Este exemplar corresponde à redação final da Tese/Dissertação devidamente corrigida e defendida por: Rodolfo Jardim de Azevedo
e aprovada pela Banca Examinadora.
Campinas, 24 de junho de 2002
COORDENADOR DE PÓS-GRADUAÇÃO
CPG-IC

Uma Arquitetura para Execução de
Código Comprimido em Sistemas Dedicados

Rodolfo Jardim de Azevedo

Tese de Doutorado

UNICAMP
BIBLIOTECA CENTRAL
SEÇÃO CIRCULANTE

Instituto de Computação
Universidade Estadual de Campinas

Uma Arquitetura para Execução de Código Comprimido em Sistemas Dedicados

Rodolfo Jardim de Azevedo[1]

18 de junho de 2002

Banca Examinadora:

- Guido Costa Souza de Araújo (Orientador)
- Claudionor José Nunes Coelho Júnior
  Departamento de Ciência da Computação - UFMG
- Edil Severiano Tavares Fernandes
  COPPE - UFRJ
- Edna Natividade da Silva Barros
  Centro de Informática - UFPE
- Mario Lúcio Côrtes
  Instituto de Computação - UNICAMP
- Paulo Cesar Centoducatte
  Instituto de Computação - UNICAMP
- Ricardo Pannain (suplente)
  Instituto de Computação - UNICAMP

[1]Financiado pela FAPESP, processo 99/09462-8

UNICAMP
BIBLIOTECA CENTRAL

UNIDADE BC
Nº CHAMADA T/UNICAMP
Az25a
V ___ EX ___
TOMBO BC/ 49955
PROC 16-837/02
C ___ D X
PREÇO R$ 11,00
DATA ___
Nº CPD ___

CM00170466-2

BIB ID 246997

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Azevedo, Rodolfo Jardim de

Az25a Uma Arquitetura para Execução de Código Comprimido em Sistemas Dedicados / Rodolfo Jardim de Azevedo -- Campinas, [S.P. :s.n.], 2002.

Orientador : Guido Costa Souza de Araújo

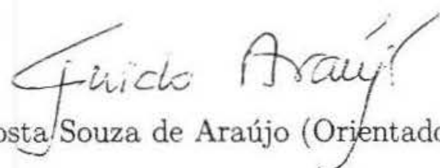
Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computadores. 2. Sistemas embutidos de computador. 3. Circuitos integrados. 4. Compressão de dados (Computação). I. Araújo, Guido Costa Souza de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Uma Arquitetura para Execução de Código Comprimido em Sistemas Dedicados

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Rodolfo Jardim de Azevedo e aprovada pela Banca Examinadora.

Campinas, 20 de junho de 2002.



Guido Costa Souza de Araújo (Orientador)

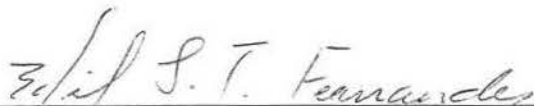
Tese apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

00231739

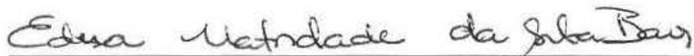
TERMO DE APROVAÇÃO

Tese defendida e aprovada em 18 de junho de 2002, pela Banca Examinadora composta pelos Professores Doutores:

Claudionor José Nunes Coelho Júnior
DCC - UFMG



Prof. Dr. Edil Severiano Tavares Fernandes
COPPE - UFRJ



Profª. Drª. Edna Natividade da Silva Barros
CIn - UFPE

Prof. Dr. Mário Lúcio Côrtes
IC - UNICAMP

Prof. Dr. Paulo César Centoducatte
IC - UNICAMP

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

© Rodolfo Jardim de Azevedo, 2002.
Todos os direitos reservados.

Resumo

Projetos de sistemas dedicados modernos têm exigido cada vez mais memória de programa para incluir novas funcionalidades como interface com o usuário, suporte a novos componentes. etc. O aumento no tamanho dos programas tem feito com que a área ocupada pela memória em um circuito integrado moderno seja um dos fatores determinantes no seu custo final bem como um dos maiores responsáveis pelo consumo de potência nestes dispositivos. A compressão de código de programa vem sendo considerada como uma estratégia importante na minimização deste problema. Esta tese trata da compressão de programas para execução em sistemas dedicados baseados em arquiteturas RISC. Um amplo estudo demonstra que a utilização do método proposto neste trabalho, *Instruction Based Compression* (IBC), resulta em boas razões de compressão e implementações eficientes de descompressores. Para a arquitetura MIPS foi obtida a melhor razão de compressão (tamanho final do programa comprimido e do descompressor em relação ao programa original) conhecida (53,6%) utilizando como *benchmark* programas do SPEC CINT'95. Uma arquitetura *pipelined* para o descompressor é proposta e um protótipo foi implementado para o processador Leon (SPARC V8). Esta é a primeira implementação em hardware de um descompressor para a arquitetura SPARC, tendo produzido uma razão de compressão de 61,8% para o mesmo *benchmark* e uma queda de apenas 5,89% no desempenho médio do sistema.

Abstract

The demand for program memory in embedded systems has grown considerably in recent years, as a result of the need to accommodate new system functionalities such as novel user interfaces, additional hardware devices, etc. The increase in program size has turned memory into the largest single factor in the total area and power dissipation of a modern *System-on-a-Chip* (SoC). Program code compression has been considered recently a central technique in reducing the cost of memory in such systems. This thesis studies the code compression problem for RISC architectures. A thorough experimental study shows that the *Instruction Based Compression* (IBC) technique proposed herein results in very good compression ratios and efficient decompressor engine implementations. For the MIPS architecture this approach results in the best compression ratio (size of the compressed program divided by the size of the original program) known in the literature (53.6%), when it is evaluated using the SPEC CINT'95 benchmark programs. A decompressor pipelined architecture was developed and prototyped for the Leon (SPARC V8) processor. This is the first implementation of a hardware decompressor on the SPARC architecture, having resulted in a 61.8% compression ratio for the same benchmark, at the expense of a fairly small performance overhead (5.89% on average).

Agradecimentos

Gostaria de agradecer ao meu orientador, Guido Araújo, pelo apoio, dedicação, direcionamento, críticas e sugestões durante o doutorado.

Ao professor Paulo Centoducatte pelas críticas e sugestões durante o desenvolvimento deste trabalho e pelas revisões desta tese.

Ao professor Pedro Rezende por ter confiado em mim e pela ajuda na matrícula como aluno especial no segundo semestre de 1998.

À minha família, por todo apoio que recebi e a Deus, pois sem ele nada seria possível.

Aos amigos de república, do Laboratório de Sistemas de Computação e do Instituto de Computação pela ótima convivência, apoio e pelos momentos de descontração.

Ao Instituto de Computação, pelo ambiente de pesquisa do qual participei e pela infra-estrutura que utilizei.

Ao CNPQ, pela bolsa que recebi em 1999 (processo 146200/1999-3). À FAPESP pela bolsa a partir de 2000 (processo 1999/09462-8) e pelos equipamentos fornecidos (processos 1997/10982-0 e 2000/15083-9). À FAEP também pelos equipamentos fornecidos (processos 1123/01 e 1124/01).

À minha família.

Sumário

Resumo vi

Abstract vii

Agradecimentos viii

1 Introdução 1
- 1.1 Compressão de Código e Compressão de Dados . . . 3
- 1.2 Custos e Benefícios da Compressão de Código . . . 6
- 1.3 Contribuição . . . 7
- 1.4 Organização . . . 9

2 Trabalhos Relacionados 11
- 2.1 Parâmetros de Comparação . . . 12
- 2.2 Trabalhos Relacionados . . . 14
  - 2.2.1 Utilizando Instruções Menores . . . 14
  - 2.2.2 Utilizando Alterações no Software . . . 19
  - 2.2.3 Utilizando Compressão do Programa . . . 23
  - 2.2.4 Outros Trabalhos . . . 30
- 2.3 Quadro Comparativo . . . 34

3 Análise de Técnicas de Compressão 39
- 3.1 Compressão Baseada em Árvores (TBC) . . . 41
  - 3.1.1 Análise do Método TBC . . . 41
  - 3.1.2 O Descompressor Baseado em TBC . . . 45
- 3.2 Compressão Baseada em Árvores Fatoradas (PBC) . . . 47
  - 3.2.1 Análise do Método PBC . . . 47
  - 3.2.2 O Descompressor Baseado em PBC . . . 53

4 **Compressão Baseada em Instruções (MIPS)** **57**
4.1 Análise do Método . . . 59
4.2 Tabela de Conversão de Endereços . . . 66
4.3 Tabela de Instruções . . . 72
4.4 Descompressor IBC *Pipelined* para MIPS R4000 . . . 75
4.5 Um Exemplo do Funcionamento do Descompressor . . . 82
4.6 Comparação dos Métodos IBC vs. TBC e PBC . . . 91
4.7 Comparação entre IBC e Outros Métodos . . . 92

5 **Compressão Baseada em Instruções (SPARC)** **95**
5.1 Ambiente de Prototipagem . . . 95
5.2 Análise do Método para SPARC . . . 99
5.3 Descompressor IBC *pipelined* para SPARC V8 . . . 108
5.4 Um Exemplo do Funcionamento do Descompressor . . . 114
5.5 Análise de Desempenho do Descompressor . . . 123
5.6 Comparação entre IBC e Outros Métodos . . . 124

6 **Conclusões e Trabalhos Futuros** **127**
6.1 Trabalhos Futuros . . . 128

Bibliografia **131**

Lista de Tabelas

2.1 Quadro comparativo do conjunto de instruções DLXe e D16 15
2.2 Resultados comparativos considerando D16=1.00 15
2.3 CodePack: Codificação dos 16 bits mais significativos 28
2.4 CodePack: Codificação dos 16 bits menos significativos 28
2.5 Quadro comparativo dos métodos de compressão de código. 35

3.1 Parâmetros utilizados e número de instruções geradas 41
3.2 Número de árvores distintas nos programas. Os números entre parênteses indicam a porcentagem em relação ao total. 42
3.3 Todas as combinações de tamanhos de *codewords* para o programa *li* utilizando 4 classes. 44
3.4 Partições que resultam nos melhores resultados para 4 classes e as razões de compressão obtidas. 45
3.5 Número de padrões de árvores e de operandos. Os números entre parênteses mostram a porcentagem em relação ao total de árvores de expressões. . . . 48
3.6 Razão de compressão composta quando os padrões de árvore e de operando são combinados. 53

4.1 Número de instruções únicas para a arquitetura MIPS. 59
4.2 Resultados obtidos para a arquitetura MIPS. 67
4.3 Exemplo de código a ser comprimido. 82
4.4 Quadro comparativo dos métodos de compressão de código 93

5.1 Programas específicos para implementação IBC/Leon. 97
5.2 Rotinas do programa Stanford. 98
5.3 Número de instruções únicas para SPARC. 100
5.4 Resultado da divisão em classes para SPARC. 104
5.5 Número de instruções que ocorrem uma ou duas vezes no programa. 105
5.6 Razão de compressão para tamanhos diferentes da IT (*benchmark* SPEC CINT'95). 106
5.7 Razão de compressão para tamanhos diferentes da IT (*benchmark* Leon). . 107

5.8 Comparação dos Resultados para as arquiteturas MIPS e SPARC. 108
5.9 Exemplo de código a ser comprimido. 114
5.10 Resultados do programa `stanford` 124
5.11 Quadro comparativo dos métodos de compressão de código 125

Lista de Figuras

1.1 Exemplo de compressão pelo método Lempel-Ziv 4
1.2 Seqüência de descompressão de dados e código 5

2.1 *Pipeline* do Thumb e do ARM 16
2.2 Instrução ARM `ADD Rd, #constante` representada nos dois formatos . . . 16
2.3 Diagrama de blocos de um processador MIPS com suporte ao conjunto de instruções MIPS16 17
2.4 Conversão de uma instrução MIPS16 para o formato MIPS-I (32 bits) . . . 18
2.5 Seqüencia de passos do compilador modificado por Fraser 20
2.6 CCRP: *Compressed Code RISC Processor* 24
2.7 Organização de uma linha da *Line Address Table* (LAT) 24
2.8 Processador para execução de programas comprimidos proposto por Lefurgy 26
2.9 Arquitetura do IBM CodePack 28
2.10 Interpretações para o *Instruction Pointer* dadas por Breternitz 31
2.11 Mapas de memória equivalentes 32

3.1 Exemplos de árvore de Expressão 40
3.2 Porcentagem das árvores do programa cobertas por árvores distintas. . . . 43
3.3 Codificação das árvores 43
3.4 Razão de compressão para diferentes quantidades de classes de árvores. . . 44
3.5 Árvores comprimidas na memória 45
3.6 Descompressor para o Método de Compressão Baseado em Árvores 46
3.7 Razão de compressão final para TBC. 47
3.8 (a) Árvore de Expressão; (b) Padrão de árvore; (c) Padrão de operandos. . 48
3.9 Porcentagem acumulada das árvores de expressão. 50
3.10 Padrões de árvores comprimidos na memória. 51
3.11 Razões de compressão com diferentes números de classes. 52
3.12 Descompressor para o método de Compressão Baseado em Árvores Fatoradas (PBC). 54
3.13 Razão de compressão final para o PBC. 56

4.1 Distribuição do tamanho médio das árvores de expressões (aproximação de Bezier). 58
4.2 Ciclo de projeto. 60
4.3 Porcentagem do programa coberta por instruções distintas. 61
4.4 Passos executados pelo compressor de código. 62
4.5 Razão de compressão para diferentes números de classes. 64
4.6 Versão simplificada do descompressor para IBC. 65
4.7 Razão de compressão final para IBC aplicado ao MIPS. 66
4.8 Tabela de Conversão de Endereços (ATT). 68
4.9 Exemplo do uso da ATT. 71
4.10 Formas de codificação das instruções 73
4.11 Possíveis divisões das classes na Tabela de Instruções 74
4.12 Descompressor *pipelined* para MIPS usando IBC (4 estágios). 76
4.13 *Codeword Register*. 79
4.14 Descompressor *pipelined* para MIPS usando IBC (3 estágios). 81
4.15 Trecho de código comprimido na memória. 82
4.16 Exemplo do IBC para MIPS: Passo 1. 83
4.17 Exemplo do IBC para MIPS: Passo 2. 84
4.18 Exemplo do IBC para MIPS: Passo 3. 85
4.19 Exemplo do IBC para MIPS: Passo 4. 86
4.20 Exemplo do IBC para MIPS: Passo 5. 87
4.21 Exemplo do IBC para MIPS: Passo 6. 88
4.22 Exemplo do IBC para MIPS: Passo 7. 89
4.23 Exemplo do IBC para MIPS: Passo 8. 90
4.24 Razão de compressão final para IBC/TBC/PBC. 91

5.1 Diagrama de blocos do conjunto Processador/Descompressor/Memória utilizado. 96
5.2 Diagrama de blocos da placa XESS XSV800. 99
5.3 Porcentagem do programa coberta por instruções distintas. 101
5.4 Razão de compressão para diferentes quantidades de classes. 103
5.5 Mapa de memória para o Leon incluindo a região de código comprimido. . 109
5.6 Descompressor para Leon. 110
5.7 Trecho de código comprimido na memória. 114
5.8 Exemplo do IBC para SPARC: Passo 1. 115
5.9 Exemplo do IBC para SPARC: Passo 2. 116
5.10 Exemplo do IBC para SPARC: Passo 3. 117
5.11 Exemplo do IBC para SPARC: Passo 4. 118

5.12 Exemplo do IBC para SPARC: Passo 5. 119
5.13 Exemplo do IBC para SPARC: Passo 6. 120
5.14 Exemplo do IBC para SPARC: Passo 7. 121
5.15 Exemplo do IBC para SPARC: Passo 8. 122

Capítulo 1

Introdução

O aumento do nível de integração dos componentes eletrônicos levou ao desenvolvimento de circuitos integrados que implementam em um único *core* a funcionalidade de um sistema complexo contendo processador, memória (RAM e ROM) e periféricos. A esse modelo de desenvolvimento é dado o nome de *System-on-a-Chip* (SoC). Juntamente com esse aumento da densidade do hardware, também vem ocorrendo um aumento da demanda por novas funcionalidades em software. Dentre as causas desta demanda, podem ser citados uma maior necessidade de suporte aos novos dispositivos de comunicação, interfaces com o usuário, a utilização de linguagens de mais alto nível juntamente com novas metodologias de desenvolvimento de software, a demanda por novos serviços, etc. Esse aumento dos requisitos ocasiona uma taxa de crescimento entre 50% e 100% ao ano no tamanho dos programas, impondo um aumento entre meio e um bit de endereçamento por ano enquanto a densidade das células de memória cresce a uma taxa inferior a 60% ao ano [35, cap.1]. O resultado disto é um aumento explosivo da memória nesses sistemas, tornando-a um componente central em um SoC moderno (a memória ocupa em torno de 60% da área total de um SoC, *circa 2001*).

O custo dessa memória passa a ser um dos fatores determinantes do custo do circuito

integrado completo, visto que este é uma função da área do circuito total, dado pela Equação 1.1 [35, cap.1].

$$custo\ do\ circuito = f(área^4) \tag{1.1}$$

Outro aspecto importante do projeto de um SoC é a disponibilidade de ferramentas de desenvolvimento para a arquitetura alvo. Ao se propor um SoC não basta apenas ter uma arquitetura que garanta um bom desempenho, o conjunto de ferramentas de desenvolvimento deve ser estável e bem mantido. Este é um dos motivos pelos quais a escolha do processador do SoC é muito importante e também pelo qual a troca do processador utilizado em um projeto é demorada e custosa.

Este trabalho propõe um método de compressão de código e a arquitetura de um descompressor em hardware visando diminuir o aumento do uso de memória. A redução do código conseguida por este método implica em uma área total menor do circuito, e conseqüentemente, em um custo final de produção menor. Não é necessária nenhuma mudança nas ferramentas de desenvolvimento existentes, a compressão é feita em uma fase isolada, anterior à transferência do programa para o SoC. Do ponto de vista do processador, a compressão também é transparente, visto que o módulo descompressor proposto está localizado entre a *cache* e a memória principal.

Nesta tese é feita uma análise detalhada da literatura desta área e é mostrado que os trabalhos relacionados não conseguem uma redução equivalente e, em alguns casos, não fornecem informações suficientes para avaliar o *overhead* do circuito descompressor, dificultando uma comparação precisa entre eles.

O método proposto neste trabalho é direcionado para arquiteturas RISC face à crescente utilização destes processadores em SoCs modernos. As características das arquiteturas RISC como desempenho, facilidade de decodificação das instruções pelo processador,

simplicidade do código e melhor capacidade de geração de código de qualidade por parte dos compiladores tornam-as mais adequadas ao uso em sistemas dedicados, desde que haja uma redução na quantidade de memória utilizada pelos programas. Por possuírem instruções de tamanho fixo, e com isso, uma grande quantidade de bits não utilizados, programas extremamente redundantes são gerados para estas arquiteturas, basicamente devido ao pequeno conjunto de instruções. As arquiteturas CISC possuem instruções mais compactas já por construção, de modo que a compressão de código nestas arquiteturas não produz resultados tão expressivos quanto para arquiteturas RISC. O método proposto neste trabalho foi utilizado para um processador MIPS II e posteriormente em uma implementação SPARC V8, para a qual foi implementado um protótipo funcional.

1.1 Compressão de Código e Compressão de Dados

Algoritmos de compressão de dados tradicionais não podem ser utilizados para compressão de programas e descompressão em tempo real, pois estes algoritmos precisam descomprimir o programa seqüencialmente, não sendo capazes de descomprimí-los seguindo o fluxo de execução. Os algoritmos para compressão de código surgiram para satisfazer essa necessidade.

Uma das características mais importantes dos algoritmos de compressão de dados, é a capacidade de adaptação ao próprio conteúdo que está sendo comprimido fazendo com que as repetições que ocorrem na entrada sejam compactadas de forma cada vez melhor. Assim, eles são capazes de quantificar a freqüência dos símbolos já vistos e codificá-los de forma mais compacta cada vez que eles aparecem novamente no conjunto de dados. Boa parte dos métodos de compressão de dados modernos são baseados nos métodos de Lempel-Ziv [55, 71, 72] e Huffman [36]. O método de Lempel-Ziv, utiliza novos símbolos que representam ponteiros para ocorrências anteriores já comprimidas fazendo

com que elas sejam representadas de forma mais compacta. A Figura 1.1 mostra um exemplo do método Lempel-Ziv aplicado à seqüência *abcdeabcabcde*. Os símbolos A e B são criados e colocados na tabela após a leitura do terceiro e quinto caracter de entrada respectivamente. Essa entrada é copiada para a saída para que o descompressor seja capaz de reconstruir a mesma tabela do compressor. Na próxima ocorrência da seqüência *abc*, ela é substituída pelo símbolo A e a seqüência *abcde* pelo símbolo B. O método de Huffman cria um novo conjunto de símbolos, no qual os símbolos que mais ocorrem são representados por uma menor quantidade de bits. Uma versão adaptativa do método de Huffman atualiza a tabela de codificação à medida que o conjunto de dados é codificado. Em ambos os casos, essa adaptação ao conjunto de dados obriga que a descompressão seja seqüencial (Figura 1.2(a)), pois durante a descompressão será necessário montar as mesmas estruturas de dados feitas durante a compressão de modo a decodificar cada um dos símbolos que existem na entrada. No caso da Figura 1.2(a), não seria possível saber o significado do símbolo B sem antes ter lido até a quinta letra da entrada.

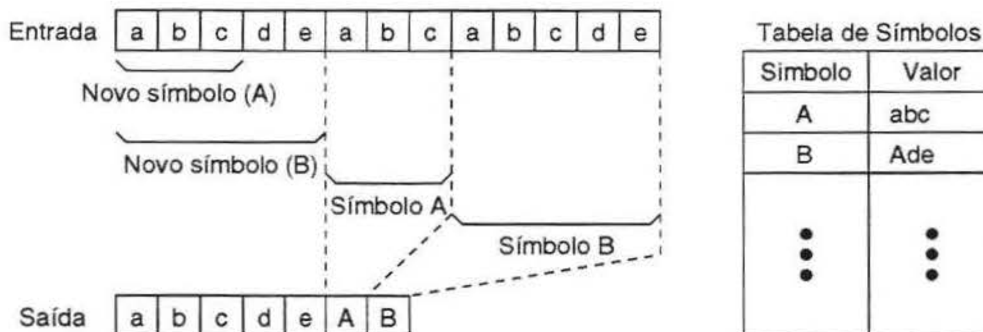
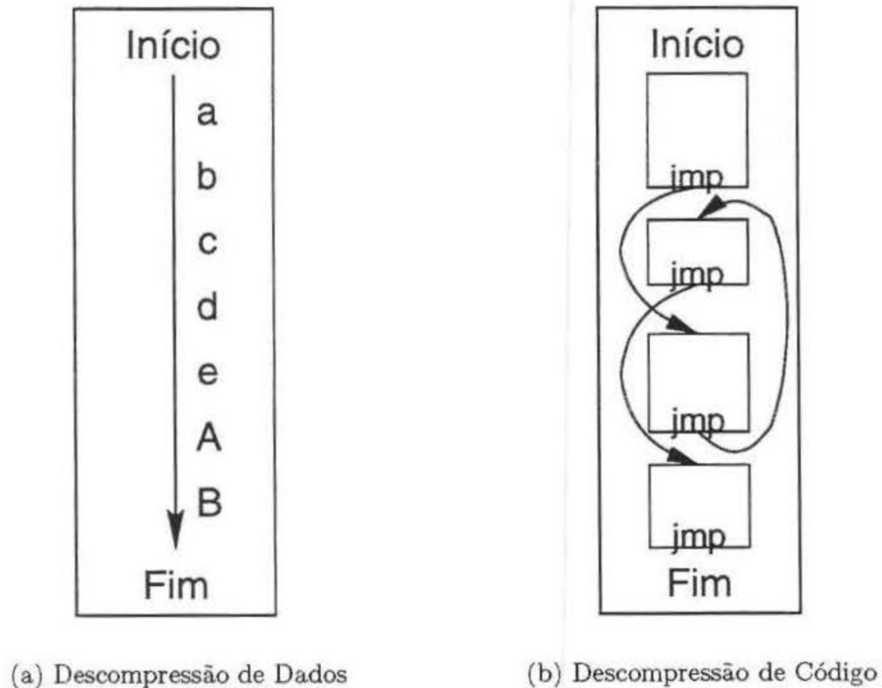


Figura 1.1: Exemplo de compressão pelo método Lempel-Ziv

Para a execução (em tempo real) de um código comprimido, é necessário que o método de descompressão seja capaz de fornecer qualquer instrução do programa sem precisar descomprimí-lo desde o início. Essa característica faz com que os algoritmos acima não possam ser utilizados uma vez que os programas possuem instruções de desvios que in-

terrompem o fluxo de descompressão, reiniciando-o em outro ponto do programa. A Figura 1.2(b) mostra 4 blocos básicos[1] terminados por instruções de salto. Observe que, devido ao fluxo de execução, o terceiro bloco básico deve ser descomprimido antes do segundo. Garantir acesso completamente aleatório ao programa, permitindo que todas as instruções sejam descomprimidas sem ter que descomprimir nenhuma outra instrução, pode gerar um *overhead* alto no desempenho. A alternativa utilizada neste trabalho é comprimir blocos de instruções, fazendo com que o descompressor seja capaz de iniciar seu trabalho a partir do início de qualquer um destes blocos de instruções. Esse custo extra em descomprimir instruções que podem não ser utilizadas será compensado por uma compressão final melhor e minimizado pelo uso da *cache*, uma vez que só é necessário utilizar o descompressor quando ocorrer um *cache-miss*.



(a) Descompressão de Dados (b) Descompressão de Código

Figura 1.2: Seqüência de descompressão de dados e código

[1]Um bloco básico [6] é uma seqüência de instruções na qual o fluxo de execução só pode iniciar pela primeira instrução do bloco e a única saída é através da última instrução.

Visando medir a qualidade da estratégia de compressão, será utilizada como métrica a razão de compressão, que é dada pela Equação 1.2.

$$razão\ de\ compressão = \frac{tamanho\ do\ programa\ comprimido}{tamanho\ do\ programa\ original} \quad (1.2)$$

Uma razão de compressão menor, indica que o programa foi melhor comprimido, e que será necessária uma quantidade menor de memória para armazena-lo.

1.2 Custos e Benefícios da Compressão de Código

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

Tamanho do Programa: O principal benefício da compressão é reduzir o tamanho da memória utilizada pelo programa. Em alguns casos, a diminuição do tamanho do programa pode não ser suficiente para promover a remoção de módulos de memória do sistema final, o que anula ou reduz o ganho da compressão. Por exemplo, se para um programa de 500KB é necessário utilizar um módulo de 512KB de memória, então uma razão de compressão de 60% não será suficiente para trocar esse módulo de memória por um de 256KB pois serão necessários 307KB para o programa comprimido. Nesse trabalho, não estamos considerando o ganho de compressão como o número de componentes discretos de memória que podem ser removidos do sistema, mas sim como a quantidade de área final do SoC que deixa de ser necessária por causa da compressão do código, uma vez que a memória é projetada e implementada

junto com todo o SoC.

Desempenho: Neste caso, o sistema de descompressão pode produzir tanto um ganho como uma perda de desempenho, dependendo do modelo a ser utilizado. Perdas podem vir da execução de um número maior de instruções (menores) necessárias para reduzir o tamanho do código [10, 40, 57] quanto do tempo gasto para descomprimir as instruções [28, 32]. Já os ganhos podem ser encontrados quando o tempo de acesso à memória é muito grande e, devido à menor quantidade de leituras feitas da memória, o sistema como um todo executa mais rapidamente. Vale ressaltar que a existência de *caches* no sistema auxilia neste ganho por não exigir que as instruções sejam descomprimidas quando ocorre um *cache hit*.

Consumo: A menor quantidade de memória utilizada e também uma redução no número de transações no barramento de memória podem ou não ser suficiente para compensar o consumo do módulo descompressor quando ele é implementado em hardware. Também vale ressaltar que o número de transições de barramento durante a leitura de um código comprimido tende a ser maior que na leitura dos programas descomprimidos [51].

Segurança: Embora seja um aspecto pouco considerado, a compressão de programas torna o código executável ininteligível e a simples leitura da memória ROM do sistema não é suficiente para decodificar (*disassembly*) o programa armazenado nela.

1.3 Contribuição

Neste trabalho é proposto o método *Instruction Based Compression* (IBC), que foi desenvolvido como uma evolução de dois métodos anteriores (*Pattern Based Compression* e *Tree Based Compression*) juntamente com uma arquitetura *pipelined* para o descompres-

sor, que foi projetada e implementada para um processador SPARC V8.

