplifica o isomorfismo de instruções.

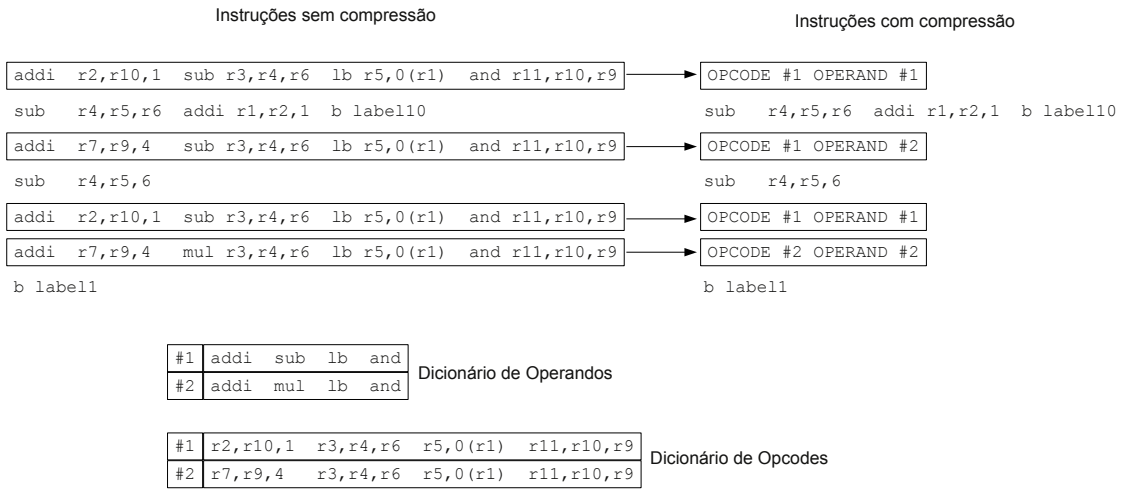



Figura 2.4: Compressão usando isomorfismo de instruções [3].

No processo de descompressão, o descompressor verifica se a instrução de entrada está comprimida ou não. Se a instrução não estiver comprimida é diretamente encaminhada ao processador. Caso contrário, a instrução é direcionada ao descompressor, que de acordo com os ponteiros da instrução busca o opcode e os operandos em seus respectivos dicionários e então, reconstitui a instrução original que é emitida para o processadores.

Em [3], os autores realizaram simulações para processador VLIW com base na arquitetura SPARC, com as respectivas emissões de instruções: *4-issue, 8-issue e 12-issue*. Considerou-se que não há limite no tamanho dos dicionários. Os resultados apontaram uma taxa de compressão média de 63%, 69% e 71% para *4-issue, 8-issue e 12-issue* respectivamente. Instruções isomorfas obtiveram taxa de compressão 17% menor (o que significa melhor) que instruções idênticas. Também observou-se que taxa de compressão de código é bastante afetada pelo número de entradas dos dicionários e a frequência com que cada instrução comprimida é executada, pois quanto maior o dicionário mais custosa será a busca, que será repetida proporcionalmente à frequência com que essa instrução ocorre.

## 2.2 Técnicas de Codificação

A codificação de instruções tem como objetivo reduzir o tamanho do código do programa, alterando a forma e/ou o tamanho da representação binária de uma instrução, porém mantendo seu significado semântico. Dessa forma, deve-se ponderar cuidadosamente sobre a inclusão de codificação em um programa uma vez que o procedimento de decodificação de uma instrução codificada pode gerar mais custos de desempenho do que uma instrução não-codificada. Além disso, há de se considerar que a codificação pode gerar instruções extras no código do programa e, com isso, reduzir a capacidade de compressão e também afetar o desempenho. Assim, é primordial que técnicas de codificação tenham um mecanismo ágil de decodificação e avaliem se a aplicação da técnica não afeta negativamente o desempenho do programa.

Processadores baseados em conjunto de instruções *Reduced Instruction Set Computer* (RISC) como ARM [38], MIPS [39] e SPARC [40] são amplamente utilizados em aplicações na indústria e como objeto de pesquisas no meio acadêmico. Nesta seção são apresentadas algumas técnicas desenvolvidas especialmente para esses processadores, como a codificação Thumb (seção 2.2.1), codificação MIPS16 (seção 2.2.2), codificação SPARC16 (seção 2.2.3) e codificação PBIW (seção 2.2.4). Destaca-se, entretanto, que a lista de propostas de codificação e compressão para tais conjuntos de instruções não se limitam aos trabalhos abordados neste texto. Algumas outras propostas encontradas na literatura da área são [22, 33, 41–43].

### 2.2.1 Codificação Thumb

A técnica de codificação de instruções Thumb [4, 6], tem como objetivo aumentar a densidade de código sem que haja grandes perdas de desempenho. O conjunto de instruções Thumb é um subconjunto do conjunto de instruções ARM, codificado em 16 bits (Figura 2.5).

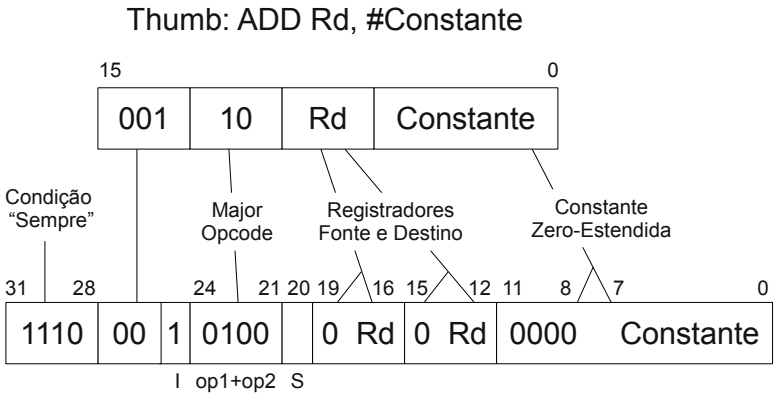



Figura 2.5: Exemplo de mapeamento de uma instrução [4].

O mapeamento entre instruções Thumb e instruções ARM é modelado através de uma função injetora com o conjunto de instruções Thumb sendo o domínio e o conjunto de instruções ARM o contradomínio. Três tipos de instruções ARM foram priorizadas para codificação de código: as mais frequentemente utilizadas em aplicações de clientes; as usadas

pelo compilador para gerar código compactado; e as que possuem redundância e opcode de tamanho fixo. O conjunto de instruções Thumb contém um subconjunto de 36 instruções embasadas no conjunto de instruções da arquitetura (ISA) ARM original. O resultado da codificação de código é um aumento em 30% na densidade de código ARM nativo [4].

Devido às limitações de tamanho das instruções de 16 bits, nem todas as operações do conjunto de instruções ARM podem ser representadas em Thumb. Ao executar uma dada rotina, são necessárias mais instruções de 16 bits do que instruções de 32 bits.

```
if (x >= 0)
  return x;
else
  return -x;
```

Figura 2.6: Código C exemplo para as rotinas em ARM e Thumb [4].

```
CMP     r0, #0;         %Compare r0 com zero;
RSBLT   r0, r0, #0;     %Se r0<0 então r0=0-r0;
MOV     pc.lr;          %Retorne;
```

Figura 2.7: Código ARM equivalente ao código C da Figura 2.6 [4].

```
CMP   r0, #0;     %Compare r0 com zero;
BGE   return;     %Retorne se é maior ou igual;
NEG   r0, r0;     %Se não, negue r0;
MOV   pc.lr;      %Retorne;
```

Figura 2.8: Código Thumb equivalente ao código C da Figura 2.6 [4].

Na Figura 2.6, o código em C recebe como parâmetro um número inteiro e retorna o valor absoluto. Esse procedimento em código assembly ARM pode ser visto na Figura 2.7, onde é representado em três instruções requerendo 12 Bytes (3 instruções de 4 Bytes cada), enquanto o código em assembly Thumb, demonstrado na Figura 2.8, necessita de 8 Bytes (4 instruções de 2 Bytes), uma economia de 33%.

Código Thumb obtém desempenho 30% superior ao do código ARM em sistemas de memória com barramentos mais estreitos (8 bits ou 16 bits), devido ao número reduzido de acessos à memória. Por outro lado, em memórias com barramentos de 32 bits o código Thumb requer instruções extras para completar uma tarefa, obtendo desempenho 15% inferior ao do código ARM. Processadores ARM "*Thumb-aware*" possuem a característica de executar tanto o conjunto de instruções ARM padrão, como as novas instruções Thumb. O projetista pode optar entre tamanho e desempenho, sub-rotina por sub-rotina, escrevendo em código Thumb rotinas de tamanho crítico ou em código ARM rotinas de desempenho crítico [4].

ARM7TDMI foi o primeiro processador com decodificador Thumb implementado no pipeline da via de dados (Figura 2.9), em uma fase do *clock* não utilizada. Dessa forma, instruções codificadas podem ser executadas na mesma taxa de instruções normais, o que significa que não há perda de desempenho durante a decodificação [44].

O processo de decodificação é simples, já que uma instrução Thumb é mapeada em exatamente uma instrução ARM. A Figura 2.5 apresenta um exemplo, onde a instrução Add Thumb é decodificada para instrução Add ARM.

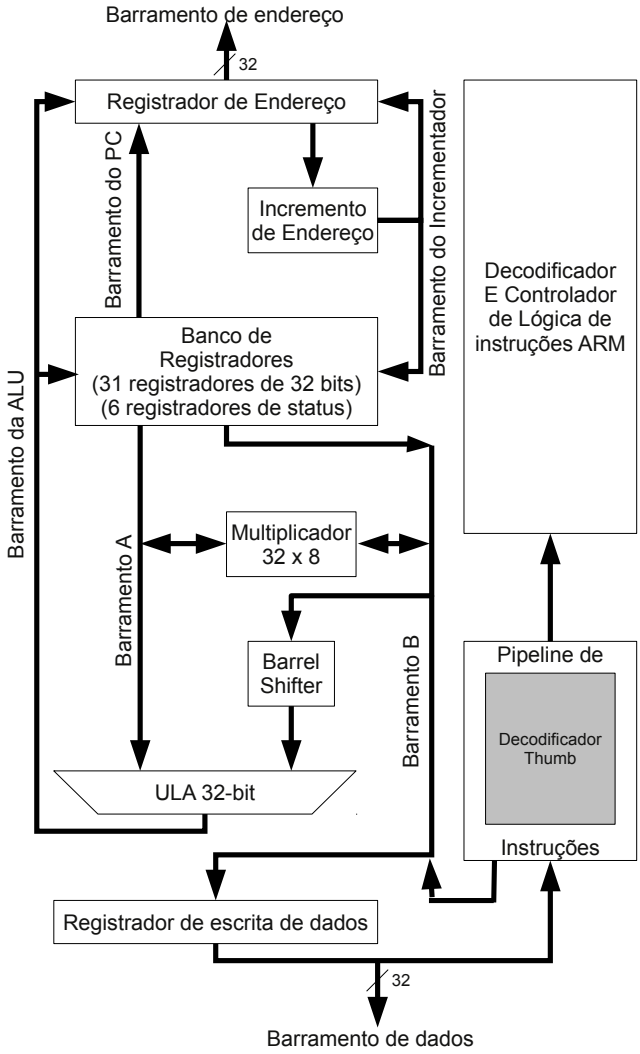



Figura 2.9: Núcleo do processador ARM7TDMI contendo decodificador Thumb [5].

Em 2003, a ARM anunciou a tecnologia Thumb-2 [6], que oferece uma maior extensão para densidade de código. Essa tecnologia aumenta a densidade de código misturando instruções de 16 e 32 bits, ambas no mesmo fluxo de instruções. Para alcançar isto, os desenvolvedores incorporaram acessos a endereços não alinhados no projeto do processador [44]. Um exemplo do formato de instrução Thumb-2 de 32 bits é apresentado na Figura 2.10.


Figura 2.10: Formato de instrução Thumb-2 de 32 bits [6].

O comprimento da instrução e funcionalidade são determinados pela primeira *halfword* (*hw1*). Se a instrução é decodificada como longa (32 bits), a segunda *halfword* (*hw2*) da instrução é buscada tendo como base o endereço da instrução mais dois bits. Thumb-2 tem desempenho próximo ou melhor que do conjunto de instruções ARM e densidade de código semelhante ao da ISA Thumb original [6].

### 2.2.2 Codificação MIPS16

MIPS16 [7] é um mecanismo de codificação para instruções MIPS. Cada instrução MIPS16 corresponde a exatamente uma instrução MIPS. Instruções MIPS16 podem ser decodificadas em instruções MIPS usando um decodificador em hardware relativamente simples. A Figura 2.11 exemplifica a aplicação de MIPS16 utilizando-se um fluxo de instruções codificadas e não codificadas.

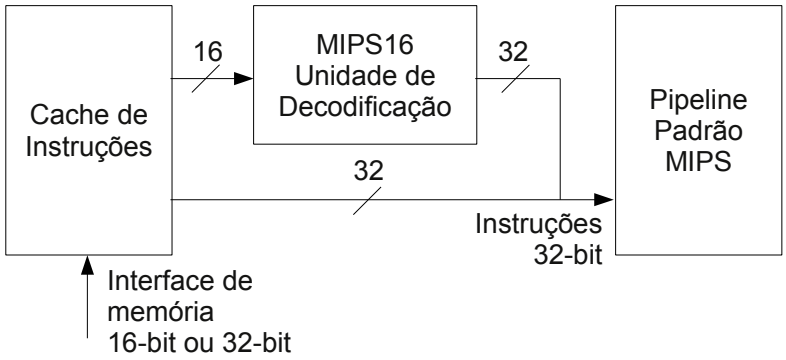



Figura 2.11: Exemplo de decodificação de uma instrução MIPS16 [7].

Para reduzir o número de bits da instrução pela metade são utilizados os três componentes da instrução: opcodes, número de registradores e valores imediatos. Enquanto o conjunto de instruções MIPS tem campo de opcode e campo *function* de 6 bits cada, MIPS16 reduz ambos para 5 bits definindo um total de 79 instruções, das quais 24 são para MIPS-III, que apoia implementações de palavras de dados de 64 bits.

A MIPS, por meio de análises estatísticas de desempenho, observou que na maior parte do tempo o código gerado pelo compilador usava oito ou menos registradores. Por isso, MIPS16 foi restrito a oito registradores, permitindo especificar registradores de três bits ao invés de cinco do MIPS original. Os 24 registradores gerais do MIPS que não são visíveis diretamente às instruções MIPS16, podem ser acessados usando as instruções MOV32R e MOVR32 que realiza cópia entre um registrador MIPS e registrador MIPS16.

A maior economia é proveniente das restrições no tamanho de valores imediatos expressivos. Ao contrário do formato de instruções MIPS-I (que são instruções com imediatos), onde o campo para imediatos tem 16 bits, no MIPS16, em sua maioria, o campo imediato está restrito a cinco bits e em alguns casos restrito a três ou quatro bits. Um exemplo do mapeamento resultante é apresentado na Figura 2.12.

A codificação MIPS16 básica restringe significativamente o número de registradores e o intervalo de operandos imediatos rigorosamente. Para ajudar a superar essas restrições, MIPS16 oferece alguns mecanismos, como: instrução EXTENDed, que contém somente um opcode e um imediato, para contribuir com 11 bits, concatenando-o com o imediato de 5 bits de uma instrução MIPS16; endereçamento relativo ao PC (*Program Counter*) de 32 ou 64 bits, por meio da inserção de constantes no código pelo programador ou compilador; SP (*Stack Pointer*), que não está presente na ISA MIPS e não está projetado no hardware, é mantido por convenção em um registrador de propósito geral ($29). No MIPS16, SP é referenciado implicitamente por meio de um opcode especial; Loads e Stores podem ser

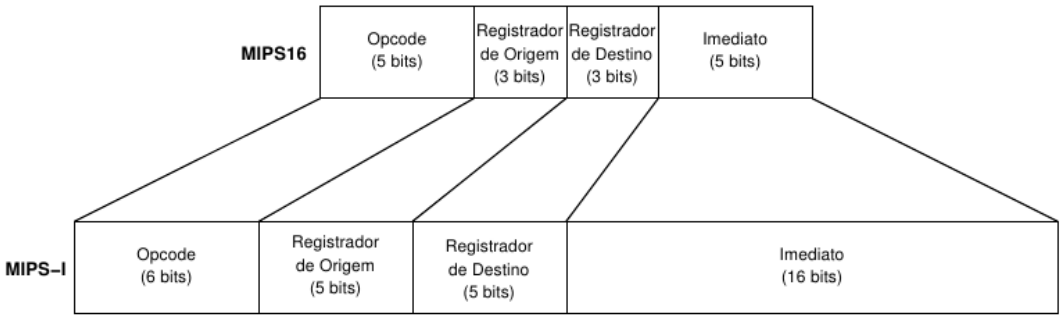



Figura 2.12: Conversão de uma instrução MIPS16 para o formato MIPS-I (32 bits) [8].

realizados relativos ao SP e PC com imediato de 8 bits (o usual são 5 bits) de deslocamento, desde que o registrador base esteja implícito no opcode.

A arquitetura MIPS16 disponibiliza a operação JALX (*Jump And Link with eXchange*) para escolha em tempo de execução entre os modos codificado ou 32 bits (não codificado). JALX estende a semântica por alternar o estado da lógica de decodificação de instruções entre o modos MIPS 32 bits e MIPS16. Uma sub-rotina de 32 bits pode chamar uma sub-rotina de 16 bits e vice-versa. O estado anterior é fundido com o endereço de retorno e restaurado automaticamente no retorno da sub-rotina. Boa parte dos *kernels* de sistemas operacionais nativamente são compatíveis para apoiar CPUs e binários MIPS16.

MIPS16 é um eficiente mecanismo de codificação de instruções que preserva a semântica e a compatibilidade binária da arquitetura MIPS. Apesar de suas instruções possuírem a metade do tamanho das instruções do ISA MIPS padrão, mais instruções são necessárias para realizar algumas operações. Com compiladores otimizados para MIPS16, a taxa de compressão atinge 60%, além de seu código ter maior densidade, melhorando a taxa de hit na *cache* de instruções.

### 2.2.3 Codificação SPARC16

De forma similar às técnicas de codificação de instruções Thumb (subseção 2.2.1) para processador ARM e MIPS16 (subseção 2.2.2) para processador MIPS, a técnica SPARC16 [9] tem como objetivo reduzir o tamanho das instruções de 32 bits para 16 bits, para o processador SPARCv8.

A ISA SPARC foi selecionada após uma análise de 15 variações de 7 diferentes ISAs e representa uma boa escolha entre oportunidade de codificação e impacto do resultado. As ISAs candidatas que foram avaliadas são de arquiteturas que ainda estão em uso atualmente e que tem baixa densidade de código. A arquitetura SPARCv8 representa um compromisso muito bom entre ganho potencial na redução do tamanho do código e uso de ISA atual. Para reduzir a ISA SPARCv8 para 16 bits foi necessário analisar como os campos da ISA são usados. Os campos mais importantes para redução, são os de valores imediatos que podem ocupar mais da metade da instrução.

As instruções SPARC16 são simples o suficiente para serem traduzidas em tempo de execução para instruções SPARCv8 equivalentes. A tradução é realizada por meio de um

decodificador localizado entre a *cache* de instruções e o pipeline do SPARCv8, como mostrado na Figura 2.13.

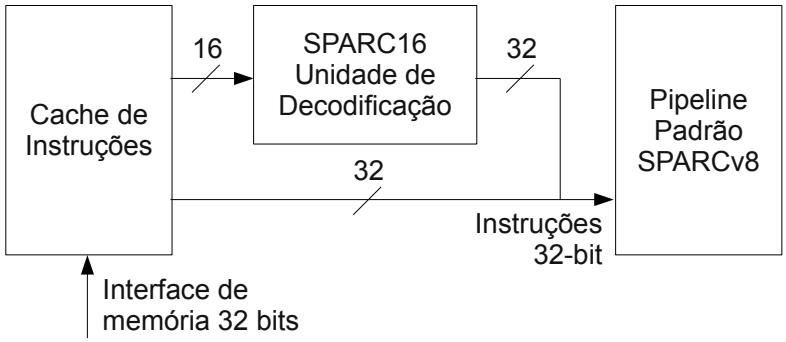



Figura 2.13: Exemplo de decodificação de uma instrução SPARC16 [9].

SPARC16 usa um subconjunto de registradores SPARCv8. Dos 32 registradores SPARCv8, oito são visíveis e podem ser explicitamente referenciados pelo SPARC16. O acesso a registradores "ocultos" acontece por meio das instruções especiais MOV8to32 e MOV32to8, sendo que a primeira move dados de registradores visíveis para registradores ocultos e o segundo faz o inverso.

Constantes grandes são manipuladas usando uma instrução auxiliar semelhante a EXTEND do MIPS16 (subseção 2.2.2). Qualquer instrução que precise de constante maior, pode ser estendida usando alguns bits da instrução estendida. Esse mecanismo possibilita a algumas instruções, o uso de registradores adicionais (por exemplo, um terceiro registrador), permitindo que algumas operações do SPARCv8 não representadas no SPARC16 possam ser representadas com um *overhead* de 16 bits, ao invés de um modo de troca para instrução SPARCv8.

A alternância entre os modos de execução pode ser realizada pela instrução *jmpl*, disponível no SPARCv8 e SPARC16. O último bit menos significativo no endereço de 32 bits determina o modo de execução da rotina (0 - SPARCv8 e 1 – SPARC16).

Conforme experimentos realizados pelos autores em [9], SPARC16 obteve uma taxa de codificação média de 60% em relação ao SPARCv8 nativo. Também houve redução significativa no tamanho do código e no número de *cache misses* para *caches* de vários tamanhos.

### 2.2.4 Codificação PBIW

A técnica de codificação PBIW [28] tem como objetivo principal, representar instruções de forma compactada em memória sem a necessidade de hardware complexo para decodificar a instrução. Além disso, PBIW possibilita sua extensão para diferentes conjuntos de instruções, uma vez que não fixa um formato específico para instruções que podem ser codificadas. Ao codificar uma instrução usando a técnica PBIW, tem-se como resultado uma instrução codificada e um padrão correspondente a esta instrução. PBIW explora a sobrejeção entre a instrução e seu padrão.

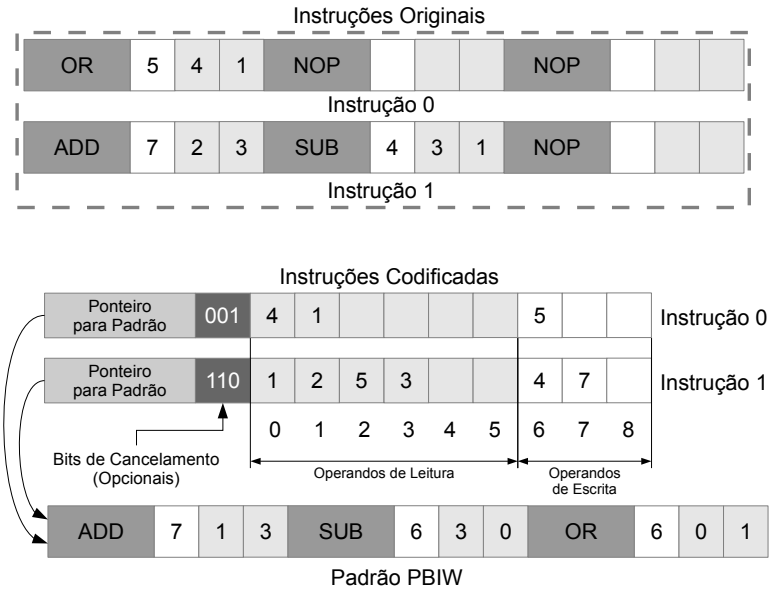



Figura 2.14: Exemplo com instruções originais, codificadas e padrão PBIW [1].

O funcionamento da codificação PBIW consiste de um codificador que usa uma representação bem definida para as instruções codificadas e seu respectivo padrão. As instruções são processadas pelo algoritmo de codificação, sendo criada sua representação codificada e um padrão (caso ainda não exista um padrão idêntico na *cache* de padrões). As instruções são armazenadas em uma *Instruction cache* (I-Cache) e os padrões em uma tabela de padrões (pode ser uma ROM dedicada ou uma outra memória *cache*). Um exemplo da codificação PBIW é mostrado na Figura 2.14, que mostra as instruções 0 e 1 em duas versões, originais (parte superior da Figura) e codificadas (parte inferior da Figura). As instruções 0 e 1 foram codificadas em duas instruções e um padrão. Pode-se observar que as operações ADD e SUB são executadas apenas pela instrução 1 e a operação OR é executada pela instrução 0, porém ambas utilizam o mesmo padrão para realizar a reconstituição da instrução original no processo de decodificação. Os bits de cancelamento para ambas as instruções são 110 e 001, respectivamente. Os bits de anulação indicam quais operações serão utilizadas pela instrução, por exemplo, os bits de cancelamento na instrução 1 (110) indicam que as operações ADD e SUB deverão ser escritas em um registrador (ou em memória se fosse uma operação STORE) e a terceira operação (OR) não deve ser executada: o resultado não será escrito em um registrador ou memória (efeito NOP).

No processo de decodificação quando a instrução é buscada na *cache* de instruções, o decodificador lê na instrução o índice para a tabela de padrões e faz a busca do mesmo. Com ambos, o decodificador reconstitui a instrução original.

A técnica de codificação PBIW será apresentada com mais detalhes no Capítulo 3.

## 2.3 Considerações Finais

Neste capítulo de revisão bibliográfica foram apresentadas pesquisas e propostas de técnicas de compressão e codificação cujo resultado é relacionado a este trabalho. As técnicas apresentadas são atrativas por terem como principal objetivo reduzir o tamanho dos programas e minimizar as latências de busca de instruções na memória. As técnicas de compressão realizam a redução de tamanho do código, sem a necessidade de manter significado semântico. As técnicas de codificação, ao contrário das técnicas de compressão, mantêm significado semântico nas instruções, permitindo que o programa codificado seja buscado na memória e decodificado já no *pipeline* do processador. Apesar das taxas de compressão provenientes das técnicas de compressão serem atrativas, o emprego destas têm sido mais efetivo com utilização de dicionários de código. Nas técnicas de codificação de instruções, apesar de também serem efetivas na redução, tais técnicas são específicas para determinados conjuntos de instruções. Com o intuito de que uma técnica de codificação possa ser aplicada por mais de um conjunto de instruções, foi apresentada a técnica de codificação de instruções PBIW, que é baseada em padrões e inicialmente foi projetada com o intuito de suprir essa necessidade para conjuntos de instruções voltados para arquiteturas VLIW.

Diante do exposto, têm-se como objeto de estudo deste trabalho a técnica de codificação PBIW e sua extensão para o projeto e aplicação de um codificador para o conjunto de instruções SPARCv8. Para implementação do codificador utilizar-se-á uma infraestrutura de software para codificação PBIW. O detalhadamento da técnica PBIW e dessa infraestrutura de codificação será apresentado no Capítulo 3.

# Capítulo 3

# Infraestrutura de Codificação PBIW

Neste capítulo apresenta-se o projeto, funcionamento e principais detalhes da infraestrutra em software (*framework*) para codificar instruções com suporte à técnica de codificação baseada em padrões PBIW. Na Seção 3.1 apresenta-se os principais conceitos da técnica de codificação PBIW. A Seção 3.2 apresenta a otimização de junção de padrões, cujo objetivo é minimizar a quantidade final de padrões. Uma análise da complexidade de tempo e espaço da técnica PBIW é apresentada na Seção 3.3. A Seção 3.4 descreve a infraestrutura de codificação PBIW. Na Seção 3.5, descreve-se o fluxo de execução da infraestrutura de codificação PBIW. A Seção 3.6 aborda brevemente a extensão da infraestrutura de codificação e aplicabilidade da técnica PBIW para arquiteturas escalares. E por fim, na Seção 3.7 são apresentadas as considerações finais do capítulo.