O método *Pattern Based Compression* [9] (PBC) trata como símbolos para compressão os padrões de árvores de expressão e os padrões de árvores de operandos. O método *Tree Based Compression* [17] surgiu da aglutinação dos padrões de operandos e expressão de árvores.

O método proposto, *Instruction Based Compression* [7, 8] (IBC) mostrou que as árvores de expressão no código executável final são, em média, muito pequenas, sendo mais vantajosa a compressão das instruções individualmente, quando codificadas com o algoritmo aqui proposto. O resultado é um descompressor mais compacto e eficiente, que proporciona uma melhor razão de compressão final e um maior desempenho durante a descompressão.

No desenvolvimento da arquitetura de descompressão, foram tratados os problemas de resolução de endereços para o mapeamento dos endereços reais (vistos pelo processador) em endereços comprimidos (endereços das instruções armazenadas na memória) e o limite no tamanho das tabelas de descompressão. A arquitetura desenvolvida foi implementada em FPGA através de um *kit* de prototipagem [69].

Em resumo, as contribuições desta Tese são:

- Um algoritmo de compressão de código que resulta na melhor razão de compressão final conhecida (considerando o custo do descompressor) para a arquitetura MIPS (*circa maio 2002*);

- O primeiro descompressor implementado em hardware para a arquitetura SPARC;

- Uma arquitetura *pipelined* para o descompressor proposto que, pelos experimentos realizados, ocasionou uma queda de apenas 5,89% no desempenho;

1.4 Organização

Esta Tese está organizada da seguinte forma: No Capítulo 2, os trabalhos relacionados são mostrados, indicando as vantagens e desvantagens de cada um deles. No Capítulo 3 são analisados os métodos *Pattern Based Compression* e *Tree Based Compression*. No Capítulo 4 é apresentado o método *Instruction Based Compression* proposto nesse trabalho, com o desenvolvimento teórico deste e uma descrição do Módulo Descompressor para a arquitetura MIPS. Uma implementação para a arquitetura SPARC V8 é mostrada no Capítulo 5, juntamente com os resultados obtidos e um Módulo Descompressor *pipelined* IBC. Finalmente as conclusões e possíveis extensões deste trabalho estão descritas no Capítulo 6.

Capítulo 2

Trabalhos Relacionados

A quantidade de memória gasta por um programa sempre foi motivo de preocupação. No início do desenvolvimento da Computação, os conjuntos de instruções eram grandes e possuíam uma grande expressividade [1], visando com isto, tanto simplificar o código *assembly* quanto reduzir a demanda pela limitada quantidade de memória. Com o passar do tempo, a quantidade de memória disponível cresceu, juntamente com a habilidade dos programadores em ocupar essa memória com novos recursos como interface com usuário, tecnologias multimídia, novos paradigmas de programação, etc. O surgimento dos processadores RISC aumentou essa necessidade de memória pois uma das características destas arquiteturas é um conjunto de instruções simples, onde todas as instruções ocupam exatamente o mesmo espaço de memória (por exemplo, um `NOP`[1] ocupa o mesmo espaço que uma instrução de `LOAD` que utilize imediatos). Recursos como Memória Virtual [35, cap.5] permitem que um programa seja executado mesmo quando não há memória física suficiente para ele. No entanto, isso requer a existência de um dispositivo de memória secundária, normalmente inexistente em sistemas dedicados.

Com o aumento da integração e o projeto de transistores cada vez menores, a área ocupada pelos processadores e pela memória têm diminuído, mas a demanda por memória

[1] Do inglês *No Operation*, instrução que não executa tarefa alguma.

continua aumentando, fazendo com que a proporção entre a área ocupada pelo processador e a memória seja cada vez mais dominada por esta. Sendo assim, reduzindo a quantidade de memória utilizada, estamos reduzindo um grande componente do sistema dedicado e com isso, o seu custo final. Uma abordagem que vem sendo pesquisada para proporcionar esta redução de memória sem inibir o crescimento das funcionalidades nos sistemas dedicados é a compressão de código.

Nesse capítulo estão descritos alguns dos métodos de compressão de código já encontrados na literatura. Na Seção 2.1 são descritos os parâmetros de comparação entre os diversos métodos, seguidos por uma descrição resumida dos trabalhos na Seção 2.2. Finalmente um quadro comparativo é mostrado na Seção 2.3.

2.1 Parâmetros de Comparação

Para comparar os diferentes métodos de compressão de código, os seguintes parâmetros serão usados:

Razão de Compressão: Esse é o indicador de quanto o programa é reduzido. Quanto menor a razão de compressão, menor a área de memória ocupada pelo programa final e melhor o resultado do método. Ela é calculada através da Equação 1.2

Alteração no Processador: Alguns métodos podem ser implementados sem que o processador alvo precise ser modificado, quer seja descomprimindo o código de forma totalmente transparente ao processador ou exigindo apenas algumas rotinas implementadas em software para tratar a descompressão. Outros métodos exigem algum tipo de modificação que pode ser uma reformulação do conjunto de instruções para uma forma mais compacta, mantendo ou não o conjunto de instruções anterior. Neste caso é comum criar algumas poucas instruções novas para agilizar a execução

do código comprimido, bem como alterar a forma de cálculo do endereço das instruções para resolver de forma mais direta o problema de endereçamento que surge com a utilização de instruções comprimidas.

Desempenho: Embora a redução do tamanho do código executável seja o principal objetivo, não se pode deixar de lado o desempenho resultante após a implementação do método. Alguns desses métodos utilizam uma quantidade maior de instruções comprimidas para conseguir uma diminuição no tamanho final do código. Essa maior quantidade de instruções influencia na quantidade de ciclos que o processador precisa para executar o programa. A não ser que haja ganho suficiente em outras partes do sistema, como um aumento no *hit ratio* da *cache* ou redução no número de acessos à memória, haverá uma perda de desempenho com a compressão.

Módulo Descompressor: Alguns dos métodos são implementados puramente em software através de reagrupamento ou remoção de instruções, sem a necessidade de modificação do hardware do processador. Outros precisam que um módulo descompressor seja incluído junto ao processador. Para efeitos de comparação, o hardware necessário para implementar quaisquer novas instruções ou funcionalidades será considerado como módulo descompressor. O cálculo da área ocupada por esse módulo descompressor deve fazer parte do cálculo da razão final de compressão no caso de sistemas dedicados, pois também implicam no aumento de área do circuito final. Isso faz com que a fórmula para o cálculo da razão de compressão agora seja dada pela Equação 2.1, onde os tamanhos são expressos em bits.

$$razão\ de\ compressão = \frac{tamanho\ comprimido + tamanho\ do\ descompressor}{tamanho\ original} \tag{2.1}$$

2.2 Trabalhos Relacionados

Os trabalhos apresentados nesta seção foram divididos de acordo com a abordagem adotada. Primeiramente serão tratadas as propostas que utilizam a abordagem de redução do tamanho das instruções (Seção 2.2.1), seguidas por métodos que implementam a compressão através de alterações no software (Seção 2.2.2). Posteriormente serão descritos os métodos que utilizam algum algoritmo de compressão sem alterar o conjunto de instruções (Seção 2.2.3) e ao final serão mostrados outros trabalhos que procuram resolver problemas que surgem com a compressão de código (Seção 2.2.4).

2.2.1 Utilizando Instruções Menores

Um dos primeiros estudos sobre compressão de código data dos anos 70, quando a memória era escassa e os conjuntos de instruções foram projetados para minimiza sua utilização. Em 1972, os projetistas do Borroughs B1700 [67] desenvolveram um conjunto de instruções que atribuía campos menores a instruções mais freqüentes e campos maiores a instruções que ocorriam pouco no programa. O conjunto de instruções do VAX [1] também foi projetado para minimizar a utilização de memória.

Um dos primeiros estudos na década de 90 que relacionam o impacto de um conjunto de instruções de tamanho fixo com o desempenho e tamanho do código executável [16] faz uma crítica ao modelo de 32 bits amplamente utilizado por processadores RISC. Reconhecendo a grande vantagem na decodificação das instruções das arquiteturas RISC sobre as CISC (advindas do tamanho fixo das instruções), é proposto um conjunto de instruções de 16 bits chamado D16 no lugar do tradicional conjunto de 32 bits para a arquitetura DLXe. A arquitetura DLXe é uma variação do conjunto de instruções DLX de Hennessy e Patterson [35] que não permite operações de *load* e *store* nos registradores da unidade de ponto flutuante.

As características dos dois conjuntos de instruções estão descritas na Tabela 2.1. Para realizar as medições comparativas, foram configurados compiladores para DLXe com cada uma das restrições do D16 separadamente e posteriormente foram implementadas todas as restrições do D16 de uma só vez.

DLXe	D16
Registradores de 32 bits	Registradores de 16 bits
Instruções com 3 operandos	Instruções com 2 operandos
Imediatos maiores	Imediatos menores
Mais instruções com imediatos	Menos instruções com imediatos

Tabela 2.1: Quadro comparativo do conjunto de instruções DLXe e D16

O uso de um conjunto de registradores menor fez com que o tráfego entre processador e memória aumentasse em 10% e que o número de instruções executadas ficasse ligeiramente maior. A redução nos campos de imediato gerou uma queda de desempenho de 9,5% pela necessidade de mais instruções para codificar os imediatos maiores. O uso de dois registradores por instrução no lugar de três causa um pequeno impacto negativo tanto no tamanho do programa quanto na quantidade de instruções executadas. Na configuração com 16 registradores e dois operandos por instrução, o código de 32 bits do DLXe ficou 1,62 vezes maior que o do D16 e foi executado em 95% do tempo (5% mais rápido que o D16). Os outros resultados finais obtidos estão na Tabela 2.2.

Registradores	Tamanho do Programa		Instruções Executadas	
DLXe	2 Operandos	3 Operandos	2 Operandos	3 Operandos
16	1,62	1,61	0,95	0,94
32	1,57	1,52	0,90	0,87

Tabela 2.2: Resultados comparativos considerando D16=1.00

Em 1995, a ARM [59] lançou o Thumb [10] como a versão do conjunto de instruções de 16 bits do seu processador ARM7. Essa versão é capaz de executar tanto instruções de 16 bits quanto instruções de 32 bits e a distinção é feita através de um bit de estado do processador que pode ser trocado através da instrução de salto `BX`. Os *pipelines* do Thumb e do ARM7 são mostrados na Figura 2.1. Deve ser ressaltado que esta figura é apenas ilustrativa pois não se tratam de dois *pipelines* distintos e sim de um único e integrado com uma distinção apenas no estágio de decodificação de instruções para converter as novas instruções do Thumb em instruções do ARM (veja exemplo da Figura 2.2).

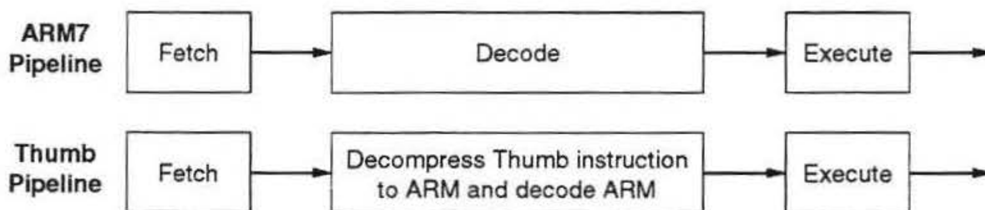


Figura 2.1: *Pipeline* do Thumb e do ARM

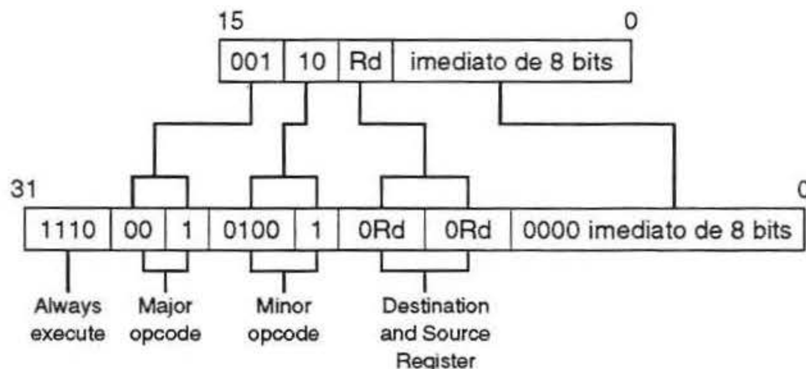


Figura 2.2: Instrução ARM `ADD Rd, #constante` representada nos dois formatos

A visibilidade do conjunto de registradores foi afetada. No entanto, é possível utilizar algumas instruções do Thumb para acessar os registradores que ficam ocultos para as demais instruções. Os valores dos registradores não são alterados durante a transferência de modo entre 16 e 32 bits.

Os códigos convertido para Thumb apresentam uma razão de compressão entre 55% e 70% e executam em um tempo de 10% a 20% maior se colocados em um barramento de 32 bits. O desempenho do Thumb pode superar o do ARM em até 30% caso o barramento utilizado seja de 16 bits pois nesse caso, são necessárias 2 leituras da memória para formar instruções de 32 bits enquanto as instruções de 16 bits são lidas de uma só vez.

A MIPS também desenvolveu sua versão do conjunto de instruções de 16 bits, o MIPS16 [40] que, de forma similar ao ARM, faz um mapeamento entre as instruções de 16 bits nas instruções de 32 bits, utilizando para execução o mesmo *core* de 32 bits já disponível (Figura 2.3). A troca de um conjunto de instruções para o outro é feita através da instrução `JALX` (*Jump And Link with eXcahge*). O estado do processador é mantido entre chamadas de procedimentos com troca do conjunto de instruções e também no processamento de interrupções.

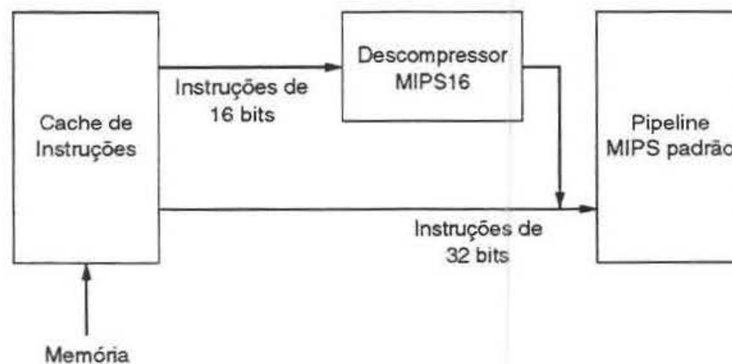


Figura 2.3: Diagrama de blocos de um processador MIPS com suporte ao conjunto de instruções MIPS16

As instruções são mapeadas campo a campo, com algumas restrições de tamanho. A Figura 2.4 mostra a conversão de uma instrução com o campo de imediato, que é o que mais sofreu restrições. O conjunto de registradores é restrito a apenas 8, mas é possível utilizar as instruções `MOV32R` e `MOVR32` para acessar os outros registradores que não estão visíveis para as demais instruções no modo 16 bits. O conjunto de instruções

do MIPS16 exclui as instruções de co-processador e de ponto flutuante, fazendo com que elas só possam ser utilizada por rotinas com instruções de 32 bits.

Figura 2.4: Conversão de uma instrução MIPS16 para o formato MIPS-I (32 bits)

Para superar as restrições por falta de bits nas instruções, os seguintes mecanismos foram criados:

Instrução EXTEND: É uma instrução que não executa nenhuma operação, a não ser carregar 11 bits de imediato que será utilizado pela próxima instrução. Dessa forma, um imediato de 16 bits pode ser criado com duas instruções MIPS16.

Endereçamento relativo ao PC: Para simplificar a carga de constantes, é possível no MIPS16 especificar um deslocamento em relação ao PC para o acesso à memória, permitindo que o compilador inclua no segmento de código, constantes que serão carregadas por essas instruções.

Endereçamento relativo à pilha: Enquanto na arquitetura MIPS convencional não há registrador específico para a pilha, o registrador $29 pode ser referenciado implicitamente através de alguns opcodes.

Deslocamentos em *Loads/Stores*: De acordo com o tamanho do dado a ser buscado da memória, será automaticamente efetuado um deslocamento dos imediatos. Assim, se o processador efetuar um acesso a 32 bits da memória, o campo de imediato

será deslocado 2 bits para a esquerda. No caso de acessos de 16 bits, ele será deslocado 1 bit apenas.

O artigo sobre o MIPS16 [40] cita uma razão de compressão média 60% mas não informam o impacto no desempenho.

2.2.2 Utilizando Alterações no Software

Os trabalhos descritos nessa sub-seção implementam a compressão de código totalmente em software, quer seja através de alterações no compilador de forma a gerar menos código, ou através de módulos de software responsáveis pela descompressão em tempo de execução, ou ainda através da interpretação do código comprimido.

Fraser e Proebsting [32] implementaram alterações no compilador *lcc* [31] para que, ao invés deste gerar código executável, fossem gerados simultaneamente uma representação intermediária compacta e um interpretador para esta, conforme mostrado no diagrama da Figura 2.5. A representação intermediária é baseada na representação por árvores do compilador com alguns operandos fatorados. Após a leitura do programa fonte, todas as suas árvores são descritas em ASCII e depois transformadas em padrões que são enviados ao gerador de código. Este então aloca o conjunto de instruções de acordo com as ocorrências de cada padrão de árvore. Na seqüência, um interpretador é gerado para essas novas instruções.

Os resultados mostram uma razão de compressão de 50% no tamanho do programa para arquitetura SPARC incluindo uma estimativa para o tamanho do código do interpretador. O custo dessa compressão está no seu tempo de execução, 20 vezes mais lento que o do programa original.

Em uma continuação desse trabalho por Ernst [26], foi proposto o formato *wire code* que é direcionado a sistemas onde o gargalo está na transferência dos dados e não na execução. No *wire code*, o código intermediário do *lcc* é transformado em padrões de

UNICAMP
BIBLIOTECA CENTRAL
SEÇÃO CIRCULANTE

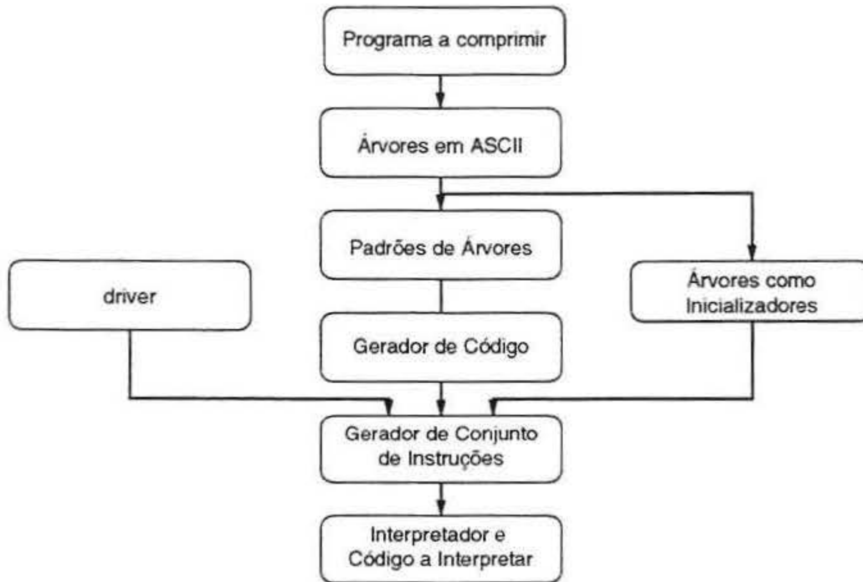


Figura 2.5: Seqüencia de passos do compilador modificado por Fraser

árvores e de expressões e separados em *streams* diferentes que são codificados usando a técnica *move-to-front* [14] (MTF). Nesta técnica, os símbolos são substituídos por índices em uma tabela que é alterada dinamicamente de forma que o último símbolo utilizado seja sempre movido para o início da tabela. Após a codificação MTF, os índices são comprimidos usando Huffman e o conjunto completo (*streams* comprimidos e tabelas) é comprimido utilizando *gzip*. O resultado é um código extremamente compacto (mais compacto que com a utilização de *gzip* no código original). No entanto, o código deve ser completamente descomprimido antes de ser executado. No mesmo artigo é também definido um padrão para código interpretável por uma máquina virtual chamada BRISC. O código para BRISC é gerado através de duas operações:

Especialização de operandos: Essa operação transforma uma instrução válida na representação intermediária em uma representação que já possui pré-codificado um ou mais de seus operandos. O objetivo é tirar proveito de operações como carga de elementos da pilha que sempre referenciam um mesmo registrador base.

Combinação de *opcodes*: Essa operação transforma dois *opcodes* em um único na representação intermediária. Ele é usado para compactar operações que normalmente ocorrem em seqüência.

Os dois passos são executados em um laço que se encerra após o esgotamento das tabelas da representação intermediária, ou caso uma iteração do laço não gere resultados melhores que a iteração anterior. As instruções são então codificadas utilizando um modelo de Markov de primeira ordem em conjuntos de 8 ou 16 bits. Como resultado final, a razão de compressão média fica em 59%, novamente ao custo de um tempo de execução muito pior que o programa original (12,6 vezes mais lento). Outros dois trabalhos deste grupo [27, 30] seguem linhas similares, e portanto direcionam-se mais para o caso em que a transferência dos programas é o gargalo do sistema. Franz [28, 29] propôs uma representação intermediária comprimida similar à de Fraser, com a implementação do sistema descompressor e gerador de código diretamente no sistema operacional, ou em navegadores web para o caso de código móvel.

Em seu trabalho sobre compressão, Liao et al. [56, 57] desenvolveu dois métodos baseados em dicionários, com uma das versões totalmente implementada em software. Nesta versão, seqüências de instruções repetitivas são transformadas em sub-rotinas e substituídas no programa por instruções de chamada ao dicionário. Essa implementação exige apenas que as rotinas sejam tiradas de blocos básicos estendidos[2]. O custo desse método é ter que incluir duas novas instruções de salto (`Call` e `Ret`) para cada chamada ao dicionário. Em um segundo método, Liao et al. supõe a existência de uma instrução específica para chamadas ao dicionário, essa instrução, `CallD` recebe como parâmetro o endereço da primeira instrução do dicionário e o número de instruções a executar. A instrução de retorno é então executada implicitamente após o número definido de ins-

[2] Um bloco básico estendido é um conjunto de instruções com apenas um sucessor em comum fora do bloco.

truções. Embora esta última estratégia reduza o custo extra do método, ela impõe que as sub-rotinas do dicionário executem um número fixo pré-determinado de instruções, o que dificulta a inclusão de instruções de salto dentro delas[3]. Como resultado final, o primeiro método gera uma razão de compressão de 88,2% e o segundo método uma razão de 84,1%, ambos para o processador TMS320C25. O desempenho do programa sofreu uma degradação de 15% a 17% com o método. Uma nova implementação com ênfase no desempenho fez com que não fossem comprimidas as partes que mais executam do programa. Dessa forma, houve uma perda de compressão em torno de 2% a 3% e o desempenho caiu apenas 1% a 2% com relação ao programa original. Em um outro trabalho [58], Liao mostrou que a simples reestruturação dos dados da memória e a utilização de instruções de auto-incremento e auto-decremento podem fazer com que o código seja reduzido para 80% a 97% do tamanho original também para o TMS320C25.

Kirovski [39] propôs um método cuja unidade de compressão é um procedimento completo. Esse método armazena o conjunto de procedimentos comprimidos em um dicionário e os descomprime para uma área na memória RAM (chamada *pcache*) à medida em que são requisitados pelo programa. As instruções de chamada a procedimentos do programa são reescritas para fornecer o identificador do procedimento no dicionário. A execução de um procedimento segue os passos do Algoritmo 1.

O retorno de procedimento também é uma tarefa complicada, pois o procedimento a receber de volta o fluxo de controle pode não estar mais na *pcache*. Para tratar esse problema, o endereço de retorno é armazenado como uma tripla contendo o identificador do procedimento, o endereço de retorno no momento da chamada e o deslocamento do endereço de retorno em relação ao início do procedimento. O retorno então é executado verificando se o procedimento está na memória e restaurando-o caso seja necessário através

[3] A inclusão de uma instrução de salto dentro de uma mini sub-rotina criaria mais de um caminho de execução que não necessariamente teriam o mesmo tamanho.

Algoritmo 1 Passos para execução de um procedimento

```
Uma chamada ao procedimento é executada com o identificador do dicionário
Se o procedimento não estiver na pcache
    Encontrar o procedimento comprimido no dicionário
    Se não existir espaço desfragmentado suficiente na pcache
        Enquanto não existir espaço desfragmentado suficiente na pcache
           Marca procedimentos para remoção na pcache
        Compacta o espaço desfragmentado em um bloco contínuo
    Descomprime o procedimento destino na área livre da pcache
Ajusta o endereço de chamada do procedimento
Executa o procedimento
```

dos dados armazenados na pilha. O autor fez uma estimativa de que o desempenho cairia em torno de 11% com uma estimativa de razão de compressão de 60% para a arquitetura SPARC.

2.2.3 Utilizando Compressão do Programa

Os trabalhos dessa sub-seção implementam algoritmos diversos de compressão e incluem um módulo descompressor que é responsável por tornar transparente ao processador todo o esquema de compressão.

Em 1992, Wolfe propôs o *Compressed Code RISC Processor* [68] (CCRP) que utiliza código de Huffman e o código *limitado* de Huffman na compressão dos programas. Estes programas são descomprimidos ao serem lidos da memória por um hardware dedicado chamado *Cache Refill Engine*. A conversão dos endereços descomprimidos (solicitados pelo processador) nos endereços comprimidos é realizada na *Line Address Table* (LAT), e no *Cache Line Address Lookaside Bufer* (CLB). Um diagrama resumido desse sistema é mostrado na Figura 2.6. A implementação foi feita considerando-se processadores MIPS R2000 com cada símbolo do código Huffman representando um byte.

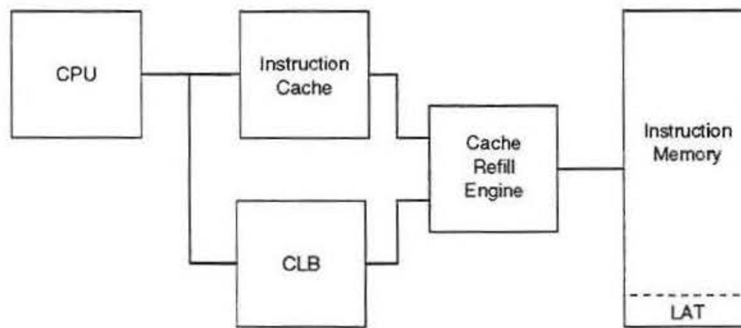


Figura 2.6: CCRP: *Compressed Code RISC Processor*

A LAT é organizada de forma a receber o endereço da *cache-line* e fornecer o endereço de memória onde está localizada a primeira instrução comprimida. Cada linha da LAT (Figura 2.7) armazena endereços para decodificar blocos de 256 bytes através de um endereço base de 24 bits que indica a posição de memória do primeiro bloco e 8 conjuntos de 5 bits que indicam o tamanho em bytes dos blocos armazenados. Desta forma, o endereço de um bloco é obtido somando-se o endereço base com os tamanhos dos blocos anteriores a ele dentro do conjunto de blocos endereçados. O custo da LAT armazenada dessa forma é de 3% do tamanho do programa original (8 bytes em cada entrada da LAT para cada 256 bytes do programa). Para diminuir a quantidade de leituras à LAT em memória, foi definido o CLB, que é um buffer que armazena as últimas entradas da LAT consultadas, sendo atualizado juntamente com as leituras da memória quando não contiver o endereço solicitado.

Endereço Base	T0	T1	T2	T3	T4	T5	T6	T7
24 bits	5 bits	5 bits	5 bits	5 bits	5 bits	5 bits	5 bits	5 bits

Figura 2.7: Organização de uma linha da *Line Address Table* (LAT)

Diversos experimentos foram realizados medindo o desempenho da arquitetura resultante, através da simulação da execução dos programas. A razão de compressão obtida pelo modelo ficou pouco acima de 70% para um conjunto de 10 programas. Não foi re-

alizada uma implementação em hardware e a conseqüente medição dos parâmetros de temporização. Foram feitas apenas estimativas e, nos diversos experimentos, os resultados oscilaram entre pequenos ganhos e pequenas perdas de desempenho. O tráfego entre processador e memória foi reduzido entre 5% e 35%. Embora grande parte do módulo descompressor possa ser implementado externamente ao processador, através dos comentários dos autores supõe-se que ao menos a CLB tenha que ser implementada internamente para que seu acesso possa ser feito juntamente com a TLB.