## 3.1 Técnica de Codificação PBIW

A técnica de codificação PBIW [45, 46] é focada na exploração da sobrejeção entre conjuntos de instruções codificadas e seus respectivos padrões, com o intuito de reduzir o tamanho do programa em memória. Esta técnica foi inicialmente projetada para atender arquiteturas VLIW [32]. A técnica é composta por um algoritmo baseado em fatoração de operandos [33–35] e por uma tabela de padrões P-Cache. O algoritmo percorre as instruções do programa e extrai operandos redundantes. Essa estratégia gera uma redução considerável no tamanho da instrução codificada pois mantém apenas operandos não-redundantes nessa instrução. O algoritmo mantém a posição original e o opcode dos operandos em um padrão da instrução, que é criado e então armazenado na tabela ou cache de padrões. As instruções codificadas são armazenadas da mesma forma que as instruções originais (memória principal ou cache de instruções).

Considerando que PBIW não é uma técnica de codificação com enfoque em uma ISA ou processador específico, o projeto do formato do padrão e instrução codificada devem levar em consideração algumas características da arquitetura alvo [45, 46]:

- Número de registradores de leitura;
- Número de registradores de escrita;

- Número de imediatos;
- Tamanho do índice para a tabela de padrões na P-Cache;

Quando uma instrução armazena mais de uma operação (instruções do tipo VLIW), pode-se definir uma quantidade de bits que indicam quais Unidades Funcionais (UFs) devem cancelar ou executar suas operações.

Ao projetar a instrução codificada, além das características mencionadas é necessário que os operandos de uma instrução sejam mantidos na instrução codificada, enquanto que os campos que possuem sinais de controles devem ser armazenados no padrão. Isso se justifica pela possibilidade dos operandos serem lidos no banco de registradores, paralelamente ao processo de codificação de instruções.

Cada instrução decodificada tem no máximo $\mathcal{R} > 0$ campos para representar registradores de leitura, onde $\mathcal{R}$ é a quantidade total de registradores de leitura disponíveis no banco de registradores. A instrução precisa ter $\lceil \log_2 Regs \rceil$ bits, para endereçar cada registrador, onde $Regs$ é o número total de registradores da arquitetura.

Além dos campos extraídos da instrução original, a instrução codificada possui um índice para a tabela de padrões, que armazena o endereço de seu padrão correspondente na tabela de padrões. O tamanho do campo de índice para tabela de padrões é $\lceil \log_2 |P - Cache| \rceil$, onde $|P - Cache|$ indica a quantidade de entradas da tabela de padrões. O padrão é uma estrutura de dados que armazena os campos que combinados aos operandos armazenados na instrução codificada, reconstitui a instrução original. Cada operação armazenada no padrão, contém um opcode e um número fixo de campos que representam os operandos. Os campos que representam os operandos no padrão, conservam suas posições originais e armazenam o índice referente ao campo na instrução codificada que armazena seu respectivo operando.

Um exemplo da técnica de codificação de instruções PBIW sobre uma ISA VLIW genérica é apresentada na Figura 3.1. Nessa Figura são apresentadas duas instruções originais, duas instruções codificadas e um padrão, que é compartilhado por ambas instruções, demonstrando uma situação de reúso.

O esquema de codificação PBIW possibilita o uso de bits de cancelamento. Os bits de cancelamento são campos de 1 bit que fazem parte da instrução codificada e servem para informar quais operações presentes na instrução terão seus resultados escritos. A quantidade de bits de cancelamento presentes nas instrução é igual à quantidade de operações existentes no padrão. Essa funcionalidade permite a utilização de uma otimização denominada Junção de Padrões (Seção 3.2).

Na Figura 3.1 as operações ADD e SUB são executadas apenas pela instrução 1 e a operação OR é executada pela instrução 0. Os bits de cancelamento são 001 para a instrução 0 e 110 para a instrução 1. Nos bits de cancelamento da instrução 0, a sequência 001 indica que apenas a terceira instrução (OR) do padrão, deve ser escrita em um registrador, enquanto as duas primeiras (ADD e SUB) não devem ser executadas, ou seja, ambas terão efeito de NOP, não tendo os resultados armazenados em registradores.

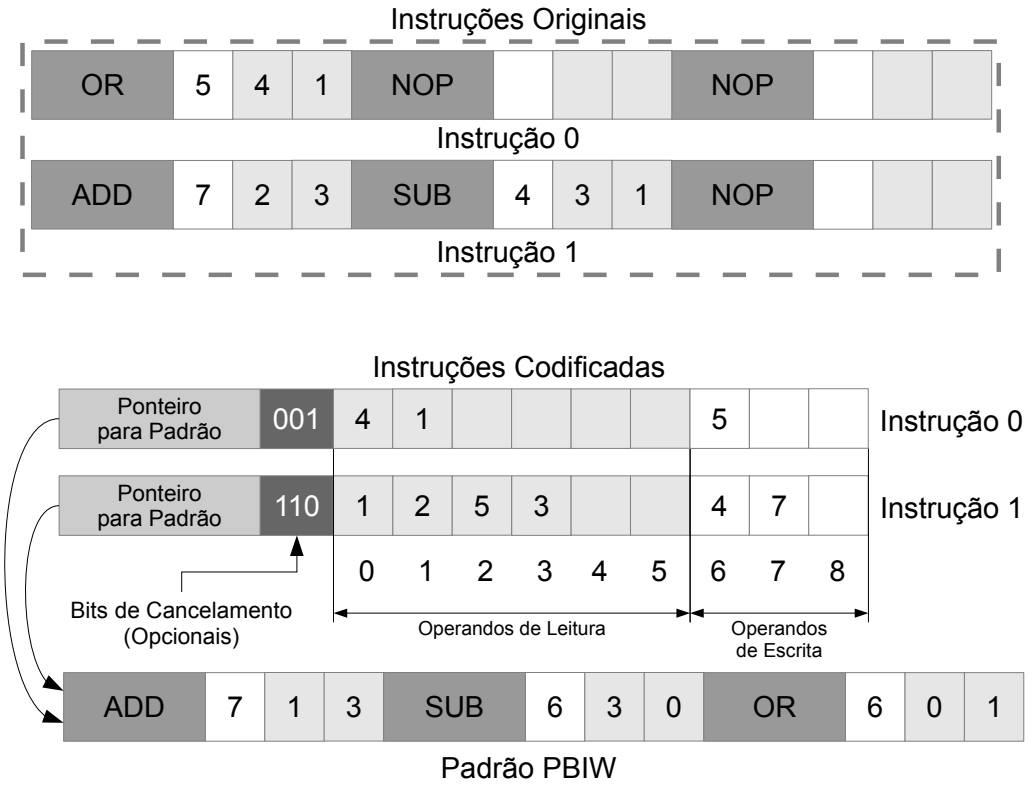



Figura 3.1: Instruções originais, codificadas e padrão PBIW [1].

## 3.2 Junção de Padrões

A junção de padrões [10, 47] é uma técnica com o objetivo de reduzir a quantidade total de padrões gerados pela maximização da ocupação dos campos disponíveis nesses padrões. Essa otimização é aplicada em padrões cuja soma das operações armazenadas em seu formato seja menor que a quantidade de operações suportadas por um padrão. Essa otimização é aplicada em projetos de codificação PBIW em que as instruções originais (não-codificadas) podem armazenar mais de uma operação simultaneamente. A resolução do problema de junção de padrões é uma abordagem do problema de subconjuntos de soma máxima. Se a junção de dois padrões A e B resulta no padrão C:

1. Todas as instruções cujo campo de índice endereça o padrão A ou B, têm seu índice atualizado com o endereço do novo padrão (C);
2. Todas as instruções que endereçam o padrão A devem atualizar seus bits de cancelamento para cancelar as operações do padrão B e vice-versa;
3. Os padrões A e B são apagados da tabela de padrões.

Foi desenvolvida uma heurística para a criação do algoritmo de junção de padrões dividida em três passos [10]:

1. **Pré-processamento:** organiza os padrões em categorias e subcategorias, de acordo com a quantidade de operações nos padrões e o tipo de cada operação. Cada subcategoria é a combinação simples dos tipos de operações que compõe o padrão, considerando as restrições arquiteturais.

   Por exemplo, para o processador $\rho$-VEX (Figura 3.2), as categorias P1, P2, e P3 correspondem a padrões com uma operação, duas operações e três operações, respectivamente. Padrões completos (com o total de operações suportadas) não são considerados. Os padrões P1 e P3 são organizados em categorias de acordo com o tipo da operação (por exemplo: operações de *branch*, operações de memória, operações lógicas e aritméticas, etc.). Padrões P2 seguem uma organização diferente de P1 e P3, de acordo com a combinação das operações suportadas pela arquitetura.

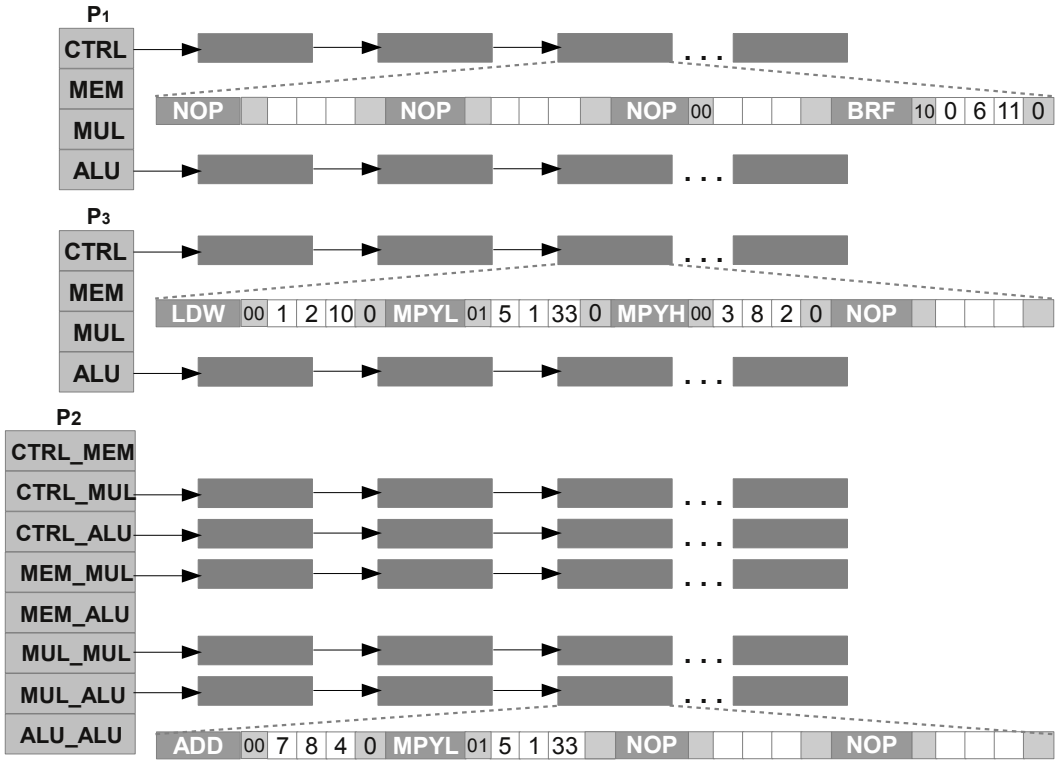



Figura 3.2: Categorias de padrões para o conjunto de instruções VEX [10].

2. **Processamento:** executa o algoritmo de junção de padrões tendo como entrada os padrões organizados. A junção é realizada em pares de padrões. O algoritmo cria os pares com base nas categorias, criadas na fase anterior. Se houver junção entre um padrão com uma operação e um padrão com duas operações, resultará em um padrão com três operações, que poderá ser unido a outro padrão com uma operação. O mesmo acontece na junção entre padrões com uma operação. Um exemplo de junção de padrões da categoria P2 é mostrado na Figura 3.3.

3. **Pós-processamento:** remove padrões redundantes do conjunto de padrões e atribui endereços contíguos aos padrões do conjunto final.

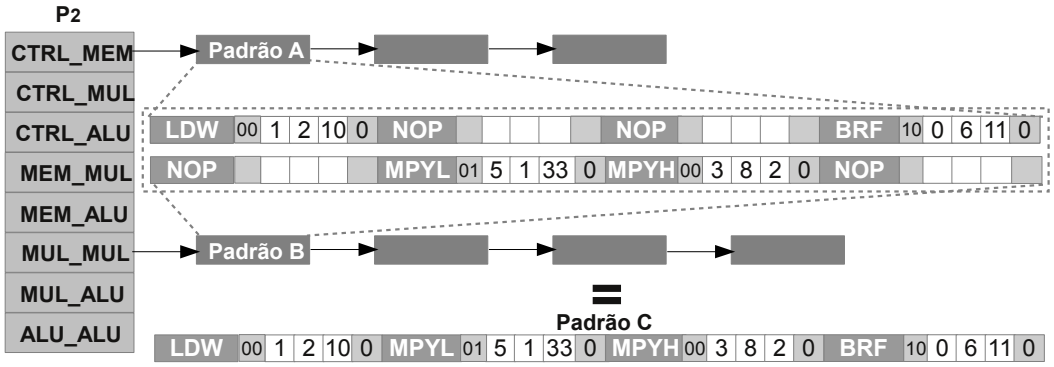



Figura 3.3: Junção entre padrões da categoria P2 [10].

## 3.3 Análise da Complexidade de Tempo e Espaço da Técnica PBIW

O Algoritmo 1 apresenta uma versão do algoritmo de codificação PBIW independente de conjunto de instrução. Esse algoritmo recebe como entrada um conjunto de instruções $CI$ originais (não-codificadas) e retorna como saída o conteúdo de instruções codificadas $CI'$ e padrões $CP$. Basicamente, o algoritmo verifica as operações existentes em cada instrução não-codificada e "copia" os operandos para uma nova instrução codificada e o(s) *opcode(s)* para um novo padrão. Ao finalizar a codificação de uma instrução, o algoritmo verifica se já existe um padrão similar ao atual. Em caso positivo, o padrão recém-criado é excluído e a instrução passa a apontar para o padrão já existente.

A complexidade do algoritmo de codificação PBIW é diretamente relacionada ao algoritmo de busca utilizado para verificar se existe um padrão similar ao recém codificado (linhas 11 e 24). No pior caso, para cada instrução original será gerado um novo padrão, fazendo com que $|CI| = |CP|$[1]. Caso seja usada uma busca linear sobre o conjunto de padrões, a complexidade assintótica de pior caso do algoritmo de codificação será $O(|CI| * |CP|)$ onde $|CP|$ é o número de padrões criados. Assim, a complexidade de tempo, no pior caso, do algoritmo de codificação PBIW é limitado por $O(|CI|^2)$. Dependendo da estrutura de dados utilizada, pode-se reduzir a complexidade assintótica do algoritmo de codificação. Como exemplo, se os padrões estiverem organizados numa estrutura do tipo árvore binária balanceada, a busca por um padrão usa $O(\log_2 |CP|)$ passos, tornando o algoritmo de codificação com complexidade $O(|CI| * \log_2 |CI|)$.

Os cenários de melhor e pior caso na análise assintótica de complexidade de espaço para o programa codificado são diretamente relacionados ao projeto da técnica de codificação PBIW sobre a ISA alvo. Os conjuntos de instruções codificados $CI'$ e padrões $CP$ são relacionados por meio de uma função sobrejetora $\varphi(I') = P$ que mapeia uma instrução codificada $I'$ para seu respectivo padrão $P$. Com isso, o tamanho e a quantidade gerada de instruções codificadas é, obviamente, um dos fatores determinantes para o menor tamanho alcançável para um programa codificado. Importante observar que ao considerar a codificação PBIW sobre todas as instruções de programa, tem-se que $|CI'| \geq |CI|$. Outros fatores

[1]Cardinalidade do conjunto $CI$ é igual ao do conjunto $CP$

**Algoritmo 1:** Algoritmo PBIW independente de conjunto de instruções.

**Entrada**: Conjunto de operações $CI$
**Saída**: Conjunto de instruções codificadas $CI'$ e padrões $CP$

```
1  início
2      Seja O um conjunto, cuja cardinalidade TO = |O| (total de operações suportadas
       simultaneamente pelo hardware alvo), de operações escalonadas;
3      para cada O ∈ CI faça
4          cria novo I;
5          cria novo P;
6          para cada op ∈ O faça
7              P.add(op.opcode);
8              para cada opnd ∈ op faça
9                  se op.opnd ∉ I então
10                     se espaço_livre(I) < 1 então
11                         se P ∉ CP então
12                             CP = CP ∪ P;
13                         fim
14                         I.add(índice de P em CP);
15                         CI' = CI' ∪ I;
16                         cria novo I;
17                         cria novo P;
18                     fim
19                     I.add(op.opnd);
20                 fim
21                 P.add(índice de op.opnd em I);
22             fim
23         fim
24         se P ∉ CP então
25             CP = CP ∪ P;
26         fim
27         I.add(índice de P em CP);
28         CI' = CI' ∪ I;
29     fim
30 fim
```

determinantes são o tamanho do padrão e, principalmente, a taxa de reúso $TR$, Equação 3.1, de padrões por instruções codificadas.

$$TR = \frac{\text{número de instruções PBIW}}{\text{número de padrões PBIW}} \tag{3.1}$$

Desconsiderando as áreas de memória dedicadas aos dados (inicializados ou não) de um programa, pode-se argumentar que o tamanho $To$ de um programa é determinado pela somatória do produto entre as instruções $i$ que compõem o programa e o tamanho de cada instrução, por meio da função $T(i)$, conforme a Equação 3.2:

$$To = \sum_{i=1}^{|CI|} T(i) \tag{3.2}$$

Em programas codificados com a técnica PBIW, além de $CI'$ e $T(i')$, o tamanho do programa codificado ($Tc$) é determinado pela quantidade de padrões gerados $|CP|$ e o tamanho projetado de cada padrão, por meio da função $T(j)$. Logo, o tamanho $Tc$ do programa codificado é dado pela Equação 3.3.

$$Tc = \sum_{i=1}^{|CI'|} T(i') + \sum_{j=1}^{|CP|} T(j) \tag{3.3}$$

Com o exposto, nota-se que um programa codificado com PBIW pode, no pior caso, ocupar um espaço em memória significativamente maior do que um programa não-codificado, uma vez que $|CI'| \geq |CI|$[2] e, para determinar o tamanho do programa codificado, há de se considerar a quantidade de padrões gerados e o tamanho de cada padrão. Nessa observação deve-se considerar, entrentanto, que a existência de uma taxa de reúso $TR > 1$ pode limitar a ocorrência dessa situação de pior caso. Observe que a taxa de compressão $\frac{(T'_i+T_p)}{T_i}$ de uma instrução do programa determina o valor mínimo para que a taxa de reúso $TR$ possa contribuir para a melhoria (redução) do tamanho final do programa codificado.

Como exemplo, na codificação PBIW-SPARC (apresentada com mais detalhes no Capítulo 4), uma instrução codificada possui tamanho (em bytes) $T(i') = 2$, o padrão possui tamanho $T(j) = 3$ e a instrução original possui tamanho $T(i) = 4$. Logo, se for obtido uma taxa de reúso $TR \geq 1,25$, então o programa codificado não ocupará mais espaço que o programa original. Importante atentar que na codificação PBIW-SPARC $|CI'| = |CI|$.

Para casos em que $TR >= \frac{(T'_i+T_p)}{T_i}$ e considerando que $N = |CI|$ e $|CI'| = |CI|$, $g(N)$ é a função que determina o tamanho do programa codificado e $f(N)$ a função que determina o tamanho do programa não-codificado. A Equação 3.4 apresenta a complexidade assintótica de pior caso para determinação do espaço (tamanho) necessário por um programa codificado com a técnica PBIW.

$$g(N) = O(f(N)) \tag{3.4}$$

[2]Considera-se que o programa foi totalmente codificado com a técnica PBIW. Para programas parcialmente codificados, $|CI'|$ pode ser menor que $|CI|$.

Isso significa que, no pior caso, o tamanho do programa codificado é limitado superiormente pela função que determina o tamanho do código original. Ademais, significa que o tamanho e quantidade de padrões não afeta de maneira determinante o tamanho do código final. Nesse caso ainda, considerando $|CP|$ e $T_p$ insignificantes, a taxa de compressão de código máxima alcançável por uma codificação PBIW é:

$TC = \frac{(|CI'|*T_i')}{(|CI|*T_i)}$

Então, a taxa de compressão é a razão entre a quantidade de instruções e tamanho das instruções codificadas sobre o código não-codificado. Assim, uma observação importante e válida para determinar a qualidade de uma técnica de codificação consiste em determinar um limite de complexidade assintótica inferior em função do tamanho do programa codificado. A determinação desse limite mostra as taxas de compressão máximas alcançáveis por meio de uma codificação PBIW. Pela equação assintótica 3.5, observa-se que, no melhor caso, $g(N)$ é limitado inferiormente pelo produto do tamanho do programa original $f(N)$ com $K$ que é o fator de redução da instrução codificada sobre a instrução original (não-codificada).

$$g(N) = \Theta(f(N) * K) \tag{3.5}$$

A Equação 3.5 mostra que para $N$ suficientemente grande, um programa codificado pode ter seu tamanho reduzido em função do tamanho do programa não-codificado pelo fator de redução $K$ de cada instrução. Note ainda que essa limitação faz com que o limite entre as razões das funções $f(N)$ e $g(N)$, com $N \to \infty$, aproxima-se de zero.

## 3.4 A Infraestrutura de Codificação PBIW

A codificação PBIW tem como principal característica ser aplicável em várias ISAs diferentes. Com base nessa premissa, foi desenvolvida uma infraestrutra de software modular e extensível no trabalho de [1], com o intuito de facilitar o uso do algoritmo PBIW em diversos tipos de ISAs e representações específicas de entrada, saída e também a possibilidade de troca do algoritmo genérico por uma versão específica e otimizada para uma determinada ISA. A infraestrutura de codificação pode ser estendida para suportar alguns tipos de formatos de programas (entradas para a infraestrutura).

- Texto em linguagem de montagem: arquivos de linguagem de montagem (.s, .asm, etc.) gerados por um compilador;
- Estruturas de dados finais providos pelo *backend* de um compilador: possibilidade de integração direta entre o *backend* de um compilador e o codificador PBIW;
- Binários ELF: arquivos executáveis no formato ELF gerados por um compilador.

Exemplos de algoritmos PBIW genéricos, especificamente otimizados e inicialmente disponíveis na infraestrutura de codificação incluem:

- Codificação PBIW genérica: algoritmo parametrizável capaz de se adequar a diversas ISAs de entradas;

- Codificação VEX PBIW [1, 10]:
  - Versão 1.0: gera instruções codificadas VEX PBIW com restrições, onde os campos de leitura e escrita na instrução $\rho$-VEX PBIW estão fisicamente separados na instrução. Operandos de leitura não podem ocupar campos de escrita e vice-versa.
  - Versão 2.0: gera instruções codificadas VEX PBIW sem restrições, não há separação entre operandos de leitura e escrita. Ambos podem ocupar qualquer lugar livre presente na instrução PBIW $\rho$-VEX.

Algumas funcionalidades disponíveis na infraestrutura de codificação:

- Possibilidade de incluir e encadear algoritmos de otimização no fluxo de execução da codificação PBIW;
- Independência dos conjuntos de instruções e padrões gerados no processo de codificação: cada contexto de codificação e otimizador PBIW possui seu próprio conjunto de instruções e padrões sem interferir nos dados dos demais.

A infraestrutura de codificação PBIW, por padrão, está preparada para realizar a codificação PBIW sobre o conjunto de instruções baseado na ISA VEX [48]. O codificador está integrado com uma gramática Flex-Bison (disponível no pacote do compilador C VEX [49]) que reconhece de forma completa o arquivo de montagem gerado pelo compilador C VEX. Essa gramática fornece um *parsing* que resolve a limitação do compilador de não gerar arquivo binário nativo para o processador $\rho$-VEX. Essa limitação existe devido à ISA VEX ser totalmente configurável (quantidade de unidades funcionais, tamanho do pipeline, entre outros), não sendo possível conhecer a implementação específica da arquitetura do processador e consequentemente gerar código binário nativo para execução. O codificador PBIW gera dois tipos de arquivos de saída: de depuração e de arquivos de descrição de *hardware*. A saída de depuração foi utilizada para checagem e teste durante a construção do codificador. As saídas de descrição de *hardware* são arquivos-fonte contendo código *VHSIC hardware description language* (VHDL), que implementam as memórias de padrões (P-Cache) e instruções (I-Cache), contendo os padrões e instruções $\rho$-VEX codificadas. Essas memórias são utilizadas para síntese e prototipação para execução e validação em *Field-Programmable Gate Array* (FPGA) utilizando processador $\rho$-VEX.

## 3.5 Arquitetura e Fluxo de Dados

A arquitetura da infraestrutura é composta por três módulos [1]: o contexto de montagem, o contexto de codificação e o contexto de otimização.

No contexto de montagem encontra-se o *front-end* do codificador e as estruturas de dados necessárias para representar em memória o(s) programas(s) antes da codificação. Esse contexto é responsável por preparar os dados de entrada para codificação.

No contexto de codificação há o algoritmo de codificação, bem como rotinas necessárias para manipulação e armazenamento das estruturas de dados, úteis para a representação em

memória das instruções e padrões do(s) programa(s) codificado(s). É possível, no contexto de codificação, incluir um algoritmo de decodificação, com a finalidade de testar a corretude do algoritmo de codificação durante seu desenvolvimento. Dessa forma, é possível escrever testes de unidade para codificação e para decodificação, possibilitando verificar se a semântica original do programa não é afetada.

Por fim, no contexto de otimização encontram-se os algoritmos de otimização para a técnica de codificação utilizada no contexto de codificação. O contexto de otimização possui as rotinas e estruturas de dados responsáveis pela manipulação e armazenamento das estruturas de dados, utilizados para representação em memória dos padrões e instruções otimizados do(s) programa(s) codificado(s).

O fluxo de dados e controle do processo de codificação PBIW simplificado pode ser visto na Figura 3.4.

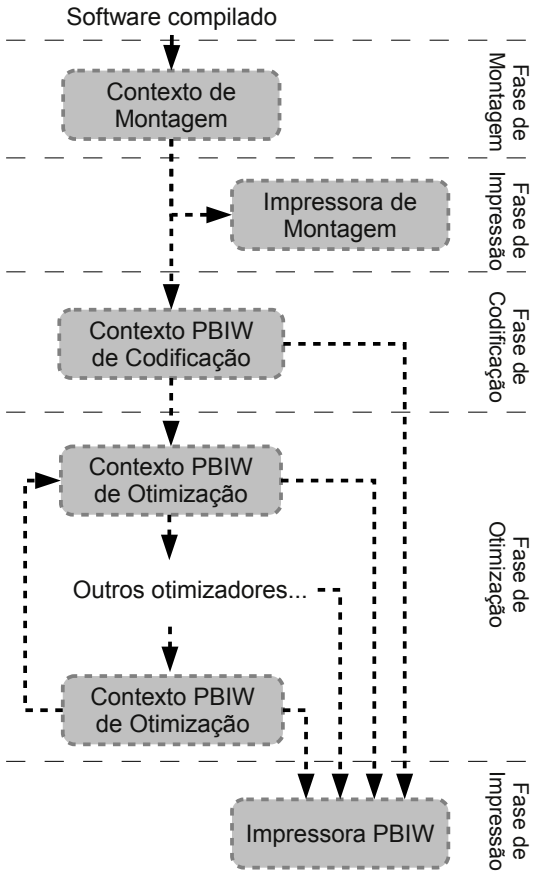



Figura 3.4: Fluxo de dados da infraestrutura de codificação utilizando a técnica PBIW [1].