Em um estudo posterior do mesmo grupo, Kozuch [41] calculou as entropias dos programas utilizando os mesmos símbolos de 8 bits, e mostrou que os resultados já obtidos eram muito próximos do máximo possível. A alternativa para aumentar a compressão utilizando o mesmo método seria aumentar o tamanho do símbolo a ser comprimido. No entanto, símbolos de 16 e 32 bits podem fazer com que o decodificador Huffman fique muito grande. Uma informação muito relevante levantada por esse artigo é o tamanho total dos programas utilizados como *benchmark* para cada uma das arquiteturas avaliadas, mostrando que o código para VAX é o mais compacto do conjunto, seguido por MIPS (~2,6x maior), 68020 (~2,8x), SPARC (~3,2x), RS6000 (~4,3x) e MPC603 (~4,8x). Esses valores por si só não fornecem muitas informações, pois são dependentes também dos compiladores utilizados, mas ao menos permitem ter-se uma noção do tamanho de um mesmo conjunto de programas em diversas arquiteturas, e da capacidade de expressão de cada um dos conjuntos de instruções desses processadores. Posteriormente, um ambiente de simulação foi desenvolvido [42] para um estudo mais detalhado desse método. Em um trabalho posterior, Beneš projetou um circuito decodificador Huffman específico para o CCRP [12, 13] que é capaz de decodificar 32 bits em 25ns.

O método inicial proposto por Lefurgy [44, 45, 46] baseia-se em um dicionário similar àquele usado no método de Liao, mas ao invés de tratar as seqüências de código repetitivas como mini sub-rotinas, ele atribui *codewords* a cada uma delas e mistura código compri-

mido com código não comprimido na memória. Ao encontrar uma *codeword*, a Lógica de Decodificação de *codeword* gera o índice para o dicionário, juntamente com a quantidade de instruções que ela representa e o dicionário fornece ao processador as instruções referentes à *codeword*. Um diagrama do modelo proposto é mostrado na Figura 2.8. O código comprimido é alinhado em *nibles* (4 bits), e a unidade de endereçamento, juntamente com as instruções de desvio foram modificadas para aceitar o novo modelo da memória. Mesmo com estas alterações, as instruções de desvio relativo não são comprimidas pois precisam ser corrigidas, após a compressão, de modo a apontarem para o destino correto.

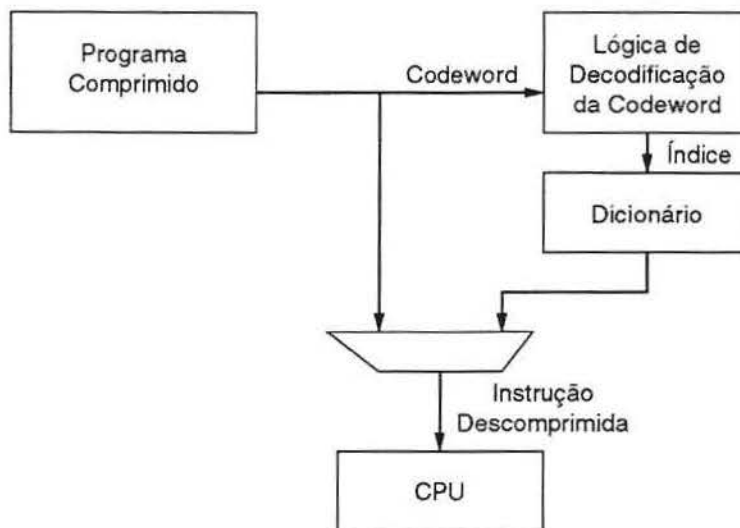


Figura 2.8: Processador para execução de programas comprimidos proposto por Lefurgy

Lefurgy et al. realizaram dois experimentos, um com *codewords* de tamanho fixo e outro com *codewords* de tamanho múltiplos de 4 bits para arquiteturas PorwerPC, ARM e i386. As razões de compressão médias foram de 61%, 66% e 74% respectivamente. Não foram fornecidos dados de desempenho.

Uma simplificação desse método foi proposta posteriormente [47] para o DSP SHARC. Nesta simplificação, foram incluídas no dicionário todas as instruções do programa e as *codewords* foram representadas por 16 bits. Também foi realizado um experimento fa-

zendo a codificação de algumas instruções SHARC em 15 bits, mesclando essas instruções comprimidas com as *codewords*. A razão de compressão obtida variou entre 41,6% a 64%, mas os programas foram compilados ou com otimização `-O1`[4] ou sem otimização, o que dificulta a comparação desses números uma vez que a otimização `-O2` reduz consideravelmente o tamanho do programa.

Outra proposta de Lefurgy é a descompressão através de software [48]. Novamente é utilizado um dicionário, mas a descompressão é realizada por rotinas em software chamadas pelo processador quando ocorre um *cache-miss*[5]. Como resultado, a razão de compressão varia de 65,4% até 82,5% com o desempenho sendo degradado em 51% para *caches* grandes. Os resultados foram obtidos através de simulações.

A IBM lançou em 1998 o PowerPC 405 CodePack [25, 34, 37] (Figura 2.9) e seguiu uma abordagem diferente da MIPS e ARM utilizando o método de dicionário ao invés de criar um conjunto compacto de instruções. Dois dicionários são utilizados, cada um correspondendo a 16 bits da instrução que são codificados de forma separada. Cada parte de 16 bits é codificada por um *tag* e um índice. Os *tags* podem ocupar 2 ou 3 bits e os índices, de 0 a 16 bits, a menor codificação possível ocupa 7 bits e a maior 38 bits (Tabelas 2.3 e 2.4). As instruções são armazenadas na forma de 4 campos, primeiro os dois *tags* e depois os dois índices.

O espaço de endereçamento de 32 bits é dividido em 64 regiões de 64MB, sendo essas regiões divididas em grupos de 128 bytes que por sua vez são divididos em 2 blocos de 64 bytes. Os blocos são alinhados em limites de bytes. Para localizar um bloco na memória, é usada a *Compression Index Table*, que faz o mapeamento entre endereços reais e endereços comprimidos. Para isso, ela possui o endereço em bytes do primeiro bloco de cada grupo e um deslocamento também em bytes do segundo bloco dentro do grupo. Como resultado,

[4] O autor citou que otimizações acima de `-O1` *quebravam* o compilador e por isso restringiu as otimizações.

[5] O processador deve ser modificado para satisfazer esse requisito.

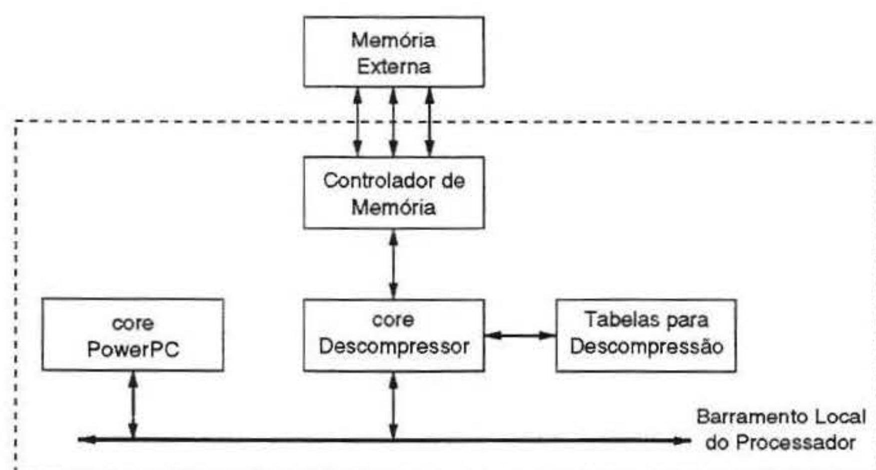


Figura 2.9: Arquitetura do IBM CodePack

tag	Tamanho do Índice	Tamanho Total	Descrição
00	3	5	8 valores mais freqüentes
01	5	7	próximos 32 valores mais freqüentes
100	6	9	próximos 64 valores mais freqüentes
101	7	10	próximos 128 valores mais freqüentes
110	8	11	próximos 256 valores mais freqüentes
111	16	19	valor não compactado

Tabela 2.3: CodePack: Codificação dos 16 bits mais significativos

tag	Tamanho do Índice	Tamanho Total	Descrição
00	0	2	valor 0 (zero)
01	4	6	próximos 16 valores mais freqüentes
100	5	8	próximos 32 valores mais freqüentes
101	7	10	próximos 128 valores mais freqüentes
110	8	11	próximos 256 valores mais freqüentes
111	16	19	valor não compactado

Tabela 2.4: CodePack: Codificação dos 16 bits menos significativos

a razão de compressão fica em torno de 60% a 65%. A variação no desempenho informada é de ±10%.

Dois modelos foram propostos por Lekatsas [53] para compressão: SAMC e SADC. Inicialmente, foram estudados apenas a forma de compressão, através de um modelo de Markov semi-adaptativo e codificação aritmética e o uso de dicionários para instruções consecutivas. Uma breve revisão dos modelos é descrita a seguir:

SAMC: É um método de codificação independente de arquitetura. As instruções são divididas em *streams* de bits que são analisados e comprimidos utilizando Compressão Semi-adaptativa de Markov.

SADC: É dependente da arquitetura. Seqüências de instruções são armazenadas em um dicionário e depois o programa é codificado com índices para o dicionário.

Experimentos foram realizados para MIPS e i386 conseguindo razões de compressão em torno de 50% para o primeiro e 65% para o segundo nesse primeiro trabalho, sem incluir informações sobre o tamanho do descompressor.

Em um trabalho posterior, Lekatsas [54] tratou o mecanismo de resolução de endereços. A compressão passou a ser efetuada em blocos de um byte que necessitam de descompressão seqüencial, os blocos comprimidos foram alinhados em bytes e para conversão de endereços foi utilizada uma LAT. Não foi mostrada nenhuma estimativa do desempenho final do descompressor projetado, que era o principal objetivo do artigo.

Em outra versão do seu descompressor para SAMC [52], Lekatsas propôs a substituição da LAT pela codificação dos novos endereços diretamente nas instruções de salto. Para o processador SHARC, a razão de compressão média ficou em 48%[6]. Para o ARM, a média da razão de compressão ficou em 55%. O problema desse descompressor é o

[6] Cada instrução da arquitetura SHARC possui 48 bits.

elevado número de ciclos para descomprimir as instruções e nesse caso, ele só é capaz de descomprimir em média 6,84 bits por ciclo, o que o torna muito lento.

Posteriormente, Lekatsas direcionou seus trabalhos de compressão na busca de economia de energia [49, 50, 51] conseguindo queda no consumo de energia total entre 22% e 82%. Para isto, um dos artifícios utilizados foi colocar o módulo descompressor entre a *cache* e o processador. Num desses trabalhos [51] fornece uma razão de compressão para a arquitetura SPARC de 54% utilizando um método baseado em tabelas. No entanto, não ficou claro se foram utilizados os tamanhos das tabelas nos cálculos das razões de compressão. Detalhes sobre a implementação não foram divulgados.

2.2.4 Outros Trabalhos

Breternitz [15] propôs em 1997 uma forma elegante para o tratamento da conversão de endereços descomprimidos em endereços comprimidos. Para implementar sua proposta, é apenas necessário mudar a forma de interpretação dos bits dos endereços de memória (Figura 2.10). No modo de endereçamento original, antes da inclusão da *cache* nos processadores, o endereço era visto como apenas um bloco de bits que indicava a posição de um byte na memória (formato "original" na Figura 2.10). Após a inclusão da *cache*, o endereço passou a ser interpretado como um mapeamento para uma certa posição desta através de um *tag*, que é comparado com o *tag* armazenado, um índice que indica a linha da *cache* que deve ser comparada e um offset que indica qual palavra dentro da linha da *cache* é a desejada. Esta visão também fornece um mapeamento direto com a memória, bastando para isso utilizar os 3 campos concatenados. Na proposta alternativa, o endereço em bytes da instrução fica separado do *offset* em relação ao início da *cache-line*. Essa interpretação reorganiza o mapeamento da memória provocando perda no espaço endereçável, mas como grande trunfo, ela torna desnecessária a conversão entre endereços comprimidos e descomprimidos quando o programa é processado para utilizar endereços

neste formato. Um aspecto interessante desta estratégia é a inexistência de continuidade no mapa de memória descomprimida. Note que esse fato pode ser ignorado a não ser pelas últimas instruções de cada *cache-line* que precisam ser ajustadas para indicar a localização da próxima instrução que não será necessariamente a próxima da memória.

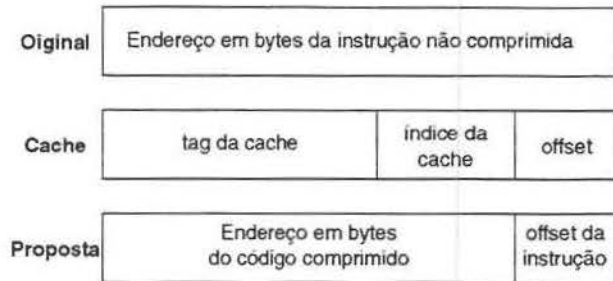


Figura 2.10: Interpretações para o *Instruction Pointer* dadas por Breternitz

A Figura 2.11 mostra três mapas de memória indicando a relação entre os endereços original, comprimido e descomprimido. Considerando a como o endereço inicial do programa original e c como o endereço inicial do programa comprimido, temos a equivalência entre $a0$ e $c0$, $a1$ e $c1$ e assim sucessivamente. A notação utilizada divide o endereço base e o *offset* da instrução. No caso de endereços não comprimidos (original), e *cache-lines* de 4 instruções (conforme a Figura 2.11), se o endereço $a0$ for mapeado no endereço 40000000_h, o endereço $a1$ será mapeado no endereço 40000010_h (4 instruções após $a0$). No exemplo, a instrução `jmp (a3)2` significa saltar para o endereço $a3$ e executar a instrução da posição 2 desse bloco de memória (*cache-line*). Como o mapeamento é feito de forma direta, $(a3)2$ corresponde ao endereço 40000038_h na memória. Considerando agora o mesmo exemplo sob o ponto de vista comprimido, a instrução de salto passa a ser `jmp (c3)2`, que continua significando saltar para o endereço $c3$ e executar a instrução da posição 2 dentro desse bloco. Como agora o bloco está comprimido, ele terá que ser descomprimido para que a instrução possa ser localizada. Mas isso não causa problemas pois o *refill* da *cache* fará a leitura do bloco inteiro. Como resultados, o autor cita que a razão de compressão foi

de 56%, obtida aplicando esse método juntamente com método do CCRP para PowerPC. Infelizmente o conjunto de testes foi baseado apenas em um programa e nas variações do algoritmo de compressão do CCRP.

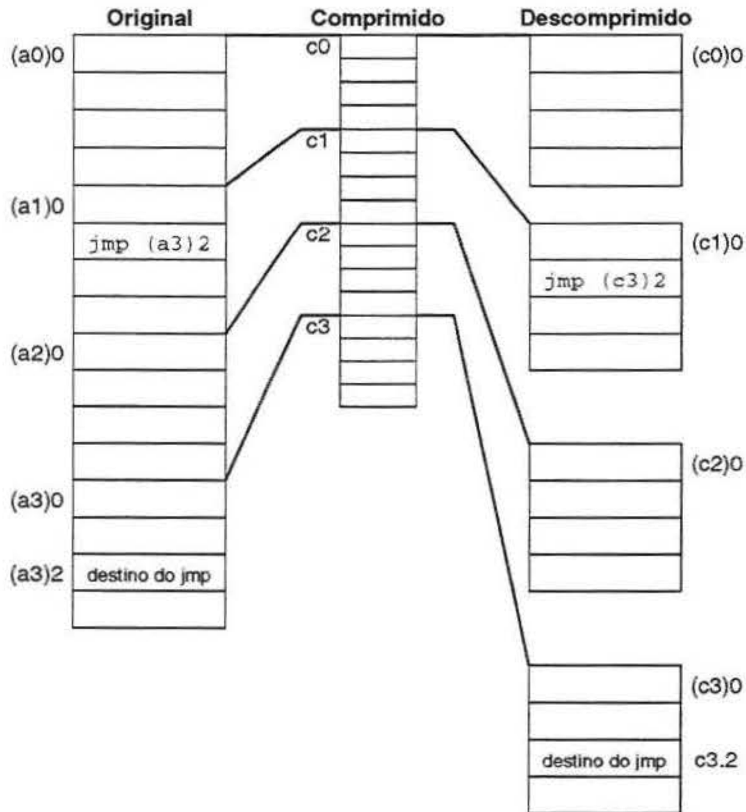


Figura 2.11: Mapas de memória equivalentes

Baxter et al. [11] realizou um trabalho de detecção de clones no código fonte dos programas e obteve uma razão de compressão média de 87,3%. Clones são trechos de código repetidos no programa, normalmente devido a técnicas como copiar e colar amplamente disponíveis nos ambientes de desenvolvimento de software.

Clausen et al. [19] comprimiu *Bytecodes* Java e criou uma nova versão da máquina virtual capaz de interpretar diretamente o código comprimido, que é um conjunto de *Bytecodes* e macros que podem ser traduzidas diretamente para os *Bytecodes* originais. Como resultado, eles obtiveram uma razão de compressão de 70% com uma perda de desempenho de 30%.

Okuma et al. [61] propôs um método de codificação dos campos de imediatos das instruções para o DLX. Ele utiliza a técnica de dicionário para armazenar os valores imediatos e substitui suas ocorrências pelos índices do dicionário. Em uma variação do método, os imediatos que ocuparem o mesmo tamanho em bits dos índices são armazenados na própria instrução. A razão de compressão obtida ficou entre 85,3% e 88,3% considerando o custo do dicionário.

Cooper et al. [20] propôs um método de detecção de sufixos semelhantes nos blocos básicos do programa, substituindo-os por saltos dentro do programa. Não é necessário nenhum dicionário nem modificação no processador. A implementação proposta tende a tirar proveito de otimizações do compilador que foram direcionadas para a compactação, tal como a troca de nome dos registradores. A razão de compressão foi de apenas 95% com uma queda de desempenho em torno de 5%.

Debray et al. [22, 23, 24] utilizou técnicas de compilação voltadas para a compressão de código, implementados sobre o código binário já compilado. Entre essas técnicas estão algumas otimizações entre procedimentos, fatoração de código e abstração de procedimentos. A razão de compressão média obtida foi de 70% com melhora do desempenho de 10% para processadores Alpha.

Kwon et al. [43] propôs o TOE, que é uma variação do conjunto de instruções do Thumb. Embora ele seja capaz de executar código compactado, a inovação foi incluir marcadores para indicar quais das novas instruções de 16 bits geradas podem ser executadas em paralelo com outras. No conjunto de programas utilizado, 33,4% das instruções podem ser executadas em paralelo.

Nam et al. [60] direcionou seu trabalho para uma implementação VLIW do processador SPARC (apenas geração do programa comprimido). O método é baseado na fatoração das instruções e uso de dicionários. Foram conseguidas razões de compressão de 63%, 69% e 71% para arquiteturas VLIW com 4, 8 e 12 unidades de execução.

Nystrom et al. [64] e Runeson [65] implementaram abstração de procedimentos em um compilador proprietário e obtiveram uma razão de compressão de até 88%. Não foram fornecidas informações sobre a arquitetura utilizada.

2.3 Quadro Comparativo

A Tabela 2.5 mostra um quadro comparativo entre os diversos métodos de compressão descritos neste capítulo, levando em conta as métricas descritas na Seção 2.1: (a) A arquitetura base; (b) A razão de compressão obtida; (c) O desempenho final do sistema; (d) O método de compressão utilizado. Neste quadro, considera-se como 100% a razão de compressão e o tempo de execução do programa original.

Para comparar as razões de compressão resultantes de cada um dos métodos, é necessário avaliar com muito cuidado como estas foram calculadas. Alguns fatores afetam fortemente a razão de compressão de um método:

Arquitetura base: A arquitetura utilizada na implementação é um grande diferencial entre dois métodos distintos ou mesmo entre duas implementações de um mesmo método. Como pode ser observado nos trabalhos de Lefurgy [44, 45, 46], o código para o PowerPC permitiu mais compressão que o do ARM que por sua vez mostrou resultados melhores que o i386, mas o i386 utiliza instruções de tamanho variável enquanto os outros dois processadores utilizam instruções de tamanho fixo (32 bits). O mesmo acontece no trabalho de Lekatsas [52], onde os resultados para o SHARC superam os resultados do ARM, mas o processador SHARC utiliza 48 bits por instrução enquanto o ARM utiliza apenas 32. Desta forma, as razões de compressão só devem ser comparadas se a arquitetura utilizada for a mesma entre os dois métodos;

Parâmetros do compilador: O compilador é um fator determinante na qualidade do código gerado. Como será mostrado no Capítulo 3 (Tabela 3.1), a troca de parâmetros

Seção	Modelo	Arquitetura Base	Razão de Compressão	Tempo de Execução[k]
2.2.1	Thumb[a b]	ARM	55%~70%	70%~120%
	MIPS16[a]	MIPS	60%	n/d
2.2.2	Fraser[c]	SPARC	50%	2000%
	Ernst[c]	SPARC	59%	1260%
	Liao[d]	TMS320C25	84%~88%	115%~117%
	Kirovski[c e]	SPARC	60%	111%
2.2.3	Wolfe[a f]	MIPS	70%	~100%
	Lefurgy[a]	PowerPC, ARM, i386	61%, 66%, 75%	n/d
	Lefurgy[a g h]	SHARC	42%~64%	n/d
	CodePack[a]	PowerPC	60%~65%	90%~110%
	Lekatsas[a i]	MIPS, i386	50%, 65%	n/d
	Lekatsas[a j]	SHARC, ARM	48%, 55%	n/d
	Lekatsas[a]	SPARC	54%	n/d

[a] Implementado em Hardware.

[b] Perda de desempenho com barramento de 32 bits e ganho com barramento de 16 bits.

[c] Totalmente implementado em Software.

[d] Hardware opcional pode acelerar o desempenho.

[e] Números estimados.

[f] Desempenho estimado.

[g] Foi utilizada otimização `-O1` apenas.

[h] O processador SHARC utiliza instruções de 48 bits.

[i] Apenas experimentos sobre a forma de comprimir os programas.

[j] O descompressor é capaz de descomprimir em média 6,84 bits por ciclo.

[k] Considerando como 100% o tempo de execução do programa original sem utilizar compressão.

Tabela 2.5: Quadro comparativo dos métodos de compressão de código.

do compilador pode causar uma redução de 13,4% no número de instruções de um programa;

Custo do descompressor: O custo do descompressor, ou ao menos uma estimativa dele, deve ser incluído na razão de compressão, uma vez que o descompressor influencia na área final ocupada pelo sistema. Esse custo nem sempre foi considerado nos trabalhos, como no caso do MIPS16 [40], Wolfe [68] e Lekatsas [44, 45, 46].

Para a arquitetura ARM, o trabalho mais completo foi o Thumb [10], que é uma versão comercial. Os trabalhos de Lefurgy [44, 45, 46] e Lekatsas [52] mostraram que existem formas diferentes de obter uma razão de compressão similar. Entretanto, em nenhum desses dois trabalhos foram obtidos resultados de desempenho do hardware (foram realizados apenas testes de compressão).

Dos trabalhos para a arquitetura MIPS, novamente tem destaque a implementação MIPS16 [40] que é uma versão comercial. Nenhum dos trabalhos forneceu medidas reais de desempenho, mas pela similaridade do MIPS16 com o Thumb, pode-se supor que o desempenho também seja similar entre os dois modelos. Sendo assim, o trabalho de Wolfe [68] possui um desempenho melhor que o do MIPS16. Infelizmente esse desempenho é baseado em estimativas apenas. Lekatsas [53] conseguiu uma razão de compressão melhor que a do MIPS16, mas esta razão de compressão foi apenas uma estimativa inicial, pois nem mesmo tratou dos problemas de resolução de endereços. No Capítulo 3 serão mostrados dois outros métodos com resultados para a arquitetura MIPS e no Capítulo 4 serão mostrados os resultados para MIPS do método proposto nesta Tese.

Para a arquitetura SPARC predominam trabalhos implementados em software, como os de Fraser [32] e Ernst [26] (Fraser é co-autor desse trabalho) que não levaram em consideração a descompressão em tempo real, por isso os resultados de desempenho são tão fracos. O trabalho de Kirovski [39] apresenta resultados bastante satisfatórios, mas

que são apenas estimativas, sem uma implementação real. Lekatsas [51] também forneceu resultados para SPARC, mas sem muitos detalhes da implementação e também sem resultados de desempenho. No Capítulo 5 mostraremos os resultados da aplicação do método proposto nesta tese para a arquitetura SPARC, incluindo razão de compressão, projeto do módulo descompressor e resultados de desempenho obtidos. Um fator muito importante a ser ressaltado para a arquitetura SPARC é a inexistência de um produto comercial que inclua compressão de código, reduzindo os estudos a trabalhos acadêmicos apenas. Esse pode ser um dos fatores da baixa utilização desta arquitetura em sistemas dedicados.

Capítulo 3

Análise de Técnicas de Compressão

O pequeno conjunto de instruções das arquiteturas RISC aliado ao código bastante regular e padronizado gerado pelos compiladores faz com que os programas possuam grande quantidade de instruções repetidas. Nesse capítulo são apresentados dois métodos de compressão que exploram as características acima: Compressão Baseada em Árvores[1](TBC) e Compressão Baseada em Árvores Fatoradas[2](PBC) que foram propostos preliminarmente para arquitetura MIPS e serviram de modelos para o método de Compressão Baseada em Instruções[3] (IBC) proposto nesse trabalho que será mostrado no próximo capítulo.

Os métodos TBC e PBC são baseados em árvores de expressões ou em partes dela. Uma instrução é a raiz de uma árvore de expressão se satisfizer uma das seguintes regras [6]:

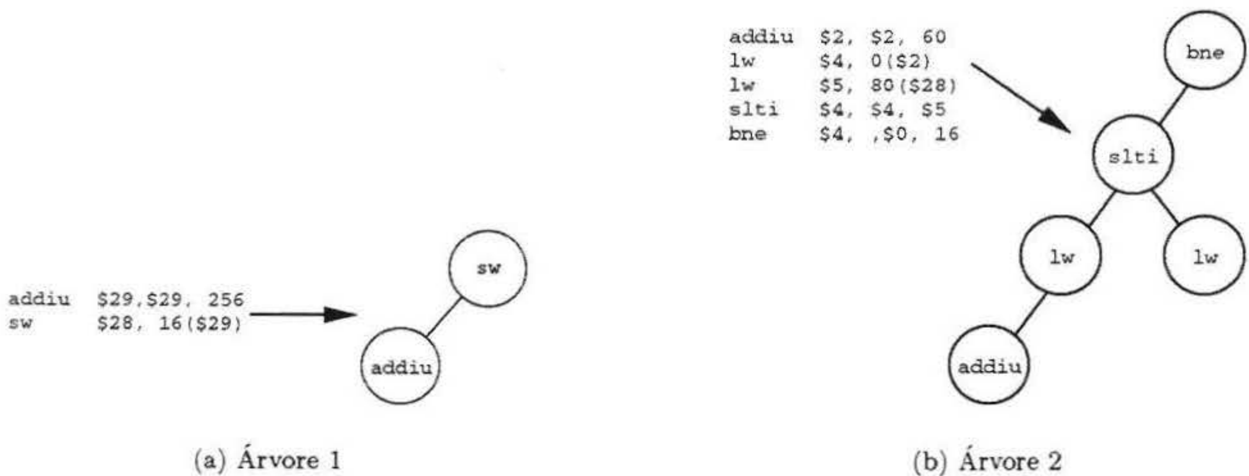
- A instrução armazena dados na memória (`store`);
- O registrador de destino da instrução é utilizado por mais de uma instrução dentro do bloco básico;

[1]Do Inglês Tree Based Compression.
[2]Do Inglês Pattern Based Compression.
[3]Do Inglês Instruction Based Compression.

- O registrador de destino da instrução é utilizado por uma instrução de outro bloco básico;

- A instrução é uma instrução de salto.

Uma árvore de expressão não ultrapassa os limites de um bloco básico. Exemplos de árvores de expressão para o código MIPS são mostrados na Figura 3.1.