Na Figura 3.4, a entrada para a infraestrutura de codificação é a saída do compilador em linguagem *assembly*. Cada unidade de compilação gerada pelo compilador será processada pelo contexto de montagem que gera uma representação interna. Esta representação interna é totalmente dependente do montador, tipo de arquitetura e da estrutura de montagem. A infraestrutura de codificação, por meio de interfaces que devem ser implementadas, torna a representação interna acessível ao contexto de codificação. As interfaces fornecem métodos de acesso aos dados internos que estão encapsulados, por meio de uma camada de abstração

entre diferentes implementações possíveis dos contextos de montagem e os algoritmos de codificação PBIW.

Após o processo de codificação é possível aplicar otimizações sobre o conjunto de padrões e instruções codificadas resultantes. As otimizações podem ser realizadas em mais de uma otimização distinta e/ou diversas vezes sobre uma única otimização. A cada otimização é fornecido um conjunto independente de instruções e padrões: uma cópia — das instruções, padrões e outros dados — do contexto original ou do contexto de outra otimização.

Depois das etapas de montagem e/ou codificação, pode-se realizar a impressão dos dados gerados, utilizando classes denominadas impressoras. As classes impressoras, têm acesso às instruções, padrões e demais dados disponíveis e podem imprimir desde informações de depuração, passando por arquivos de descrição de *hardware* e até mesmo gerar arquivos binários ELF completos.

Um exemplo de uso da infraestrutura de codificação PBIW pode ser visto na Figura 3.5 com as cinco fases de processamento: montagem, impressão da montagem (opcional), codificação PBIW, otimização PBIW (opcional) e impressão da codificação ou otimização. Cada uma dessas fases é descrita a seguir:

**Fase de montagem:** realiza o processamento do programa a ser codificado, convertendo-o para uma representação intermediária de seções, operações, funções, rótulos e outros elementos necessários para representar o programa de forma estruturada em memória. Caso o programa a ser codificado está representado em código *assembly* gerado pelo compilador, nesta etapa será necessário realizar o cálculo de endereços de desvio e chamadas de funções. Se o programa a ser codificado estiver representado em um nível já montado e *link* editado — como um arquivo ELF — o cálculo de endereços não é necessário, porém a representação intermediária do contexto de montagem deve saber quais instruções ou funções são os destinos dos desvios já calculados.

**Fase de impressão da montagem:** nesta fase (opcional) os dados coletados e armazenados na representação intermediária do contexto de montagem podem ser impressos de várias maneiras: como saída de depuração para detecção de inconsistências; representação binária do código de entrada, seja como *string* de bits contendo instruções ou como arquivos ELF.

**Fase de codificação PBIW:** realiza a codificação sobre o código em representação intermediária recebido do contexto de montagem. A versão codificada do programa que está no contexto de codificação PBIW, pode ser impresso (um processo análogo ao que acontece no contexto de montagem) ou encaminhado ao contexto de otimização.

**Fase de otimização PBIW:** nesta fase a representação codificada gerada no contexto de codificação é copiada para dentro do contexto de otimização, trabalhando com um conjunto de dados independente. Dessa forma, é possível comparar, caso necessário, os resultados da(s) otimização(ões) com o resultado inicial da codificação PBIW.

**Fase de impressão PBIW:** nesta fase os padrões e instruções PBIW gerados no contexto de codificação e/ou otimização são impressos em um processo análogo a impressão de montagem, seja como saída de depuração ou como saída formatada para execução do programa codificado.

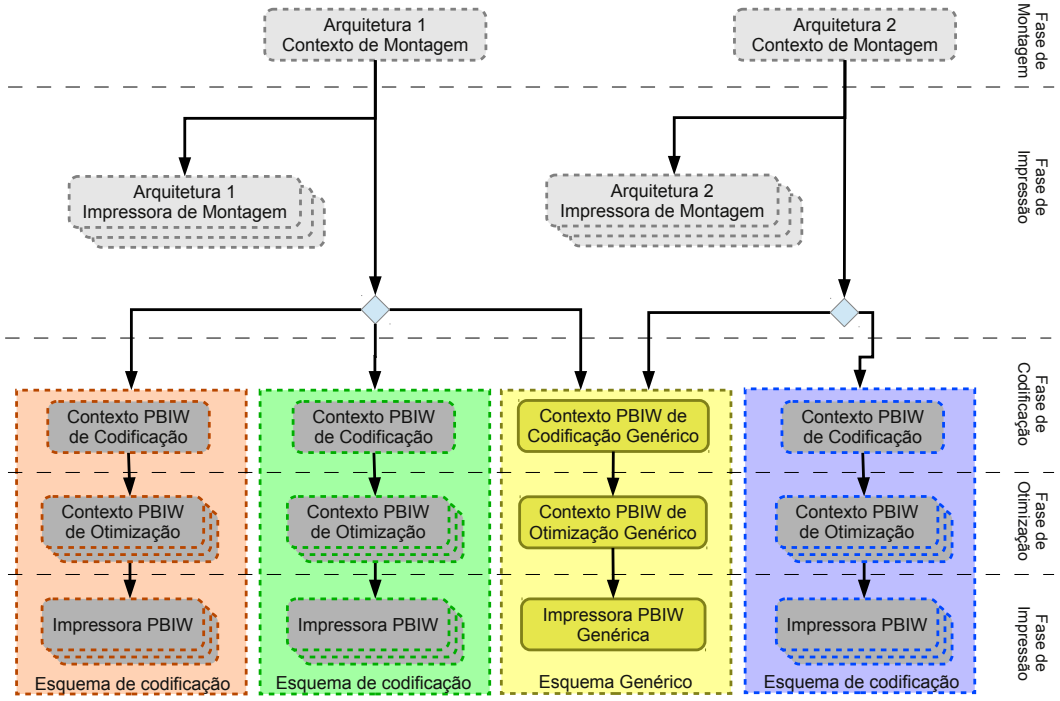



Figura 3.5: Exemplos de fluxos de dados para diversas arquiteturas e codificadores PBIW [1].

## 3.6 Extensão da Infraestrutura de Codificação PBIW para Arquiteturas Escalares

Tendo em vista que a técnica de codificação PBIW [45, 46] foi projetada com o intuito de atender conjuntos de instruções de arquiteturas VLIW [32], verificou-se a factibilidade de aplicar essa técnica em conjuntos de instruções RISC e arquiteturas escalares [50]. Arquiteturas VLIW buscam na memória e emitem ao processador instruções com mais de uma operação, enquanto as arquiteturas escalares buscam na memória e emitem ao processador uma instrução com apenas uma operação por vez. O fato de se buscar apenas uma instrução por vez, torna inviável a utilização de bits de cancelamento e da otimização de junção de padrões (Seção 3.2). Diante deste cenário, há necessidade de adaptar a forma de se aplicar a técnica PBIW para esse novo contexto de instruções e com aspectos inerentes às arquiteturas escalares.

Da necessidade de demonstrar a aplicabilidade da técnica PBIW para um novo conjunto de instruções, formatos de entrada e saída específicos, surge a necessidade de projetar um módulo de codificação para integrar a infraestrutura de codificação PBIW, tornando-a apta a gerar código codificado PBIW também para essa nova demanda.

## 3.7 Considerações Finais

Este capítulo apresentou os principais conceitos da técnica de codificação PBIW, bem como a infraestrutura em software para codificação de instruções. A abordagem sobre

a infraestrutura, considerou pontos como a extensibilidade e portabilidade da técnica, e descreveu a forma como está organizada, em cinco contextos que desempenham funções como: o recebimento do arquivo de entrada e geração de uma representação interna; codificação de instruções e a geração dos padrões; otimização sobre o conjunto de padrões / instruções codificadas e; dois contextos de impressão, um responsável por imprimir informações de depuração e outro responsável por imprimir arquivos de saída (de descrição de hardware, binários ELF, entre outros). A extensão da infraestrutura de codificação PBIW e portabilidade da técnica para suportar o conjunto de instruções SPARCv8 e gerar código para o processador Leon 3 são apresentados no Capítulo 4.

# Capítulo 4

# Técnica PBIW Aplicada Sobre o Conjunto de Instruções SPARCv8

Uma das características da técnica de codificação PBIW é sua aplicação em arquiteturas VLIW. Um dos desafios deste trabalho de mestrado é estender a técnica PBIW para conjuntos de instruções RISC [50] e demonstrar sua eficácia neste tipo de conjunto de instruções e arquitetura.

O conjunto de instruções e a arquitetura alvo escolhida para a técnica PBIW neste trabalho é a SPARC [40] por ser consagrada no meio acadêmico e comercial. SPARC permite um espectro de implementações de sistemas e *chips*, para uma gama de aplicações científicas, de engenharia, embarcadas, comerciais, entre outras [11]. Outra motivação é atribuída ao processador Leon3 [36], que implementa a arquitetura *Scalable Processor Architecture version 8* (SPARCv8) e seu código fonte é disponibilizado completamente, sob a licença *GNU General Public License* (GNU GPL) [51].

Este capítulo apresenta a arquitetura SPARCv8, seu conjunto de instruções e as principais características de ambos. Além de detalhes da arquitetura, este capítulo também aborda as características do processador Leon 3, que implementa a arquitetura SPARCv8 e outras especificidades que vão desde a estrutura do formato ELF, até os projetos do padrão e instrução codificada e do circuito decodificador. O capítulo está organizado da seguinte forma: a Seção 4.1 aborda a estrutura da arquitetura SPARCv8 e seu conjunto de instruções. O processador Leon3 tem suas principais características apresentadas na Seção 4.2. O projeto da técnica PBIW para SPARCv8 é abordado na Seção 4.3. O projeto do circuito decodificador PBIW-SPARC é mostrado na Seção 4.4. Por fim, na Seção 4.5, são apresentadas as considerações finais do capítulo.

## 4.1 Arquitetura SPARCv8

A arquitetura SPARC está publicada como padrão IEEE 1754-1994. Ela define registradores de propósito geral, de ponto flutuante, de controle/status e 72 instruções, todas codificadas em formatos de 32 bits. Um processador SPARC, logicamente compreende uma Unidade de Inteiro (UI), uma Unidade de Ponto Flutuante (UPF) e opcionalmente,

um coprocessador, cada com seus respectivos registradores. A seguir são listadas algumas características da arquitetura SPARCv8 [11]:

**Alinhamento:** Espaço de endereços linear de 32 bits.

**Poucos e simples formatos de instruções:** Todas as instruções têm 32 bits de tamanho e possuem alinhamento de 32 bits na memória. Há apenas três formatos de instruções básicos, onde campos de *opcode* e endereço de registradores estão alocados em posições uniformes. Apenas instruções *load* e *store* acessam memória e I/O.

**Poucos modos de endereçamento:** Endereços de memória são formados pelo par *registrador+registrador* ou *registrador+imediato*.

**Tríade de endereços de registradores:** A maioria das instruções opera em dois ou três operandos (ou um registrador e um imediato) e atribui o resultado em um terceiro registrador.

**Banco de registradores em janelas *"windowed"*:** A qualquer instante estão visíveis ao programa oito registradores inteiros globais, mais uma janela de 24 registradores, em um banco de registradores maior. Os registradores da janela podem ser descritos como uma *cache* de argumentos de procedimentos, valores locais e endereços de retorno.

**Banco de registradores de ponto flutuante separado:** Configurável por software sendo 32 de precisão simples (32 bits), 16 de precisão dupla (64 bits), oito de precisão quadrupla (128 bits), ou uma combinação destes.

As instruções se encaixam em seis categorias básicas:

1. Load/Store
2. Aritméticas/Lógicas/Deslocamento
3. Transferência de controle
4. Leitura/escrita em registrador de controle
5. Operações de ponto flutuante
6. Operações de coprocessador

### 4.1.1 Registradores

Um processador SPARC inclui dois tipos de registradores: de propósito geral e controle/status. Os registradores de propósito geral da UI são chamados de registradores $\boldsymbol{r}$ enquanto que os registradores de propósito geral da UPF são chamados de registradores $\boldsymbol{f}$. Os registradores de coprocessador dependem da implementação.

### Registradores de Propósito Geral

Uma implementação da UI pode conter de 40 até 520 registradores de propósito geral de 32 bits. Eles são divididos em 8 registradores globais, mais um conjunto de 16 registradores. A quantidade dos conjuntos de 16 registradores é dependente de implementação. Um conjunto de registradores é dividido em 8 registradores de entrada (***in***) e 8 registradores locais (***local***). Os agrupamentos de registradores são mostrados na Tabela 4.1.

Tabela 4.1: Endereçamento da janela [11].

| **Endereço do Registrador na Janela** | | **Endereço do Registrador *r*** |
|---|---|---|
| $in[0] - in[7]$ | $\%i[0] - \%i[7]$ | $r[24] - r[31]$ |
| $local[0] - local[7]$ | $\%l[0] - \%l[7]$ | $r[16] - r[23]$ |
| $out[0] - out[7]$ | $\%o[0] - \%o[7]$ | $r[8] - r[15]$ |
| $global[0] - global[7]$ | $\%g[0] - \%g[7]$ | $r[0] - r[7]$ |

Em um dado momento, uma instrução pode acessar os 8 registradores globais e uma janela com 24 dos registradores $r$. Uma janela de registradores compreende os 8 registradores *in* e 8 registradores *local* de um conjunto de registradores, junto com os 8 registradores *in* de um conjunto de registradores adjacente, que são endereçados pela janela atual como registradores *out*. Os registradores do conjunto adjacente, ao serem utilizados pela janela atual como registradores *out*, estão realizando uma sobreposição entre janelas. A sobreposição de três janelas e os 8 registradores globais podem ser vistos na Figura 4.1.

O número de janelas ou conjuntos de registradores, NWINDOWS, estão compreendidos no intervalo de 2 até 32 conjuntos, dependendo da implementação. O número total de registradores $r$ em uma dada implementação é 8 (para registradores globais), mais o número de conjuntos $\times$ 16 registradores (por conjunto). Assim, o número mínimo de registradores $r$ é 40 (2 conjuntos), e o número máximo é 520 (32 conjuntos).

A janela atual de registradores $r$ é dada pelo ponteiro da janela atual — *Current Window Pointer* (CWP), um campo contador de 5 bits localizado no registrador de estado do processador — *Processor State Register* (PSR). O CWP é incrementado por 1 pelas instruções RESTORE ou RETT, e decrementado por uma instrução SAVE ou trap (uma exceção). O registrador de máscara de janela inválida — *Window Invalid Mask* (WIM) é responsável por detectar *overflow* e *underflow* de janela. O registrador WIM é controlado por software no modo supervisor.

Cada janela compartilha seus registradores *ins* e *outs* com duas janelas adjacentes. Os registradores *outs* da janela CWP+1 são endereçáveis aos *ins* da janela atual, e os *outs* da janela atual são os *ins* da janela CWP-1. Os registradores locais são únicos para cada janela.

Os registradores de saída (*out*) da janela CWP são endereçáveis como registradores de entrada (*in*) na janela CWP$-1$. Do mesmo modo, os registradores de entrada (*in*) da janela CWP são endereçáveis como registradores de saída (*out*) na janela CWP+1. A Figura 4.2 mostra como é realizada a sobreposição de janelas.

A Figura 4.2 exemplifica a sobreposição de janelas em uma configuração da arquitetura com 8 janelas. Os registradores de saída em w0 *outs* são as entradas em w7 *ins* e os

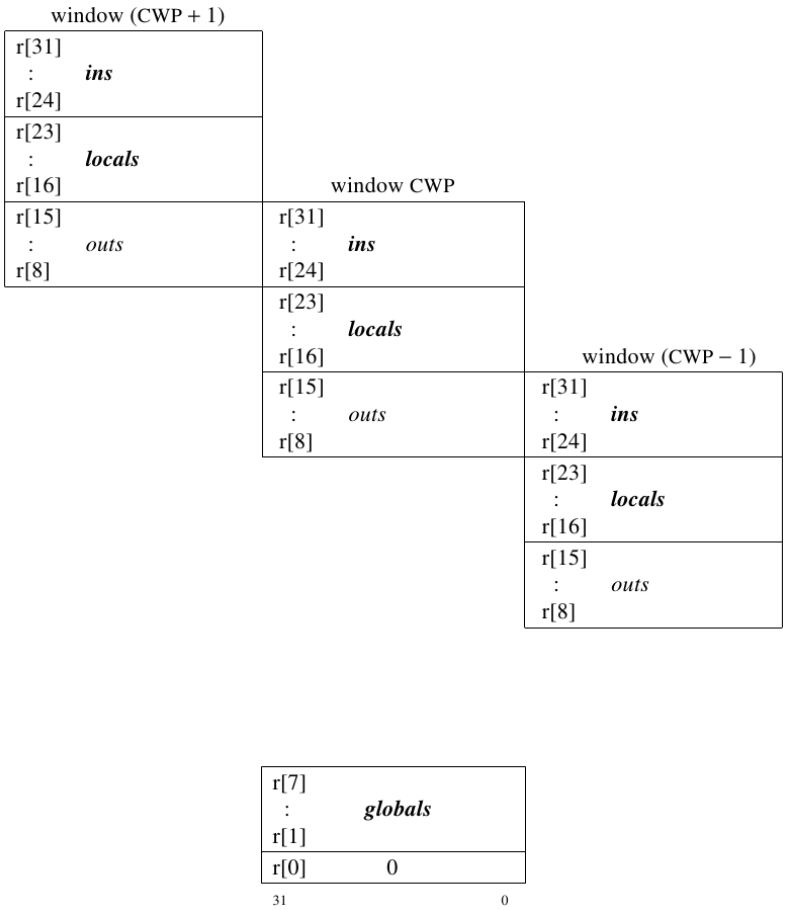



Figura 4.1: Sobreposição das três janelas e os 8 registradores globais [11].

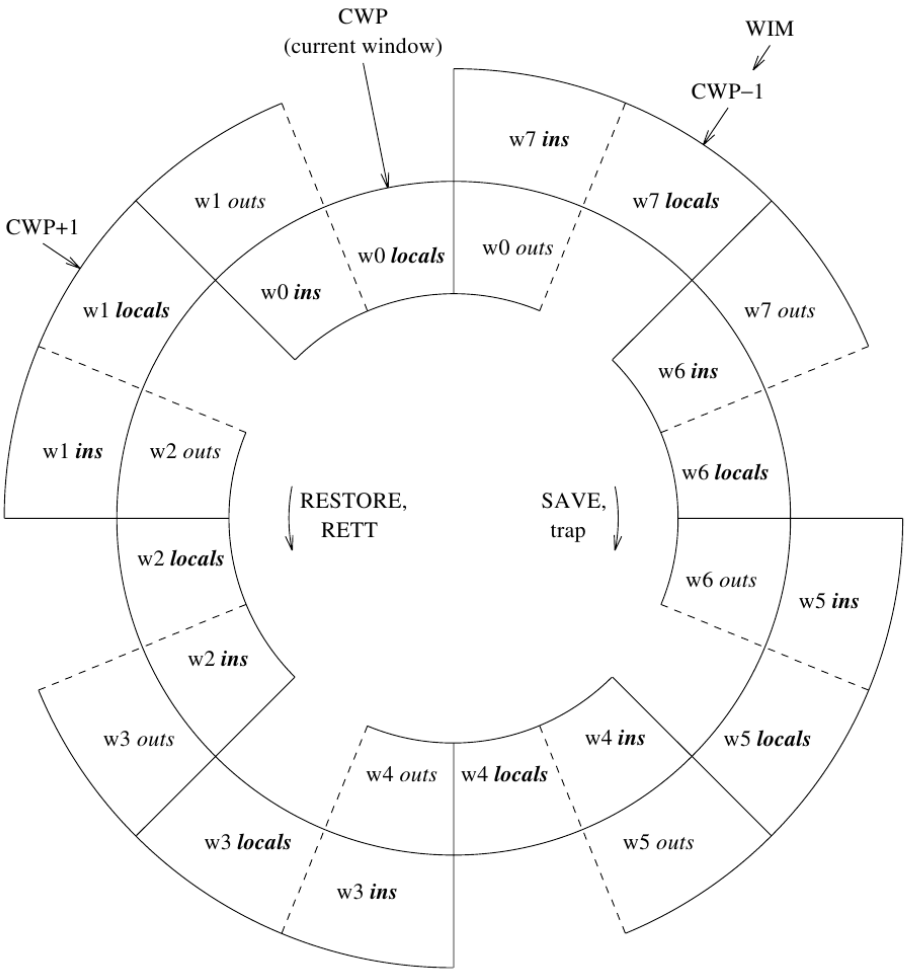



Figura 4.2: Sobreposição de janelas [11].

registradores de entrada em w0 *ins* são as saídas em w1 *outs*. As instruções RESTORE e RETT são responsáveis por incrementar a janela, enquanto as instruções SAVE e trap são responsáveis por decrementar.

### 4.1.2 Conjunto de Instruções SPARCv8

As instruções da ISA SPARCv8 são codificadas em três formatos de 32 bits. Os três formatos são mostrados na Figura 4.3.

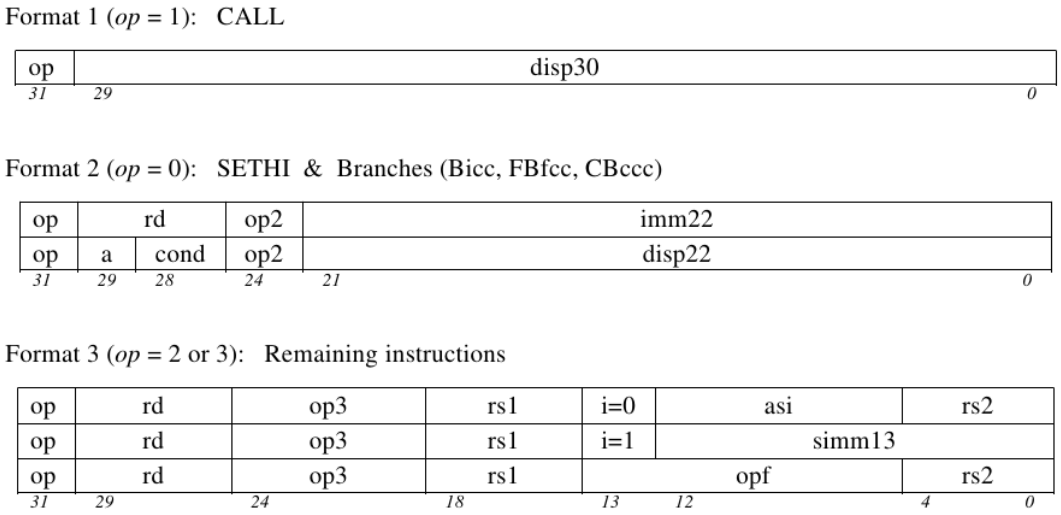



Figura 4.3: Formatos de instrução SPARC [11].

O campo *op* é o único comum aos três formatos, sendo ele o identificador do formato da instrução. A Tabela 4.2 mostra como a codificação do campo *op* define o formato. O campo *op* com valor 1, indica o formato 1, utilizado apenas pela instrução CALL, que além do campo *op* possui o campo *disp30*, um imediato de 30 bits. Esse campo é deslocado dois bits à esquerda e somado ao *Program Counter* (PC) quando a instrução CALL é executada. Dessa forma, todos os endereços de memória podem ser alcançados pela instrução CALL. Quando *op* é zero, o formato 2 é utilizado por uma instrução de extensão de sinal (SETHI), NOP ou *branch* (Bicc, FBicc, CBccc). O formato 3 é utilizado, quando *op* é 3, por instruções de memória ou quando *op* é igual a 2, por instruções aritméticas, lógicas, de deslocamentos e demais instruções.

Tabela 4.2: Codificação do campo *op* [11].

| Formato | *op* | Instruções |
|---|---|---|
| 1 | 1 | CALL |
| 2 | 0 | Bicc, FBfcc, CBccc, SETHI |
| 3 | 3 | Instruções de memória |
| 3 | 2 | Aritméticas, lógicas, de deslocamento e restantes |

Em instruções do formato 2 o valor do campo *op2* determina o tipo de instrução, conforme mostra a Tabela 4.3. O valor 0 é utilizado para codificar a instrução UNIMP, que causa

Tabela 4.3: Codificação do campo *op2* [11].

| *op2* | Instruções |
|---|---|
| 0 | UNIMP |
| 1 | unimplemented |
| 2 | Bicc |
| 3 | unimplemented |
| 4 | SETHI |
| 5 | unimplemented |
| 6 | FBfcc |
| 7 | CBccc |

Tabela 4.4: Significado dos campos das instruções SPARCv8 [11].

| Campo | Significado |
|---|---|
| op | Formato da instrução |
| op2 / op3 | *Opcode* da instrução. |
| rd | Registrador (operando) de destino. |
| a | Bit que anula a execução de um *branch*. |
| cond | Seleciona uma condição para testar uma instrução *brach*. |
| imm22 | Um valor imediato que localiza SETHI na parte mais significativa de um registrador de destino. |
| disp22 e disp30 | São campos para palavras alinhadas, sinais estendidos, deslocamentos relativos ao PC (para instruções *call* ou *branch*, respectivamente. |
| i | Bit que seleciona um segundo operando da ALU, para instruções *load store* e aritméticas (de inteiros). Se i=0, o operando é r[rs2]. Se i=1 o operando é *simm13* com sinal estendido de 13 para 32 bits. |
| asi | Utilizado por instruções *load store* alternativas, para indicar que é um programa do super-usuário e se a operação deve ser realizada na memória de instruções ou na memória de dados. |
| rs1 | Registrador (operando) de origem 1. |
| rs2 | Registrador (operando) de origem 2, utilizado quando o campo i=0. |
| simm13 | Valor imediato utilizado como segundo operando da ALU quando i=1. |
| opf | Codifica instruções de ponto flutuante ou instruções do coprocessador. |

uma interrupção do tipo *ilegal_instrucion*. Os valores 2, 6 e 7 determinam instruções de *branchs* de três tipos: *branchs* de inteiros, tipo Bicc; *branchs* de ponto flutuante, tipo FBfcc; *branchs* de coprocessador, tipo CBccc. Cada um dos três tipos possuem diversas instruções de *branchs* que são definidas com base em um código de condição, armazenado no campo *cond* da instrução. O campo *a* é utilizado para anular a execução de instrução de *branch*. Quando o valor de *a*=1, o *branch* é condicional e não tomado ou, incondicional e tomado. As instruções SETHI ou NOP são definidas pelo campo *op2*=4, que interpretam os campos *a* e *cond* como campo *rd*. Nesse caso, se *rd*=0 indica uma instrução NOP, caso contrário SETHI. Os valores de *op2* 1, 3 e 5 estão disponíveis para futuras extensões da ISA. As instruções do formato 3 utilizam *op3* em conjunto com o bit menos significativo do campo *op* (0 quando *op*=2 e 1 quando *op*=3). Esse mecanismo é necessário para distinguir entre instruções de memória (*op*=2) e instruções aritméticas, lógicas, de deslocamento e as restantes (*op*=3), pois há alguns valores idênticos de *op3* para ambos. As instruções de ponto flutuante também seguem o formato 3, porém não possuem o campo *i*. Nesse caso, essas instruções são idênticas quando *op3*=110100 (FPop1) e *op3*=110101 (FPop2). A operação em si é definida pelo campo *opf*. Os registradores *rd*, *rs1* e *rs2* são de ponto flutuante.

## 4.2 Processador Leon3

Leon3 [36] é um processador *softcore* de 32 bits que segue a especificação da arquitetura (SPARCv8) IEEE-1754. Leon foi projetado para aplicações embarcadas, combina alto desempenho com baixa complexidade e baixo consumo de energia. Dentre as principais características do Leon3, estão [12]:

**Unidade de Inteiro** Implementa completamente a unidade de inteiro do padrão SPARCv8, incluindo *hardware* para instruções de multiplicação e divisão. O número de janelas de registradores é configurável até o limite do padrão SPARC (2-32), com uma configuração padrão de 8. O *pipeline* consiste de 7 estágios com interface de cache de instruções e dados separadas (arquitetura Harvard).

***Cache de instruções e dados*** As *caches* são configuráveis, e podem ter até 4 vias, com tamanho de 1 até 256 KBytes, pode adotar uma das três políticas de substituição: *Least Recently Used* (LRU), *Least Recently Replaced* (LRR) ou aleatória.

**Unidade de ponto flutuante e coprocessador** A unidade de inteiro fornece interfaces para UPF e um coprocessador personalizado. As instruções de ponto flutuante e coprocessador executam em paralelo com a unidade de inteiro. Não há bloqueio de operações a menos que exista uma dependência de dados ou recursos.

***Hardware* de multiplicação e divisão** Suporte em *hardware* para instruções de multiplicação (UMUL, SMUL, UMULCC, SMULCC) e divisão (SDIV, UDIV, SDIVCC, UDIVCC).

**Suporte à memória virtual** A memória virtual está disponível quando há uma *Memory Management Unit* (MMU) ativa.

**Interface de Interrupção** Suporta o modelo de interrupções do SPARCv8 com um total de 15 interrupções.

**Depuração em *hardware*** Unidade de depuração é controlada por uma interface de suporte à depuração que armazena cada instrução executada em um *buffer*.

**Interface AMBA** Implementa um sistema de cache mestre *Advanced Microcontroller Bus Architecture* (AMBA) *Advanced High-performance Bus* (AHB) para *load/store* de dados para/da cache. A interface é compatível com o padrão AMBA-2.0 [52].

### 4.2.1 Pipeline da Unidade de Inteiro do Leon3

A unidade de inteiro do Leon3 implementa a parte de instruções de inteiro da ISA SPARCv8. Os 7 estágios do *pipeline* da UI do Leon3 são mostrados na Figura 4.4 e descritos a seguir:

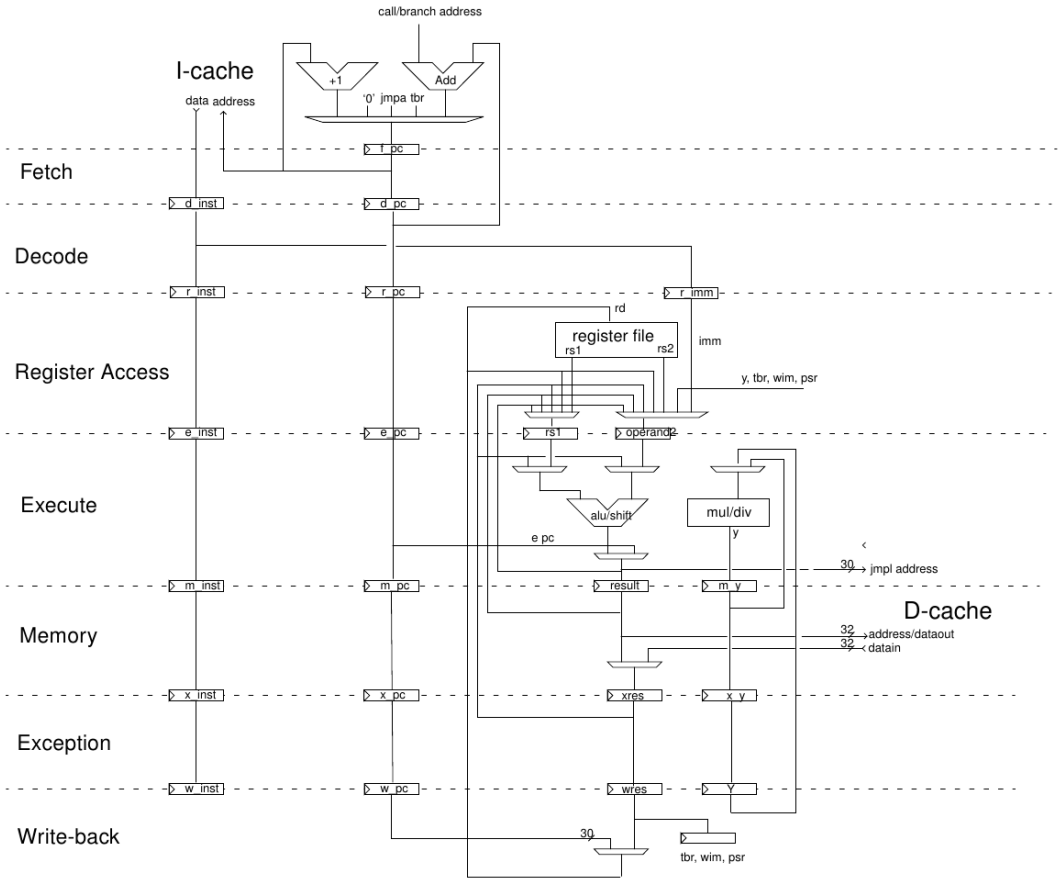



Figura 4.4: Via de dados da unidade de inteiro do Leon3 [12].

1. *FE (Instruction Fetch):* Se a *cache* de instruções está ativada, a instrução é buscada da *cache* de instruções. Caso contrário, a busca (*fetch*) é encaminhada ao barramento AHB. A instrução é válida até o fim deste estágio e é colocada dentro da UI.

2. *DE (Decode):* A instrução é decodificada e os endereços de destino de CALL e *Branchs* são gerados.

3. *RA (Register access):* Os operandos são lidos do banco de registradores ou de dados internos via *bypasses* (resultado de uma operação que é disponibilizado para operações subsequentes dependentes, antes de serem escritas no banco de registradores).

4. *EX (Execute):* Aritméticas, lógicas e operações de deslocamento são executadas. Para operações de memória (p.e.: LD) e para JMPL/RETT, os endereços são gerados.

5. *ME (Memory):* Os dados são lidos ou escritos na *cache* de dados neste momento.

6. *XC (Exception):* Resolução de exceções ou interrupções. Para leitura de *cache*, o dado é alinhado adequadamente.

7. *WR (Write):* O resultado de qualquer operação aritmética, lógica, de deslocamento ou de *cache* são escritos de volta ao banco de registradores.

### 4.2.2 Unidade de Ponto Flutuante e Coprocessador

A arquitetura SPARCv8 define a unidade de ponto flutuante e um coprocessador, ambos de implementação opcional. A UPF pode operar com precisão simples ou precisão dupla e, implementa todas as instruções de ponto flutuante da ISA SPARCv8. A UPF é ligada ao *pipeline* do Leon3 utilizando um controlador, que permite que as instruções sejam executadas em paralelo com instruções de inteiro. Somente se houver dependência de dados ou de recursos que o *pipeline* bloqueia a operação. A UPF permite o início de uma instrução a cada ciclo de *clock*, com exceção das instruções FDIV e FSQRT, que podem ser executadas apenas uma por vez. Essas instruções são executadas em uma unidade de divisão separada e não bloqueiam a UPF, que pode continuar operando com outras instruções em paralelo.

## 4.3 Projeto da Técnica PBIW Para o Conjunto de Instruções e Arquitetura SPARCv8

A codificação da técnica PBIW-SPARC codifica todas as instruções da ISA SPARCv8. Para cada instrução codificada PBIW-SPARC há exatamente uma correspondente SPARCv8. Nenhuma instrução adicional foi criada. A codificação PBIW-SPARC realiza sua codificação sobre um arquivo binário ELF gerado por uma compilador com *back-end* para o conjunto de instruções SPARCv8. A Figura 4.5 exibe o fluxo de codifição PBIW-SPARC.

O fluxo de codificação (Figura 4.5) é iniciado a partir de um arquivo binário executável ELF (SPARCv8) que é fornecido como entrada para o codificador PBIW-SPARC, que realiza a codificação e como saída retorna um arquivo binário executável ELF (PBIW-SPARC) codificado.

### 4.3.1 *Executable and Linkable Format (ELF)*

ELF é um formato padrão de arquivo *link* editável e executável que tem sido adotado pelo sistema operacional Linux, BSD e algumas variantes do UNIX. Arquivos ELF podem ser

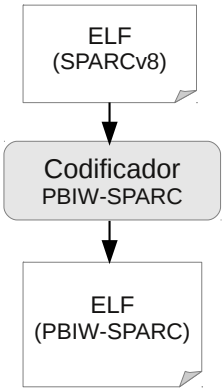



Figura 4.5: Fluxo de codificação PBIW-SPARC.

utilizados para três finalidades distintas: realocável (*relocatable*), executável e objeto compartilhado. Arquivos realocáveis são criados por compiladores e montadores mas necessitam ser processados pelo *linker* antes da execução. Arquivos executáveis têm todas as realocações feitas e todos os símbolos resolvidos, exceto os símbolos de bibliotecas compartilhadas que devem ser resolvidos em tempo de execução. Arquivos objetos são bibliotecas compartilhadas, contendo: informações de símbolos para o *linker* e código diretamente executável para tempo de execução. A codificação PBIW-SPARC é realizada sobre executáveis ELF, portanto, a partir daqui o enfoque será apenas nesse tipo de arquivo.

Arquivos ELF têm uma incomum natureza dupla (Figura 4.6). Compiladores, montadores e *linkers* tratam o arquivo como um conjunto de seções lógicas descritas por uma tabela de cabeçalho de seção (*section header table*), enquanto o carregador (*loader*) do sistema trata o arquivo como um conjunto de segmentos descritos por uma tabela de cabeçalho do programa (*program header table*). Um único segmento normalmente consiste em muitas seções. As seções são destinadas para processamento adicional por um *linker*, enquanto os segmentos são mapeados em memória.

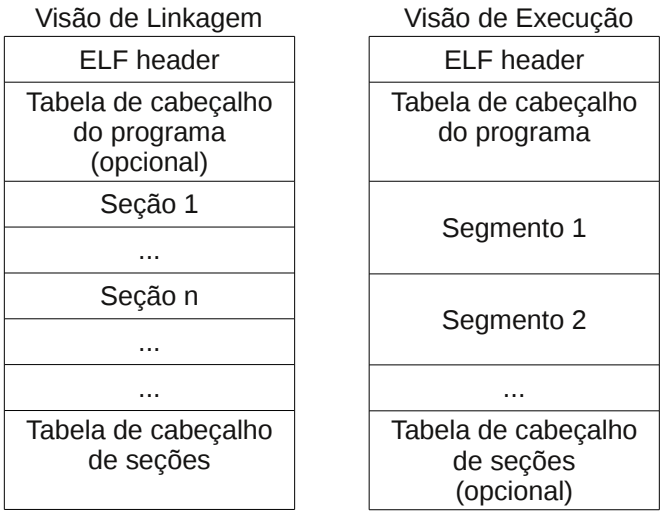



Figura 4.6: Duas visões de um arquivo ELF [13].

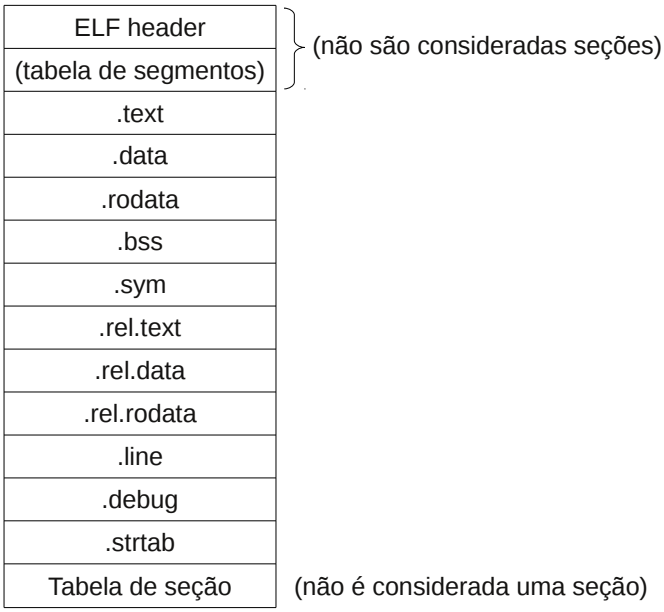



Figura 4.7: Seções de um arquivo ELF [14].

Um típico arquivo ELF completo (Figura 4.7) contém algumas seções. ELF *header* (cabeçalho do ELF) possui informações de identificação da arquitetura alvo do arquivo, ordenação de bytes, tamanho da palavra (*word*) e algumas informações sobre o restante do arquivo, como tamanho e localização do cabeçalho do programa e cabeçalho de seção, se presentes. A tabela de segmentos (ou tabela de cabeçalho do programa) traz a descrição de cada segmento presente, como: endereços virtual e físico, alinhamento, tamanhos físico e em memória, entre outros. De modo similar, a tabela de seção (ou tabela de cabeçalho de seção) descreve cada seção, informando tamanho, localização, alinhamento, entre outros. O cabeçalho do ELF, a tabela de segmentos e a tabela de seções não são considerados seções. Todo o código necessário para execução do programa (exceto o código de bibliotecas compartilhadas) é representado na seção `.text`. A codificação PBIW-SPARC é realizada sobre o código desta seção (`.text`), portanto, não serão dados detalhes sobre as demais seções. Durante a codificação PBIW-SPARC são criadas duas novas seções, uma que armazena os padrões (`.pattern`) gerados pela codificação e uma que armazena informações sobre leiaute (`.layout`) do padrão e instrução codificada utilizados.

### 4.3.2 Projeto de Instruções e Padrões PBIW-SPARC

Diante das características inerentes aos conjuntos de instruções RISC e, em particular, o conjunto de instruções SPARCv8, duas características presentes na técnica PBIW original [45] não foram utilizados nesta abordagem, são elas: a fatoração de operandos (descrita na seção 3.1) e, a utilização no padrão, de índices (ponteiros) para os campos da instrução codificada. A utilização da fatoração de operandos implica na utilização de campos de ponteiros que apontam para campos da instrução codificada, pois cada campo de ponteiro no padrão é armazenado na posição original do operando na operação. Este mecanismo é interessante em aplicações da técnica PBIW para ISAs VLIW, que geralmente possuem

instruções compostas por diversas operações e que podem se beneficiar do reúso de operandos na instrução. A não utilização da fatoração de operandos e de ponteiros para os campos da instrução codificada se justifica devido às instruções SPARCv8 representarem apenas uma operação.

A técnica PBIW concentra, no padrão, os campos que descrevem a operação (campos de *opcode*, controle, entre outros). Os campos de operandos estão presentes na instrução codificada (campos de origem e destino). Isso se justifica pela análise das Figuras 4.8 e 4.9.

```
add  %g1, -1, %g1
add  %g2, -2, %g2
```

Figura 4.8: Instruções `add`.

A instrução `add` se encaixa no formato 3 (seção 4.1.2). A representação binária das instruções `add` (Figura 4.8) é exibida na Figura 4.9. O campo *op* indica que o formato da instrução é 2, *op3* indica o *opcode* da instrução `add` e *i* quando possui valor 1, indica que a instrução utiliza um imediato. É importante observar que esses campos são idênticos para ambas instruções. De forma inversa, os campos de operandos (*rd*, *rs1* e *simm13*) são os únicos que se distinguem. Ao considerar que os campos de operandos compõem a instrução codificada e os demais campos compõem o padrão, nesse caso, seriam criados um padrão e duas instruções.


Figura 4.9: Representação binária das instruções `add` da Figura 4.8.

Seguindo a especificação do algoritmo PBIW, uma instrução codificada sempre terá um padrão associado a ela. Portanto, para reduzir o tamanho do programa, a instrução codificada deve ter tamanho menor que o tamanho da instrução original (não codificada). Além disso, há necessidade de existir um reúso mínimo (taxa de reúso) entre o conjunto de instruções e o conjunto de padrões. No caso da ISA SPARCv8, as instruções originais possuem tamanho de 32 bits.

Para definição do tamanho do padrão e instrução codificada PBIW-SPARC optou-se por reduzir o tamanho da instrução com relação ao padrão, tendo em vista explorar o reúso de padrões, pois sempre haverá uma instrução codificada para representar cada instrução original. Supondo um padrão de 24 bits, restam 8 bits dos 32 da instrução original para serem armazenadas na instrução codificada. No entanto, há de se considerar a existência de um campo/ponteiro de índice de padrões junto aos demais campos da instrução codificada. Esse campo indica o endereço/índice de memória em que o respectivo padrão da instrução pode ser encontrado.

Ao considerar uma P-Cache para 256 padrões, são necessários 8 bits para endereçá-los, deixando a instrução codificada com 16 bits (8 + 8 = 16 bits). Por exemplo, havendo duas

instruções codificadas (16 + 16 = 32 bits) associadas a um padrão (24 bits), taxa de reúso=2, resultará em 32 + 24 = 56 bits. Observe que duas instruções originais, referentes a essas duas instruções codificadas, utilizariam um tamanho de 64 bits.

De modo semelhante, ao supor um padrão com 16 bits da instrução original, os outros 16 bits devem ser armazenados na instrução codificada (16 + 16 = 32 bits). Além disso a instrução codificada necessita do campo de índice do padrão. Novamente, considerando uma P-Cache com capacidade para 256 padrões, 8 bits são necessários para endereçá-los (instrução = 8 + 16 = 24 bits). Duas instruções (24 + 24 = 48 bits) associadas a um padrão (16 bits), resultará em 16 + 48 = 64 bits, igualando-se aos 64 bits da instrução original e superando o exemplo anterior (56 bits).

Uma solução ótima para ambos os leiautes de padrão e instrução codificada, seria um programa com $N$ instruções originais que após a codificação geram $N$ instruções codificadas e apenas um padrão. Nesse caso, ao considerar os tamanhos de instrução codificada e padrão exemplificados anteriormente, teremos:

1. Tamanho do padrão ($T_p$) = 24 bits, tamanho da instrução codificada ($T'_i$) = 16 bits e tamanho da instrução original ($T_i$) = 32 bits:
   $(T_p + N \times T'_i)/(N \times T_i) \cong 50\%$
2. Tamanho do padrão ($T_p$) = 16 bits, tamanho da instrução codificada ($T'_i$) = 24 bits e tamanho da instrução original ($T_i$) = 32 bits:
   $(T_p + N \times T'_i)/(N \times T_i) \cong 75\%$

Pode-se observar no leiaute dos três formatos de instrução da ISA SPARCv8 (seção 4.1.2), que a quantidade de bits dos campos de operandos vão de 15 até 30 bits, sendo inviável mantê-los na íntegra na instrução codificada. Como a maioria dos endereços de desvios e imediatos utilizados pelos programas são valores pequenos, optou-se por armazenar apenas a parte mais significativa dos operandos no padrão e a parte menos significativa e mais susceptível a mudanças, na instrução codificada. A única exceção é o campo *rs2* (utilizado pelo formato 3) que é armazenado totalmente na instrução codificada.

Diante do exposto, o leiaute do padrão e da instrução utilizados pela codificação PBIW-SPARC foram definidos com tamanhos de 24 bits e 16 bits respectivamente e são mostrados nas Figuras 4.10 e 4.11.

A Figura 4.10 mostra o leiaute do padrão que possui 24 bits e a oganização lógica dos campos de acordo com cada formato de instrução da ISA SPARCv8 (seção 4.1.2). Os campos cujo nome contém o símbolo * armazenam apenas a parte mais significativa do campo. A instrução codificada (Figura 4.11) tem 16 bits e campos com os símbolos ** no nome armazenam a parte mais significativa do campo. Sua organização lógica prevê um campo de 8 bits para o índice do padrão e mais um ou dois campos nos 8 bits restantes, dependendo do formato e da instrução a ser codificada. Por exemplo, nas duas situações possíveis para o Formato 2, além do campo de índice do padrão, a instrução codificada pode armazenar, no primeiro caso, os 3 bits menos significativos do campo *rd* e os 5 bits menos significativos do imediato *imm22*. No segundo caso, pode armazenar os 8 bits menos significativos do imediato *simm13*.

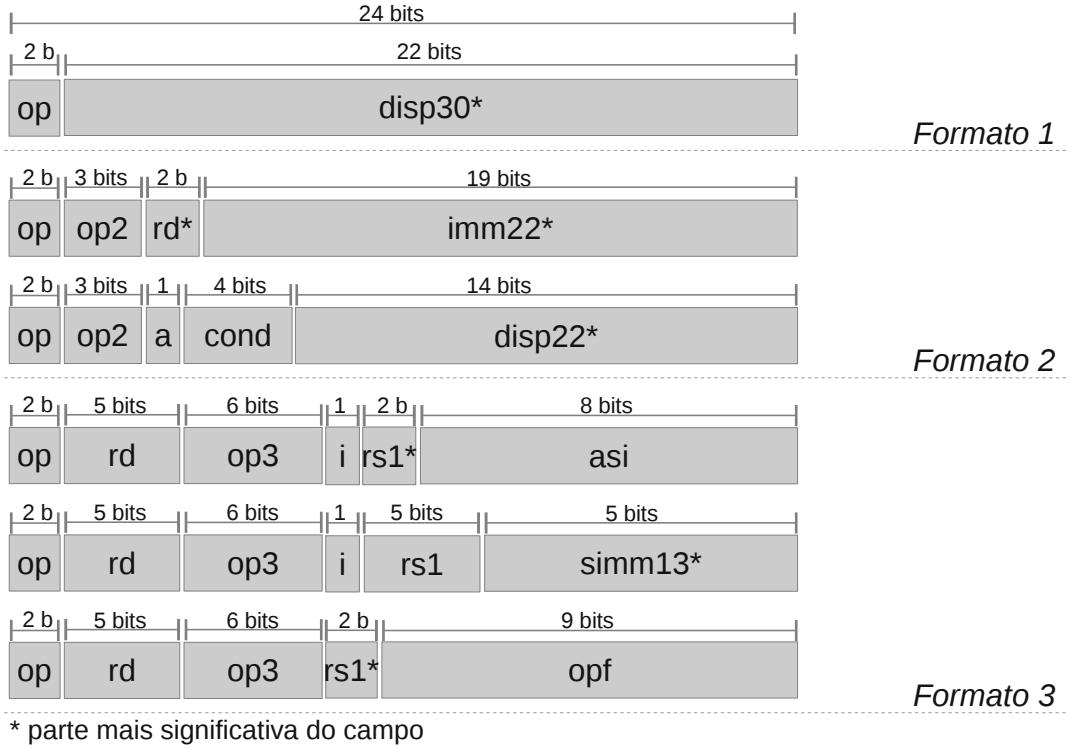



Figura 4.10: Leiaute do padrão PBIW-SPARC.

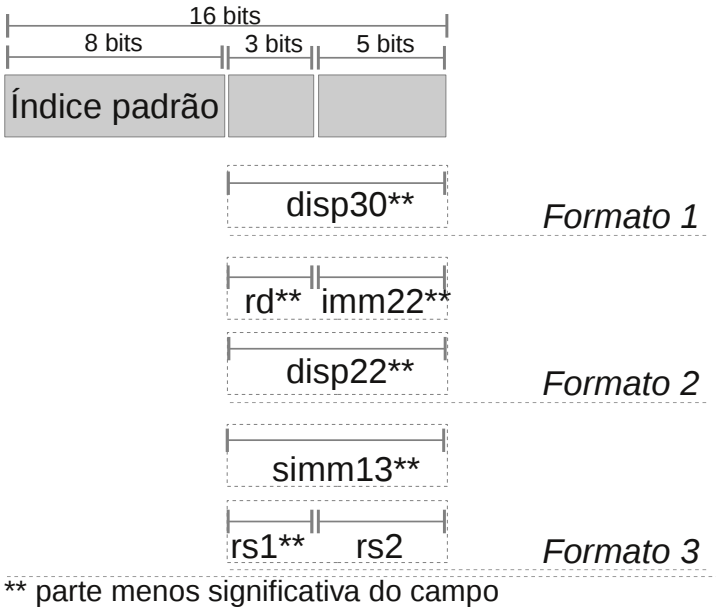



Figura 4.11: Leiaute de instrução PBIW-SPARC.

A codificação com instruções de 16 bits, conforme o leiaute mostrado na Figura 4.11, está limitada a endereçar no máximo 256 padrões (considerando os 8 bits do campo de índice de padrão). Para superar essa limitação e permitir a codificação de programas que geram mais de 256 padrões, pode-se utilizar o leiaute mostrado na Figura 4.12, que estende o campo de índice do padrão para 24 bits. A codificação PBIW-SPARC com esse leiaute está limitado inferiormente à taxa máxima de codificação de 75%, ou uma redução de tamanho máxima de até 25%.

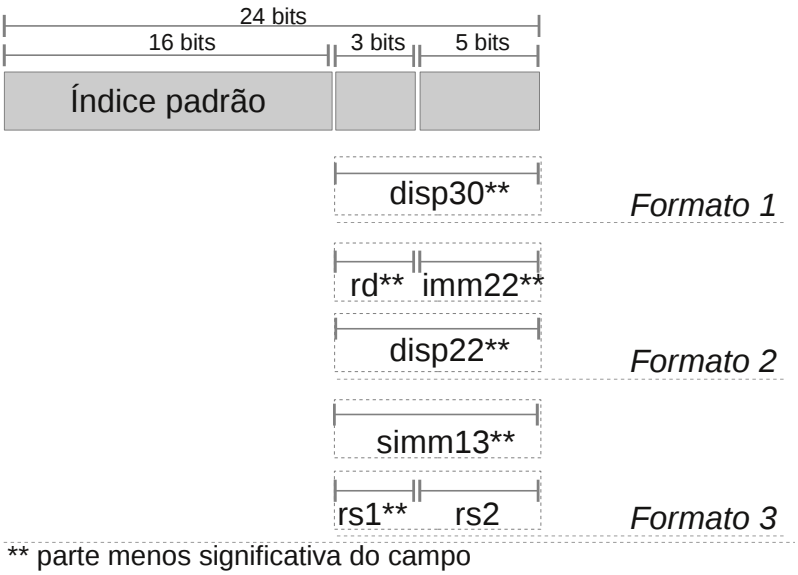



Figura 4.12: Leiaute de instrução PBIW-SPARC de 24 bits.

Após o projeto e formato da instrução codificada e padrão PBIW-SPARC, realizou-se o acoplamento desse projeto junto à infraestrutura de codificação PBIW. O intuito de incluir esse projeto sobre essa infraestrutura é minimizar o tempo de projeto de um codificador, utilizar mecanismos para validação e avaliação da técnica de codificação e, principalmente, demonstrar a flexibilidade dessa infraestrutura quanto à possibilidade de ser estendida para outros domínios de projetos codificadores. A inclusão de um novo codificador PBIW junto à essa infraestrutura inicia pela criação de um contexto de codificação. A criação do contexto de codificação PBIW-SPARC é mostrado na Figura 4.13.