(a) Árvore 1 (b) Árvore 2

Figura 3.1: Exemplos de árvore de Expressão

Os programas utilizados como *Benchmark* pelos dois métodos fazem parte do conjunto SPEC CINT95 e foram compilados para o processador MIPS usando o compilador *gcc* versão 2.8.1. Durante a compilação foram utilizadas as otimizações `-O2`, que incluem praticamente todas as otimizações do compilador, e `-Os`, que incluem das otimizações do `-O2` apenas as que não implicam em aumento do tamanho do código. Também foram utilizadas as opções `-mips1` e `-mips2` para escolha dos conjuntos de instruções MIPS I e MIPS II respectivamente. Os resultados estão na Tabela 3.1. O código gerado para `-mips1` fica maior que o para `-mips2` principalmente por causa dos `NOPs` necessários para

preencher os *delay slots*. Os parâmetros `-mips2 -Os` foram escolhidos para os programas criados para MIPS por gerarem o código mais compacto.

Programa	-mips1 -O2	-mips1 -Os	-mips2 -O2	-mips2 -Os
compress	2304	2304	2164	2152
gcc	409204	407636	364524	363560
go	79776	80284	73908	72516
ijpeg	52816	52336	48548	47988
li	20832	20652	18616	18448
perl	80308	79676	70228	69536
vortex	167212	167384	151476	151348

Tabela 3.1: Parâmetros utilizados e número de instruções geradas

3.1 Compressão Baseada em Árvores (TBC)

No método de Compressão Baseada em Árvores de Expressão, os símbolos utilizados são as árvores de expressão conforme a definição anterior. O método foi definido em [7, 8, 17, 18].

3.1.1 Análise do Método TBC

Para avaliar a capacidade de compressão do TBC, foi feito um levantamento da quantidade de árvores únicas em um programa que é mostrado na Tabela 3.2. Observa-se que o número de árvores únicas é bem menor que o número total de árvores. Na média, este valor corresponde a apenas 24% de todas as árvores do programa.

O gráfico da Figura 3.2 mostra a distribuição dessas árvores para os programas utilizados. Neste gráfico, temos que 20% de todas as árvores únicas cobrem mais de 80% dos programas em média. Isso sugere que elas devam ser comprimidas por um método que atribua códigos menores às árvores que mais ocorrem e códigos maiores às que menos ocorrem, como é o caso do código de Huffman. No entanto, decodificadores para

Programa	Árvores Totais	Árvores Distintas
compress	1844	832 (45,1%)
gcc	291758	51186 (17,5%)
go	62423	12460 (20,0%)
ijpeg	40621	11264 (27,7%)
li	15509	3072 (19,8%)
perl	52276	12793 (24,5%)
vortex	130336	17463 (13,4%)

Tabela 3.2: Número de árvores distintas nos programas. Os números entre parênteses indicam a porcentagem em relação ao total.

codewords codificadas por Huffman são mais complexos que decodificadores baseados em *codewords* de tamanho fixo [12, 13, 18]. Durante a compressão, as árvores são divididas em n_c classes, cada uma delas contendo n_k *codewords* de tamanho fixo. A cada *codeword*, é adicionado um prefixo de $\lceil log_2 n_c \rceil$ bits que indica sua classe. As *codewords* da classe k têm tamanho dado por $\lceil log_2 n_k \rceil$. À cada árvore é associado um par [prefixo, *codeword*] (Figura 3.3), onde o prefixo indica a classe a qual a árvore pertence.

O número de classes e o tamanho delas são definidos através de uma busca exaustiva tendo como limites um conjunto de 2 a 8 classes. Na Figura 3.4, são mostrados os melhores resultados para cada número de classe estudada, que ocorrem normalmente para 4 classes (em alguns casos, o melhor resultado ocorre com 5 classes, mas a diferença é muito pequena). Esse ponto mínimo na curva ocorre no momento em que o custo de adicionar mais um bit na codificação das classes passa a superar a vantagem da existência de mais classes com tamanhos distintos. Na Tabela 3.3 são mostradas as combinações de tamanhos para cada uma das classes, indicando como melhor resultado a combinação 1/5/8/12 que proporciona uma razão de compressão de 23,4%. O programa é então reescrito substituindo cada uma das árvores pelos respectivos pares [prefixo, *codeword*].

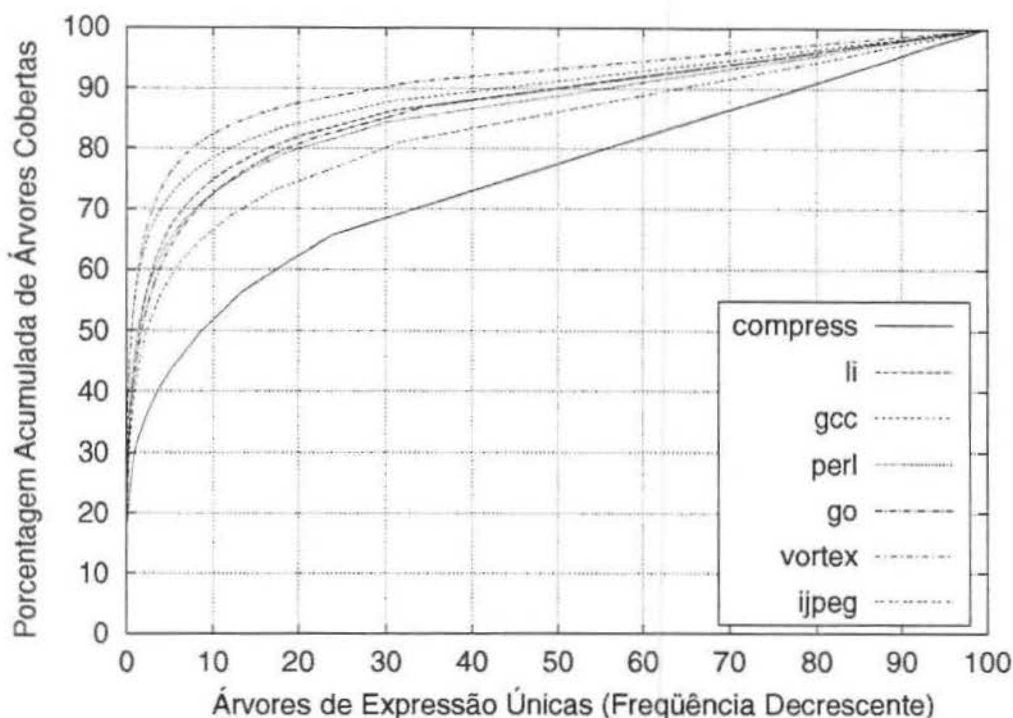


Figura 3.2: Porcentagem das árvores do programa cobertas por árvores distintas.

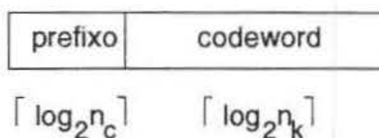


Figura 3.3: Codificação das árvores

Como o tamanho dos pares não é necessariamente um múltiplo da palavra de memória, alguns deles serão divididos em mais de uma palavra como na Figura 3.5[4], cabendo ao descompressor agrupá-los e interpretá-los corretamente.

Os resultados para os programas escolhidos são mostrados na Tabela 3.4. A razão de compressão média dos programas é de 27,2%.

[4]Os pares estão representados por (pn,cn).

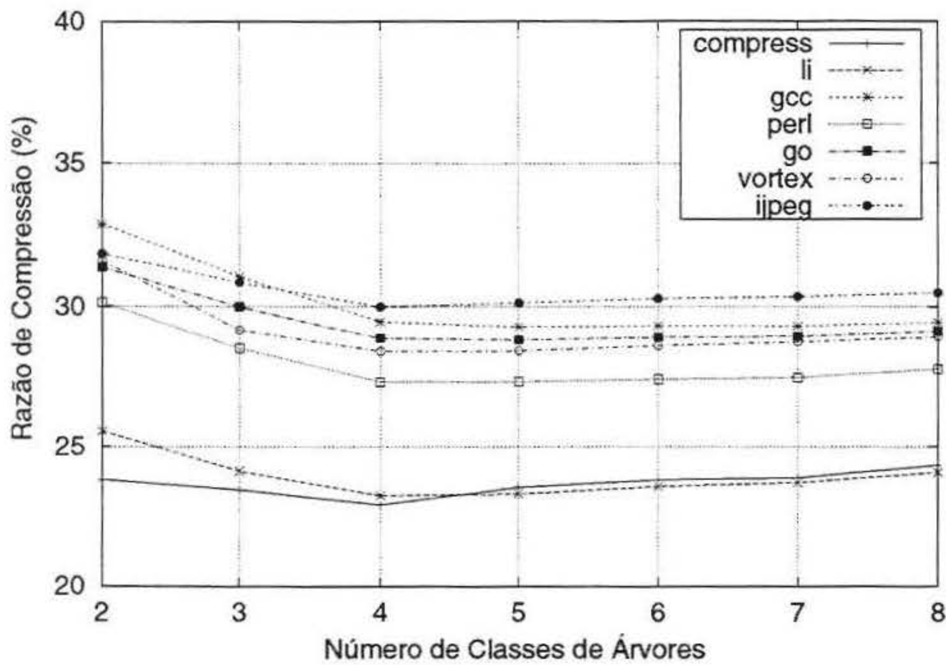


Figura 3.4: Razão de compressão para diferentes quantidades de classes de árvores.

Tamanho				Razão de
I	II	III	IV	Compressão
1	1	1	12	30,3%
1	1	2	12	29,2%
⋮	⋮	⋮	⋮	⋮
1	**5**	**8**	**12**	**23,4%**
1	5	9	12	23,5%
⋮	⋮	⋮	⋮	⋮
9	9	8	12	31,2%
9	9	9	12	30,5%

Tabela 3.3: Todas as combinações de tamanhos de *codewords* para o programa *li* utilizando 4 classes.

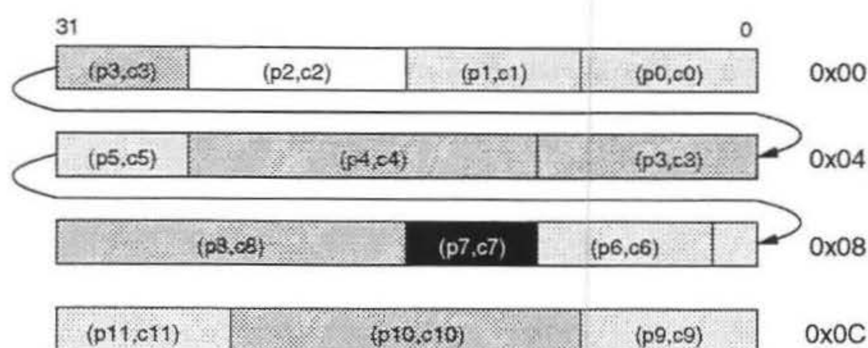


Figura 3.5: Árvores comprimidas na memória

Programa	Tam. das Classes				Razão de Compressão
	I	II	III	IV	
compress	1	5	8	10	22,9%
gcc	2	8	12	16	29,4%
go	3	8	11	14	28,8%
ijpeg	3	8	11	14	29,9%
li	2	6	9	12	23,2%
perl	2	7	10	14	27,3%
vortex	1	6	10	14	28,4%

Tabela 3.4: Partições que resultam nos melhores resultados para 4 classes e as razões de compressão obtidas.

3.1.2 O Descompressor Baseado em TBC

O descompressor proposto para o método TBC é mostrado na Figura 3.6. Ele trabalha da seguinte forma. Primeiro ele extrai os prefixos e *codewords* das palavras da memória (`Tc`), que são posteriormente decodificados pelo módulo `TGEN` e convertidos no endereço `taddr`. O endereço `taddr` indica a primeira entrada no diretório de árvores (`TD`) que armazena a árvore desejada. O diretório de árvores é composto por dois campos: `INSTR` e `END`.

INSTR é um campo de 32 bits que contém uma instrução descomprimida. O campo END é responsável por marcar a última instrução de cada árvore (END=1), sendo utilizado como entrada ld do módulo INC, que solicita a carga da próxima *codeword*.

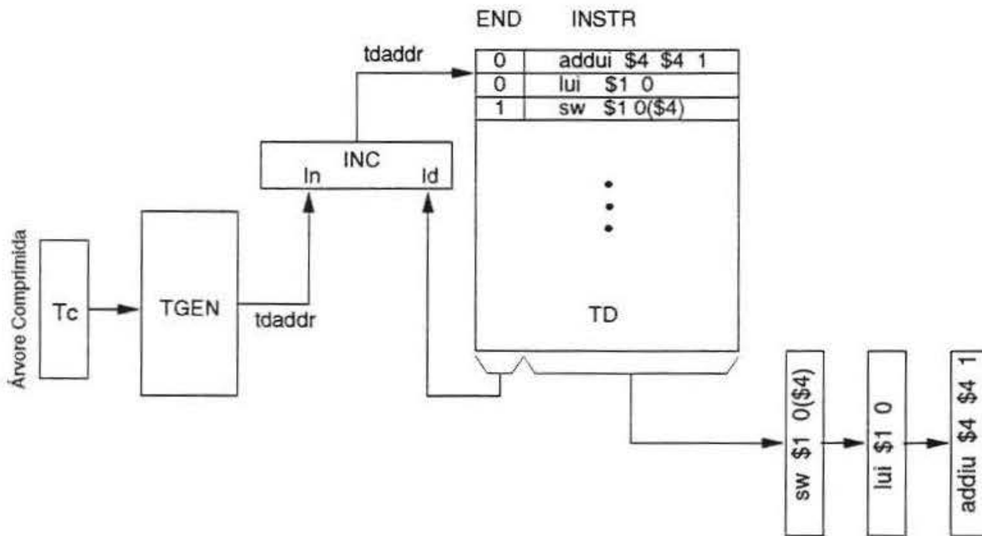


Figura 3.6: Descompressor para o Método de Compressão Baseado em Árvores

Quando ocorre um desvio no fluxo de execução (*branches*, interrupções, etc) é necessário localizar a posição da árvore comprimida na memória. Isso é feito utilizando a Tabela de Conversão de Endereços[5] (ATT), que será mostrada na Seção 4.2. Para determinar a razão de compressão total dos programas, uma estimativa da área do circuito descompressor foi realizada. Na Figura 3.7 é mostrada uma estimativa das áreas de cada um dos componentes extras somadas à razão de compressão básica. A razão de compressão final média obtida pelo método é de 60,7%.

[5] Do Inglês *Address Translation Table*.

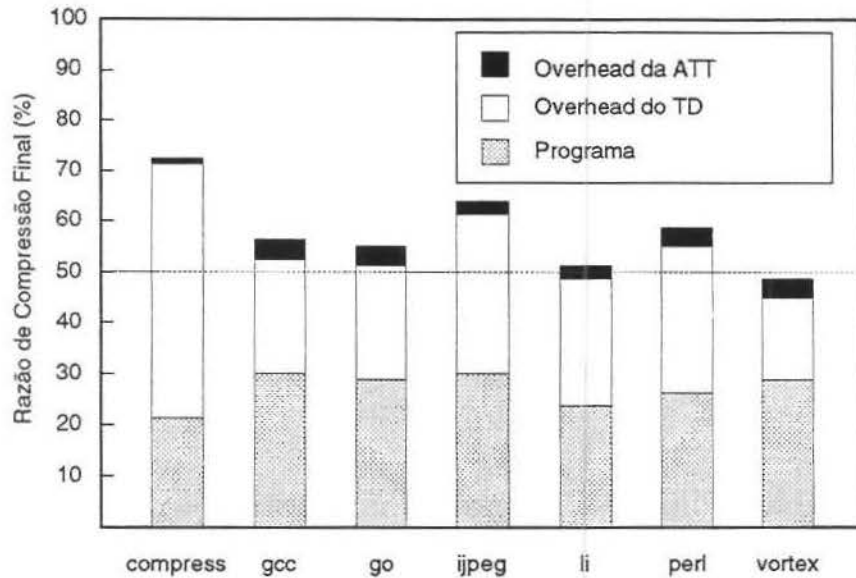


Figura 3.7: Razão de compressão final para TBC.

3.2 Compressão Baseada em Árvores Fatoradas (PBC)

No método de Compressão Baseada em Árvores de Expressão Fatoradas (PBC), os símbolos utilizados são os padrões de árvores e padrões de operandos. O método foi estudado em [7, 8, 9, 62].

3.2.1 Análise do Método PBC

A idéia principal desse método é fatorar os padrões de operandos e os padrões de árvores de expressão. Considere, por exemplo, a árvore de expressão da Figura 3.8(a). A Figura 3.8(b) mostra o padrão de árvore fatorado. Os asteriscos (*) indicam os operandos que foram removidos. O padrão de operando desta árvore é mostrado na Figura 3.8(c), e é obtido percorrendo o código da árvore de expressão e listando todos os seus operandos.

A Tabela 3.5 mostra o número de árvores e padrões de árvores e de expressões para o conjunto de programas. De acordo com a tabela, o programa *gcc*, possui 291758 árvores de expressões diferentes, que podem ser representadas por apenas 921 (0,3%) padrões

```
addiu $4, $4, 1          addiu *, *, *
lui $1, 0                lui *, *
sw $1, 0($4)             sw *, *(*)
    (a)                      (b)

        [$4,$4,1,$1,0,$1,0,$4]
                 (c)
```

Figura 3.8: (a) Árvore de Expressão; (b) Padrão de árvore; (c) Padrão de operandos.

de árvores e 45469 (15,6%) padrões de operandos. Principalmente no caso dos padrões de árvores, nota-se a regularidade do código gerado e também a pequena quantidade de opções disponíveis para o compilador gerar código. No caso do *compress*, que é o menor programa do conjunto, os padrões de árvores correspondem a 5,8% de todas as árvores e os padrões de operandos correspondem a 41,6% destacando a relação entre o esgotamento das possibilidades de padrões de árvores e o tamanho do programa.

Programa (I)	Árvores (II)	Árvores distintas (III)	Padrões de Árvores (IV)	Padrões de Operandos (V)	(III) - (V) (VI)
compress	1844	832	107 (5,8)	767 (41,6)	8,5%
gcc	291758	51186	921 (0,3)	45469 (15,6)	12,6%
go	62423	12460	256 (0,4)	11373 (18,2)	9,6%
ijpeg	40621	11264	348 (0,9)	9907 (24,4)	13,7%
li	15509	3072	169 (1,1)	2840 (18,3)	8,2%
perl	57276	12793	547 (1,0)	11579 (20,2)	10,5%
vortex	130336	17493	324 (0,2)	15592 (12,0)	12,2%
Média	85681	15585	382 (1,4)	13932 (17,4)	10,8%

Tabela 3.5: Número de padrões de árvores e de operandos. Os números entre parênteses mostram a porcentagem em relação ao total de árvores de expressões.

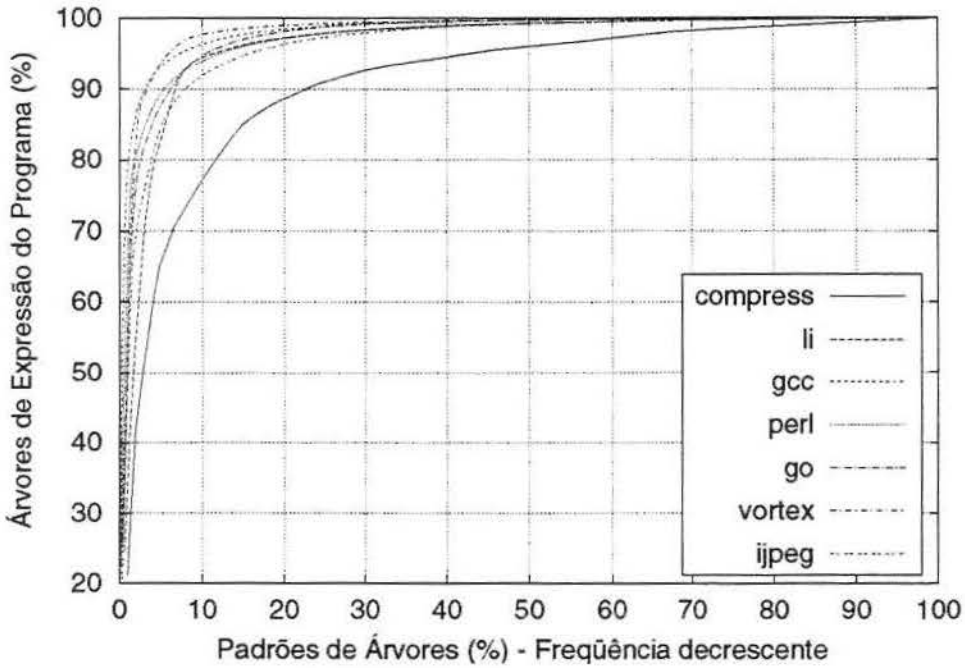
Duas árvores de expressões são diferentes se possuírem ao menos uma instrução diferente. Uma instrução é diferente de outra se elas possuírem o *opcode* e/ou pelo menos

um operando diferente. Existe uma grande relação entre a quantidade de árvores únicas mostradas na coluna (III) da Tabela 3.5 e o número de padrões de operandos na coluna (V). Essa relação pode ser visualizada na coluna (VI), que é formada pela diferença entre (III) e (V) que resulta em uma média de 10,8%. Isso significa que, para a maioria dos programas, dado um padrão de operandos só existe um padrão de árvores correspondente a ele. Esse resultado não chega a ser uma surpresa visto que a possibilidade de combinações entre registradores e imediatos é muito maior que a de *opcodes* numa arquitetura RISC. Como essa relação não é um para um, foi feito um estudo sobre as oportunidades de compressão.

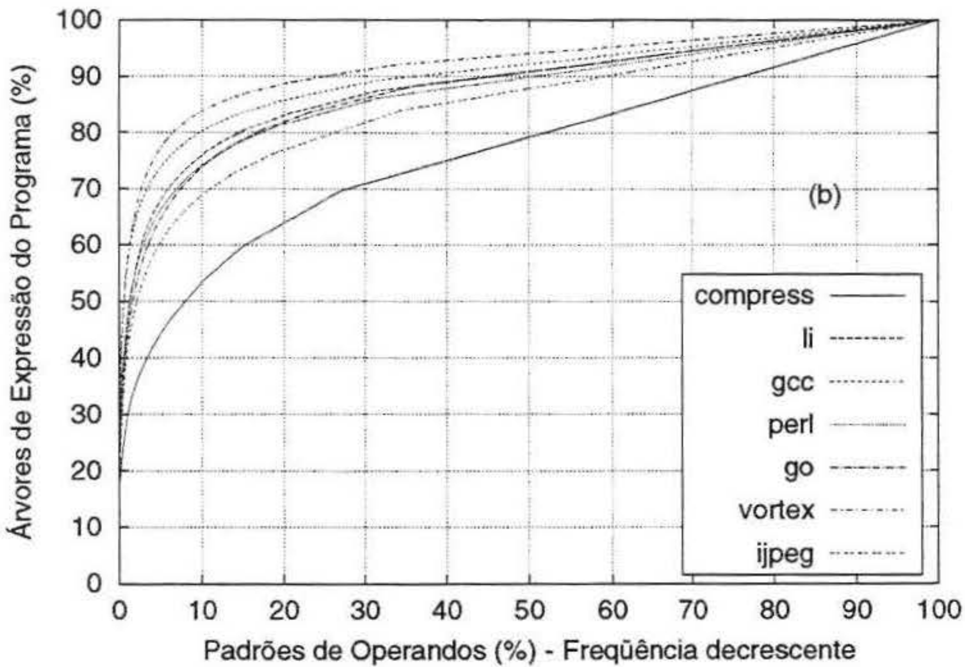
Para determinar a capacidade de compressão dos padrões de árvores e de operandos foram feitos gráficos indicando a distribuição destes (Figura 3.9). A distribuição dos padrões de árvore é mostrada na Figura 3.9(a), na qual podemos ver que 20% de todos os padrões de ávores cobrem praticamente a totalidade das árvores do programa (a não ser para o programa *compress*, que por ser muito pequeno tem uma curva diferenciada). Um comportamento semelhante também pode ser observado na Figura 3.9(b) para os padrões de operandos. Nesse caso, 20% dos padrões de operandos cobrem em torno de 80% das árvores distintas do programa[6].

A fatoração de operandos de árvores destaca o fato de que técnicas de compressão que interpretam a entrada como uma seqüência de bits não são capazes de detectar a relação entre os padrões de árvores e os de operandos separadamente. Por exemplo, um algoritmo como o de Lempel-Ziv não seria capaz de detectar um padrão de árvore comum como `[ld *,*,* : add *,*,*]`. No caso de uma árvore de expressão, o método da fatoração de operandos permitiria identificar uma semelhança maior entre as instruções `subu $2,$0, $4` e `nor $2, $0, $4`, identificando o mesmo padrão de operandos, `[$2, $0, $4]`.

[6]Observe a semelhança entre a Figura 3.9(a)(b) e o comportamento das árvores de expressões da Figura 3.2.



(a) Cobertas por Padrões de Árvores



(b) Cobertas por Padrões de Operandos

Figura 3.9: Porcentagem acumulada das árvores de expressão.

Os padrões de árvores (Tp) e os padrões de operandos (Op) são codificados como pares [Tp, Op] formando as *codewords* e são colocados em seqüência seguindo a ordem do programa para formar o programa comprimido (Figura 3.10). De maneira semelhante ao que foi feito para as árvores de expressão, a Figura 3.11(a-b) mostra as razões de compressão para os padrões de árvore e de operandos de acordo com o número de classes utilizados. No caso dos padrões de árvore, a melhor compressão é obtida com 3 classes. Isso se explica pelo fato dos padrões existirem em pequena quantidade e alguns poucos padrões cobrirem praticamente a totalidade do programa, fazendo com que o custo de um bit adicional na representação do prefixo não seja compensado pelo ganho resultante da adição de novas classes. Por outro lado, a diferença entre as razões de compressão com 3 e 4 classes é de apenas 0,49%. No caso dos padrões de operando, os melhores resultados ocorrem em 4 classes para alguns programas e 5 em outros, mas a diferença entre ambos é de apenas 0,1%.

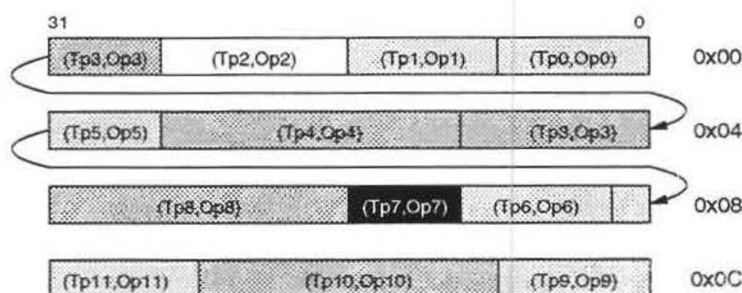
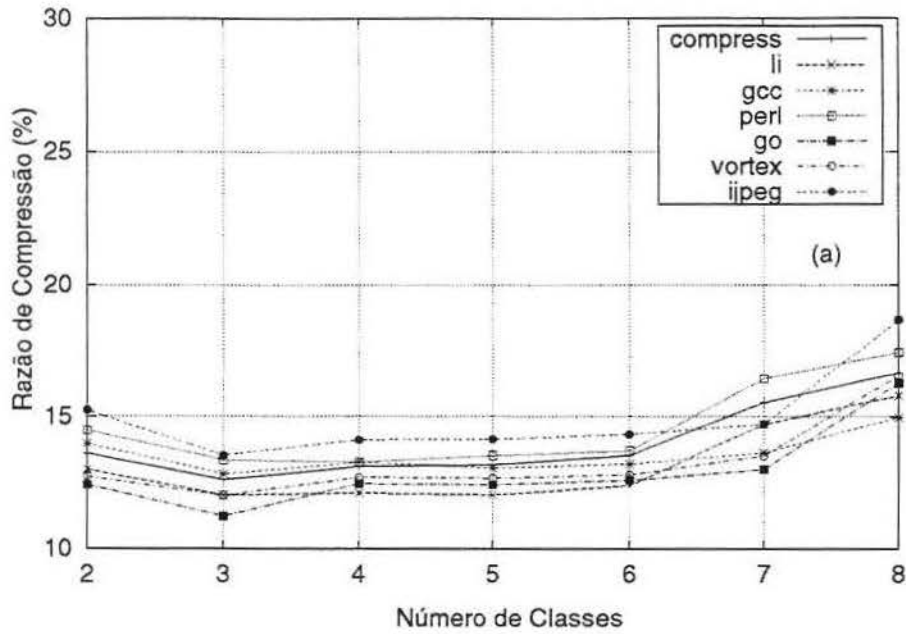
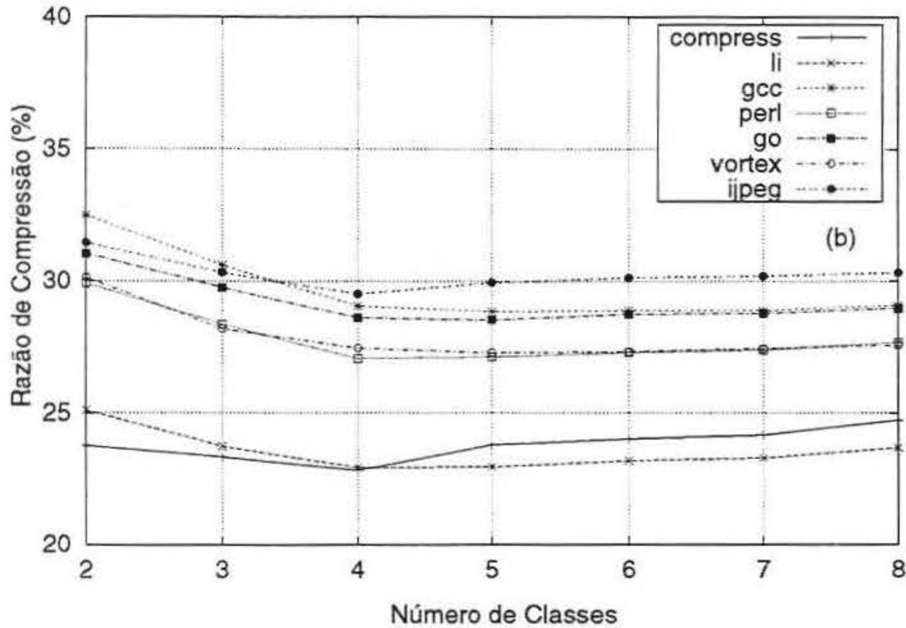


Figura 3.10: Padrões de árvores comprimidos na memória.