De acordo com o funcionamento e implementação da infraestrutura de codificação PBIW, na fase de montagem implementa-se a leitura da seção `.text` do arquivo ELF e as estruturas de dados são preenchidas com as instruções ainda em seu formato original (SPARCv8). Detalhes sobre essa etapa podem ser impressos, opcionalmente, na fase de impressão. A fase de codificação executa o algoritmo de codificação PBIW-SPARC, que é responsável por identificar cada instrução original e gerar sua versão codificada e o respectivo padrão (caso ainda já não exista um padrão idêntico). Ainda não existem otimizações implementadas para codificação PBIW-SPARC, embora tais algoritmos podem ser criados futuramente e incluídos nessa infraestrutura. Por fim, a outra fase de impressão é responsável por gerar o resultado da codificação, que no caso da codificação PBIW-SPARC é um arquivo ELF. Esse arquivo ELF tem o código original da seção `.text` substituído pelas instruções codificadas e duas novas seções são criadas: uma que armazena os padrões (`.pattern`) gerados na codificação;

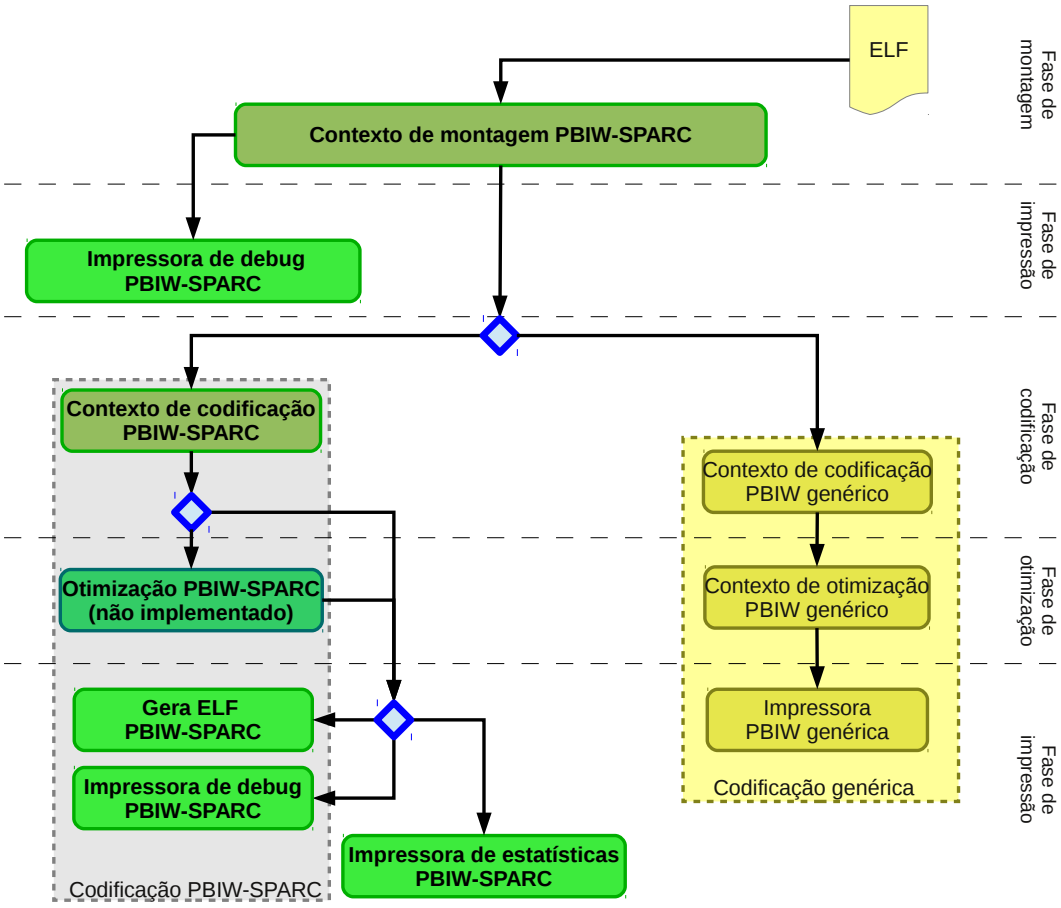



Figura 4.13: Infraestrutura de codificação PBIW após contexto de codificação PBIW-SPARC.

e outra que armazena o leiaute (`.layout`) do padrão e instrução codificada utilizados na codificação.

## 4.4 Projeto do Decodificador PBIW-SPARC para o Processador Leon3

O circuito decodificador PBIW-SPARC é responsável por reestabelecer a instrução original, por meio da combinação de campos entre uma instrução codificada e o padrão. Após a definição dos formatos e tamanhos das instruções codificadas e padrões PBIW-SPARC e após o estudo da arquitetura e projeto do processador Leon3, definiu-se o esboço de um circuito digital combinacional que realiza a decodificação de instruções PBIW-SPARC utilizando uma tabela de padrões. A visão do circuito decodificador PBIW-SPARC para instruções de 16 bits é apresentada na Figura 4.14.

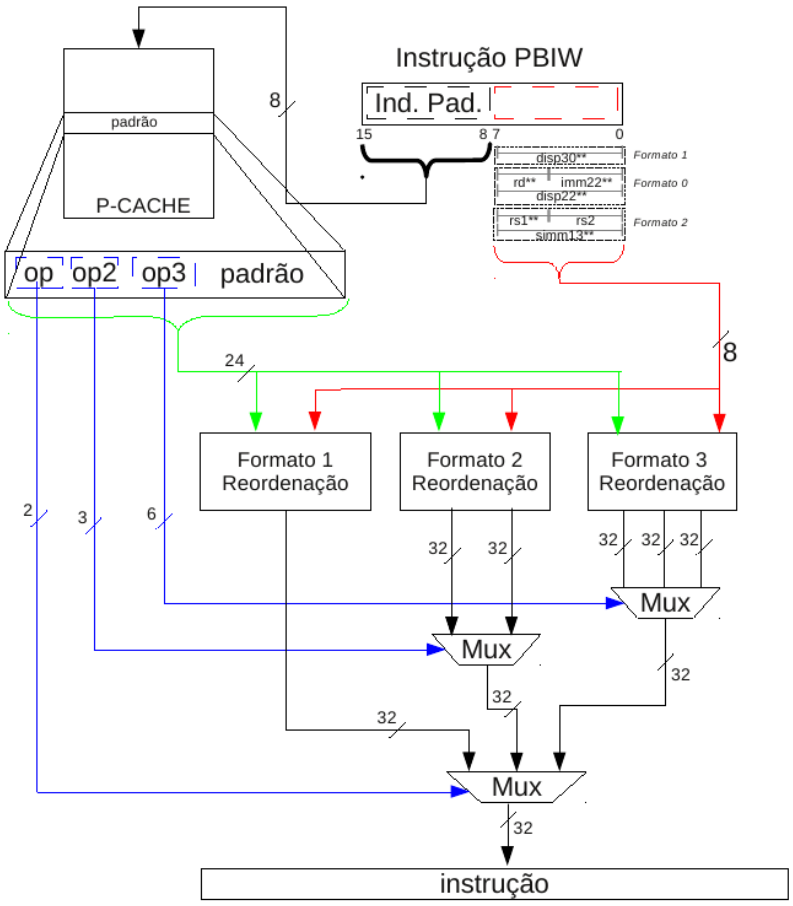



Figura 4.14: Visão geral do decodificador PBIW-SPARC para Leon3, versão de 16 bits.

Na parte superior direita da Figura 4.14 há a instrução PBIW que pode se encaixar em qualquer combinação de bits dos 3 formatos (conforme mostrado na subseção 4.3.2). A partir da busca da instrução na *cache* de instruções, realiza-se a leitura do campo de índice de padrões e então, o padrão endereçado é buscado na tabela de padrões (P-*cache*).

De acordo com o campo op presente no padrão define-se o formato da instrução, que em combinação com o campo opcode (exceto o formato 1, que não possui este campo), respectivo ao formato, configura os multiplexadores para selecionar a instrução decodificada. Observe que as entradas de dados dos multiplexadores são oriundas das unidades de reordenação que atuam, paralelamente, reorganizando os dados e sinais de controle de acordo com cada formato específico. Após esse passo, a instrução original já está recomposta e pode ser despachada para o estágio de decode do *pipeline*.

A Figura 4.15 mostra o circuito de reorganização dos campos do padrão e instrução para reconstituir instruções do formato 2.

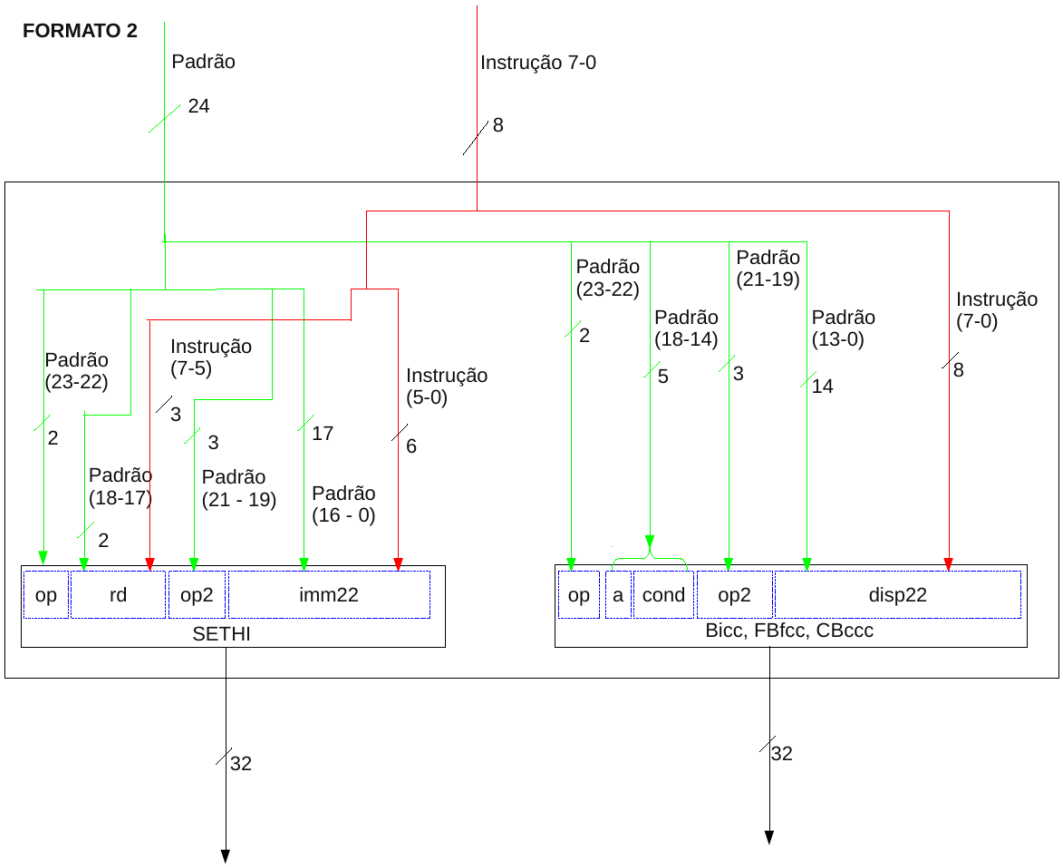



Figura 4.15: Visão geral da unidade de reordenação do formato 2 decodificador PBIW-SPARC para Leon3, versão 16 bits.

De modo geral, há dois barramentos de entrada (do padrão e instrução) que se dividem em vários fluxos para distribuir os bits de acordo com as posições especificadas nos dois sub-formatos. Cada ramo com origem em um dos dois barramentos de entrada da unidade de reordenação está com etiquetas com o nome da origem (padrão ou instrução) e indicando um intervalo de bits que será "concatenado" na posição de destino do ramo, na instrução original. Por fim, outros dois barramentos de 32 bits saem da unidade de reorganização, sendo selecionados pelo multiplexador (conforme mostrado na Figura 4.14).

A única diferença entre o circuito decodificador para instrução de 16 bits (Figura 4.14) e a versão do decodificador para instrução de 24 bits reside no campo índice do padrão, que

passa de 8 para 16 bits. Da mesma forma, o barramento de saída do campo índice do padrão e entrada na *cache* de padrões também é ampliado para 16 bits.

## 4.5 Considerações Finais

Este capítulo apresentou a arquitetura SPARC, abordando suas principais características como os bancos de registradores em janelas e o conjunto de instruções. Em seguida foi apresentado o processador Leon3 que implementa a arquitetura SPARC e é utilizado, neste trabalho, como base para a implementação do circuito decodificador PBIW-SPARC. Além disso, as principais características do conjunto de instruções SPARCv8 e da estrutura do formato ELF foram detalhadas. Durante a apresentação do projeto da técnica PBIW-SPARCv8 foram demonstrados alguns pontos relevantes à definição dos leiautes de padrão e instrução codificadas. Experimentações, resultados e análises da técnica de codificação PBIW são apresentadas no Capítulo 5.

# Capítulo 5

# Experimentos e Resultados

Neste Capítulo são apresentados experimentos e resultados sobre a utilização da codificação PBIW-SPARC. Os resultados obtidos consideram métricas como taxa de compressão, taxa de reúso de padrões, taxa de *cache miss* e melhoria de desempenho (*speedup*). Este Capítulo está organizado conforme segue: a Seção 5.1 introduz as métricas, *benchmarks* e ferramentas utilizadas ao longo do Capítulo; O resultados estáticos envolvendo a taxa de compressão de programas e taxa de reúso de padrões são apresentados na Seção 5.2; Na Seção 5.3, os resultados dinâmicos sobre cache *miss* e *speedup* são apresentados. Nessa seção também apresenta-se uma comparação entre os resultados obtidos com a técnica PBIW-SPARC e a técnica SPARC16. E por fim, na Seção 5.4 são apresentadas as codiderações finais do capítulo.

## 5.1 Introdução

Para avaliar novas técnicas de codificação de instruções, alguns indicadores são comumente utilizados:

- indicadores estáticos: taxa de compressão;
- indicadores dinâmicos: taxas de *misses* em memória cache, quantidade de bytes transferidos da memória principal, tempo de execução e desempenho.

Também há indicadores do impacto em hardware que a nova técnica pode gerar devido ao processo de decodificação: área ocupada no hardware pelo novo decodificador, consumo de potência e a frequência de operação.

No contexto da técnica PBIW-SPARC, além dos indicadores mencionados, considera-se aqui a utilização da métrica "taxa de reúso". Dependendo dos valores obtidos com o reúso de padrões por instruções codificadas, tem-se um indicativo válido e representativo sobre as taxas de compressão e, também, o desempenho final dos programas.

O experimentos visando a validação e avaliação da técnica de codificação PBIW-SPARC foram realizados utilizando alguns programas do *Simple Benchmark* [53] e programas dos

*benchmarks* MiBench [54, 55] e MediaBench [56, 57]. A escolha dos programas levou em conta o tamanho do código gerado e a utilização de algoritmos clássicos (estruturas de dados elementares, ordenação, compressão de dados, codificação de imagens, entre outros) com foco em sistemas embarcados.

Todos os programas foram compilados com o *cross-compiler* GCC 3.3.1 para arquitetura SPARCv8, disponibilizado pelo projeto ArchC [58]. Esse compilador realiza *link* edição de programas compatível com a linguagem ArchC para descrição de arquiteturas de processadores. Isso permite a simulação de chamadas de sistema em simuladores gerados sobre modelos de processadores ArchC.

Neste trabalho, dois modelos de processadores foram utilizados: um para gerar um simulador SPARCv8; e outro para gerar um simulador SPARCv8 com decodificador PBIW-SPARC embutido. Destaca-se aqui que o simulador SPARCv8 original estava disponível no site do projeto ArchC[1]. Assim, neste trabalho foi desenvolvido um novo simulador SPARCv8 com suporte à decodificação de instruções codificadas PBIW-SPARC. A importância em utilizar e desenvolver ferramentas de simulação para os programas codificados e não-codificados reside não apenas na possibilidade de obter estatísticas de desempenho e comportamento desses programas. Tais ferramentas são imprescindíveis para a validação da codificação dos programas.

## 5.2 Resultados Estáticos

Os resultados estáticos foram obtidos por meio da experimentação de programas gerados na infraestrutura de codificação PBIW (Capítulo 3) utilizando o esquema PBIW-SPARC. Esses programas foram gerados com instruções codificadas PBIW-SPARC com dois ou três bytes de tamanho.

### 5.2.1 Codificação com Instrução PBIW-SPARC de 16 bits

A codificação PBIW-SPARC com instrução codificada de 16 bits (conforme apresentado no Capítulo 4), foi avaliada por meio de experimentações sobre os programas *bmm* e *alloca_test* do *Simple Benchmark* e mais nove programas que são implementações de algoritmos clássicos e manipulação de estruturas de dados. A codificação da instrução em dois bytes é uma alternativa aos programas cujo conjunto de padrões gerados não ultrapassa 256 unidades, pois o campo de índice de padrão de oito bits é capaz de endereçar todos os padrões na *cache* de padrões (P-Cache).

A Figura 5.1 exibe o tamanho do código estático dos programas. O eixo Y indica o tamanho dos programas não-codificados (originais) e codificados com PBIW-SPARC. As barras dos programas codificados (barra à direita de cada programa) também mostram a porcentagem do tamanho do padrão e instrução em relação ao tamanho total do programa.

Os resultados mostram que a utilização do PBIW-SPARC obteve redução no tamanho de código para todos os 11 programas analisados. A taxa de compressão variou entre 58,91% e

[1] http://www.archc.org

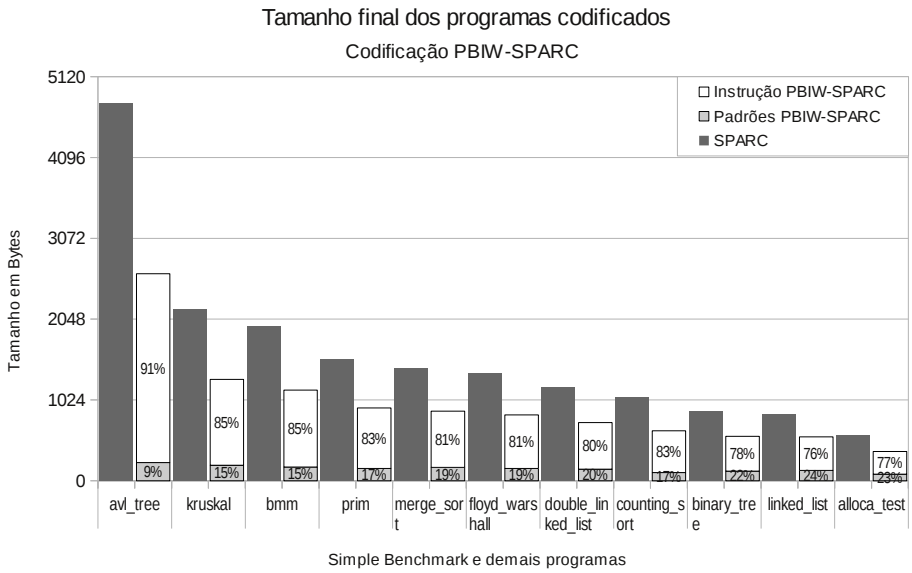



Figura 5.1: Resultados estáticos dos programas codificados no esquema PBIW-SPARC.

65,58%, com taxa de compressão média de 59,43%. Salienta-se que o tamanho do programa codificado é obtido conforme a equação:

$$\begin{aligned} \text{Tamanho Comprimido} &= (\text{Número instruções codificadas} \times \text{Tamanho instrução codificada}) \\ &+ (\text{Número padrões codificados} \times \text{Tamanho padrão codificado}) \end{aligned}$$

Conforme previamente discutido no Capítulo 3, uma das características fundamentais da técnica PBIW é a existência de sobrejeção entre o conjunto de instruções e o conjunto de padrões. A sobrejeção entre o conjunto de instruções e padrões pode ser mensurada por meio da taxa de reúso ($TR$) apresentada na Equação 3.1 da Seção 3.3. Todo projeto de codificação baseado em PBIW busca explorar o reúso de padrões, uma vez que tal reúso impactará na taxa de compressão e desempenho do programa e, também, na área ocupada pelo decodificador de instruções. Na Figura 5.1 é possível observar que programas maiores (da esquerda para a direita) tendem a obter maior redução de código, pois a sobrejeção entre o conjunto de instruções e o conjunto de padrões aumenta. A taxa de reúso de cada um dos programas avaliados é mostrada na Figura 5.2.

A taxa de reúso está compreendida no intervalo entre 4,82-15,53 padrões por instrução e com média de 7,94 padrões por instrução para o cojunto de programas avaliados. Ressalta-se que quanto maior o quociente obtido pelo cálculo da $TR$, menor é a geração de padrões com relação às instruções PBIW-SPARC.

A Figura 5.3 mostra o percentual de melhoria e a redução do programa após ser codificado em relação a sua versão não-codificada, para cada programa avaliado.

Na Figura 5.3, cada programa é representado por duas barras, sendo que a da esquerda é a versão não-codificada e da direita a versão codificada PBIW-SPARC. A barra da direita representa o programa original e a barra da esquerda, na área inferior, representa o programa

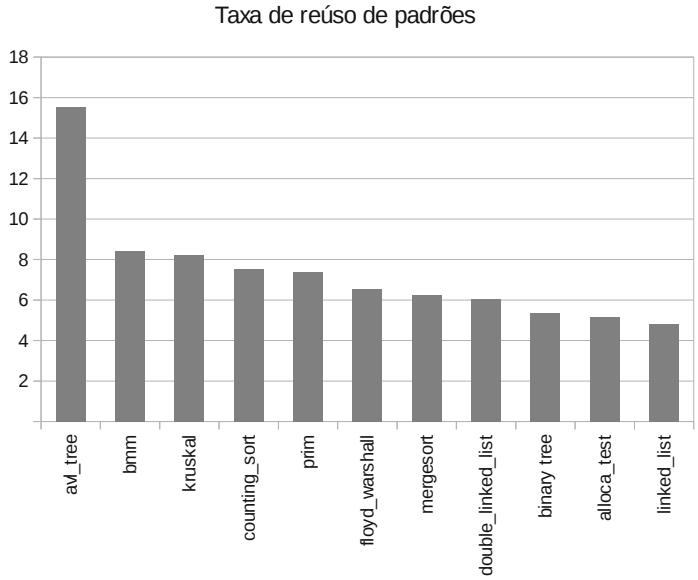



Figura 5.2: Taxa de reúso de padrões para os programas codificados no esquema PBIW-SPARC.

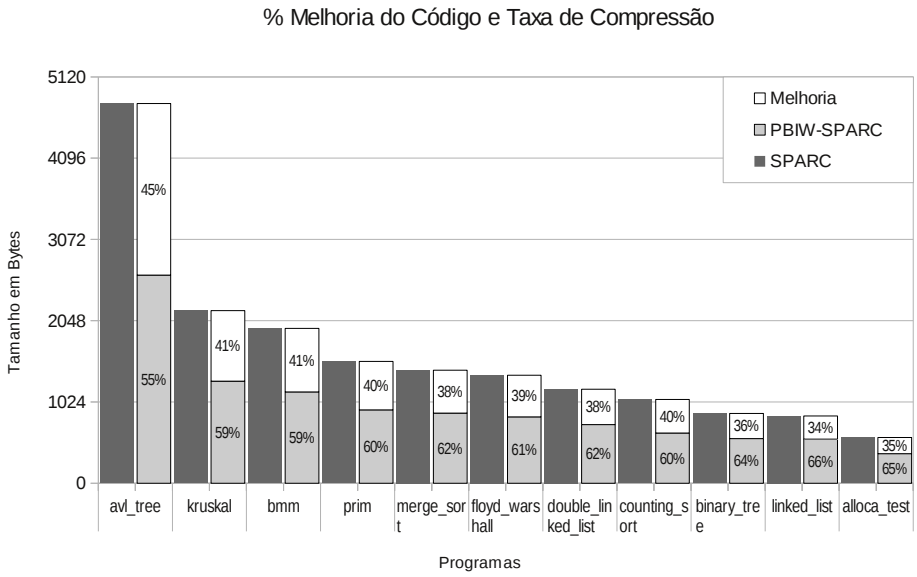



Figura 5.3: Porcentagem de melhoria e taxa de compressão para programas codificados no esquema PBIW-SPARC.

codificado e seu valor percentual em relação a sua versão não-codificada (taxa de compressão). A região superior de cada barra que representa programas codificados exibe o percentual de melhoria no tamanho de código da versão codificada.

### 5.2.2 Codificação com Instrução PBIW-SPARC de 24 bits

A codificação PBIW-SPARC com instruções codificadas de 24 bits tem como objetivo possibilitar a codificação de programas maiores, cujo número de padrões é superior a 256. Na instrução de 24 bits, 16 bits são destinados ao campo de índice de padrões e suporta até $2^{16}$ (65536) padrões. Os experimentos foram realizados com programas dos *benchmarks* MiBench e MediaBench.

As Figuras 5.4 e 5.5 mostram os tamanhos dos códigos estáticos para os conjuntos de programas dos *benchmarks* MiBench e MediaBench, respectivamente. Estas figuras estão organizadas da mesma forma que as Figuras 5.1 e 5.2.

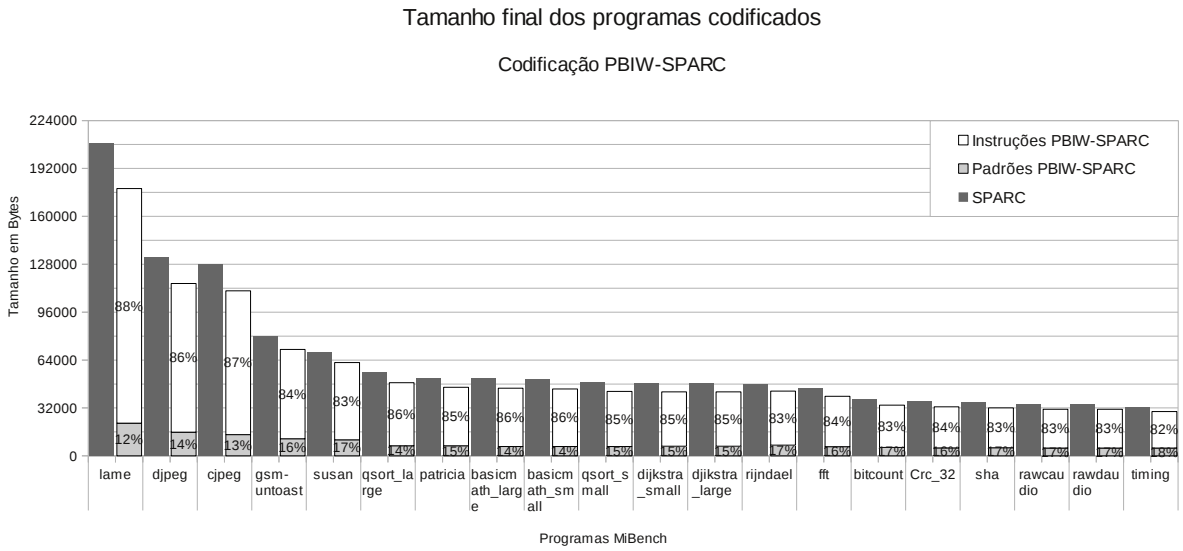



Figura 5.4: Resultados estáticos da codificação dos programas do MiBench.

Embora na codificação PBIW-SPARC de 24 bits o limite inferior de codificação aumente para 75% (conforme descrito na subseção 4.3.2), todos os programas obtiveram redução no tamanho. A taxa de compressão variou entre 85,49% e 90,91%, com média de 88,52% para os programas do MiBench. Nos programas do MediaBench a taxa variou entre 76,57% e 90,91%, com média de 86,92%.

As Figuras 5.6 e 5.7 mostram a taxa de compressão para cada programa dos *benchmarks* MiBench e MediaBench, respectivamente. Os programas *cjpeg* e *djpeg* alcançaram a maior redução de tamanho, ambos ficando 2% acima do limite de compressão inferior ótimo alcançável (75%). Os demais programas de ambos *benchmarks* reduziram seu tamanho de 9% a 15% (taxa de compressão entre 85% e 91%).