A razão de compressão média obtida para os padrões de árvore foi de 13,0% e para os padrões de operandos foi de 26,8%. A Tabela 3.6 mostra o resultado final das razões de compressão quando ambos os padrões são combinados resultando em uma média de 39,8%.



(a) Padrões de Árvores



(b) Padrões de Operandos

Figura 3.11: Razões de compressão com diferentes números de classes.

Programa	Razão de Compressão (Tp)	Razão de Compressão (Op)	Total
compress	22,8%	13,2%	35,9%
gcc	29,1%	13,3%	42,4%
go	28,6%	12,5%	41,1%
ijpeg	29,5%	14,1%	43,6%
li	22,9%	12,1%	35,0%
perl	27,1%	13,3%	40,4%
vortex	27,5%	12,7%	40,2%

Tabela 3.6: Razão de compressão composta quando os padrões de árvore e de operando são combinados.

3.2.2 O Descompressor Baseado em PBC

O descompressor proposto para o método PBC é mostrado na Figura 3.12. Primeiramente os campos `Tp` e `Op` são extraídos da palavra comprimida. Posteriormente, `Tp` é mapeado no endereço de uma seqüência de instruções descomprimidas e `Op` é usado para gerar os campos de registradores e imediatos para essas instruções. Esses dados são enviados para o módulo de montagem das instruções (`IAB`) que é responsável por colocar cada um dos campos no lugar correto, gerando as instruções para o processador. Os módulos do descompressor são descritos a seguir:

Dicionário de Padrões de Árvores (`TPD`): Armazena os *opcodes* correspondentes a cada padrão de árvore. Primeiramente o padrão de árvore (`Tp`) é decodificado através do módulo `TGEN`, que gera o endereço `tpaddr`. Cada entrada do `TPD` é composta por três campos: `OPCODE`, `ITYPE` e `END`. O campo `OPCODE` contém os bits de *opcode* da instrução. O campo `ITYPE` contém o formato da instrução, que é utilizado pelo módulo `IAB` para decidir como montar a mesma. O `IAB` agrupa o `OPCODE`, registradores (`RS1`, `RS2`, `RD`) e o imediato (`IMB`) para formar as instruções. O campo

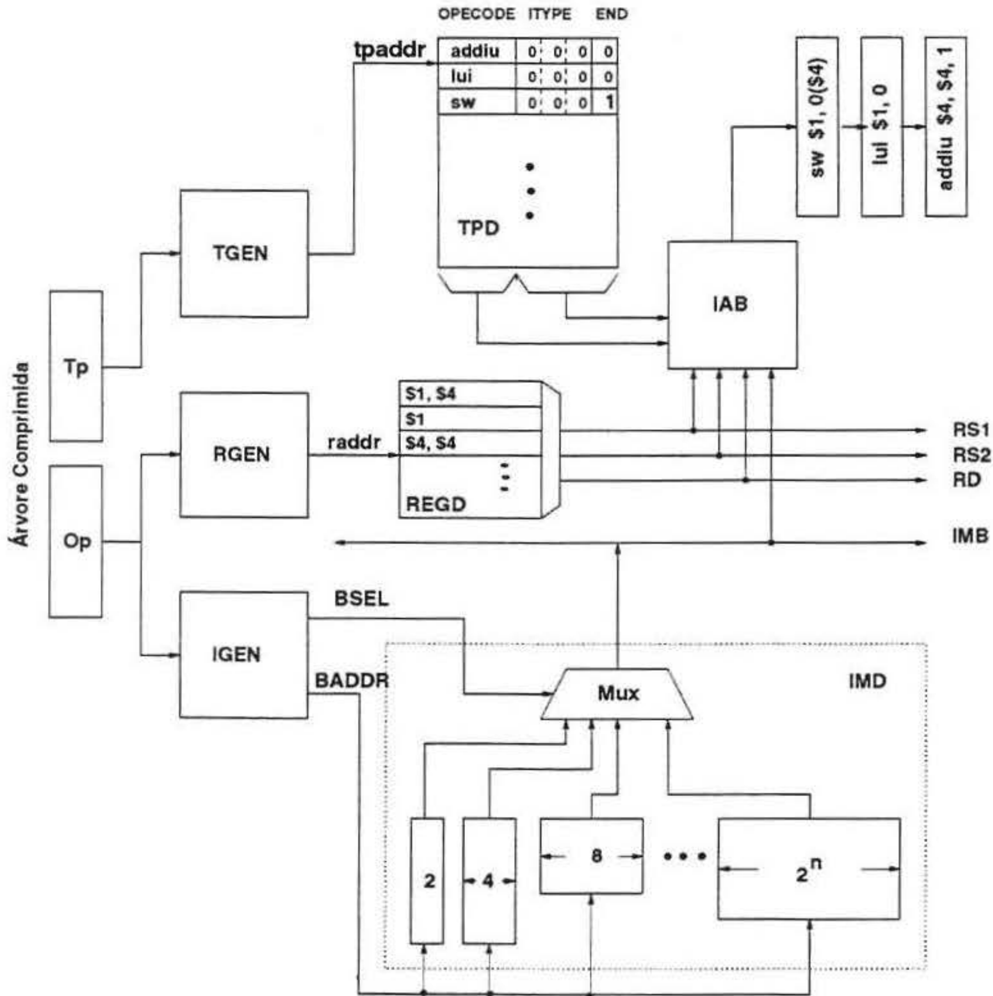


Figura 3.12: Descompressor para o método de Compressão Baseado em Árvores Fatoradas (PBC).

END é utilizado para marcar a última instrução do padrão. Na média, a área do TPD é de apenas 0,9% do programa original.

Dicionário de Registradores (REGD): O Dicionário de Registradores decodifica o padrão de operandos nos registradores necessários para preencher as instruções. A saída do REGD é formada por três barramentos: RS1, RS2 e RD que transportam os bits referentes aos registradores de origem e destino das instruções. A estimativa do

tamanho do REGD foi obtida através da soma de todos os campos de registradores nos padrões de operandos. O tamanho médio do REGD é de 3,8%.

Dicionário de Imediatos (IMD): O dicionário de imediatos armazena os imediatos utilizados pelo programa. Uma entrada no dicionário é criada para cada um dos imediatos encontrados. A variação no tamanho em bits dos imediatos é utilizada para minimizar a largura do IMD, agrupando-os em bancos de memória de acordo com a quantidade de bits necessária para armazena-los. Os sinais BADDR e BSEL são gerados pelo IGEN ao receber um Op. O sinal BADDR fornece o endereço para os bancos de memória e o BSEL seleciona o banco de memória correspondente ao tamanho do imediato. Esse método reduz consideravelmente o tamanho do IMD que ocupa em média 13% de área do programa original.

Dois métodos foram propostos para resolução de endereços. O método inicial [9, 62], foi baseado na alteração do código executável e num módulo adicional de predição de desvios que monitora as instruções enviadas para o processador em busca de endereços de origem das próximas instruções. O problema com esse método é que ele não é capaz de tratar a existência de *caches* no sistema pois não há como detectar o caminho que o processador está executando enquanto houver *cache-hit*. Outro problema é a existência de interrupções, que podem ocorrer a qualquer momento e modificar o fluxo de execução do programa sem que uma instrução de salto seja executada. O segundo método prevê o uso de uma Tabela de Conversão de Endereços [7, 8], que será discutida em detalhes na Seção 4.2.

A razão de compressão total para esse método, incluindo estimativas de custo para os módulos de hardware, é mostrada na Figura 3.13. O custo do descompressor contribui com 21,5% da razão de compressão final. A razão de compressão final média obtida pelo método PBC é de 61,3%.

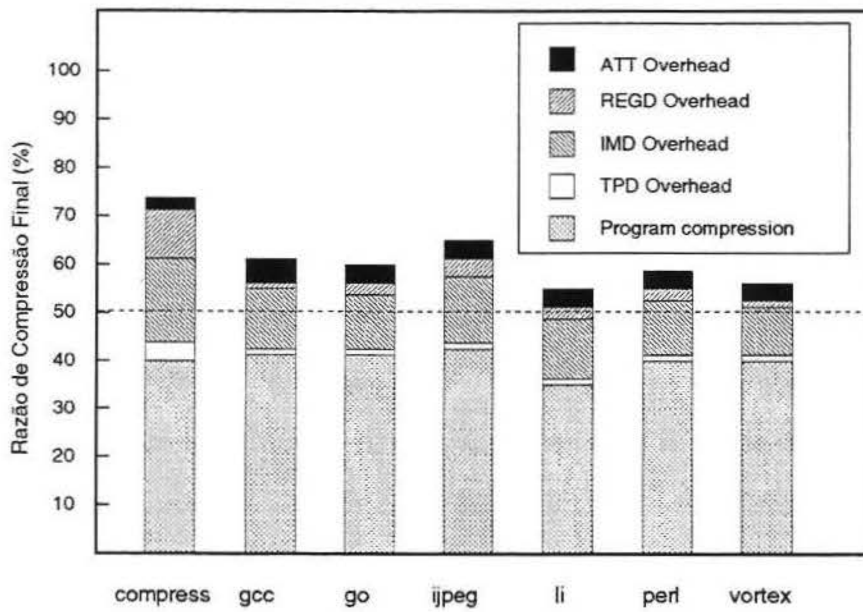


Figura 3.13: Razão de compressão final para o PBC.

Capítulo 4

Compressão Baseada em Instruções (MIPS)

Este capítulo detalha a técnica de Compressão Baseada em Instruções (IBC), que é a contribuição central deste trabalho. Esse método surgiu após verificarmos que o tamanho médio das árvores de expressão em programas típicos é de apenas 1,2 instruções (veja distribuição na Figura 4.1) tornando pouco eficiente estratégias baseadas em árvores de expressão. Um dos fatores que determinam esse pequeno tamanho médio das árvores é o escalonamento feito pelo compilador, mesclando instruções de árvores diferentes para preencher os *delay slots* do código, o que equivale à quebrar as árvores do ponto de vista do compressor[1].

Nesse capítulo são detalhados os resultados preliminares obtidos, usando IBC, para a arquitetura MIPS [7, 8], juntamente com o desenvolvimento teórico do método. Posteriormente será apresentada uma implementação para SPARC, para a qual foi desenvolvido um protótipo (descrito no Capítulo 5).

Uma das propostas do IBC é alterar o mínimo possível a forma como o projeto de um sistema dedicado é criado. Um diagrama simplificado do ciclo de projeto de um sistema sem compressão de código é mostrado na Figura 4.2(a). A compressão do código passa a

[1] O IBC, assim como o TBC e PBC adotam a premissa de não reordenar as instruções do código.

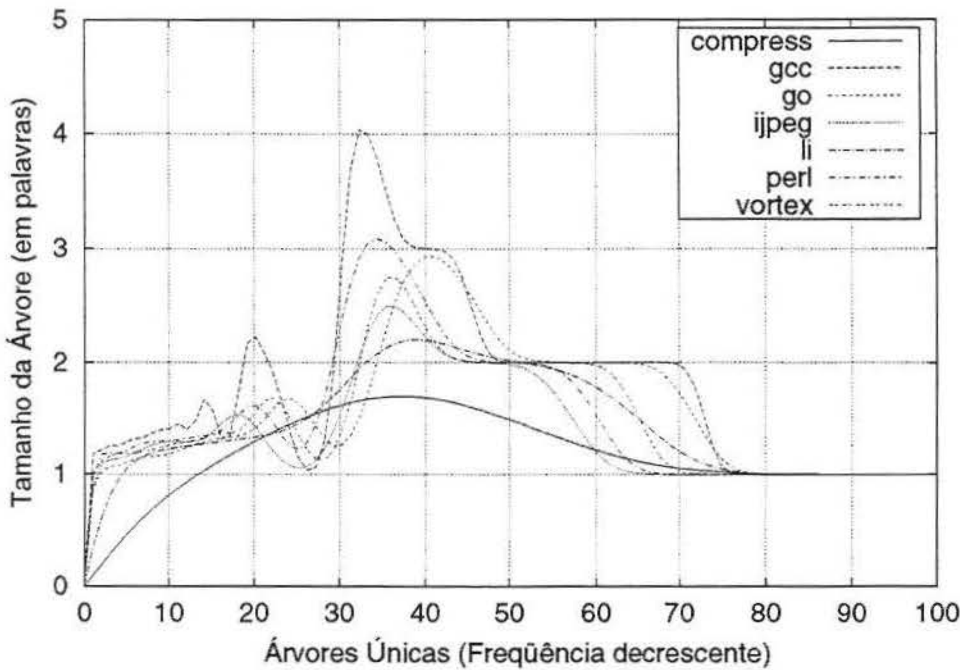


Figura 4.1: Distribuição do tamanho médio das árvores de expressões (aproximação de Bezier).

ser apenas mais uma fase do processo de desenvolvimento, sem que seja necessária a alteração de nenhuma das ferramentas existentes, e é realizada após a validação do software e antes da implementação em hardware como mostrado na Figura 4.2(b). O compressor tem como entrada o arquivo binário executável e a partir dele gera o código executável comprimido e um arquivo de configuração do modelo VHDL do descompressor. Como existem alguns parâmetros que podem ser ajustados no descompressor, a realimentação do estágio de desenvolvimento de hardware permite um retorno à compressão de código para que os novos arquivos de configuração do descompressor sejam gerados.

O modelo do descompressor implementado (descrito na Seção 4.4) é formado por vários componentes desenvolvidos em VHDL[2] que são configurados através de um pacote gerado pelo compressor de acordo com os parâmetros de compressão. A síntese final deve incluir

[2]VHDL é uma linguagem de descrição de hardware.

o descompressor entre a *cache* e a memória.

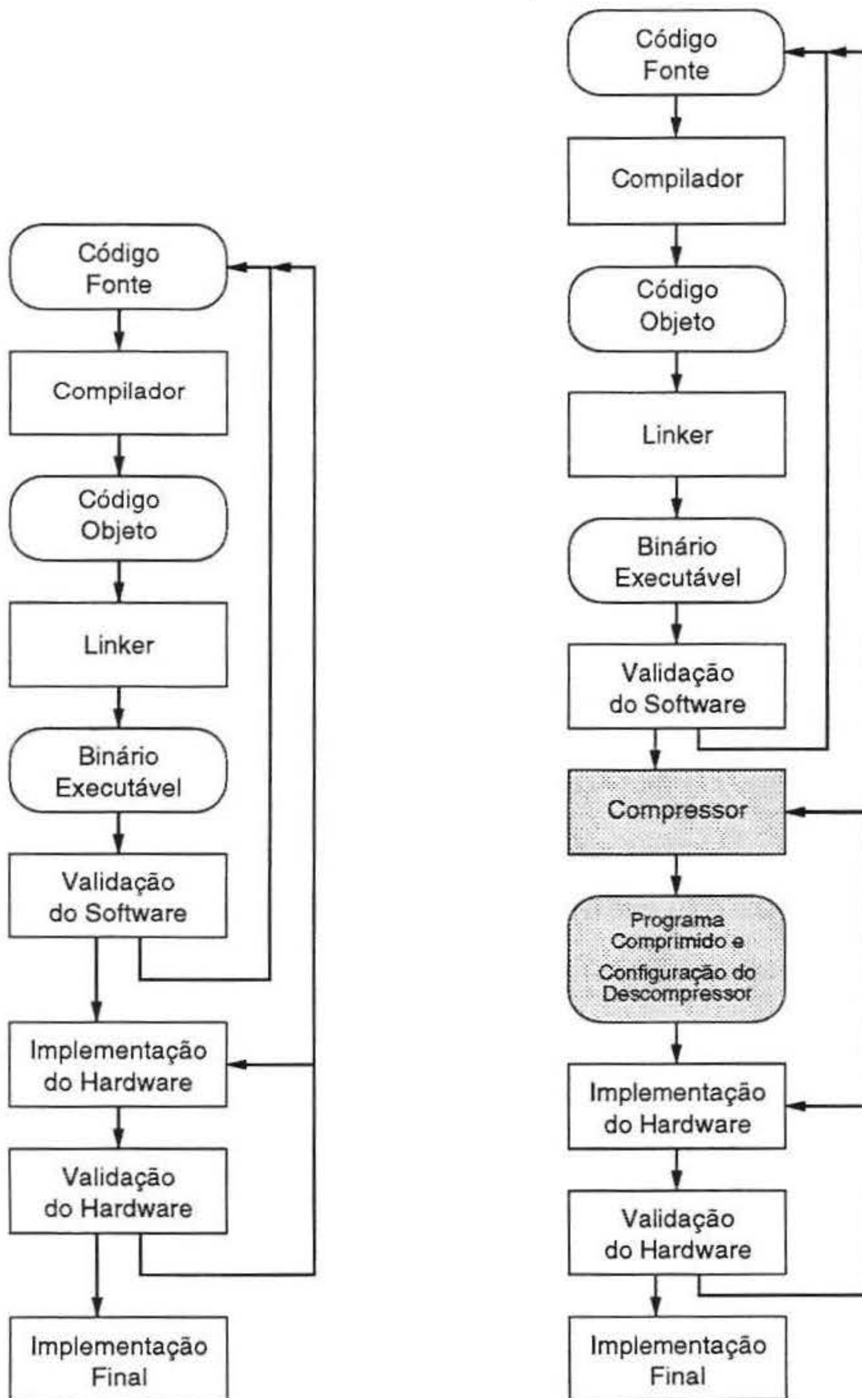
4.1 Análise do Método

No trabalho realizado para o processador MIPS foram usados os mesmos programas e parâmetros de compilação utilizados para o TBC e PBC, conforme a Tabela 3.1. Foi feito o levantamento do número de instruções únicas para cada um dos programas utilizados e o resultado está listado na Tabela 4.1. Na média, os programas possuem apenas 18,3% de instruções únicas, sendo o resto composto por repetições destas. O fato deste número ser menor que o de árvores únicas (24%), indica a existência de repetição das instruções dentro das árvores ou entre árvores distintas, o que levou ao desenvolvimento de um novo método que explora esta característica dos programas. As repetições de instruções devem-se a inúmeros fatores, entre eles, os que mais se destacam são as convenções de chamada de procedimentos, passagem de parâmetros, manipulação da pilha e retorno de funções, e as instruções de `NOP`. Por mais que as otimizações tentem evitar a utilização de `NOPs`, elas não conseguem fazer com que esta deixe de ser uma das instruções que mais ocorrem no código de um processador RISC[3].

Programa	Tamanho (em instruções)	Instruções Únicas (%)
compress	2152	846 (39,3)
gcc	363560	38600 (10,6)
go	73908	10267 (13,9)
ijpeg	47988	10536 (22,0)
li	18448	2959 (16,0)
perl	69536	11178 (16,1)
vortex	151348	15200 (10,0)

Tabela 4.1: Número de instruções únicas para a arquitetura MIPS.

[3]Os *delay slots* que não podem ser preenchidos com outras instruções são preenchidos com `NOPs`.



(a) Sem Compressão de Código

(b) Com Compressão de Código

Figura 4.2: Ciclo de projeto.

O gráfico da Figura 4.3 mostra a distribuição das instruções para os programas utilizados. Dele temos que 20% das instruções cobrem mais de 80% do código e que 60% das instruções são responsáveis por apenas 10% do código. Novamente, como nos outros métodos, o programa `compress` não segue exatamente este comportamento.

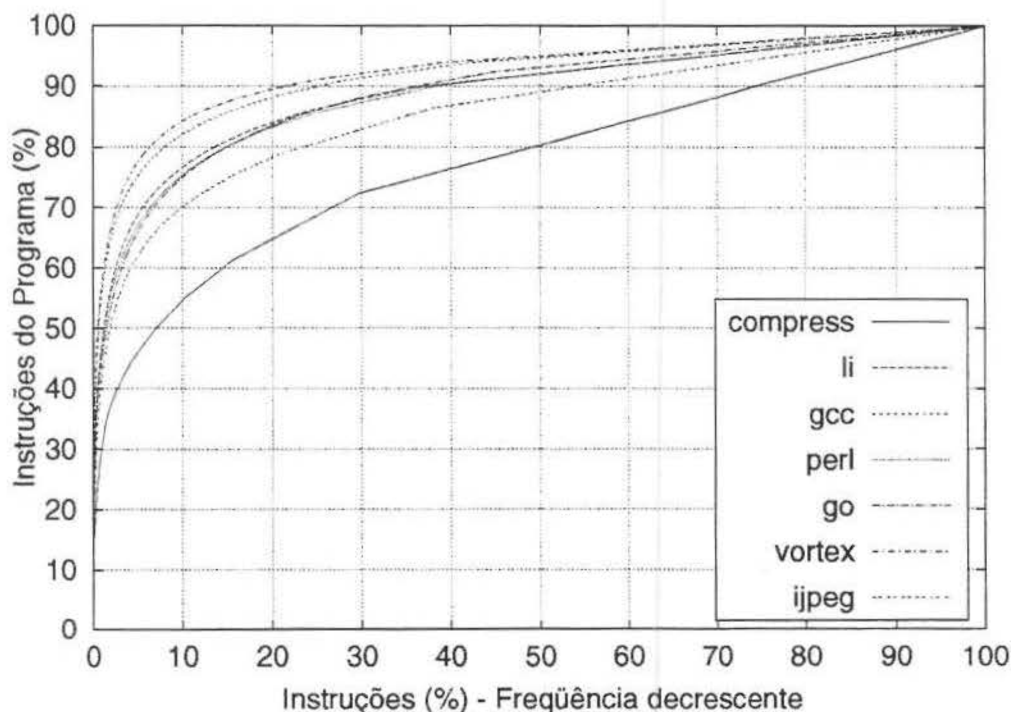


Figura 4.3: Porcentagem do programa coberta por instruções distintas.

O algoritmo de compressão adotado consiste dos seguintes passos (Figura 4.4):

Extração das Instruções Únicas: O arquivo binário original no formato ELF [2, 3, 4] é lido e analisado. Um arquivo ELF é composto por várias partes, chamadas seções. Cada uma dessas seções pode conter código, dados, informações de depuração ou informações para controle da carga do programa na memória. Todas as seções de código são analisadas e a seção correspondente ao bloco destinado a compressão[4]

[4]Uma região de memória é reservada para o código comprimido. Mais detalhes serão fornecidos na Seção 4.4 que trata da construção do descompressor.

UNICAMP
BIBLIOTECA CENTRAL

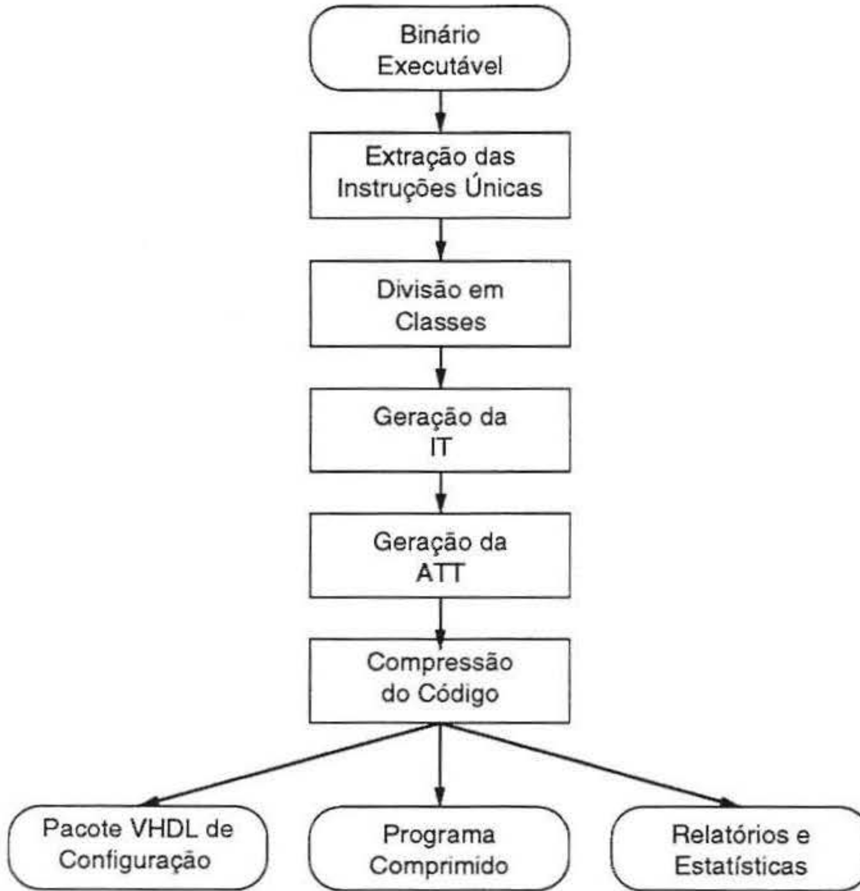


Figura 4.4: Passos executados pelo compressor de código.

tem suas instruções decodificadas, agrupadas e ordenadas de forma decrescente de ocorrências.

Divisão em Classes: As instruções são divididas em classes de acordo com o seu número de ocorrências e a quantidade de elementos por classe. São testadas todas as combinações possíveis entre 2 e 8 classes e também todas as combinações de tamanhos para as classes. O gráfico da Figura 4.5 mostra as razões de compressão obtidas para diferentes números de classes. Conforme a Figura 4.5, a melhor combinação ocorre em torno de 4 classes, sendo superado apenas em alguns casos pelo uso de 5 classes. Como a diferença entre o uso de 4 e 5 classes é pequena (0,1%), para os

resultados finais foram consideradas sempre 4 classes. Uma forma simplificada da divisão em classes foi utilizada pelo TBC. No entanto, inicialmente foram utilizadas apenas uma (sem divisão de classes) ou duas combinações de classes diferentes. A divisão em um número maior de classes e o teste para a melhor combinação entre número de classes e o tamanho de cada uma das classes foi implementado inicialmente para o IBC e somente após os resultados terem se mostrado satisfatórios essa divisão foi implementada para o TBC também.

Geração da Tabela de Instruções: Neste etapa é gerada uma tabela chamada Tabela de Instruções onde cada linha corresponde ao código objeto de uma instrução. Esta tabela será consultada durante a descompressão. A IT[5] possui 32 bits de largura e armazena em cada linha uma instrução descomprimida. A tabela de instruções é similar ao dicionário de Árvores do método TBC e também ao dicionário de padrões de árvores do PBC só que no caso da tabela de instruções, existe apenas um campo e a relação é sempre de uma leitura para cada *codeword*.

Geração da Tabela de Conversão de Endereços: Uma tabela que será utilizada para converter os endereços descomprimidos em endereços comprimidos é criada para permitir a localização das instruções comprimidas na memória pelo descompressor. A ATT[6] será descrita em detalhes na Seção 4.2.

Compressão do Código: A cada instrução é atribuído um prefixo indicando sua classe e uma *codeword* que é um índice à tabela de instruções. O programa original é reescrito substituindo cada uma das instruções originais pelo par [prefixo, *codeword*]. Variações na representação das instruções e a conseqüente redução no número de instruções da IT serão mostradas na Seção 4.3.

[5]Do Inglês *Instruction Table*.

[6]Do inglês *Address Translation Table*.