O conjunto de programas do MiBench obteve taxa de reúso de padrões entre 4,71% e 7,15% e média de 5,59%. A Figura 5.8 mostra a taxa de reúso para cada programa

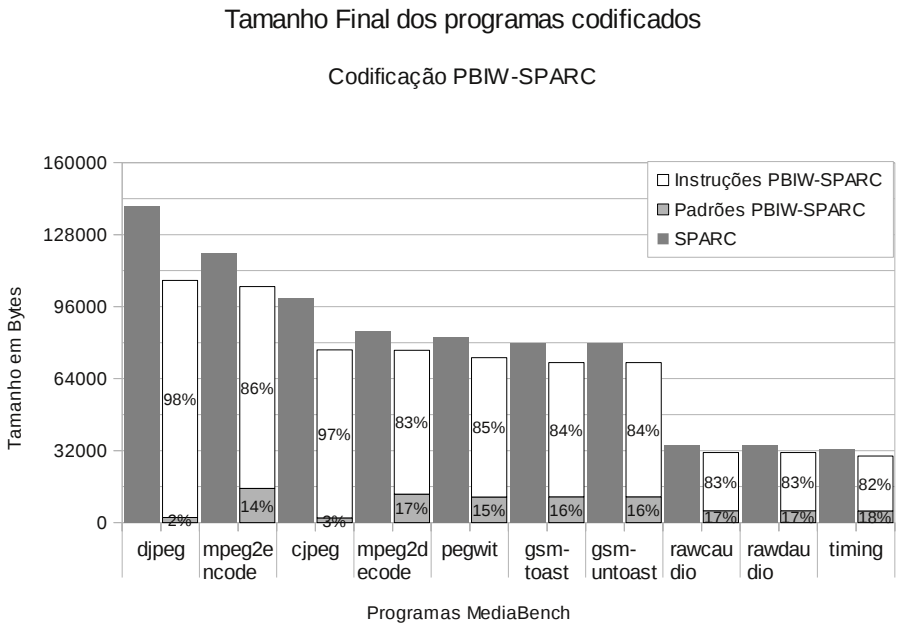



Figura 5.5: Resultados estáticos da codificação dos programas do MediaBench.

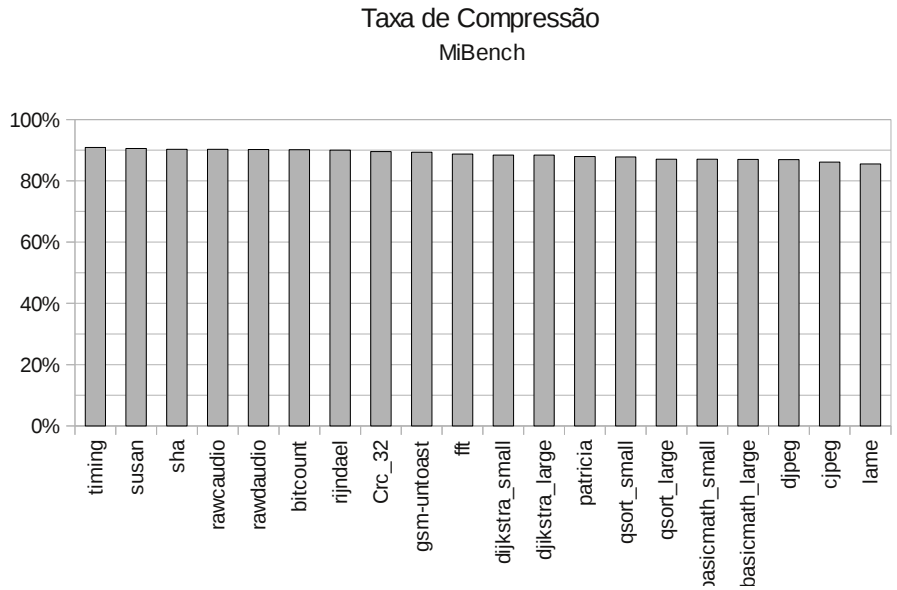



Figura 5.6: Taxa de compressão para o conjunto de programas do MiBench.

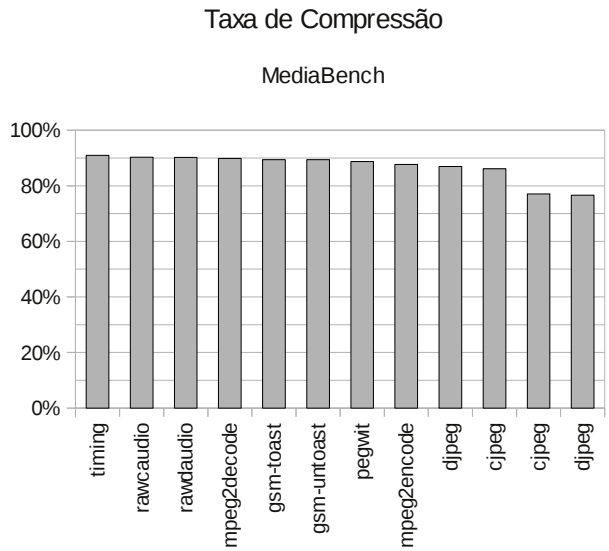



Figura 5.7: Taxa de compressão para o conjunto de programas do MediaBench.

do MiBench enquanto que a taxa de reúso para o cojunto de programas do MediaBench é apresentada na Figura 5.9. Os programas do MediaBench obtiveram taxa de reúso de padrões entre 4,71% e 47,76%, com média de 12,57%. A taxa de reúso média do MediaBench foi mais que o dobro da taxa de reúso média do MiBench, devido à alta taxa de reúso de padrões obtida por dois programas em particular: *cjpeg* (36,51) e *djpeg* (47,76). Isso se explica por haver grande quantidade de instruções (originais) idênticas e/ou similares nesses programas, ocasionando redução do número de padrões gerados e aumento do reúso.

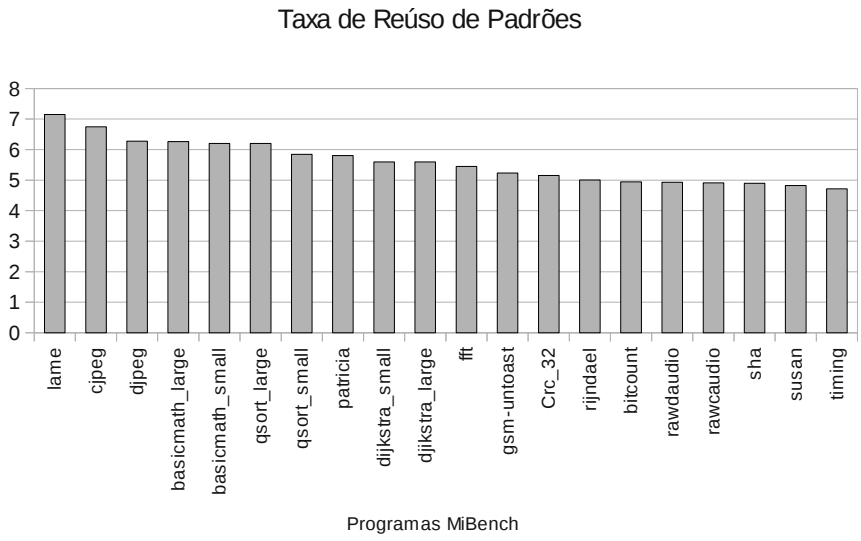



Figura 5.8: Taxa de reúso para o conjunto de programas do MiBench.

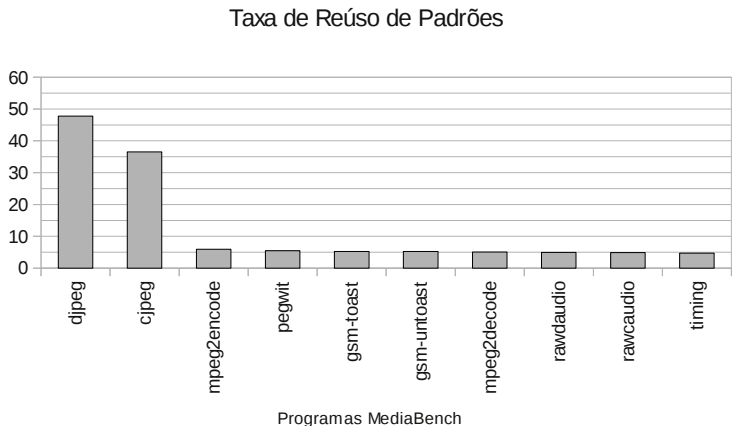



Figura 5.9: Taxa de reúso para o conjunto de programas do MediaBench.

## 5.3 Resultados Dinâmicos

Os experimentos dinâmicos mostram os efeitos dos *misses* em *cache* devido à redução do tamanho das instruções, em contrapartida à adoção de uma *cache* de padrões. O objetivo é revelar o comportamento e as implicações no desempenho dos programas com a adoção da *cache* de padrões. Os resultados dinâmicos foram obtidos com base nas seguintes métricas: quantidade de instruções emitidas; quantidade total de *misses*; quantidade de bytes transferidos da memória; tempo de execução e *speedup* PBIW-SPARC sobre SPARC.

Para obtenção dos resultados foram utilizados o simulador de *caches* Dinero [59] e o simulador da arquitetura SPARCv8 gerado pela linguagem ArchC [58].

Os passos utilizados para geração de traços de execução e avaliação do impactos na cache são apresentados a seguir:

- A partir do arquivo executável (ELF SPARC) duas versões de cada programa são mantidas, uma codificada e outra não-codificada;
- Cada versão do programa é submetida ao simulador gerado pelo ArchC, que gera os traços de execução válidos para o processador Leon3, para as versões codificada e não-codificada, respectivamente;
- Os traços de execução codificado e não-codificado de cada programa são convertidos em traços compatíveis com o Dinero, que é simulado considerando o impacto de *misses* das caches.

Os experimentos foram realizados considerando *caches* organizadas com mapeamento direto, blocos de 2 e 4 bytes e tamanhos de 128, 256 ou 512 bytes. Como a codificação PBIW-SPARC utiliza uma *cache* para padrões e outra para instruções, cada uma dessas *caches* sempre utiliza metade do tamanho da memória cache de programas não-codificados. Por exemplo, uma *cache* de 256 bytes para programas SPARC é dividida em uma *cache* de 128 bytes para padrões e 128 bytes para instruções PBIW-SPARC.

A ferramenta Dinero simula apenas *caches* com tamanhos em potências de 2. Devido a essa restrição, os experimentos foram executados com *caches* de 2 e 4 bytes por linha,

para instruções e padrões, respectivamente. Como os padrões possuem 3 bytes, há um byte excedente por linha. As estatísticas geradas pelo Dinero consideram 4 bytes por transferência, em vez de 3 bytes que cada padrão possui. Nos resultados e gráficos a seguir, esses 25% que excederam a quantidade de bytes transferidos são reduzidos da quantidade de bytes transferidos da memória e no tamanho da *cache* original mantendo a quantidade real de dados transferidos entre a memória principal e a *cache*.

Os experimentos dinâmicos foram obtidos pela simulação dos mesmos programas utilizados nos experimentos estáticos da subseção 5.2.1. Os resultados apresentados são referentes a *caches* de 256 bytes e mostram os efeitos dos *misses* na *cache* ao reduzir o tamanho da instrução em contrapartida à adoção de uma *cache* de padrões.

A Figura 5.10 mostra a porcentagem de melhoria $(1 - \frac{\text{tamanho codificado}}{\text{tamanho original}})$ na quantidade de bytes buscados da memória principal com a técnica PBIW-SPARC. Todos os programas avaliados reduziram a quantidade de bytes buscados na memória principal devido a *misses* na cache de instruções e padrões.

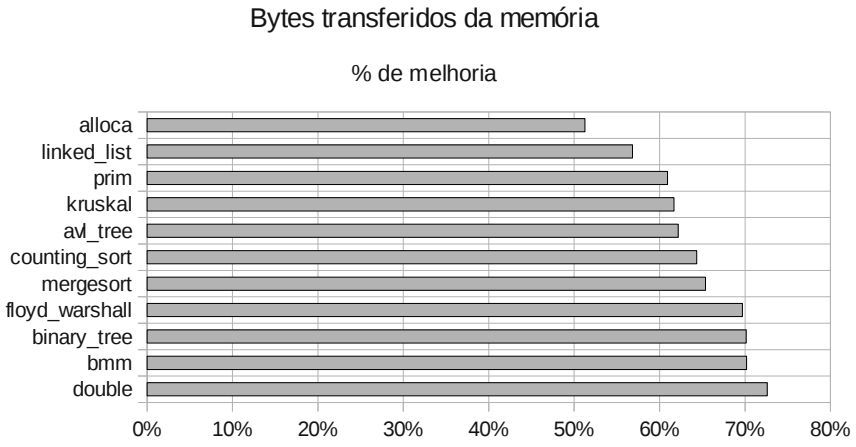



Figura 5.10: Porcentagem de melhoria na quantidade de bytes buscados da memória.

O programa que mais se beneficiou da utilização da técnica PBIW-SPARC para minimizar a ocorrência de *cache miss* foi *double_linked_list*, que buscou 72,61% menos bytes na memória, enquanto o programa *alloca_test* foi o que menos se beneficiou, buscando 51,25% menos bytes da memória. O número de bytes transferidos da memória obteve redução média de 64,11%. Essa porcentagem de melhoria está diretamente relacionada com o menor tamanho das instruções codificadas e à taxa de reúso entre instruções e padrões. Observou-se que a alta taxa de reúso (4-16) obtida com programas do *Simple benchmark* propicia a ocorrência de várias sequências de *hits* na cache de padrões mesmo diante da ocorrência de *misses* na cache de instruções.

### 5.3.1 Taxa de *Miss*: Instruções e Padrões

Esta subseção apresenta detalhes do comportamento de programas codificados com a técnica PBIW-SPARC, com relação aos acessos à memória. Além disso, também são apre-

sentados resultados de experimentos referentes à ocorrência de *cache misses*. Deve-se atentar para o fato que padrões e instruções são organizados em *caches* distintas (P-Cache e I-Cache) e que cada instrução que é buscada na memória implica na busca de um padrão. Importante salientar também que a ocorrência de *hit/miss* na busca de instrução não implica em *hit/miss* na busca do padrão e vice-versa, uma vez que os conjuntos de instruções e padrões são disjuntos e as *caches* são distintas.

O reúso de padrões também pode ser observado no contexto dinâmico. Ao considerar um padrão que é utilizado por uma única instrução e, portanto, não compartilhado, há de se considerar um impacto negativo no tamanho final do código já que |padrão| + |instrução| > |instrução original| (24 + 16 > 32). Porém, durante a execução (contexto dinâmico), essa mesma instrução pode ser buscada e executada diversas vezes, tendo como consequência a busca do respectivo padrão a cada execução. Portanto, a possível ocorrência de *hits* na busca do padrão pode compensar o prejuízo estático e ainda gerar ganhos. Em situações em que os padrões são compartilhados por diversas instruções, os ganhos podem ser ainda melhores. O reúso de padrões desacelera o processo de preenchimento da *cache* diminuindo assim a quantidade de *misses* de conflito e capacidade.

A Figura 5.11 apresenta todos os acessos à memória para os traços do programa *binary_tree* em suas versões codificada e não-codificada usando uma cache de 256 Bytes (128 bytes para as caches de instruções codificadas e padrões). As linhas representam o comportamento dos acessos à memória ao longo da execução do programa para as instruções e padrões PBIW-SPARC e instruções originais. O programa *binary_tree* foi escolhido por ser um dos menores em número de instruções, o que facilita a visualização das alterações ao longo da execução nas Figuras 5.11 e 5.12 e, principalmente, por representar bem o comportamento geral dos demais programas.

O gráfico da Figura 5.11 mostra o comportamento dos acessos à memória das instruções não-codificadas, codificadas e padrão, que possuem as respectivas legendas: SPARC, Instrução PBIW e Padrão PBIW. As instruções codificadas e não-codificadas têm suas linhas sobrepostas pois as ocorrências de *hit* e *miss* acontecem na mesma sequência. Isso se justifica devido ao tamanho da instrução codificada e da *cache* de instruções (I-Cache) serem metade do tamanho das instruções e *cache* originais, o que faz com que os padrões de acessos a tais instruções sejam similares entre os programas codificado e não-codificado.

Na Figura 5.11, o programa *binary_tree* tem sua execução iniciada na instrução 1 com 100% de *miss*. Conforme o programa realiza os acessos à memória para buscar padrões ou instruções PBIW-SPARC ou apenas instruções (versão original), a reta de *misses* permanece na mesma taxa de *miss*[2] caso aconteça um *miss*. A Figura 5.12 mostra um trecho ampliado dos traços do programa que permite visualizar ocorrências sucessivas de *misses*. No início, a reta dos padrões (mais clara) está próxima às retas das instruções codificadas (sobreposto) e não-codificadas (mais escuro), mas ao longo da execução, devido a maior ocorrência de *miss*, as retas das instruções vão se distanciando dos padrões. A execução termina na instrução 1419 com a reta dos padrões alcançando 0% de taxa de *misses* e com as retas das instruções codifcadas e não-codificadas por volta de 19% de taxa de *miss*.

[2]Não existe coeficiente angular entre dois acessos consecutivos com *misses*.

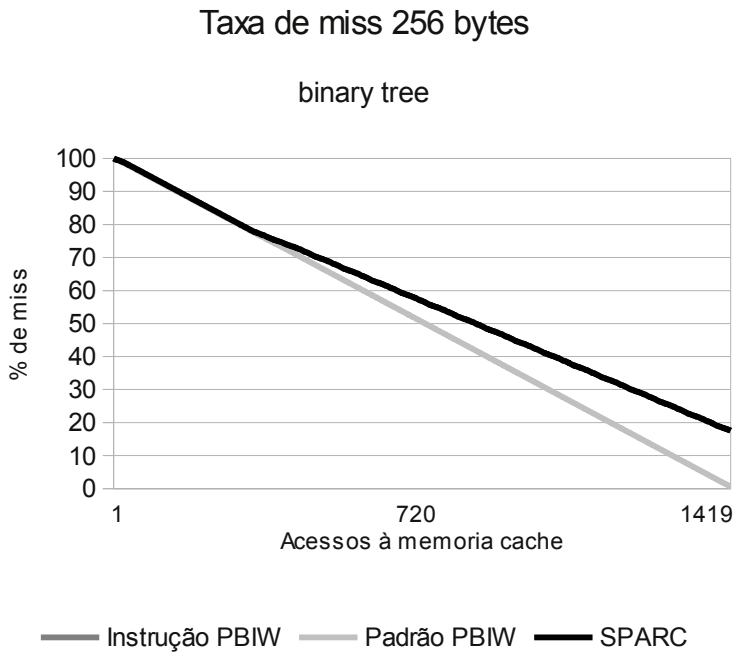



Figura 5.11: Comportamento do programa *binary_tree* quanto à ocorrência de *misses* nas caches de instruções e padrões.

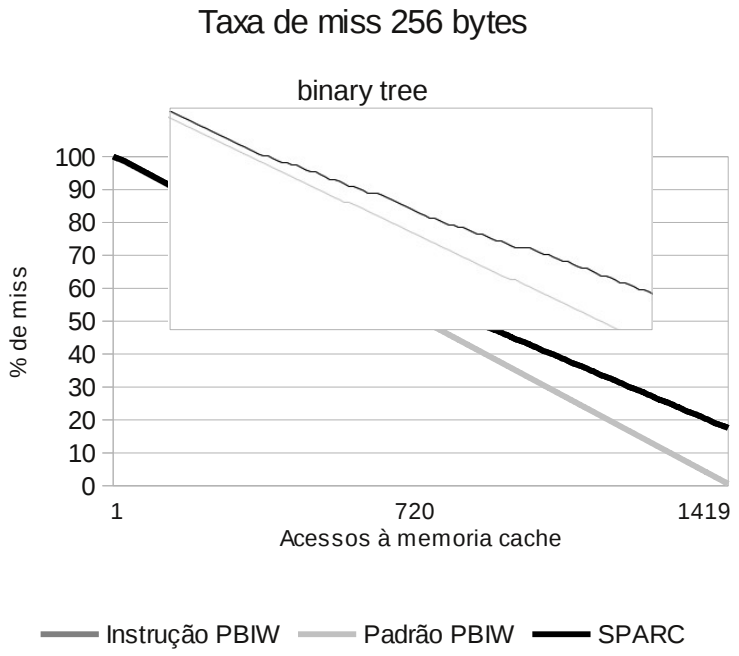



Figura 5.12: Ampliação de trecho do traço de acessos a memória da Figura 5.11.

### 5.3.2 Desempenho de Programas com a Codificação PBIW-SPARC

Esta subseção apresenta a melhoria de desempenho (*speedup*) dos programas codificados sobre a versão não-codificada. Para simplificar o cálculo de *speedup*, foi considerado que qualquer instrução pode ser executada em um ciclo de relógio, pois a ordem de execução, a quantidade de ciclos gastos ao longo do *pipeline* e a quantidade de instruções são idênticas para programas codificados e não-codificados. A ocorrência de *cache miss* é penalizada em 2 ciclos para a instrução PBIW-SPARC, 3 ciclos para o padrão e 4 ciclos para instrução SPARCv8. Optou-se por penalizar em 1 ciclo cada byte a ser buscado na memória, com o intuito de tornar a penalidade proporcional entre ambas as formas de execução, codificada ou original. Portanto o número de ciclos (CP) gastos na execução de um programa é dado por:

$$\text{CP} = \text{Número instruções executadas} + \text{Número de } \textit{misses} \times \text{Penalidade}$$

Se o programa for codificado, o Número de *misses* deve ser a soma dos *misses* de instrução e padrão PBIW-SPARC. Então, para se obter o *speedup* deve-se considerar a equação:

$$\text{Speedup} = \frac{\text{CP}_{\text{SPARCv8}}}{\text{CP}_{\text{PBIW-SPARC}}}$$

A Tabela 5.1 mostra o *speedup* obtido por programas codificados com PBIW-SPARC. Nota-se que programas com *speedup*>1 obtiveram ganhos de desempenho com a codificação PBIW-SPARC. Por exemplo, o programa *mergesort* teve $\text{CP}_{\text{SPARCv8}} = 13783$ e $\text{CP}_{\text{PBIW}} = 8235$. O *speedup* do programa *mergesort* foi $13783/8235 = 1,67$. Importante observar que as taxas de *misses* para os programas SimpleBench foram obtidas a partir da simulação em uma cache de 256Bytes para programas não-codificados. Para os programas codificados com PBIW-SPARC, essa cache é dividida em 128Bytes para instruções codificadas e 128Bytes para padrões.

Tabela 5.1: Tempo de execução e desempenho de programas do SimpleBench.

| | **Tempo de Execução (ciclos)** | | |
|---|---|---|---|
| **Programa** | **SPARCv8** | **PBIW-SPARC** | ***speedup*** |
| alloca | 1763 | 1640 | 1,07 |
| avl_tree | 12083 | 6646 | 1,82 |
| binary_tree | 3963 | 2179 | 1,81 |
| bmm | 90738 | 44108 | 2,05 |
| counting_sort | 2731 | 2260 | 1,21 |
| double | 27741 | 27229 | 1,02 |
| floyd_warshall | 38875 | 19413 | 2,00 |
| kruskal | 12591 | 8399 | 1,50 |
| linked_list | 96599 | 96299 | 1,00 |
| mergesort | 11984 | 7279 | 1,64 |
| prim | 1903 | 1201 | 1,58 |

Dentre os programas que tiveram o *speedup* avaliado (conforme mostrado na Tabela 5.1) o que obteve melhor resultado foi *bmm* com *speedup* de 2,06, significando assim um

desempenho 2× superior em relação ao programa não-codificado. O programa *linked_list*, de outra forma, não apresentou melhorias significativas em seu desempenho, uma vez que o *speedup* obtido foi 1,003. Na média, o *speeedup* obtido para o conjunto de programas foi de 1,61. Esses resultados demonstram as possibilidades da codificação PBIW-SPARC quanto à melhoria no desempenho dos programas.

As Tabelas 5.4 e 5.5 mostram a análise de tempo de execução e *speedup* para os *benchmarks* MiBench e MediaBench, simulados com uma *cache* de 2 Kbytes para os programas não-codificados. Na simulação do programa codificado, projetou-se uma cache de instruções com 1,5KBytes (aproximadamente, 75% do tamanho da *cache* original). Para padrões foi utilizada uma *cache* de tamanho aproximado de 768Bytes.

O tempo de execução foi calculado com base no número de instruções (e padrões) executados. Cada instrução original (SPARCv8), instrução codificada ou padrão buscado em sua respectiva *cache* contam 1 ciclo no cálculo do tempo de execução. A penalidade das instruções codificadas foi alterada em relação ao cálculo de desempenho dos programas SimpleBench pois, para os programas dos benchmarks MiBench e MediaBench utilizou-se uma instrução codificada de 24 bits. Nesse sentido, a penalidade dos *misses* das instruções codificadas são de 3 ciclos.

Tabela 5.2: Tempo de execução e desempenho de programas do MiBench.

| | **Tempo de Execução (ciclos)** | | |
|---|---|---|---|
| **Programa** | **SPARCv8** | **PBIW-SPARC** | ***speedup*** |
| basicmath_large | 77483455 | 67428934 | 1,15 |
| basicmath_small | 77601804 | 68171258 | 1,14 |
| bitcount | 46178898 | 46163377 | 1,00 |
| cjpeg | 25358557 | 26354139 | 0,96 |
| crc_32 | 30241274 | 30239003 | 1,00 |
| dijkstra_small | 16492 | 9426 | 1,75 |
| djikstra_large | 51354962 | 46720541 | 1,10 |
| djpeg | 8376969 | 8376988 | 1,00 |
| fft | 52424195 | 47847744 | 1,09 |
| gsm-toast | 7970 | 6794 | 1,17 |
| gsm-untoast | 8128 | 6960 | 1,17 |
| lame | 100495948 | 122425671 | 0,82 |
| patricia | 181867041 | 153602843 | 1,18 |
| qsort_large | 62399581 | 49699138 | 1,25 |
| qsort_small | 15529680 | 14782546 | 1,05 |
| rawcaudio | 34288631 | 34287094 | 1,00 |
| rijndael | 145194341 | 112322740 | 1,29 |
| sha | 13273781 | 13354768 | 0,99 |
| susan | 12632 | 11563 | 1,09 |
| timing | 1602161 | 1596845 | 1,00 |

Para o *benchmark* MiBench o melhor desempenho foi alcançado pelo programa *qsort_small*, com *speedup* de 1,75. O pior desempenho foi do programa *lame*, com *speedup* 0,82. O *speedup* médio para os programas do MiBench foi 1,11. No *benchmark* MediaBench, o pior desempenho foi do programa *cjpeg* com *speedup* de 0,96 e o melhor desempenho foi do programa *gsm-untoast* com 1,18 de *speedup*. Na média, os programas do MediaBench alcançaram um *speedup* de 1,14.

Dentre os programas analisados quanto ao *speedup*, observa-se que o programa *rijndael* do *benchmark* MiBench, foi avaliado sob a técnica de codificação SPARC16 [9], obtendo *speedup* de 1,12 considerando que cada *miss* na *cache* de instruções tem penalidade de 10

Tabela 5.3: Tempo de execução e desempenho de programas do MediaBench.

| | Tempo de Execução (ciclos) | | |
|---|---|---|---|
| **Programa** | **SPARCv8** | **PBIW-SPARC** | ***speedup*** |
| cjpeg | 15177774 | 14466009 | 0,95 |
| djpeg | 4838476 | 4948036 | 1,02 |
| gsm-toast | 7970 | 6794 | 1,17 |
| gsm-untoast | 8128 | 6960 | 1,18 |
| mpeg2decode | 68812369 | 61893189 | 1,11 |
| mpeg2encode | 61585502 | 64257217 | 0,96 |
| pegwit | 14233452 | 14033227 | 1,01 |
| rawcaudio | 7407556 | 7406485 | 1,00 |
| rawdaudio | 6412012 | 6410991 | 1,00 |
| timing | 1602161 | 1596845 | 1,00 |

ciclos e *speedup* de 1,33 com penalidade de 50 ciclos. A técnica PBIW-SPARC, considerando penalidade de 4 ciclos para instrução ou padrão, obteve *speedup* de 1,29. Na comparação de desempenho do programa *rijndael* deve-se considerar que *speedup* de 1,29 foi obtido com uma cache de 1,5KBytes enquanto que o *speedup* de 1,12 de SPARC16 foi obtido com uma cache de 2KBytes.