Como resultados da aplicação do compressor, são gerados um pacote VHDL com os parâmetros de configuração do descompressor, o programa comprimido e relatórios e estatísticas de compressão do código. Os parâmetros de configuração do descompressor são:

- O número de classes utilizadas;
- A quantidade de instruções em cada uma das classes;
- A região de memória onde se localiza o código comprimido;
- O tamanho de cada campo da máscara da Tabela de Conversão de Endereços ATT;
- A Tabela de Conversão de Endereços (ATT);
- A Tabela de Instruções.

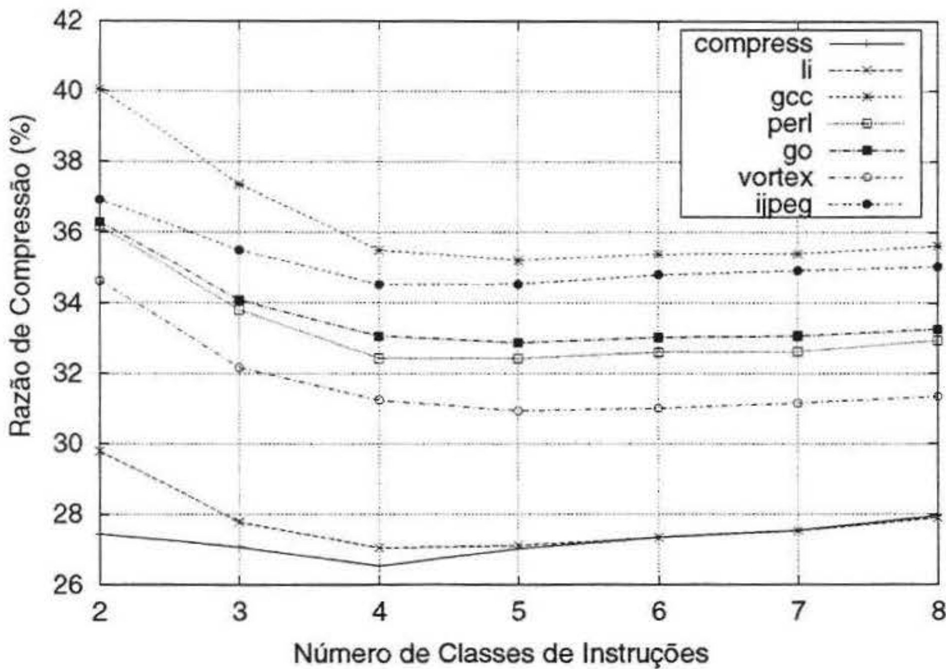


Figura 4.5: Razão de compressão para diferentes números de classes.

Uma versão simplificada do descompressor é mostrada na Figura 4.6 apenas para descrição do método, a versão detalhada está descrita na Seção 4.4. As *codewords* são extraídas das palavras lidas da memória e armazenadas no registrador `Ic`. O módulo `IGEN` é responsável por transformar `Ic` no endereço `it_addr`. A conversão entre `Ic` e `it_addr` é realizada de forma muito simples, através de uma soma do valor da *codeword* com um endereço base de cada classe na tabela. O valor de `it_addr` é utilizado para endereçar o dicionário de instruções (`ID`, que também é chamado de tabela de instruções, ou `IT`). A instrução endereçada por `it_addr` é então enviada para o processador.

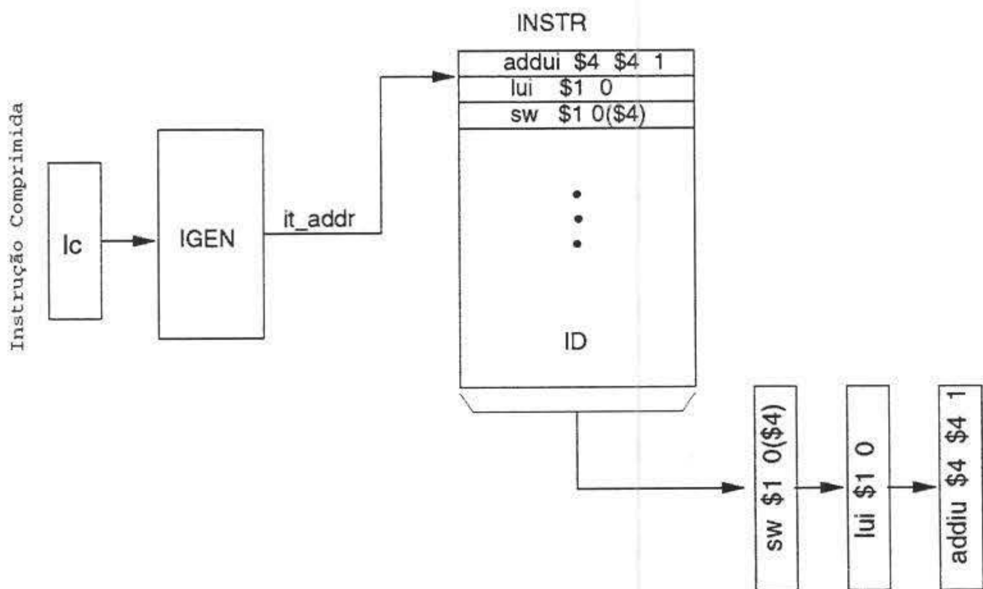


Figura 4.6: Versão simplificada do descompressor para IBC.

A razão de compressão média obtida pelo IBC é de 31,5% para os programas do *benchmark* SPEC CINT'95. O custo médio da tabela de instruções é de 18,3% do tamanho do programa original em bits. A tabela de conversão de endereços tem um custo adicional de 3,8% e, como essas duas tabelas ocupam a maior área do descompressor, o custo total médio fica em 22,1%. A Tabela 4.2 e a Figura 4.7 mostram os resultados finais obtidos para a arquitetura MIPS. A razão de compressão média final é de 53,6%, que é melhor

que as do TBC e PBC. A Tabela de Instruções tem uma influência razoável na razão de compressão final, na Seção 4.3 é descrito um método para reduzir esta influência.

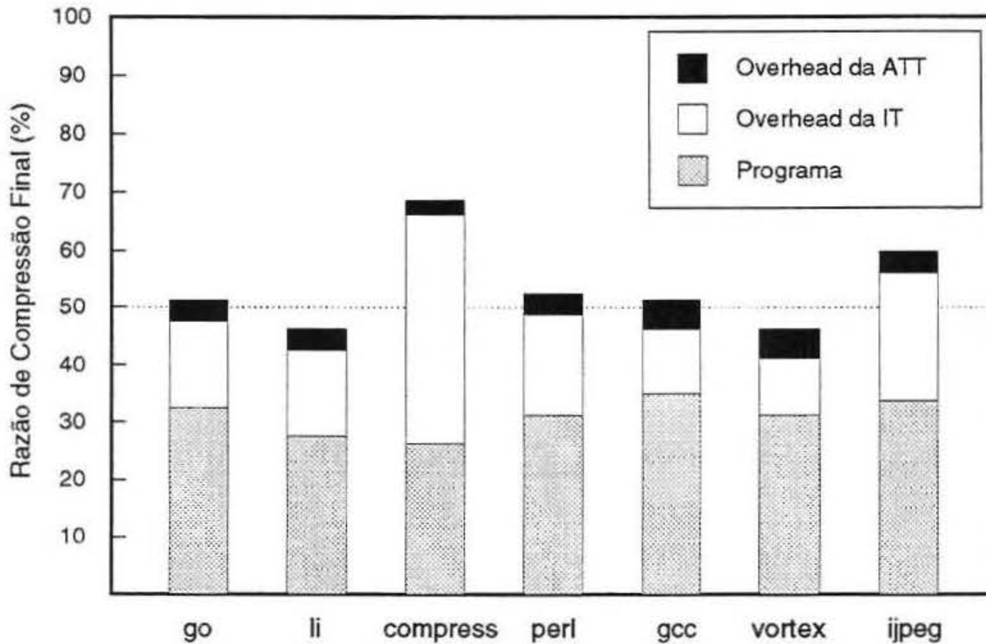


Figura 4.7: Razão de compressão final para IBC aplicado ao MIPS.

4.2 Tabela de Conversão de Endereços

O problema mais crítico encontrado durante a implementação da compressão de código de forma transparente ao processador (como no IBC) foi fazer com que o processador continue sendo capaz de ter as mesmas instruções nos mesmos endereços que antes da compressão. As soluções utilizadas por métodos anteriores para este problema podem ser divididas em duas categorias:

Nova interpretação para os endereços: Nesse método, os endereços que eram anteriormente codificados como deslocamentos em bytes em relação ao início da memória passam a ter outro significado. O trabalho de Breternitz [15] utiliza uma parte da palavra para indicar o endereço em bytes e outra parte para indicar o deslo-

Programa	Classes				Razão de Compressão			
	I	II	III	IV	Código	IT	ATT	Total
compress	1	5	8	10	26.53	39.3	2.8	65,83%
gcc	2	8	12	16	35.49	10.6	4.3	50,39%
go	3	8	11	14	33.06	13.9	3.9	50,86%
ijpeg	3	8	11	14	34.53	22.0	3.7	60,23%
li	2	6	9	12	27.05	16.0	3.5	46,55%
perl	2	7	10	14	32.43	16.1	3.9	51,43%
vortex	1	6	10	14	31.24	10.0	4.5	45,74%

Tabela 4.2: Resultados obtidos para a arquitetura MIPS.

camento da instrução dentro da *cache-line*. Esse trabalho requer uma mudança no compilador ou um processamento posterior à geração de código que seja capaz de encontrar todas as referências a instruções no código comprimido (saltos, *jump-tables*, endereçamento relativo, etc.) e atualizá-las, o que não é uma tarefa muito simples.

Conversão dos Endereços: Nessa estratégia, o código não sofre nenhuma modificação, e o processador envia exatamente os mesmos sinais ao barramento de endereço que seriam enviados sem descompressão. A conversão dos endereços descomprimidos em endereços comprimidos fica a cargo do descompressor. Esse modelo foi o adotado no método IBC.

Para realizar essa tarefa, foi desenvolvida a *Address Translation Table* (ATT) que é utilizada para a conversão dos endereços. Um diagrama da ATT é mostrado na Figura 4.8. Uma consulta à ATT é efetuada toda vez que ocorre um *cache-miss* e o endereço solicitado não está no intervalo que o descompressor está processando. Ao receber uma solicitação de uma instrução do processador, os seguintes passos são utilizados na conversão do endereço:

1. Uma máscara de bits é aplicada ao endereço fornecido pelo processador. Essa máscara é criada de forma permitir apenas a passagem de n_i bits significativos

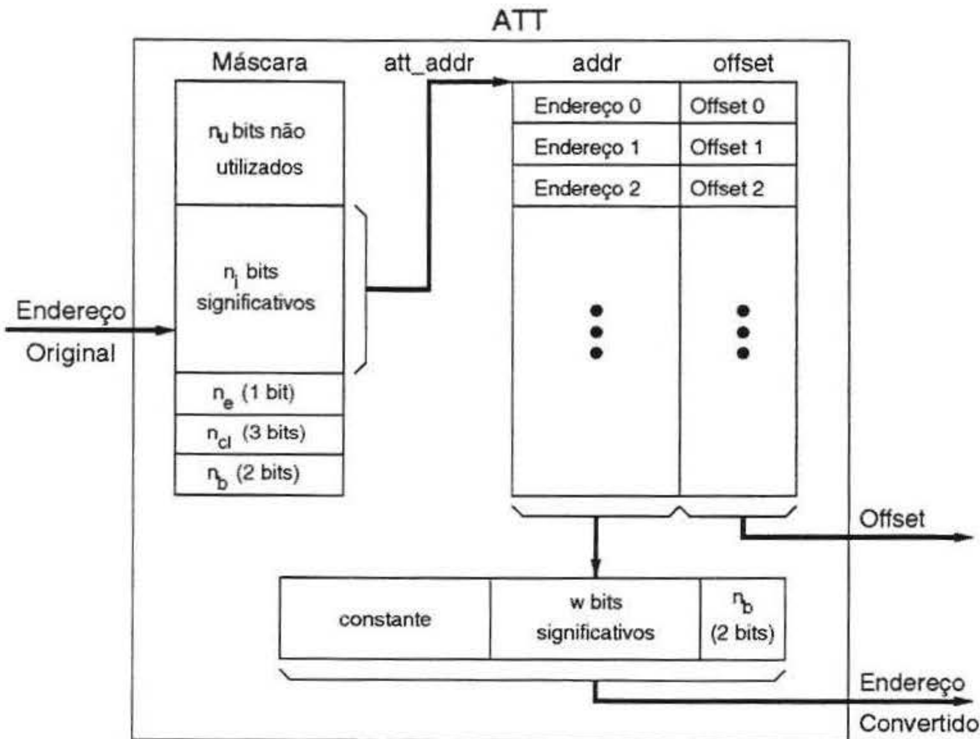


Figura 4.8: Tabela de Conversão de Endereços (ATT).

que são selecionados através da remoção dos seguintes bits do endereço original:

- n_b bits relativos ao endereçamento de bytes dentro das palavras. No caso do MIPS, esses bits não são utilizados quando o processador endereça instruções na memória pois no programa descomprimido o código fica alinhado em palavras. No caso específico da Figura 4.8, $n_b = 2$.

- n_{cl} bits relativo ao número de bits utilizados para endereçar palavras dentro das *cache-lines*. Uma vez que a política de preenchimento da *cache* fará com que toda a *cache-line* seja lida da memória no caso de um *cache miss*, só é necessária a conversão do primeiro endereço de cada *cache-line*, fazendo com que os outros só sejam descobertos através da decodificação seqüencial da *cache-line* completa. No caso da Figura 4.8, as *cache-lines* utilizadas possuem 8

palavras, o que faz com que $n_{cl} = 3$.

- n_e bits extras utilizados para reduzir o tamanho da ATT. Como cada bit incluído no endereçamento da tabela potencialmente dobra o número de linhas da ATT, um número n_e de bits extras podem ser adicionados à máscara para diminuir o tamanho da tabela. No caso da Figura 4.8, $n_e = 1$, o que significa que apenas uma em cada duas *cache-lines* terá seu endereço inicial mapeado na ATT, essa *cache-line* é chamada *cache-line* base. A decodificação do endereço das *cache-lines* que não são *cache-lines* base é feito através da decodificação seqüencial dos endereços desde a última *cache-line* base. A escolha do valor do n_e é uma relação de compromisso entre desempenho e a quantidade de memória utilizada pela ATT, que influi diretamente na razão de compressão final.

- n_u bits superiores não utilizados para endereçar o bloco de memória comprimido. Esses bits são utilizados para detectar se a leitura solicitada pelo processador está ou não numa região comprimida de memória, mas não têm nenhuma utilidade no endereçamento da ATT.

Após a remoção desses bits, sobram apenas n_i bits, que formam o campo `att_addr`. O valor de n_i pode ser calculado como:

$$n_i = \lceil \log_2 (tamanho\ do\ programa\ em\ bytes) \rceil - n_b - n_{cl} - n_e \qquad (4.1)$$

2. O valor `att_addr` é utilizado para endereçar a ATT, que possui dois campos por linha, `addr` de w bits e `offset` de 5 bits. O campo `addr` contém parte do endereço da palavra comprimida na memória que contém a instrução desejada e o `offset` contém

o deslocamento em bits da instrução comprimida dentro da palavra da memória. O valor de w pode ser calculado como:

$$w = \lceil \log_2 (tamanho\ do\ programa\ comprimido\ em\ bytes) \rceil - 2 \qquad (4.2)$$

Deve ser destacado o fato que apenas os bits necessários para endereçar as instruções do programa comprimido são armazenados na ATT, os bits que possuem sempre o mesmo valor (n_b bits inferiores e os n_u bits superiores que indicam a região da memória onde está o código comprimido) são gerados automaticamente pelo descompressor.

3. O valor `addr` é utilizado para formar o endereço de 32 bits da palavra na memória. Para isso, são adicionados zeros nos dois bits menos significativos do endereço e os bits mais significativos são preenchidos com uma máscara que define os bits da região de memória onde o código comprimido está armazenado.

4. O endereço convertido é utilizado no acesso à memória e a palavra resultante da leitura passa por um deslocamento de `offset` bits para ser processada pelo descompressor.

A Figura 4.9 mostra um exemplo do uso da ATT para conversão do endereço 40000104_h. Considerando exatamente os mesmos parâmetros utilizados na Figura 4.8, são descartados os 6 bits menos significativos do endereço original, e extraídos n_i bits significativos como valor de `att_addr` (nesse caso, 004_h). O valor de `att_addr` endereça a quinta linha da tabela, que contém os valores 0020_h para `addr` e 7_h para `offset`. O valor do campo `offset` é fornecido diretamente como uma das saídas da ATT. O campo `addr` é agrupado aos bits indicadores da região de memória (4000_h) e com os dois bits de endereçamento de palavra (que sempre devem ser zeros) para formar o endereço convertido.

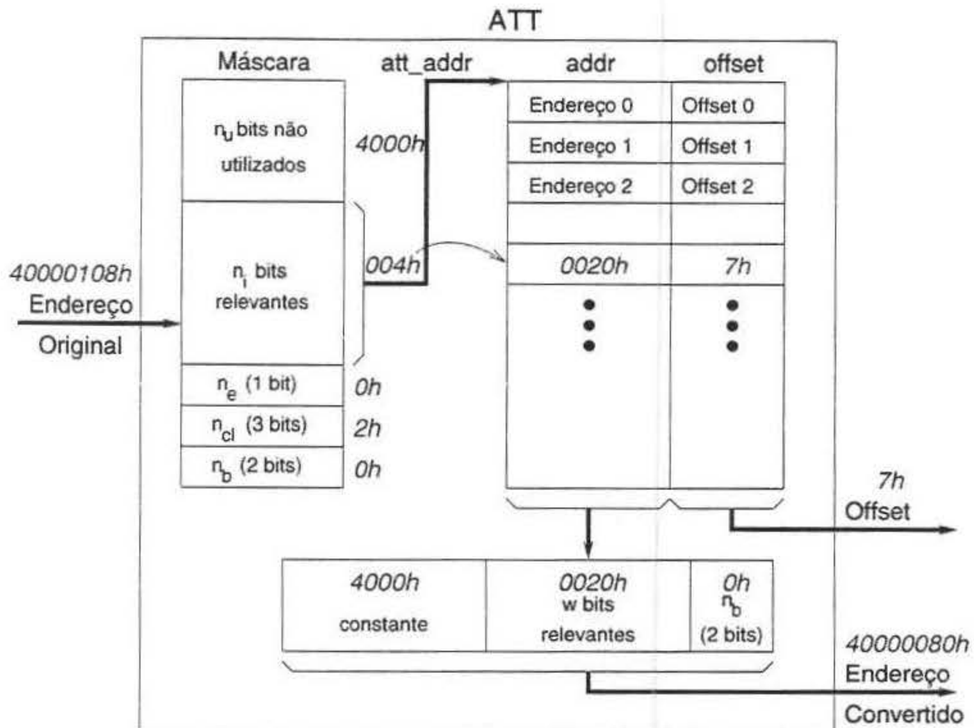


Figura 4.9: Exemplo do uso da ATT.

Dos trabalhos já realizados, o método que mais se assemelha ao aqui desenvolvido é a *Line Address Table* (LAT) desenvolvida por Wolfe [68]. Os dois métodos utilizam uma tabela para decodificar o endereço, mas a forma como as tabelas são organizadas e o local onde são armazenadas é diferente. A LAT fica armazenada na memória, possuindo um custo de acesso igual ao da memória onde fica o programa comprimido. A ATT fica armazenada em uma memória interna ao descompressor, o que permite um acesso mais rápido. O custo desse acesso mais rápido é a implementação de uma maior quantidade de memória interna ao descompressor. Para minimizar esse tempo de acesso à LAT, Wolfe propôs a CLB, que funciona como a TLB do processador, só que para linhas da LAT. O problema da CLB no trabalho de Wolfe é que ela deve ser implementada próxima à TLB para acelerar o acesso e isso implica em alteração no processador.

Outra diferença estrutural entre a ATT e a LAT é a largura das linhas na memória. A LAT tem linhas fixas de 64 bits para cada 8 *cache-lines* do processador. Essa linha é dividida em dois blocos: o primeiro deles, de 24 bits, contém o endereço base da primeira dessas 8 *cache-lines*; o outro bloco é composto por 8 conjuntos de 5 bits indicando o tamanho de cada um dos blocos endereçados. Essa abordagem permite que uma linha da LAT seja capaz de referenciar 8 *cache-lines*, mas torna necessário efetuar até 7 somas para chegar ao endereço desejado, o que torna ainda mais lento o seu acesso. Outro fato a ser destacado é que a LAT armazena 8 tamanhos, enquanto seriam necessários apenas 7 deslocamentos para calcular os endereços de cada uma das *cache-lines* (o oitavo tamanho pode ser utilizado para calcular o início da nona *cache-line*, que é a primeira endereçada pela próxima linha da LAT). O tamanho da linha da ATT não é fixo, ele é dividido em duas partes, `addr` de w bits e `offset` de 5 bits. Como ela é implementada diretamente em hardware, a largura não precisa ser um múltiplo da largura da memória, fazendo com que os bits de memória disponíveis, e conseqüentemente a área do circuito, sejam melhor aproveitados. Uma linha da ATT endereça apenas uma *cache-line* mas, dependendo do valor de n_e (bits extras utilizados na codificação) essa linha será utilizada para a localização das próximas $2^{n_e} - 1$ *cache-lines*.

4.3 Tabela de Instruções

Na tabela de instruções são armazenadas uma cópia de cada uma das instruções que ocorrem no programa. Ela tem 32 bits de largura e não possui nenhum outro bit adicional, nem mesmo o bit `END` do PBC (Seção 3.2) e TBC (Seção 3.1), pois a unidade de armazenamento da IT é uma única instrução.

A IT é implementada como uma memória interna ao descompressor e como tal ocupa uma proporção razoável em relação ao programa comprimido. Assim sendo, foi projetada

uma forma de minimizar esse espaço gasto pela IT. A solução foi baseada nas instruções que ocorrem apenas uma vez no programa. Essas instruções, além de gastar uma linha da IT para serem armazenadas, ainda precisam ser armazenadas comprimidas no programa, o que faz com que elas ocupem mais espaço quando é utilizada compressão do que quando o código fica descomprimido na memória. A solução encontrada foi fazer com que as instruções que ocorrem apenas uma vez sejam representadas diretamente no código comprimido, reservando para isso uma das classes só para elas. Na Figura 4.10(a), todas as quatro classes de instruções aceitam apenas instruções comprimidas, já na Figura 4.10(b), as instruções da classe IV são colocadas diretamente no código comprimido, gastando para isso 34 bits, mas economizando uma linha da IT que seria utilizada para armazená-la. Esse método foi estendido no compressor de forma que é possível especificar apenas o tamanho máximo da tabela de instruções e as instruções menos freqüentes que não puderem ser representadas na tabela serão codificadas diretamente no programa comprimido.

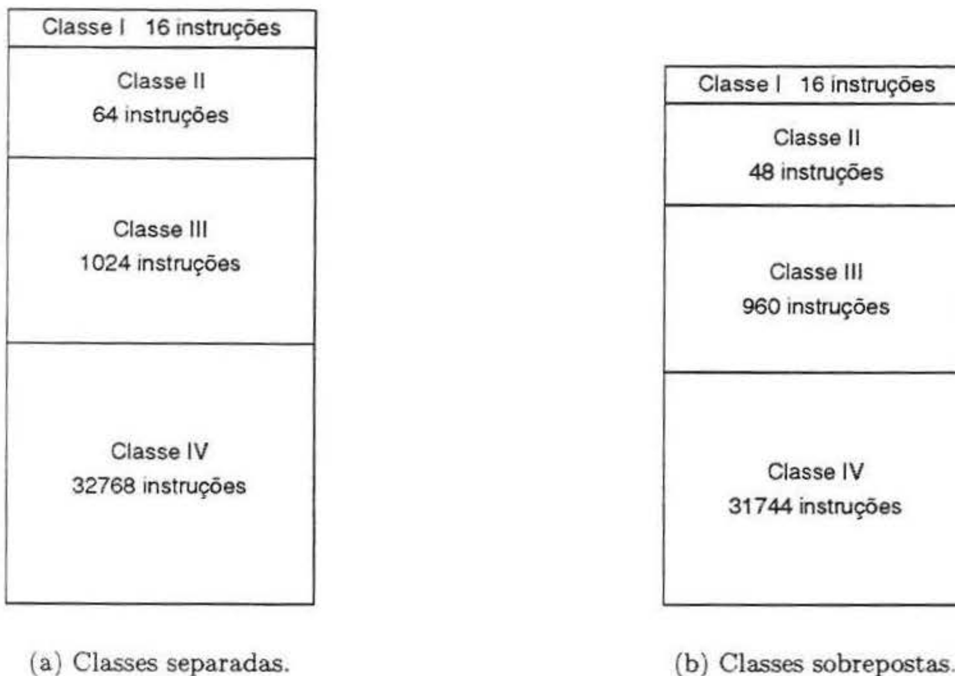
(a) Instruções codificadas.

(b) Prefixo para instrução sem codificação.

Figura 4.10: Formas de codificação das instruções

Duas formas de decodificação da tabela de instrução foram estudadas e suas implementações estão mostradas na Figura 4.11 considerando a divisão de classes da Figura 4.10(a). A primeira delas, mostrada na Figura 4.11(a), divide a IT em quatro par-

tes, de acordo com a capacidade de endereçamento de cada uma das classes. Na primeira parte, correspondente à Classe I, ficam 16 instruções, na segunda parte ficam 64 instruções e assim sucessivamente. Como a IT é um módulo de memória, para implementar esse método e calcular os endereços de cada *codeword*, são implementados 3 registradores base, um para cada uma das classes II, III e IV. O valor do registrador base é então somado ao valor da *codeword* para gerar o endereço da instrução comprimida dentro da IT. Esse método aproveita melhor o espaço de endereçamento das *codewords* ao custo de um somador e registradores de endereço base.



(a) Classes separadas. (b) Classes sobrepostas.

Figura 4.11: Possíveis divisões das classes na Tabela de Instruções

O outro método desenvolvido não necessita do somador nem dos registradores de endereço base. Para isso, o espaço de endereçamento das classes é sobreposto, todos começando no endereço zero, fazendo com que exista mais de uma forma de endereçar uma mesma instrução, o que na prática significa reduzir a capacidade de endereçamento de

cada uma das classes conforme mostrado na Figura 4.11(b). Como exemplo, a Classe II da figura, embora possua 6 bits para codificação, só é utilizada para codificar 48 instruções, visto que o compressor, ao atribuir os códigos comprimidos para cada uma das instruções, o faz sempre escolhendo a menor representação possível. A diferença entre esse método e o anterior não pode ser medida simplesmente através do espaço de endereçamento perdido pois, uma vez que o compressor testa todas as combinações de tamanhos de classes, normalmente os tamanhos das classes escolhidos por um dos métodos é diferente dos tamanhos escolhidos pelo outro. Caso seja escolhida a opção por codificar instruções diretamente na última classe, esse método permite que os bits de endereçamento da IT sejam utilizados completamente[7].

4.4 Descompressor IBC *Pipelined* para MIPS R4000

A primeira versão do descompressor (Figura 4.12) foi implementada para o processador MIPS R4000 e utilizou um *pipeline* de quatro estágios, uma unidade de controle e um *buffer* como interface ao barramento. Cada um desses estágios é responsável por parte do trabalho de descompressão e como estão implementados em um *pipeline*, várias dessas partes ocorrem em paralelo, acelerando o processo de descompressão. Os estágios e seus componentes são descritos a seguir:

***Address Decoder*:** Esse estágio é responsável por decodificar, converter e armazenar os endereços das leituras feitas pelo processador. Ele é composto por três módulos:

***Requested Address Register* (RAR):** É um registrador que armazena o endereço solicitado para comparação com o endereço processado por cada um dos estágios.

[7] Note que no caso do primeiro método, são necessários 16 bits para endereçar a IT, enquanto a maior classe utiliza apenas 15 bits.

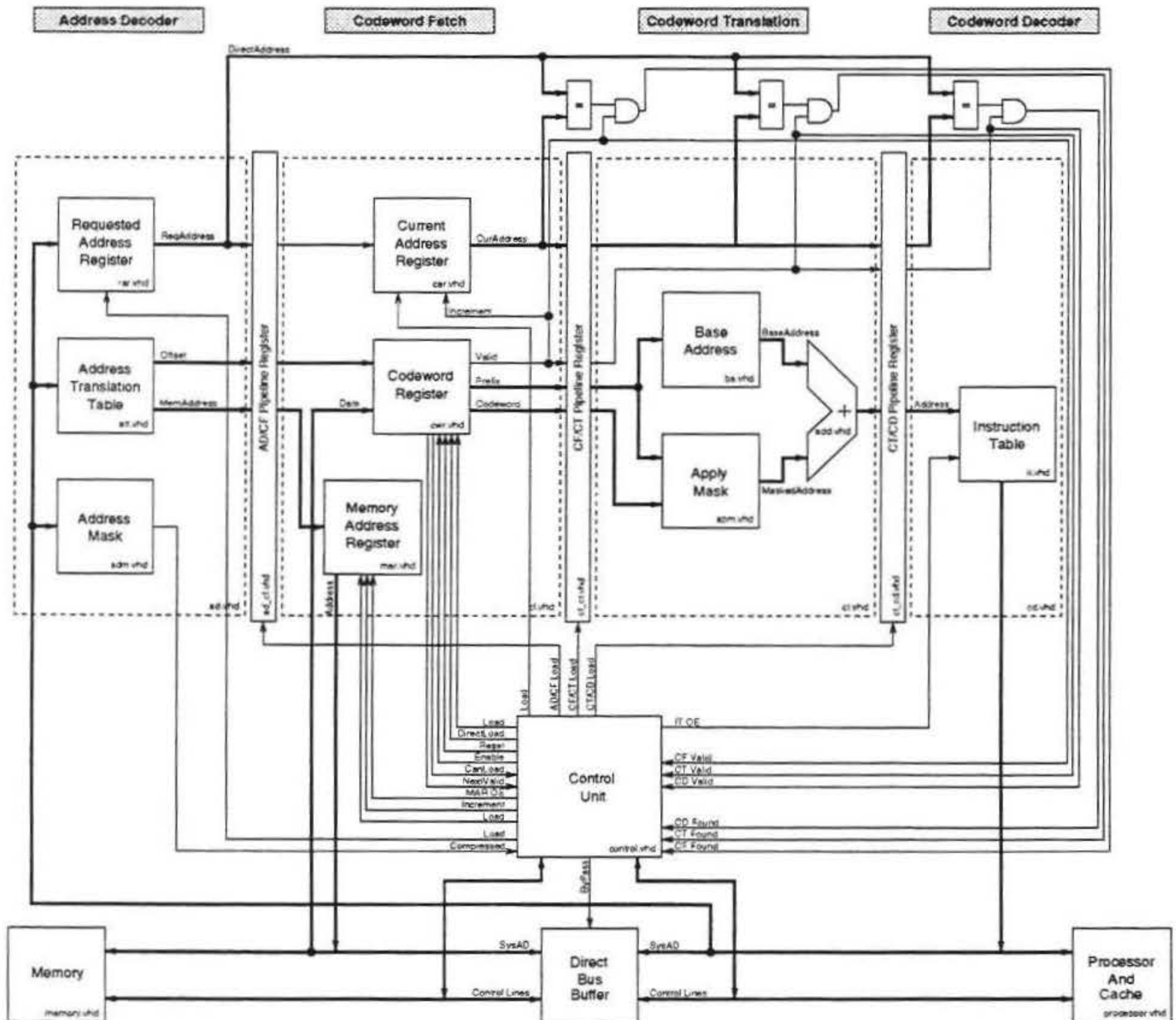


Figura 4.12: Descompressor *pipelined* para MIPS usando IBC (4 estágios).

Address Mask (`ADM`): É responsável por detectar se a instrução solicitada pelo processador é uma instrução de leitura em área comprimida de memória. Para isso, é feita uma comparação entre os bits superiores do endereço com um valor padrão que é gerado pelo compressor. O valor e o número de bits que serão comparados dependem do programa original e da região de memória que será utilizada para o programa comprimido. Todo o código comprimido deve ficar em uma mesma região de memória, chamada região comprimida e todos os endereços dessa região possuem os mesmos bits mais significativos, facilitando assim a detecção de um endereço dentro da mesma.

Address Translation Table (`ATT`): Realiza a conversão de endereços descomprimidos em endereços comprimidos conforme descrito na Seção 4.2.

Codeword Fetch: Esse estágio é responsável: pela busca das palavras com instruções comprimidas da memória; pelo agrupamento das palavras lidas e separação das *codewords*. Ele é composto por três módulos:

Current Address Register (`CAR`): Guarda o endereço da instrução atualmente processada. O `CAR` só é carregado quando há transferência de fluxo (salto) no programa. Durante a decodificação seqüencial, ele é apenas incrementado para indicar o endereço da próxima instrução. Quando há uma transferência de fluxo no programa, um novo endereço precisa ser carregado e a unidade de controle ativa o sinal `Load`. O `CAR` carregará o endereço da *cache-line* base a ser processada[8]. A extração do endereço da *cache-line* base é feita através da transformação dos $n = n_b + n_d + n_e$ bits inferiores do `RAR` para zero.

[8] O *pipeline* só é capaz de descomprimir instruções a partir de uma *cache-line* base conforme já dito na Seção 4.2.

Memory Address Register **(`MAR`):** Armazena o endereço de memória onde deverá ser feita a próxima leitura. Sempre que uma palavra é lida da memória, esse endereço é incrementado para indicar a próxima palavra a ser lida. A carga do `MAR` é feita quanto há uma transferência no fluxo de dados e o valor carregado é o campo `addr` da ATT (mostrado como `MemAddr` no diagrama).

Codeword Register **(`CWR`):** É responsável por concatenar as palavras lidas da memória e extrair os prefixos e as *codewords* resultantes. Esse é o módulo mais complexo dessa versão do descompressor, um diagrama dos componentes internos é mostrado na Figura 4.13. O `CWR` possui uma pequena unidade de controle que é responsável por gerenciar os componentes internos e fornecer os sinais de estado para a unidade de controle do descompressor. Os três sinais de estado são: `CanLoad`, que indica que há espaço dentro do `CWR` para mais uma palavra da memória e portanto pode ser realizado um *pre-fetch* da memória; `Valid` que indica que os sinais de saída `Codeword` e `Prefix` possuem valores válidos e podem ser processados pelo descompressor; e `NextValid` que indica que o `CWR` será capaz de fornecer um sinal válido no próximo ciclo. O `CWR` funciona em dois modos distintos e exclusivos: carga de dados e processamento. A carga de dados pode ser de dois tipos: carga normal, ativada pelo sinal `Load`, e carga direta que é ativada pelo sinal `DirectLoad`. Em qualquer um dos dois tipos de carga, os dados lidos da memória estão armazenados no registrador de entrada (`InputRegister`). No caso de uma carga direta, o `CWR` descarta todo o conteúdo que possui internamente e efetua a transferência dos dados do registrador de entrada para o registrador de saída (`BigRegister`) deslocando-o de `Offset` bits, seguindo o fluxo número 1 mostrado na Figura 4.13 (em azul). No caso de uma carga normal, o fluxo seguido é o de número 2 (em verde),

onde o CWR concatena o valor do registrador de saída ao valor de entrada. O fluxo número 3 (em vermelho) é o utilizado para o processamento normal do CWR, no qual o registrador de saída passa por um deslocamento para remover a instrução comprimida atual e dar lugar à instrução seguinte.

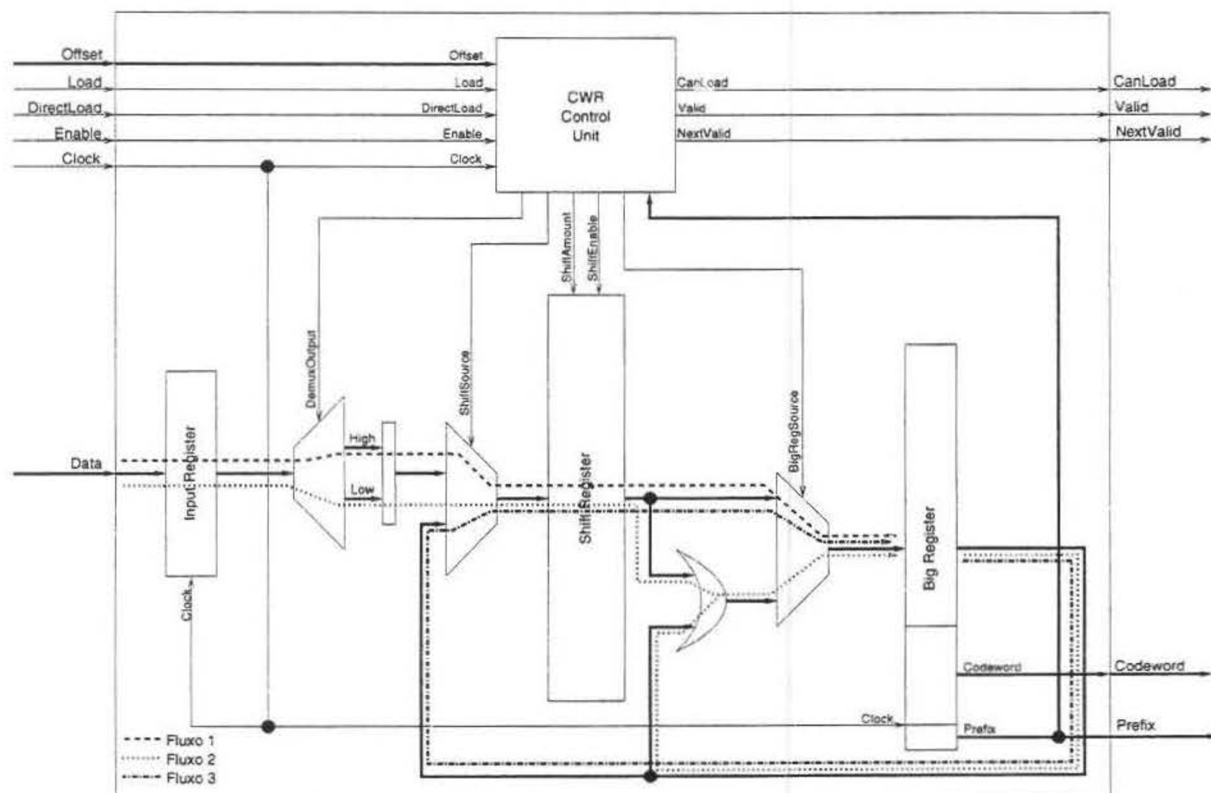


Figura 4.13: *Codeword Register.*

***Codeword Translation*:** Este estágio é responsável por transformar as *codewords* lidas da memória em endereços para a Tabela de Instruções. Ele pode ser dividido em três partes:

***Base Address* (BA):** É uma pequena memória que armazena os registradores base para cada uma das classes de instrução. Utiliza a entrada `Prefix` como endereço.

***Apply Mask* (`APM`):** Aplica uma máscara à *codeword* para eliminar os bits que não fazem parte dela. Como o barramento da *codeword* tem tamanho fixo, é preciso mascarar os bits superiores das *codewords* menores.

***Adder* (`ADD`):** Soma a *codeword* já com a máscara aplicada ao endereço base da classe. O resultado da soma é o endereço da instrução na IT.

***Codeword Decoder*:** Este é o estágio mais simples do *pipeline*. O único componente é a IT, que recebe um endereço e um sinal para habilitar a escrita da instrução no barramento do processador.

Unidade de Controle: É responsável por interpretar e gerar os sinais do barramento de controle do processador (`Control Lines`), ler o estado e enviar sinais de controle para os módulos do descompressor. Como os barramentos de endereços e de dados são multiplexados, a unidade de controle também deve gerenciar a temporização para a leitura e escrita correta neles.

***Direct Bus Buffer*:** Devido a forma como o barramento do MIPS é implementado, é necessário que um buffer seja colocado entre o processador e o controlador de memória para que as leituras de instruções comprimidas sejam tratadas apenas pelo descompressor. O barramento então é dividido em duas partes que podem ser acessadas separadamente, e que são conectadas quando ocorrem leituras de áreas não comprimidas ou escrita na memória.

A utilização de classes sobrepostas nesse *pipeline* pode ser implementada simplesmente zerando todos os endereços base. Mas tal implementação desperdiçaria recursos além de incluir um estágio desnecessário no *pipeline*. A versão da Figura 4.14 foi desenvolvida especificamente para esse caso, seu funcionamento é similar ao da Figura 4.12 e portanto não será detalhado novamente. A codificação de instruções diretamente em uma classe

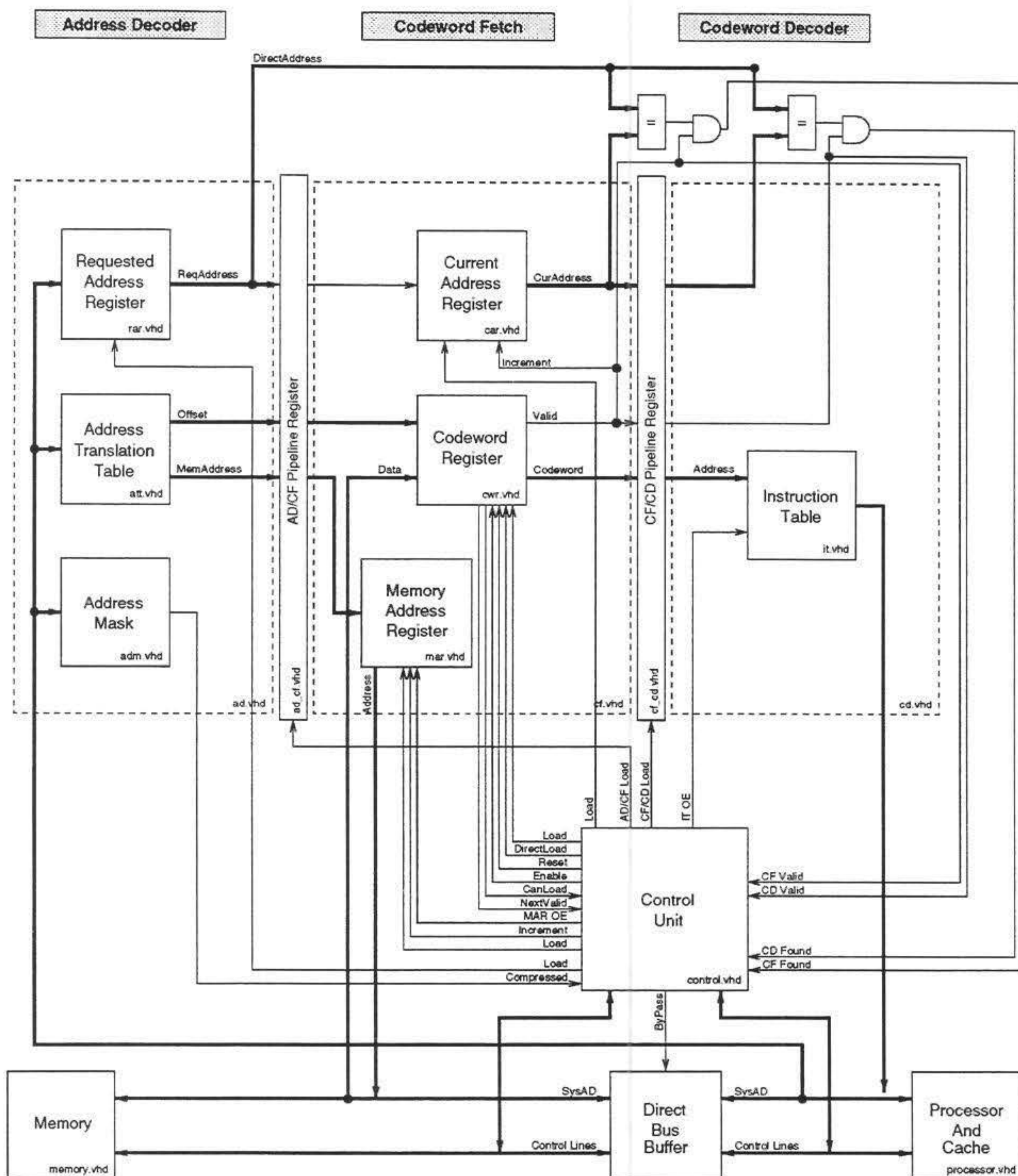


Figura 4.14: Descompressor *pipelined* para MIPS usando IBC (3 estágios).

requer uma alteração na lógica de controle e no tamanho dos registradores e barramentos internos, além de uma forma de transferir a *codeword* diretamente para o barramento sem passar pela tabela de instruções. Essa alteração só foi implementada na versão para o processador SPARC que será tratada no próximo capítulo.

4.5 Um Exemplo do Funcionamento do Descompressor

Nessa seção, será mostrado um exemplo de como o descompressor funciona através da descompressão de um bloco de dados da memória seguindo a implementação da Figura 4.14 que utiliza classes sobrepostas. O código original é mostrado na Tabela 4.3 e o código comprimido na Figura 4.15.

Endereço (hexadecimal)	Instrução	Instrução Comprimida		
		Codeword (binário)	*Codeword* (hexadecimal)	Prefixo (binário)
40000000_h	`addiu $2, $2, 60`	0000_b	0_h	00_b
40000004_h	`lw $4, 0($2)`	0011100101_b	$E5_h$	10_b
40000008_h	`slti $4, $4, 17`	100001010000011_b	4283_h	11_b
$4000000C_h$	`bne $4, 0, 16`	011111_b	$1F_h$	01_b

Tabela 4.3: Exemplo de código a ser comprimido.

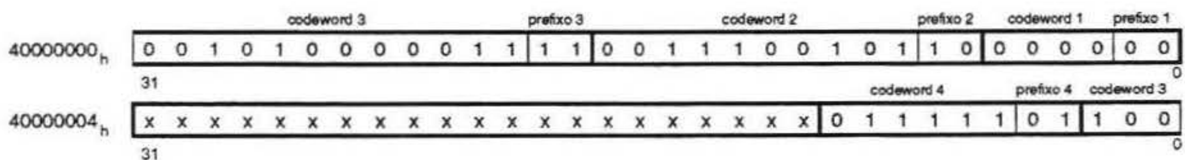


Figura 4.15: Trecho de código comprimido na memória.

Quando o fluxo de execução do processador for transferido para o endereço de memória 40000000_h (endereço de uma *cache-line base*), os seguintes passos serão executados para a leitura das 4 instruções comprimidas:

1. O processador ativa os sinais de controle solicitando uma leitura do endereço de memória 40000000_h;

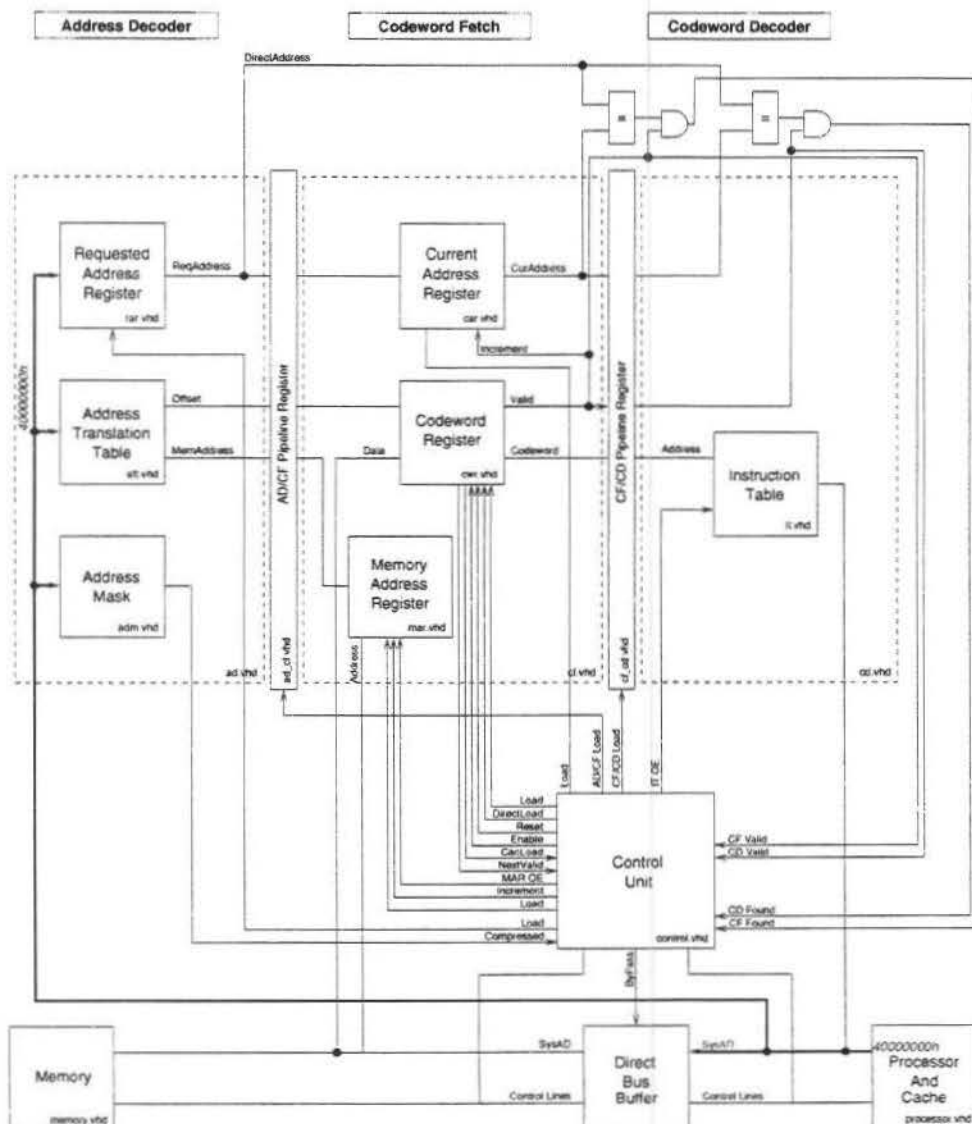


Figura 4.16: Exemplo do IBC para MIPS: Passo 1.

2. A unidade de controle observa o pedido de leitura e recebe o sinal `compressed` do `ADM`, que fica constantemente monitorando o barramento de endereços da CPU, indicando que se trata de um acesso à região de código comprimido; A unidade de controle então ativa o sinal de `Load` do `RAR` e também ativa o sinal de `Load` do registrador de *pipeline* `AD/CF`. Nenhum dos outros registradores de *pipeline* carregam dados neste ciclo;

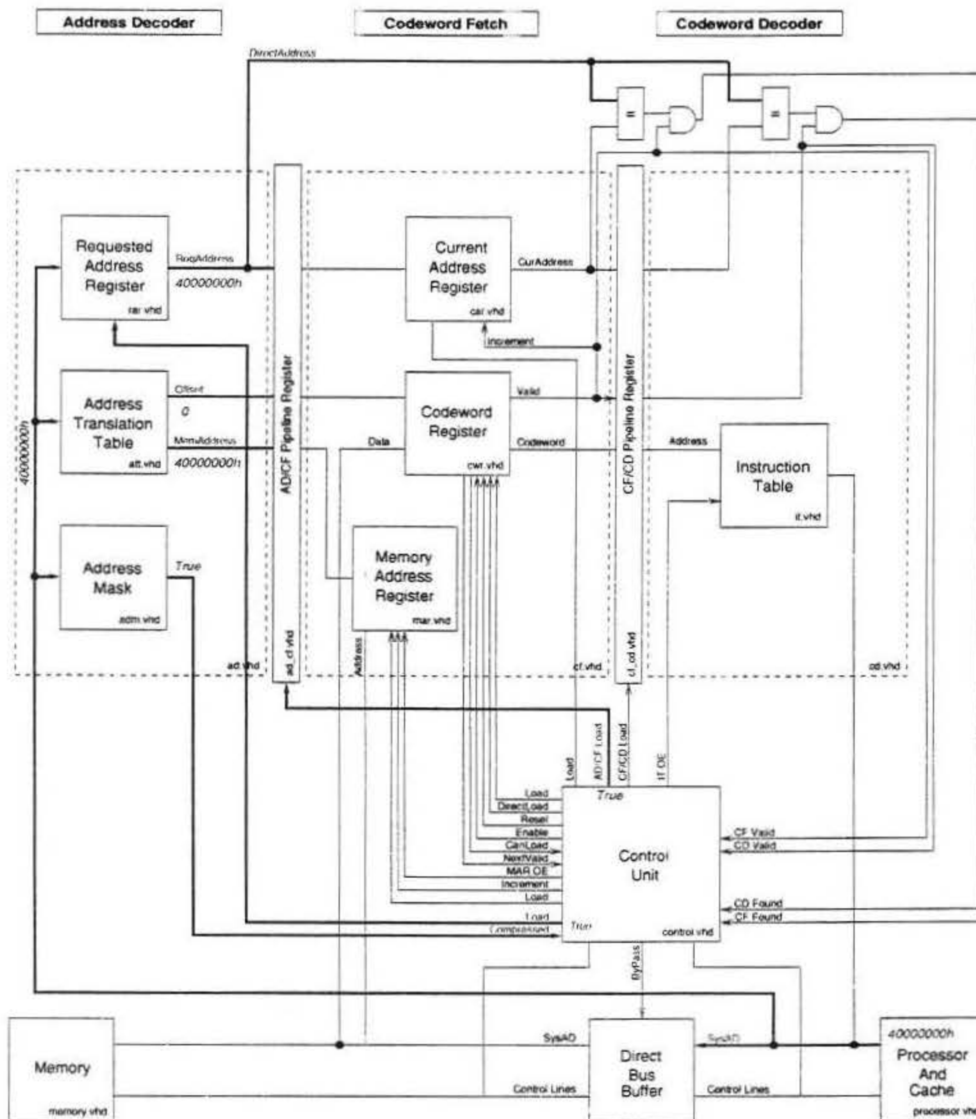


Figura 4.17: Exemplo do IBC para MIPS: Passo 2.

3. No próximo ciclo, os sinais de `Found` e `Valid` de cada estágio são observados para saber se a instrução solicitada já se encontra em decodificação. Como a instrução não está disponível, os sinais `Load` do `CAR` e do `MAR` são ativados, e tem-se início um ciclo de leitura da memória, com o `MAR` fornecendo o endereço de busca na memória e o resultado da leitura sendo transferido para o `CWR` através de uma leitura direta. A leitura da memória pode demandar vários ciclos;

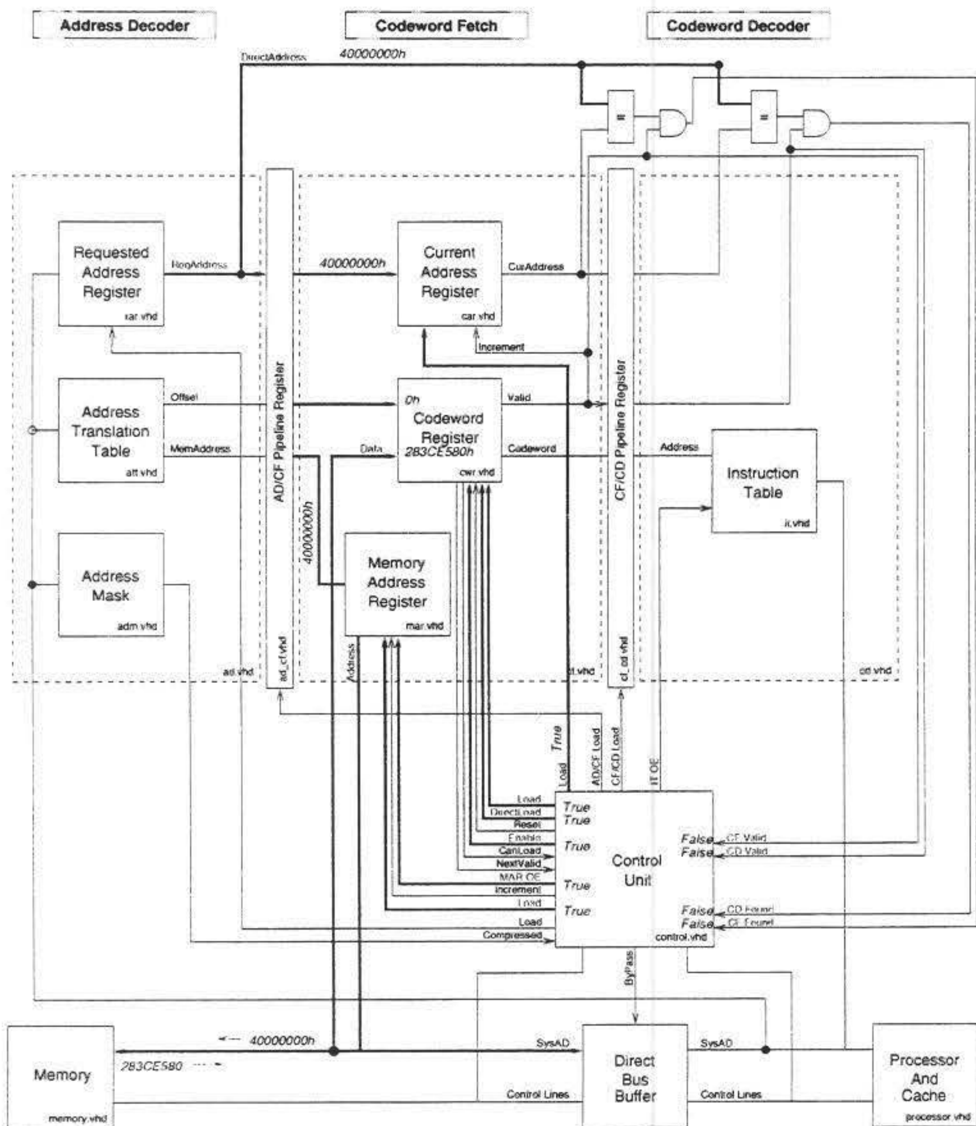


Figura 4.18: Exemplo do IBC para MIPS: Passo 3.