### 5.3.3 Comparações das codificações PBIW-SPARC e SPARC16

Nesta subseção é mostrado um comparativo entre as codificações PBIW-SPARC e SPARC16 [9], tendo como referência a versão original SPARCv8. Para efeito de comparação, foram considerados os *benchmarks* MediaBench e MiBench. As métricas utilizadas para comparação se dividem em resultados estáticos de taxa de compressão e área utilizada pelo processador Leon3 nas versões com decodificador SPARC16 e PBIW-SPARC.

#### Resultados Estáticos

As Tabelas 5.4 e 5.5 mostram a taxa de compressão obtida para cada programa dos *benchmarks* MiBench e MediaBench respectivamente, utilizando a técnica de codificação PBIW-SPARC e SPARC16.

Tabela 5.4: Codificação de programas MiBench: taxa de compressão.

| **Programa** | **SPARC16** | **PBIW-SPARC** | **Programa** | **SPARC16** | **PBIW-SPARC** |
|---|---|---|---|---|---|
| bmath_large | 58,3% | 86,98% | lame | 63,9% | 85,49% |
| bmath_small | 58,3% | 81,10% | patricia | 57,2% | 87,91% |
| bitcnts | 56,8% | 90,16% | qsort_large | 57,1% | 87,10% |
| cjpeg | 56,8% | 86,12% | qsort small | 56,7% | 87,83% |
| crc_32 | 56,7% | 89,56% | rawcaudio | 56,6% | 90,28% |
| dijkstra_small | 57,0% | 88,40% | rawdaudio | 56,6% | 90,21% |
| dijkstra_large | 57,0% | 88,40% | rijndael | 58,7% | 89,99% |
| djpeg | 57,8% | 86,95% | sha | 56,7% | 90,33% |
| fft | 57,2% | 88,76% | susan | 57,0% | 90,55% |
| gsm_toast | 58,9% | 89,32% | timing | 56,4% | 90,91% |
| gsm_untoast | 58,9% | 89,32% | | | |

A técnica de codificação SPARC16 obteve melhor taxa de compressão para ambos *benchmarks*, oscilando entre 56,4% e 63,9% e, com média de 61,02%. A codificação PBIW-SPARC

Tabela 5.5: Codificação de programas MediaBench: taxa de compressão.

| Programa | SPARC16 | PBIW-SPARC | Programa | SPARC16 | PBIW-SPARC |
|---|---|---|---|---|---|
| cjpeg | 56,8% | 77,05% | rawcaudio | 56.6% | 90,28% |
| djpeg | 57,8% | 76,57% | rawdaudio | 56.6% | 90,21% |
| mpeg2dec | 59,9% | 89,86% | timing | 56.4% | 90,91% |
| mpeg2enc | 59,9% | 87,68% | toast | 58.9% | 89,36% |
| pegwit | 58,7% | 88,75% | untoast | 58.9% | 89,36% |

obteve, para ambos *benchmarks*, taxas de compressão oscilando entre 77,05% e 90,91%, com média de 87,86%. Um dos motivos da técnica PBIW-SPARC gerar uma redução dos tamanhos dos programas abaixo da técnica SPARC16 consiste no limite inferior de compressão (para instruções codificadas de 24 bits) de 75%. Por outro lado, é interessante observar que a compressão da técnica PBIW-SPARC obtém valores mais próximos do resultado ótimo do que a técnica SPARC16. Nos resultados obtidos, o melhor resultado de SPARC-PBIW é apenas 2,7% acima do resultado ótimo enquanto que SPARC16 obtém melhor resultado 12,8% acima do ótimo. Importante também destacar que a taxa de compressão de SPARC16 tem um custo associado às trocas de contexto entre execução de instruções de 16 bits para 32 bits. Diante disso, como pode-se notar pela comparação de desempenho do programa *rijndael*, essa taxa de compressão obtida por SPARC16 nem sempre resulta em ganhos significativos de desempenho na execução do programa.

### Resultados da Síntese em FPGA

A Tabela 5.6 apresenta uma comparação aproximada da área ocupada em uma FPGA Cyclone II, entre os decodificadores das técnicas SPARC16 e PBIW-SPARC sobre o processador *soft-core* Leon 3. A configuração adotada para síntese do processador Leon3 é semelhante a utilizada no projeto do SPARC16. Ressalta-se, entretanto, que a quantidade de elementos lógicos ocupada pelo decodificador SPARC16 foi inferida a partir da síntese desse circuito em uma FPGA Altera Stratix II EP2S60.

Tabela 5.6: Área ocupada em cada técnica de codificação.

| *Recurso* | *Configuração — SPARC16* | *Configuração — PBIW-SPARC* |
|---|---|---|
| número de janelas de registradores | 8 | 8 |
| I-cache | 2 sets de 8 Kbytes | 2 sets de 8 Kbytes |
| D-cache | 2 sets de 4 Kbytes | 2 sets de 4 Kbytes |
| jtag e uart | sim | sim |
| pci | sim | sim |
| unidade de gerenciamento de memória | sim | sim |
| mul/div em hardware | sim | sim |
| trace buffer | 128 linhas | 128 linhas |
| unidade de ponto flutuante | não | não |
| número original de elementos lógicos | 12170 | 12170 |
| número de elementos lógicos com o descompressor | 13320 | 12776 |
| aumento da área | 9,4% | 4,9% |

Nessa comparação de área, vale ressaltar que embora não exista unidade de ponto flutuante implementada na síntese do Leon 3, tanto o circuito decodificador quanto o algoritmo de codificação PBIW-SPARC são compatíveis com instruções de ponto flutuante, não havendo necessidade de alteração em ambos, caso a implementação do Leon ofereça suporte

para instruções de ponto flutuante. Embora o circuito decodificador para SPARC16 tenha aumentado a área do processador em 9,4%, o circuito decodificador e a técnica de codificação SPARC16 não oferecem suporte a instruções de ponto flutuante, sendo necessárias adaptações em ambos, o que pode aumentar a área e número de elementos lógicos do processador. O circuito decodificador PBIW-SPARC foi sintetizado considerando apenas a ocupação de elementos lógicos para os circuitos combinacionais de acesso à memória de padrões, reordenação de formatos e multiplexadores. Mesmo sendo um circuito consideravelmente simples e com pouco impacto na área ocupada pelo processador, o decodificador está para realizar a decodificação de programas codificados com a técnica PBIW-SPARC.

Deve-se ressaltar que a área ocupada pelo decodificador PBIW-SPARC pode ser aumentada se a tabela de padrões for sintetizada junto à plataforma alvo na forma de uma ROM (tabela ROM de padrões). Experimentos realizados mostraram que uma tabela de padrões com, 256 padrões ocupa 2191 elementos lógicos. Logo, ao utilizar essa tabela de padrões sintetizada em hardware, a área ocupada pelo circuito decodificador PBIW-SPARC passa a ser de 14361 com impacto de 18% na área ocupada pelo processador Leon3.

## 5.4 Considerações Finais

Este capítulo apresentou os experimentos e resultados realizados para avaliar a codificação PBIW-SPARC, tanto no contexto estático, obtido via infraestrura de codificação (*software*), quanto no contexto dinâmico obtido por meio da simulação dos programas codificados e não-codificados. Na avaliação dinâmica foram utilizados o simulador de *caches* Dinero e modelos ArchC para geração de simuladores funcionais para arquitura SPARCv8 com e sem decodificador. A técnica PBIW-SPARC alcançou taxa de compressão média de 59,43% na codificação com instruções de 16 bits e 87% com instruções de 24 bits. Foi demonstrado o comportamento da codificação PBIW-SPARC e traço de execução considerando as ocorrências de *miss* e *hit*. O próximo Capítulo apresenta as conclusões finais sobre o trabalho, as contribuições e possíveis trabalhos futuros.

# Capítulo 6

# Conclusões e Trabalhos Futuros

Este trabalho apresentou o projeto e implementação de um codificador de instruções, baseado na técnica de codificação PBIW, para o conjunto de instruções SPARCv8 (PBIW-SPARC). A aplicação do algoritmo de codificação PBIW foi realizada utilizando-se a infraestrutura de codificação PBIW. Além da proposta de uma nova codificação para processadores baseados na ISA SPARCv8, este trabalho explorou o projeto de decodificadores de instruções junto à via de dados do processador Leon 3.

O algoritmo de codificação foi projetado incialmente para instruções codificadas de 16 bits que suportam endereçar até 256 padrões e possuem limite inferior de compressão de aproximadamente 50%. Devido à limitada capacidade de endereçamento de padrões da instrução codificada de 16 bits, houve a necessidade de projetar uma versão extendida da instrução codificada, com 24 bits, capaz de endereçar até 65536 padrões. Com novas instruções codificadas de 24 bits o limite inferior de compressão passa a ser de, aproximadamente, 75%. A especificação (tamanho e formato) do padrão de instruções é único para ambas as versões de instruções que se diferenciam apenas pelo tamanho do campo de índice de padrões.

O algoritmo de codificação extrai o código binário do arquivo executável ELF do programa (seção `.text`), organiza o código e então realiza a codificação. Após o processo de codificação, as instruções codificadas são armazenadas na seção `.text` do arquivo ELF, que passa a ter duas novas seções (`.pattern` e `.layout`) que armazenam os padrões gerados na codificação e o leiaute de padrão e instrução codificada utilizados.

Os programas codificados sob a técnica de codificação PBIW-SPARC obtiveram redução de tamanho do código (taxa de compressão inferior a 100%) em todos os programas avaliados. A versão com instrução de 16 bits, conseguiu uma taxa de compressão média de 59,43%, dentro de uma faixa de 58,91%-65,58%. Já a versão com instrução de 24 bits obteve taxa compressão entre 76,57%-90,91% com média de 87,72%. Ambas as versões da codificação PBIW-SPARC para o processador Leon3 obtiveram resultados satisfatórios no quesito de redução de memória necessária aos programas experimentados.

Na simulação do comportamento dinâmico, foram avaliados taxa de *cache miss* e desempenho (*speedup*). Para geração dos traços de execução dos programas, dois simuladores gerados com ArchC foram utilizados. Um simulador SPARCv8 (disponibilizado no site do ArchC), para gerar os traços de execução dos programas originais e um simulador PBIW-SPARC que foi desenvolvido para gerar os traços de execução e validação dos programas

codificados. Os traços de execução foram utilizados como entrada para o simulador de *caches* Dinero para geração de dados estatísticos referentes aos acessos a memória, utilizados para obtenção das taxas de *cache misses* e para o cálculo do tempo de execução, utilizado no cálculo do *speedup*. Na versão com instrução de 16 bits, o *speedup* médio foi 1,61. O menor desempenho obteve *speedup* de 1,003 e o melhor desempenho alcançado obteve *speedup* de 2,06. Já a versão com instrução de 24 bits, o pior desempenho com *speedup* de 0,82 e o melhor com *speedup* de 1,75. O *speedup* médio foi 1,12.

Quanto ao circuito decodificador, o processador Leon 3 teve sua área aumentada em 4,9%. Deve-se ressaltar que a área ocupada pelo decodificador PBIW-SPARC considera a existência apenas do circuito decodificador de instruções. Se uma tabela de padrões for sintetizada, como memória ROM, junto ao circuito, essa área pode aumentar significativamente, ultrapassando, inclusive, a área ocupada pelo decodificador SPARC16, com impacto de 8,70% na área ocupada pelo processador Leon3.

Além dos resultados demonstrando a viabilidade da técnica de codificação PBIW-SPARC e da técnica PBIW para conjuntos de instruções RISC, o algoritmo construído pode ser facilmente estendido para outras ISAs similares à SPARCv8, como MIPS e ARM. Além da codificação PBIW-SPARC, foi desenvolvida a heurística de junção de padrões (seção 3.2) que foi incluída na infraestrutura de codificação PBIW e ajudou a melhorar as taxas de compressão para o processador $\rho$-VEX. Esses resultados podem ser conferidos em [10].

## 6.1 Trabalhos futuros

Alguns trabalhos futuros podem ser enumerados para a melhoria da infraestrutura de codificação PBIW desenvolvida neste trabalho:

1. Desenvolvimento de um algoritmo paramétrico para codificação PBIW. O algoritmo codifica e gera instruções e padrões com base em um arquivo de configuração onde são descritos os parâmetros e formatos de instrução e padrão PBIW desejados;
2. Inclusão, na infraestrutura de codificação, de mecanismos automáticos para validação, simulação e avaliação de desempenho da técnica de codificação projetada;
3. Integração da infraestrutura de codificação como *back-end* de algum compilador de produção para emissão de código já codificado em PBIW ao final da compilação;
4. Desenvolvimento de técnicas de otimização para codificação PBIW-SPARC;
5. Experimentos envolvendo a codificação parcial de programas PBIW-SPARC de forma parecida com as demais técnicas de codificação de programas.
6. Adaptação do algoritmo para outras ISAs na infraestrutura de codificação como exemplo: MIPS, ARM, x86, etc;
7. Avaliação experimental do decodificador de instruções PBIW-SPARC considerando potência dinâmica dissipada e aumento do caminho crítico do processador.

# Referências Bibliográficas


[1] R. A. MARKS, "Infraestrutura para Codificação de Instruções Baseada em Fatoração de Padrões," Master's thesis, UFMS, Brazil, Novembro 2012.

[2] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic Coding for Data Compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[3] S.-J. Nam, I.-C. Park, and C.-M. KYUNG, "Improving Dictionary-Based Code Compression in VLIW Architectures," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E82-A, pp. 2318–2324, 1999.

[4] L. Goundge and S. Segars, "Thumb: Reducing the Cost of 32-bit RISC Performance in Portable and Consumer Applications," *Proceedings of Computer Society Conference*, 1996.

[5] A. Holdings, *ARM7TDMI Technical Reference Manual*, 2001. Página 1-7.

[6] A. Holdings, "ARM1156T2-S Technical Reference Manual," tech. rep., 2005-2007. pp. 35-36.

[7] K. D. Kissell, "Mips16: High-density MIPS for the Embedded Market," Real Time Systems, 1997.

[8] L. L. Ecco, "SPARC16: Uma Nova Visão para Compressão de Código para Processadores SPARC." Exame de Qualificação de Mestrado, 2008. Instituto de Computação - Universidade Estadual de Campinas - Campinas-SP.

[9] L. L. Ecco, B. C. Lopes, E. C. Xavier, R. Pannain, P. Centoducatte, and R. J. de Azevedo, "SPARC16: A New Compression Approach for the SPARC Architecture," in *Proceedings of the 21st International Symposium on Computer Architecture and High Performance Computing - SBAC-PAD*, (Washington, DC, USA), pp. 169–176, IEEE Computer Society, 2009.

[10] R. Marks, F. Araújo, R. Santos, F. Yonehara, and R. Santos, "Design and Implementation of the PBIW Instruction Decoder in a Softcore Embedded Processor," in *13th Simpósio de Sistemas Computacionais (WSCAD-SSC)*, pp. 110–117, IEEE, 2012.

[11] I. SPARC International, "The SPARC Architecture Manual," tech. rep., SPARC International, Inc., 1992.

[12] A. Gaisler, *GRLIB IP Core User's Manual*, 2012. Página 694-719.

[13] Linux Foundation: Referenced Specification, "Executable and Linkable Format (ELF)." http://refspecs.linuxfoundation.org/, 1995.

[14] J. R. Levine, *Linkers and Loaders*. Morgan Kaufmann Publishers, 1 ed., 2000.

[15] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM SigArch*, vol. 23, pp. 20–24, March 1995.

[16] A. Saulsbury, F. Pong, and A. Nowatzyk, "Missing the Memory Wall: The Case for Processor/Memory Integration," in *ISCA*, (Philadelphia), pp. 90–101, ACM, October 1996.

[17] S. A. McKee, "Reflections on the Memory Wall," in *Procs. of the* $1^{st}$ *ACM Computing Frontiers*, pp. 162–167, ACM, April 2004.

[18] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, "Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings," in *Proceedings of the ACM/IEEE* $29^{th}$ *Annual International Symposium on Microarchitecture*, pp. 201–211, October 1996.

[19] Y. Xie, W. Wolf, and H. Lekatsas, "A Code Decompression Architecture for VLIW Processors," in *Proceedings of the* $34^{th}$ *IEEE/ACM International Symposium on Microarchitecture*, pp. 66–75, IEEE Computer Press, 2001.

[20] K. Kissell, "MIPS16: High-density MIPS for the Embedded Market," tech. rep., Silicon Graphics MIPS Group, 1997.

[21] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting, "Code Compression," in *Proceedings of Programming Language Design and Implementation (PLDI)*, pp. 358–365, ACM, 1997.

[22] M. Kozuch and A. Wolfe, "Compression of Embedded System Programs," in *IEEE ICCS*, (Washington, DC, USA), pp. 270–277, IEEE Computer Society, 1994.

[23] G. Araujo, P. Centoducatte, R. J. Azevedo, and R. Pannain, "Expression Tree Based Algorithms for Code Compression on Embedded RISC Architectures," *IEEE Transactions on VLSI Systems*, vol. 8, pp. 530–533, October 2000.

[24] S. K. Menon and P. Shankar, "Space/Time Tradeoffs in Code Compression for the TMS320C62x Processor," Tech. Rep. IISc-CSA-TR-2004-4, Indian Institute of Science, India, 2004.

[25] R. Montserrat and P. Sutton, "Compiler Optimization and Ordering Effects on VLIW Code Compression," in *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pp. 95–103, ACM, November 2003.

[26] J. Prakash, C. Sandeep, P. Shankar, and Y. N. Srikant, "Experiments with a New Dictionary Based Code-Compression Tool on a VLIW Processor," Tech. Rep. IISc-CSA-TR-2004-5, Indian Institute of Science, India, 2004.

[27] M. Ros and P. Sutton, "Code Compression Based on Operand-Factorization for VLIW Processors," in *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pp. 559–569, ACM, September 2004.

[28] R. Batistella, "PBIW: Um Esquema de Codificação Baseado em Padrões de Instrução," Master's thesis, Instituto de Computação - Universidade Estadual de Campinas, Campinas-SP, Fevereiro 2008.

[29] R. Santos, R. Batistella, and R. Azevedo, "A Pattern Based Instruction Encoding Technique For High Performance Architectures," *Int. J. High Perform. Syst. Archit.*, vol. 2, pp. 71–80, Mar. 2009.

[30] M. S. Schlansker and B. R. Rau, "EPIC: An Architecture for Instruction-Level Parallel Processors," Tech. Rep. 99-111, Hewlett Packard Laboratories Palo Alto, February 2000.

[31] M. S. Schlansker and B. R. Rau, "EPIC: Explicitly Parallel Instruction Computing," *IEEE Computer*, vol. 33, pp. 37–45, February 2000.

[32] M. Len and I. Vaitsman, "VLIW: Old Architecture of the New Generation," Mar. 2011. http://ixbtlabs.com/articles2/vliw/.

[33] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain, "Code Compression Based on Operand Factorization," in *Proceedings of the* $31^{st}$ *Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pp. 194–201, IEEE Computer Society, 1998.

[34] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting, "Code Compression," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pp. 358–365, ACM Press, 1997.

[35] M. Franz and K. Thomas, "Slim Binaries," *Communications of the ACM*, vol. 40, no. 2, pp. 87–94, 1997.

[36] A. Glaiser, "LEON 3 Processor." http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=13&Itemid=53, Mar. 2011.

[37] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," vol. 11, pp. 91–99, Springer India, in co-publication with Indian Academy of Sciences, 2006. 10.1007/BF02837279.

[38] Advanced RISC Machines Ltd. http://www.arm.com, Março 2012.

[39] MIPS Technologies. http://www.mips.com, Fevereiro 2012.

[40] SPARC International, Inc. http://www.sparc.org, Fevereiro 2012.

[41] W. R. A. Dias and E. D. Moreno, "CPB-ARM – A New Code Compression Method for Embedded Systems," in *Proceedings of the 13th Symposium on Computer Systems (WSCAD-SSC)*, pp. 25–32, 2012.

[42] A. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto, "Survey of Code-Size Reduction Methods," *ACM Computing Survey*, vol. 35, no. 3, pp. 223–267, 2003.

[43] E. A. Billo, ""Projeto e Implementação de um Descompressor PDC-ComPacket em um Processador SPARC"," Master's thesis, Unicamp, Abril 2005.

[44] J. Goodacre and A. N. Sloss, "Parallelism and the ARM Instruction Set Architecture," *Computer*, vol. 38, pp. 42–50, July 2005.

[45] R. Batistella, R. Santos, and R. Azevedo, "A New Technique for Instruction Encoding in High Performance Architectures," Tech. Rep. IC-07-27, Institute of Computing - State University of Campinas, September 2007.

[46] R. Batistella, "PBIW: Um Esquema de Codificação Baseado em Padrões de Instrução," Master's thesis, Unicamp, Brazil, Fevereiro 2008.

[47] R. F. dos Santos, "Otimizações para Codificadores de Instruções." Exame de Qualificação de Mestrado, 2012. Faculdade de Computação - Universidade Federal de Mato Grosso do Sul - Campo Grande-MS.

[48] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Elsevier, 2005.

[49] Hewlett-Packard Laboratories, "VEX Toolchain." Disponível em: http://www.hpl.hp.com/downloads/vex/, Março 2011.

[50] D. A. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 4 ed., 2007.

[51] I. Free Software Foundation, "GNU General Public License." http://http://www.gnu.org/licenses/gpl-2.0.html, June 1991.

[52] ARM, "AMBA protocol specifications and design tools.." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.amba/index.html, ARM 2010.

[53] L. N. Chakrapani, J. Gyllenhaal, W. Mei, W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah, "Trimaran - An Infrastructure for Research in Instruction-Level Parallelism," *Lecture Notes in Computer Science*, vol. 3602, pp. 32–41, 2004.

[54] T. M. A. T. M. Matthew R. Guthaus; Jeffrey S. Ringenberg; Dan Ernst, "MiBench Version 1.0." [online], 2001. http://www.eecs.umich.edu/mibench/.

[55] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, (Washington, DC, USA), pp. 3–14, IEEE Computer Society, 2001.

[56] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench Consortium." [online], 1997. http://euler.slu.edu/~fritts/mediabench/.

[57] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," in *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pp. 330–335, 1997.

[58] L. Computers Systems Laboratory IC UNICAMP, "ArchC - Architechture Description Language." [online]. http://archc.sourceforge.net/.

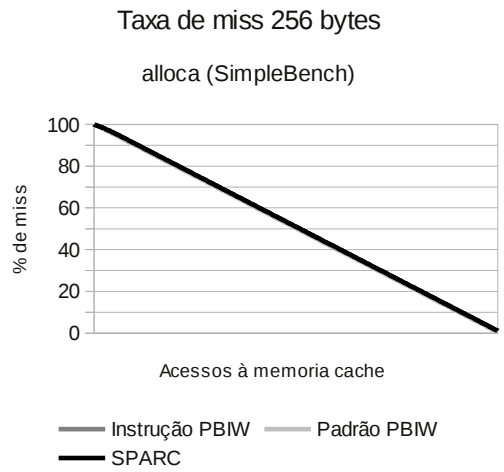
[59] J. Edler and M. D. Hill, "Dinero IV Trace-Driven Uniprocessor Cache Simulator." [online], 1995. http://www.cs.wisc.edu/~markhill/DineroIV/.

# Apêndice A

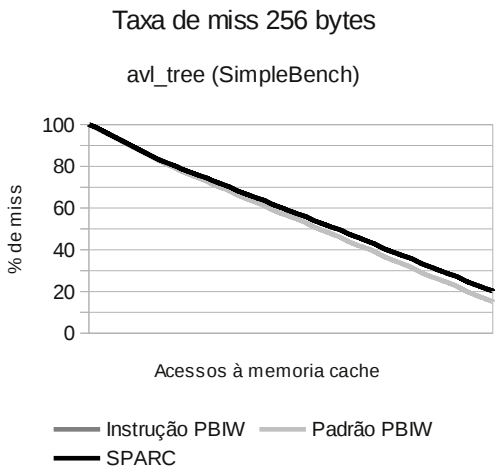
# Gráficos para taxa de *miss*

Os gráficos contidos neste apêndice são referentes ao comportamento dinâmico de execução dos *benchmarks* utilizados neste trabalho. Nos programas do *benchmark* SimpleBench, os gráficos representam o comportamento dos acessos à memória *cache* de 256 bytes. Para os programas dos *benchmarks* MiBench e MediaBench os gráficos mostram as taxas de *cache misses* para *caches* de tamanho: 0,5 Kbytes, 1 Kbytes, 2 Kbytes, 4 Kbytes e 8 Kbytes.

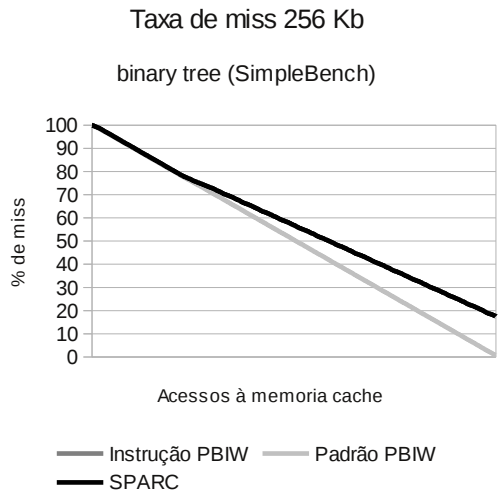
## A.1 SimpleBench



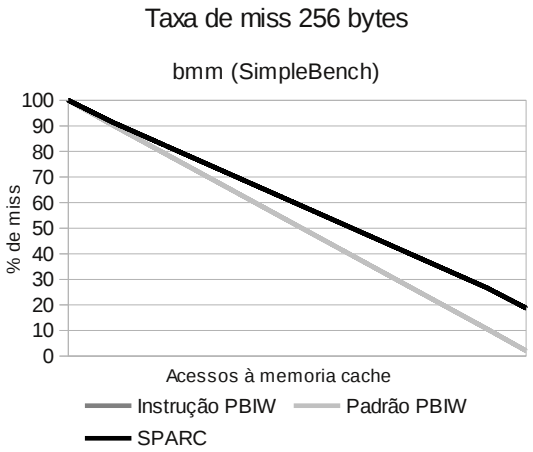

(A.1) alloca



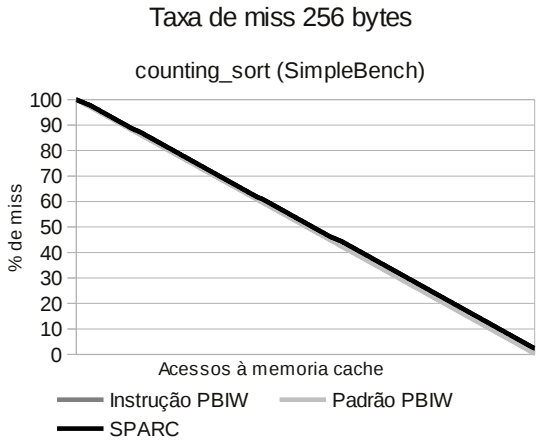

(A.2) avl_tree



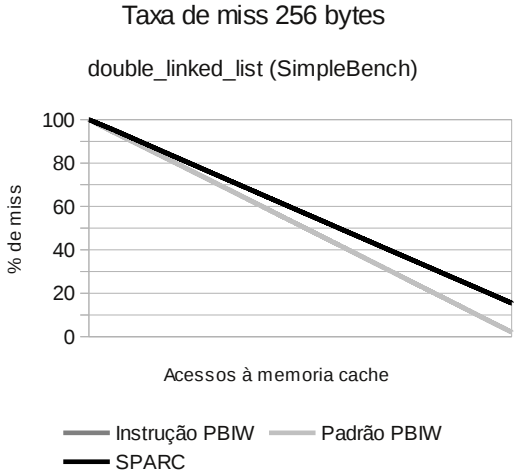

(A.3) binary_tree



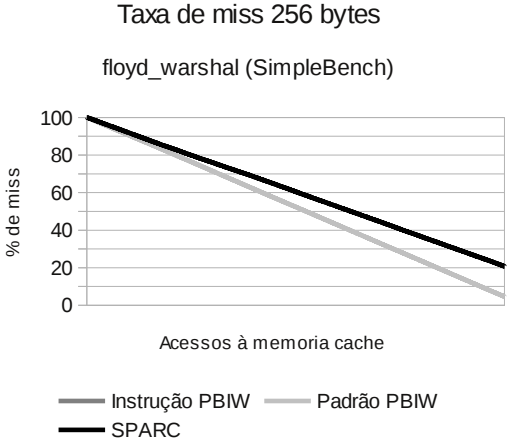

(A.4) bmm



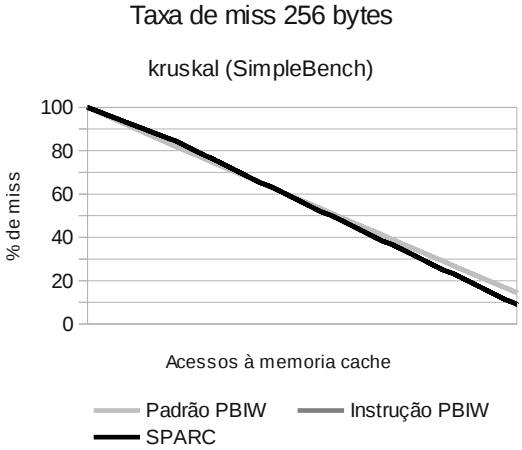

(A.5) counting_sorte



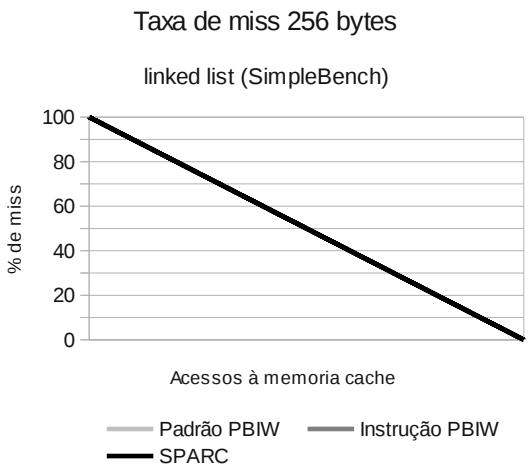

(A.6) double_linked_list



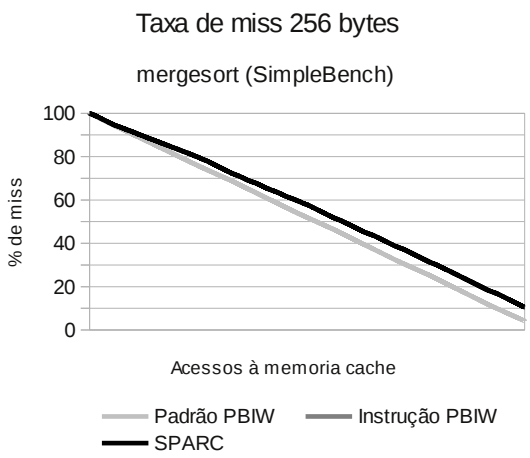

(A.7) floyd_warshall



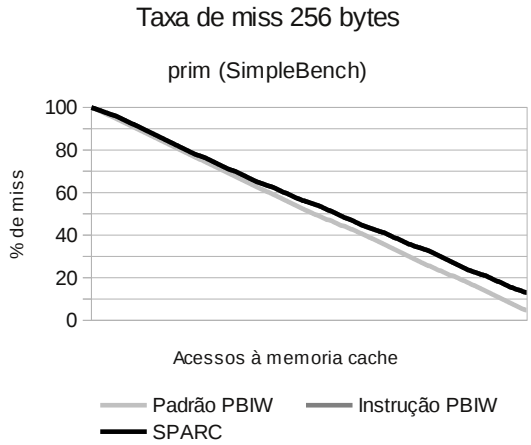

(A.8) kruskal




(A.9) list_list

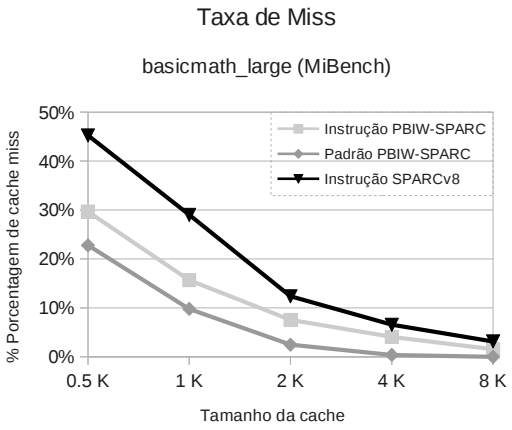



(A.10) mergesort

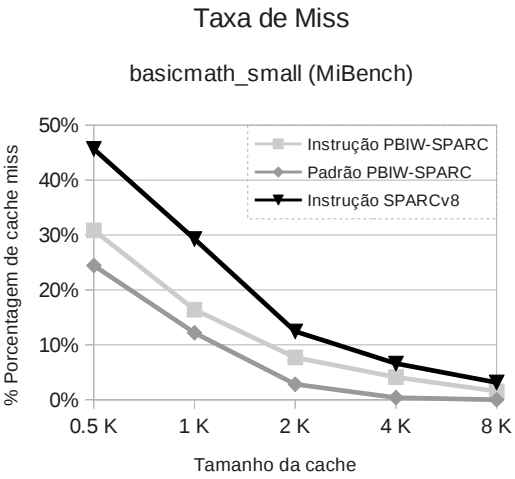



(A.11) prim

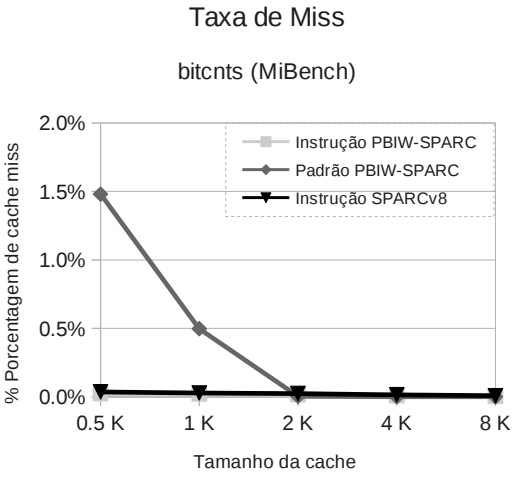
## A.2 MiBench



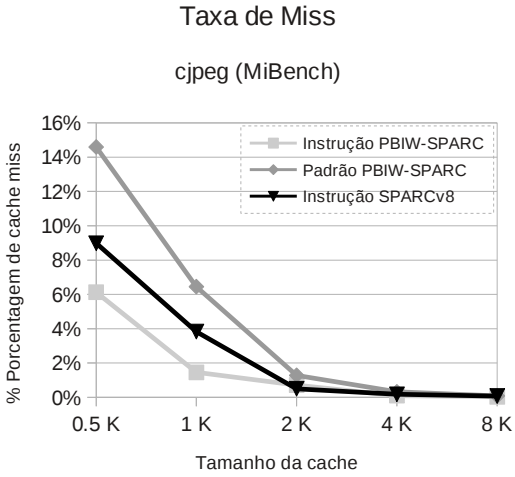

(A.12) basicmath_large



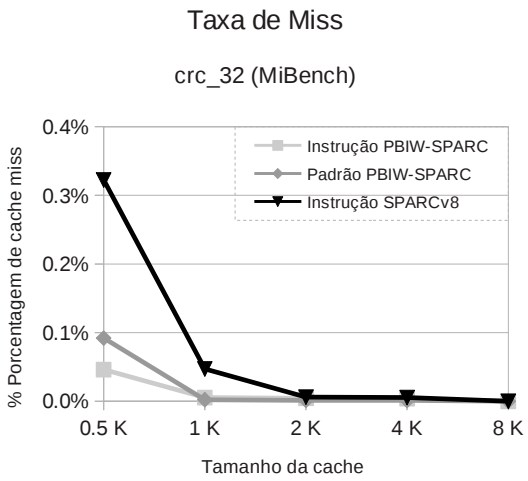

(A.13) basicmath_small



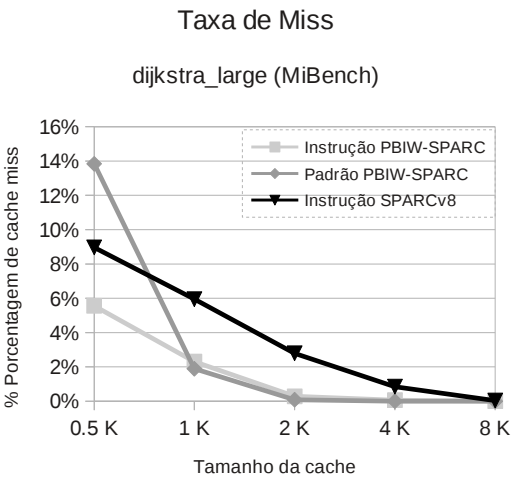

(A.14) bitcnts



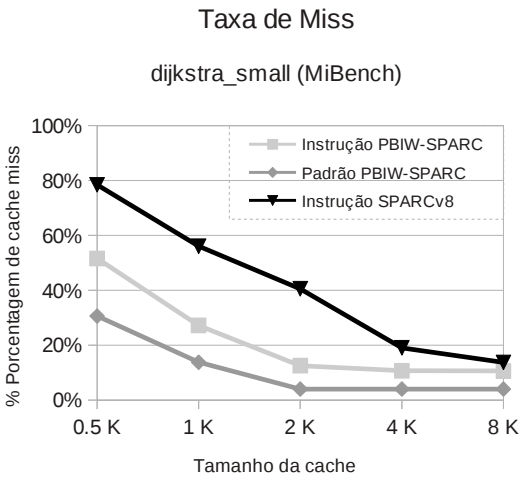

(A.15) cjpeg



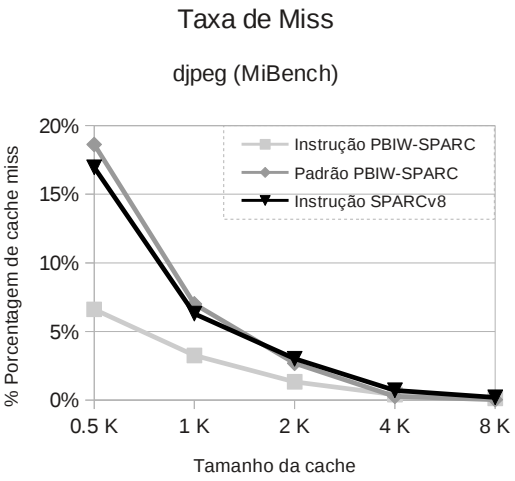

(A.16) crc_32



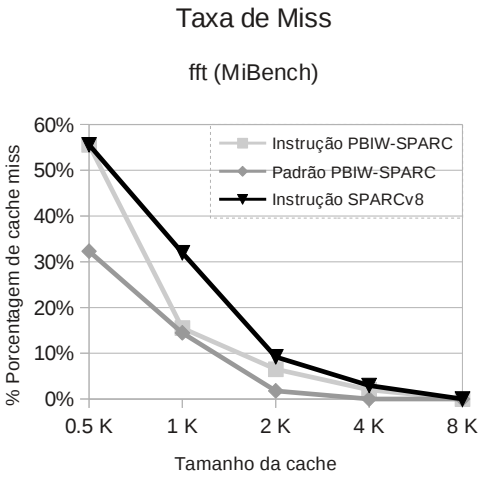

(A.17) dijkstra_large



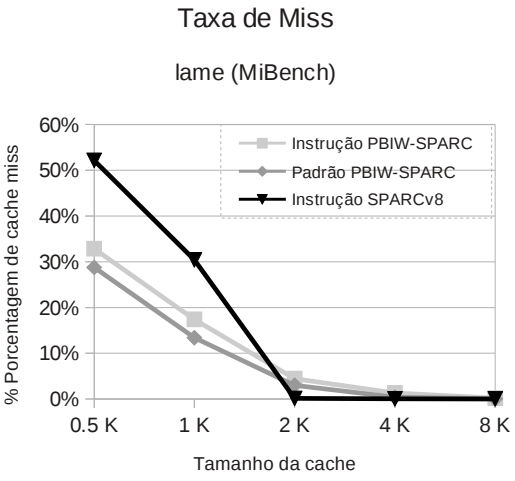

(A.18) dijkstra_small



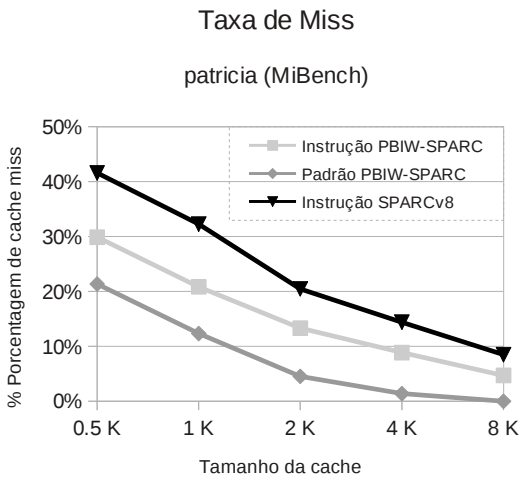

(A.19) djpeg



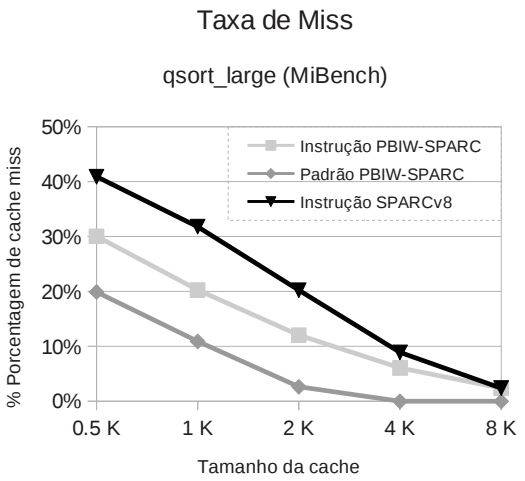

(A.20) fft



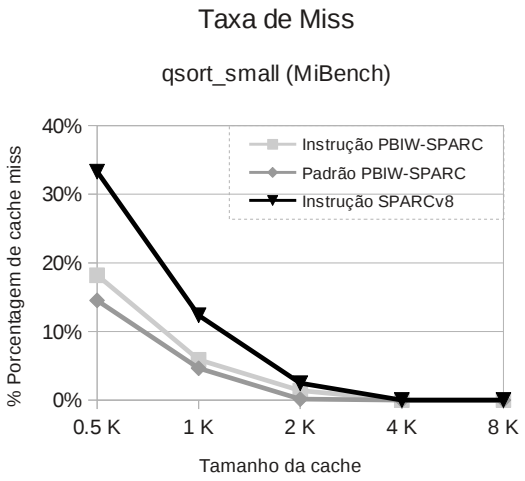

(A.21) lame



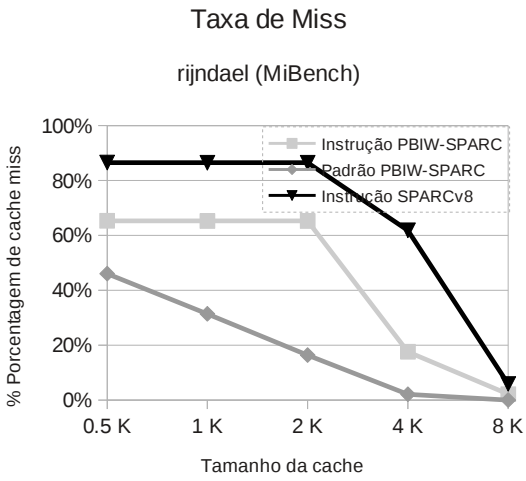

(A.22) patricia



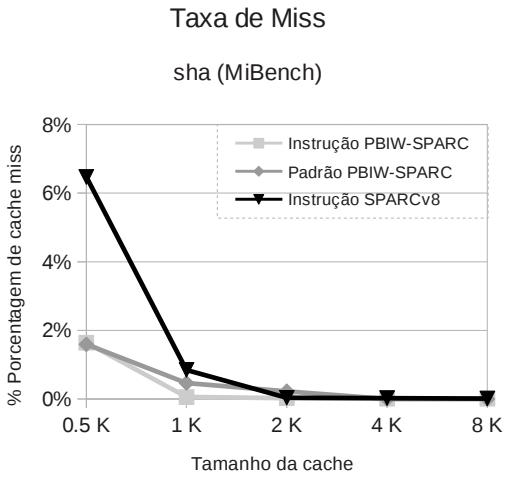

(A.23) qsort_large



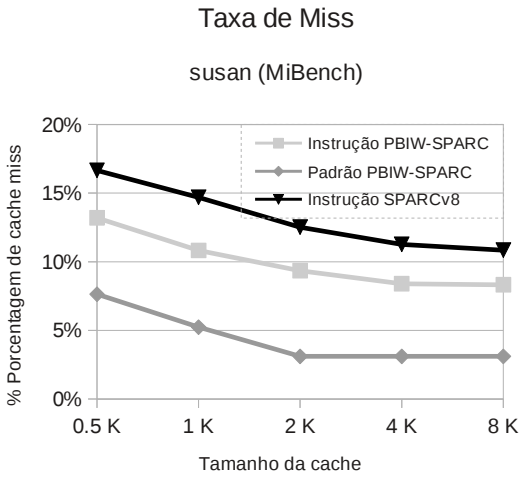

(A.24) qsort_small



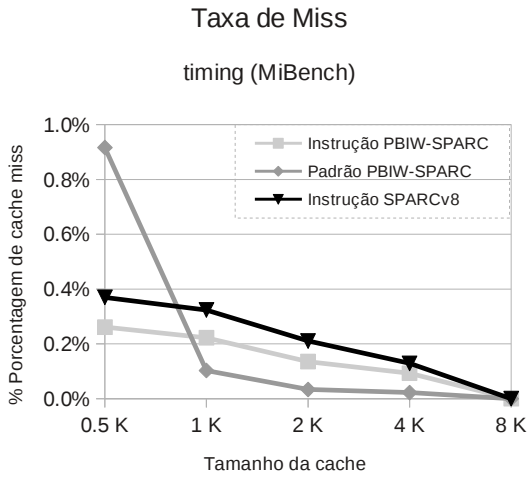

(A.25) rijndael



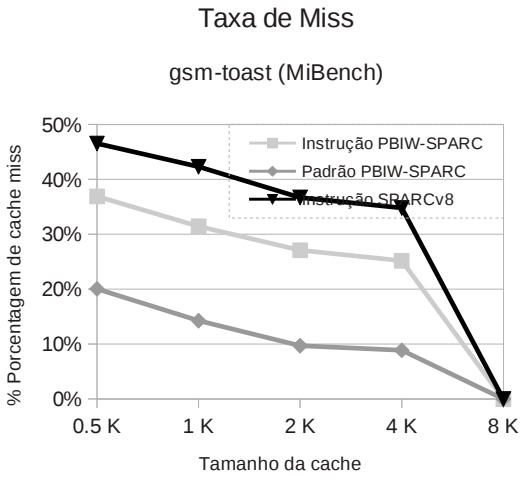

(A.26) sha



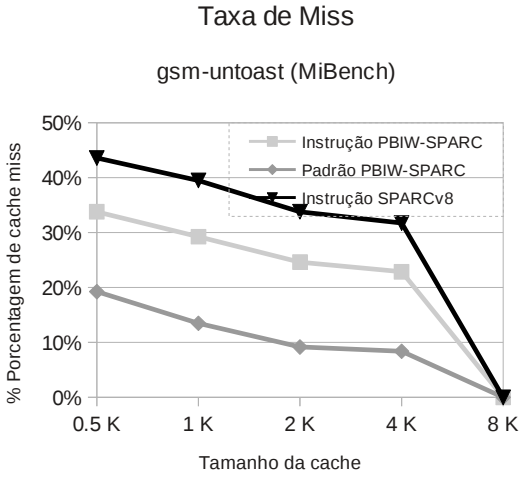

(A.27) susan




(A.28) timing

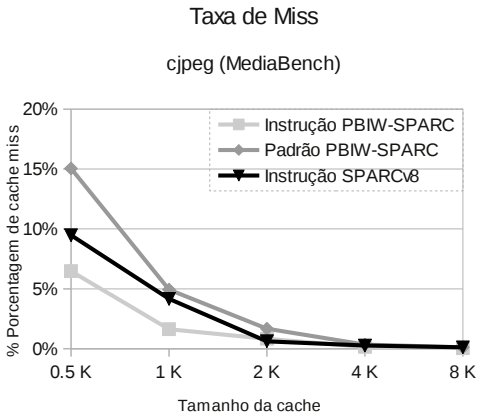



(A.29) toast

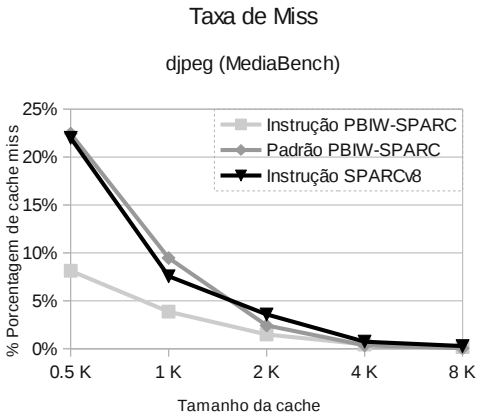



(A.30) timing

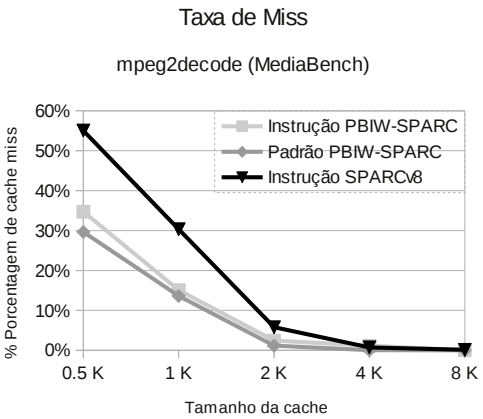
## A.3 MediaBench



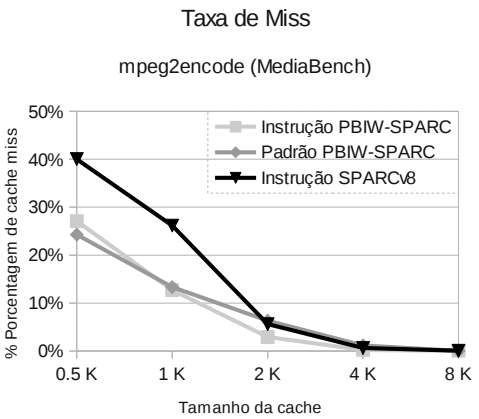

(A.31) cjpeg



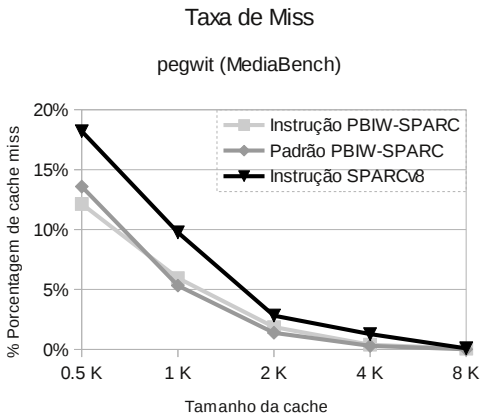

(A.32) djpeg



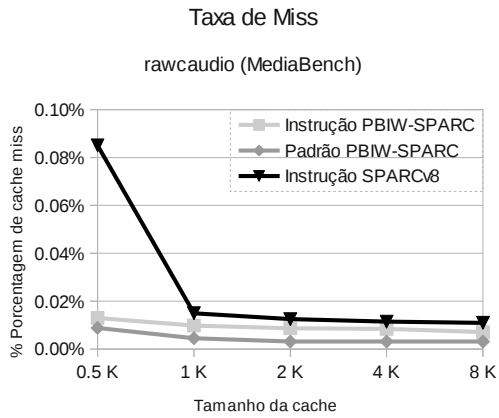

(A.33) mpeg2decode



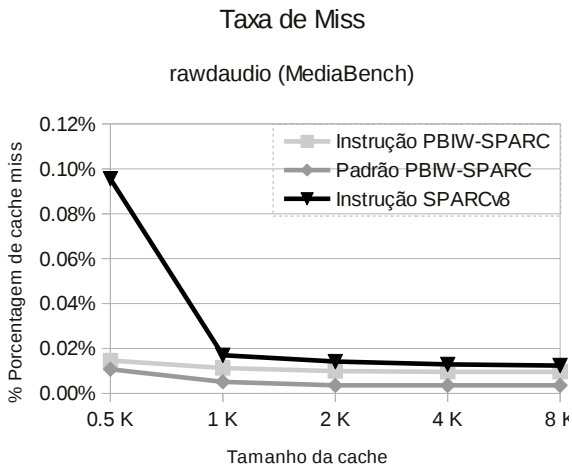

(A.34) mpeg2encode



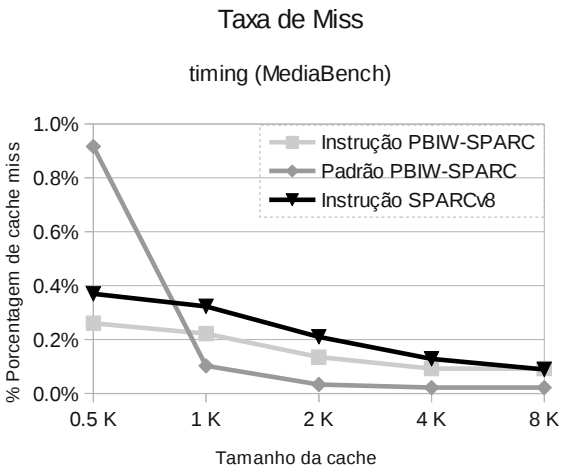

(A.35) pegwit



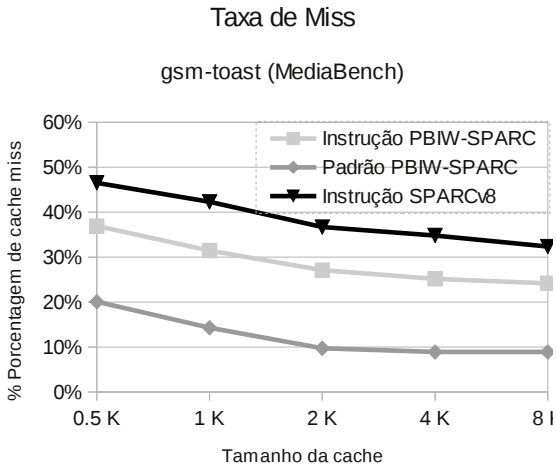

(A.36) rawcaudio



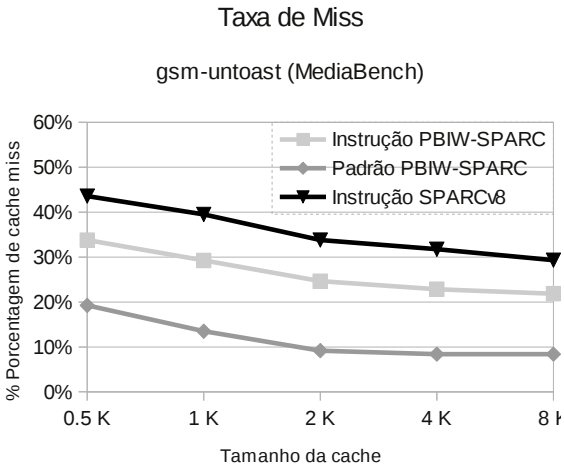

(A.37) rawdaudio




(A.38) timing




(A.39) toast




(A.40) untoast