4. Após a leitura da memória e carga direta no CWR, a unidade de controle aguarda que o sinal `Valid` do CWR fique ativo e dá então prosseguimento à descompressão, ativando o registrador de *pipeline* CF/CD e ativando os sinais `Increment` do CAR e MAR. Como nesse caso, a instrução desejada é exatamente a primeira da palavra, o sinal CF_Found será ativado;

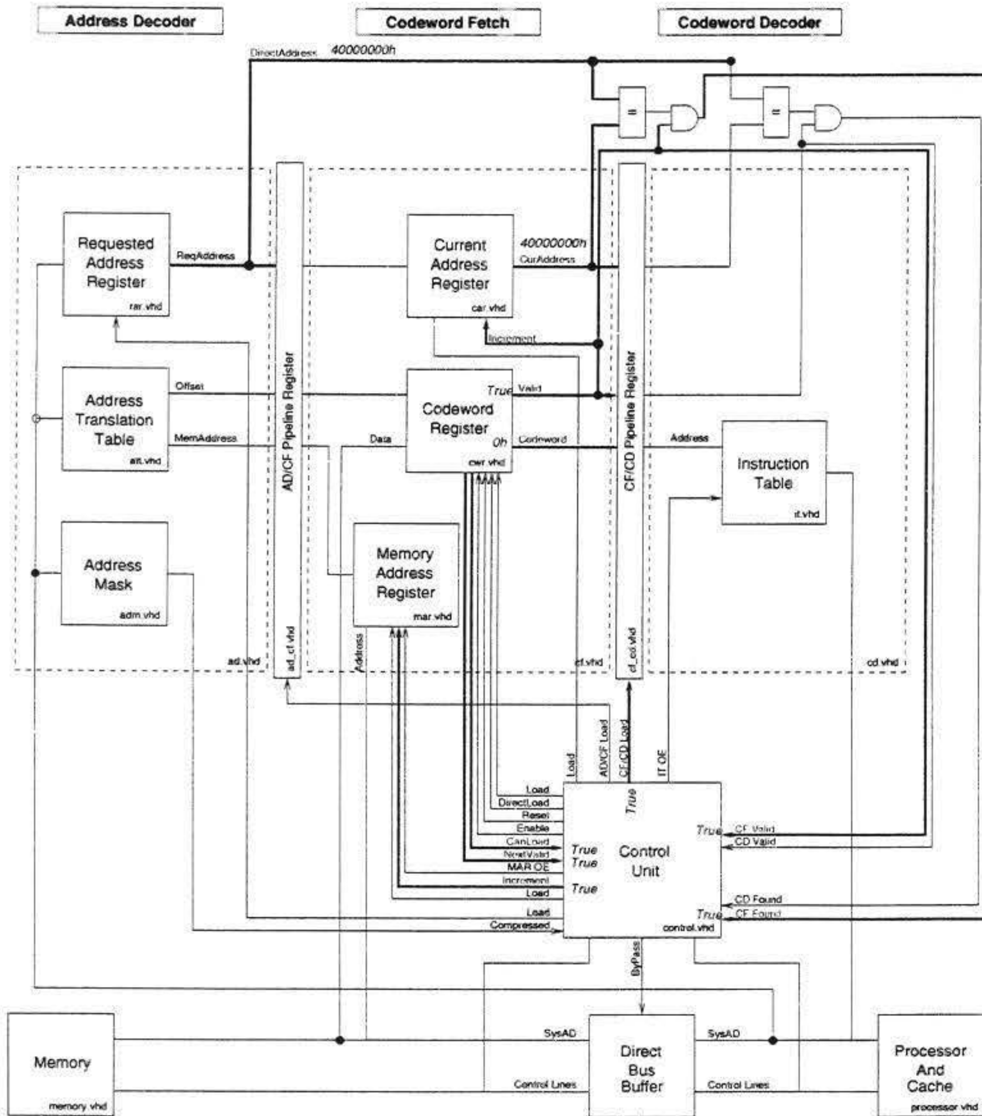


Figura 4.19: Exemplo do IBC para MIPS: Passo 4.

5. No próximo ciclo, a *codeword* é utilizada como endereço da Tabela de Instruções, e a unidade de controle, já tendo observado o sinal CD_Found ativo, inicia o processo de resposta ao processador. Nesse momento, o CWR já descartou 6 bits referentes à *codeword* e ao prefixo dessa instrução e já está pronto para liberar a próxima *codeword* após o final do ciclo de resposta ao processador;

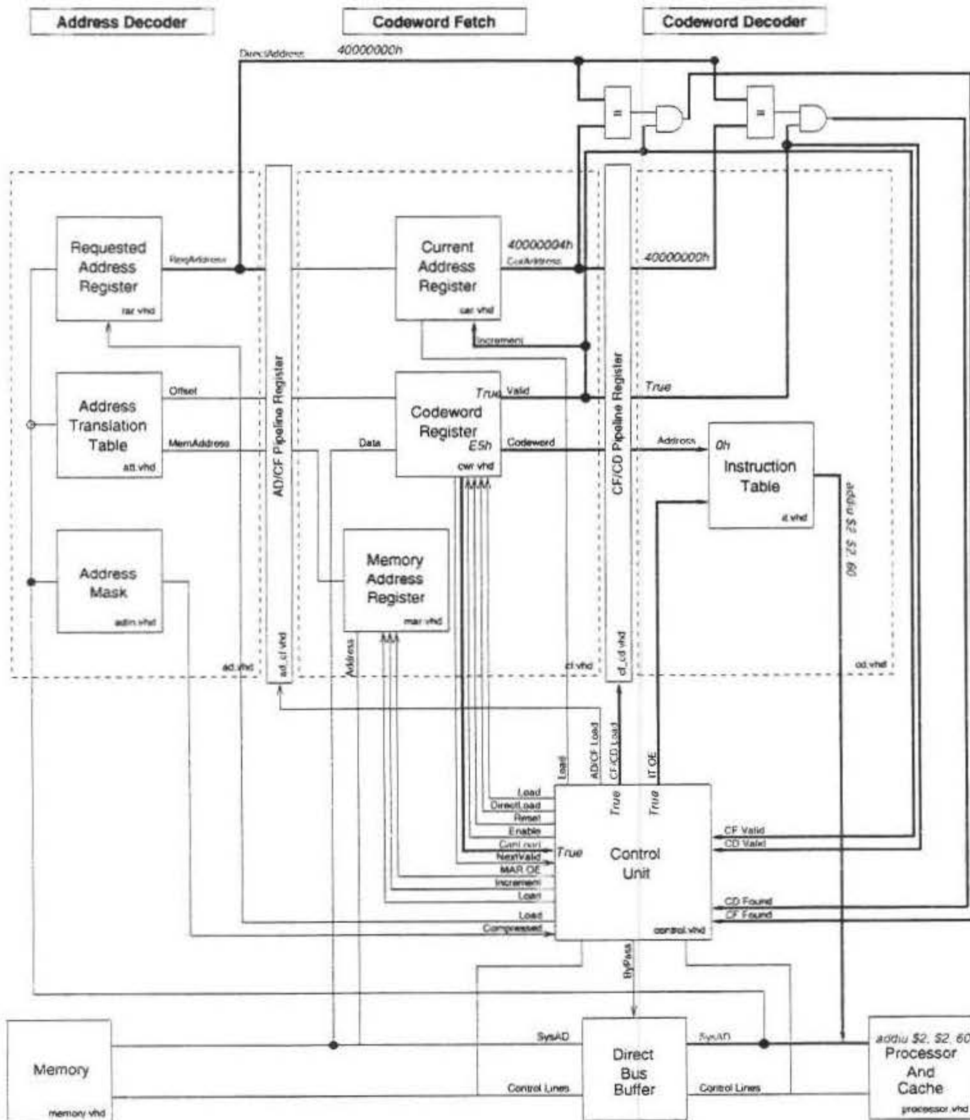


Figura 4.20: Exemplo do IBC para MIPS: Passo 5.

6. Após a resposta ao processador, a unidade de controle ainda permite que um ciclo de processamento do descompressor seja executado, de forma a transferir a *codeword* da instrução 40000004_h do estágio CF para o estágio CD, fazendo com que o CWR descarte mais 12 bits, e deixando-o sem condições de gerar a próxima *codeword* por ter apenas 14 dos 17 bits necessários para decodificá-la. A unidade de controle no entanto, só permitirá um ciclo de *pre-fetch* quando o processador fizer a solicitação de uma instrução comprimida, para não provocar contenção no barramento;

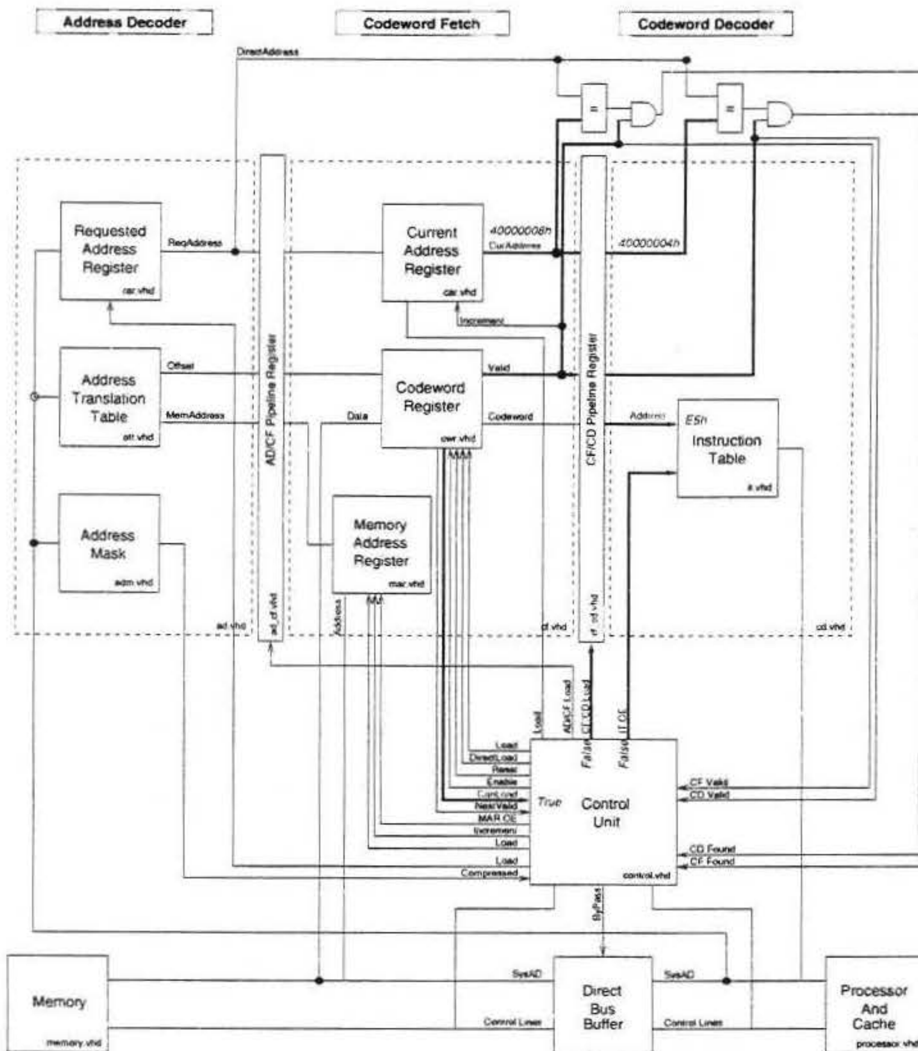


Figura 4.21: Exemplo do IBC para MIPS: Passo 6.

7. Após o início do próximo ciclo de leitura de instrução, o endereço solicitado é 40000004_h (próximo na seqüência do preenchimento da *cache-line*). O descompressor detecta que a instrução solicitada já está no estágio CD, e inicia a resposta ao processador já no ciclo seguinte. Enquanto isso, como há espaço no CWR, um ciclo de *pre-fetch* também é iniciado e a palavra lida da memória será agrupada com os bits já presentes no CWR através de uma carga normal, fazendo com que 46 bits estejam disponíveis para decodificação neste momento, e a *codeword* do endereço 40000008_h fique disponível no estágio CF. Mais um ciclo será necessário para que essa *codeword* chegue ao estágio CD, o que ocorre juntamente com a disponibilidade da próxima *codeword* (endereço $4000000C_h$) no estágio CF;

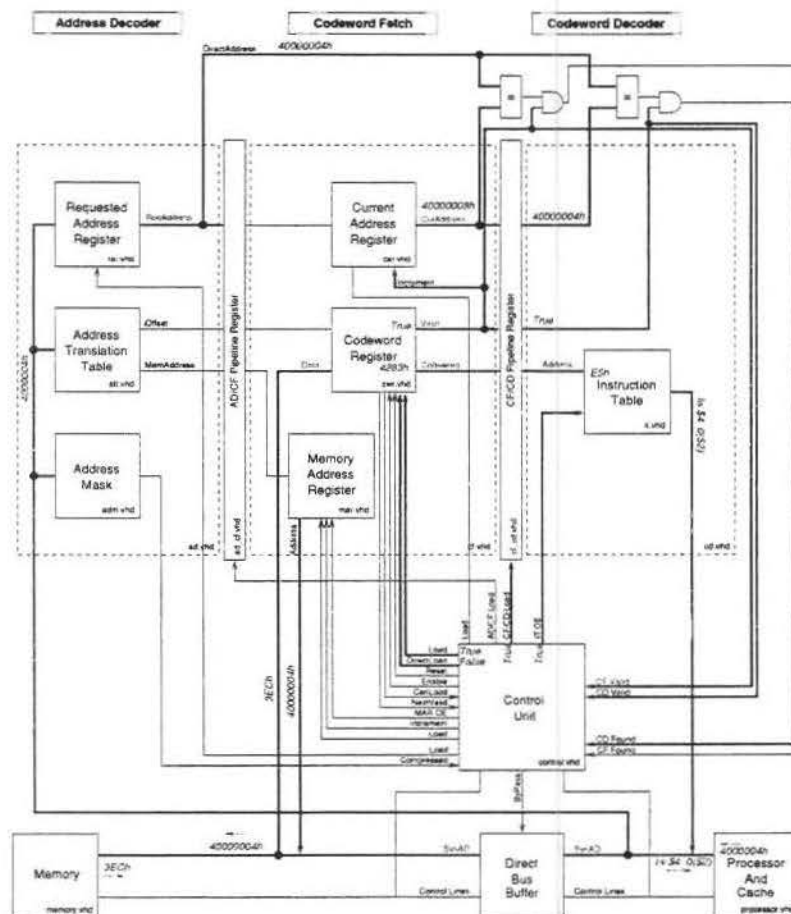


Figura 4.22: Exemplo do IBC para MIPS: Passo 7.

8. As demais leituras seguem o esquema anterior, efetuando o *pre-fetch* da memória sempre que possível na tentativa de manter o CWR com bits disponíveis para fornecer uma *codeword*.

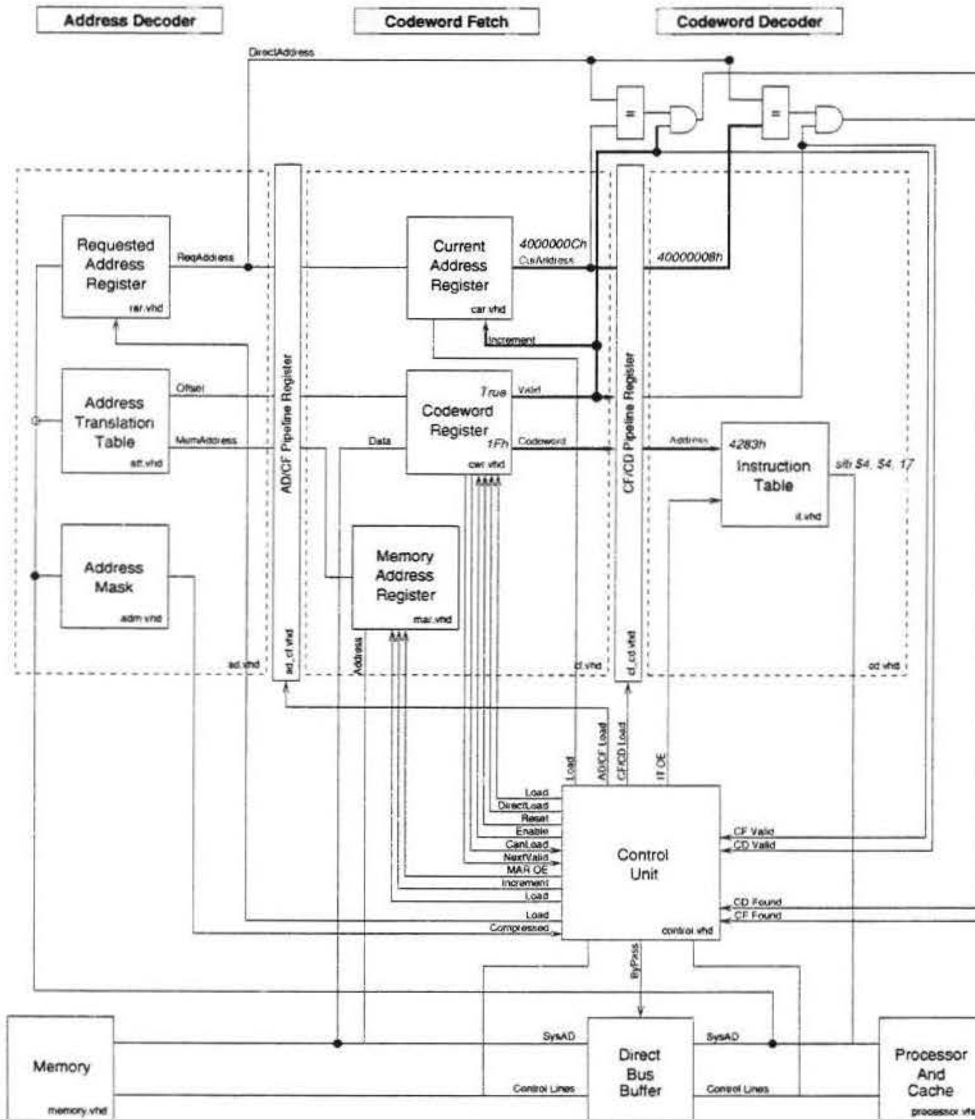


Figura 4.23: Exemplo do IBC para MIPS: Passo 8.

Caso a transferência inicial de fluxo fosse para o endereço $4000000C_h$ (que não é o endereço da primeira instrução de uma *cache-line* base) ao invés do endereço 40000000_h, todos os passos anteriores teriam que ser executados pois a política de preenchimento da *cache* exigiria todas as instruções para preencher uma *cache-line*.

4.6 Comparação dos Métodos IBC vs. TBC e PBC

O IBC pode ser considerado como uma simplificação do método TBC que por sua vez é uma simplificação do método PBC. A Figura 4.24 mostra a razão de compressão obtida por cada um dos três métodos. O IBC produz a melhor razão de compressão média final (53,6%), considerando também o impacto devidos às tabelas do descompressor. O TBC produz 60,7% e o PBC 61,3%. Quando o custo do descompressor é desconsiderado, o TBC produz a melhor razão de compressão média (27,2%), seguido pelo IBC com 31,5% e depois pelo PBC com 39,8%. Esse resultado mostra que o custo do descompressor IBC é menor que o do TBC, e que esta diferença é suficiente para compensar uma pequena perda na compressão do código.

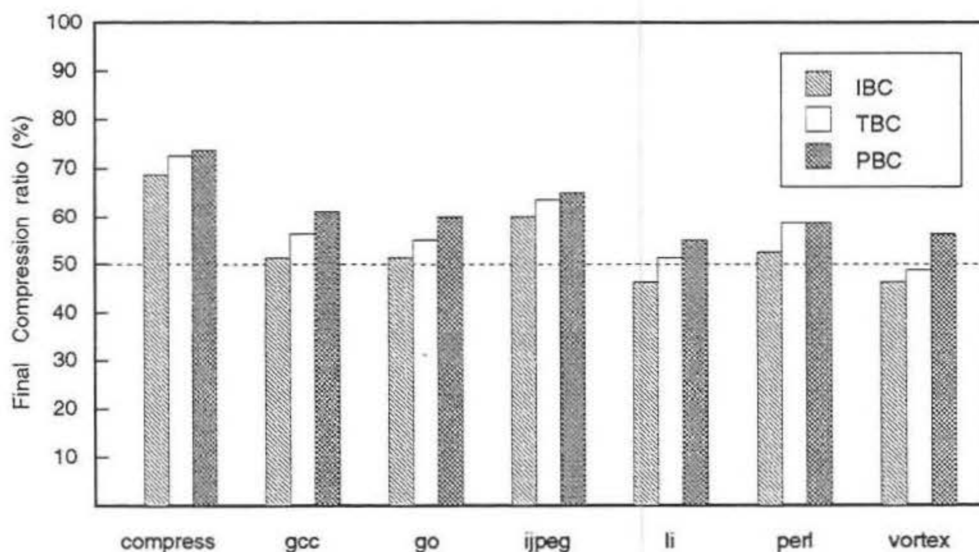


Figura 4.24: Razão de compressão final para IBC/TBC/PBC.

O descompressor TBC ocupa mais espaço que o IBC por não conseguir tirar proveito da igualdade entre duas instruções que estão em árvores diferentes, armazenando-as duas vezes no dicionário. Também é preciso levar em consideração que o decodificador das *codewords* do TBC tem que ser capaz de gerar o endereço do dicionário da árvore. No

caso do IBC, ele pode ser codificado diretamente na *codeword*. Outra diferença é o fato de o TBC precisar de 33 bits por instrução no dicionário (32 para a instrução propriamente dita e o bit `END`), enquanto o IBC precisa de 32.

Em relação ao PBC, a vantagem do IBC se destaca também na maior simplicidade do descompressor, que não requer vários módulos de memórias de tamanhos e larguras diferentes. Outro aspecto importante que piora a razão de compressão do PBC é que cada padrão de árvore e de operandos, ao serem codificados separadamente, incluem em dobro os custos de compactação (dois prefixos, dicionários, etc).

Com base nos argumentos acima, o método IBC pode ser considerado superior aos métodos PBC e o TBC para compressão de código.

4.7 Comparação entre IBC e Outros Métodos

A Tabela 4.4 mostra como fica o quadro comparativo da Tabela 2.5 após a inclusão dos métodos PBC, TBC e IBC. Para comparar as razões de compressão resultantes de cada um dos métodos, é necessário avaliar com muito cuidado como estas foram calculadas, principalmente no que diz respeito aos parâmetros levantados na Seção 2.3. Sendo assim, a comparação dessa seção se restringirá aos métodos que utilizam a arquitetura MIPS.

A razão de compressão final do IBC supera a dos métodos usados no MIPS16 [40] e Wolfe [68], mesmo não considerando o custo do descompressor em nenhum dos dois[9] casos. O trabalho de Lekatsas [53] reportou uma razão de compressão de 50%, mas esta foi baseada apenas em experimentos sobre a compressibilidade dos programas, não levando em consideração sequer um método para descompressão aleatória do código. Posteriormente Lekatsas deu continuidade ao seu método, mas utilizou outros processadores, o que inviabilizou a comparação de seus resultados finais. Como não faz sentido comparar a

[9] Wolfe considera apenas o custo da LAT.

Seção	Modelo	Arquitetura Base	Razão de Compressão	Tempo de Execução[l]
2.2.1	Thumb[a b]	ARM	55%~70%	70%~120%
	MIPS16[a]	MIPS	60%	n/d
2.2.2	Fraser[c]	SPARC	50%	2000%
	Ernst[c]	SPARC	59%	1260%
	Liao[d]	TMS320C25	84%~88%	115%~117%
	Kirovski[c e]	SPARC	60%	111%
2.2.3	Wolfe[a f]	MIPS	70%	~100%
	Lefurgy[a]	PowerPC, ARM, i386	61%, 66%, 75%	n/d
	Lefurgy[a g h]	SHARC	42%~64%	n/d
	CodePack[a]	PowerPC	60%~65%	90%~110%
	Lekatsas[a i]	MIPS, i386	50%, 65%	n/d
	Lekatsas[a j]	SHARC, ARM	48%, 55%	n/d
	Lekatsas[a]	SPARC	54%	n/d
3.1	TBC	MIPS II, TMS320C25	60,7%,75%	n/d
3.2	PBC	MIPS II	61,3%	n/d
4	IBC[k]	MIPS II	53,6%	88%~184%

[a] Implementado em Hardware.
[b] Perda de desempenho com barramento de 32 bits e ganho com barramento de 16 bits.
[c] Totalmente implementado em Software.
[d] Hardware opcional pode acelerar o desempenho.
[e] Números estimados.
[f] Desempenho estimado.
[g] Foi utilizada otimização `-O1` apenas.
[h] O processador SHARC utiliza instruções de 48 bits.
[i] Apenas experimentos sobre a forma de comprimir os programas.
[j] O descompressor é capaz de descomprimir em média 6,84 bits por ciclo.
[k] Desempenho estimado através de simulação de leitura da memória, e não de *traces* de execução do programa. O desempenho e razão de compressão da implementação para processador SPARC serão mostrados no próximo capítulo.
[l] Considerando como 100% o tempo de execução do programa original sem utilizar compressão.

Tabela 4.4: Quadro comparativo dos métodos de compressão de código

razão de compressão do IBC com métodos que utilizam outros processadores, o IBC gera a melhor razão de compressão (considerando o custo do descompressor) para a arquitetura MIPS (*circa maio 2002*).

O trabalho sobre o MIPS16 não revelou informações sobre desempenho, mas devido a similaridade com o Thumb, o desempenho pode ser estimado como sendo 120% do original. Wolfe também trabalhou com estimativas de desempenho, e elas ficaram em torno de 100%. No caso do IBC, as estimativas foram muito simples, desconsiderando a existência da *cache* tanto na simulação do programa original quanto no caso do programa comprimido. Para obter essas estimativas foram feitos dois conjuntos de simulações do modelo VHDL do descompressor. Para o primeiro conjunto foi feita a simulação de uma leitura seqüencial do programa, da primeira até a última instrução. A média do tempo de execução para os programas do SPEC CINT'95 foi de 88%. No segundo conjunto foi feita uma simulação de um *trace* de execução dos programas do SPEC CINT'95. Como não foi possível obter um *trace* com a seqüência real de execução dos programas, a simulação foi feita assumindo que todos os desvios foram tomados. Dessa forma foram obtidos os tempos de descompressão para cada um dos blocos básicos. Foi feita então uma média entre os tempos de descompressão ponderada pelo tamanho de cada bloco básico e também pelo número de execuções de cada um deles obtidos através da execução numa estação Silicon O2. O valor médio obtido foi 184%.

A comparação de desempenho entre IBC e os outros métodos não foi efetuada porque os métodos possuem apenas estimativas e não um valor real (medido). A implementação para SPARC teve seu desempenho medido e será analisado na Seção 5.5.

Capítulo 5

Compressão Baseada em Instruções (SPARC)

Esse capítulo trata da implementação do método IBC para a arquitetura SPARC. O protótipo desenvolvido foi baseado no processador Leon [33, 63], que é um *core* compatível com a arquitetura SPARC V8 [66] descrito em VHDL. Esta implementação foi feita em um *kit* de desenvolvimento XESS XSV800 [21, 69].

A motivação em utilizar o Leon deve-se principalmente ao fato deste ser um *core* aberto já validado e que pode ser configurado, se necessário, para atender a algumas restrições específicas do projeto (principalmente no que diz respeito à área total ocupada na FPGA, que afeta o tamanho da *cache* e alguns periféricos internos). No entanto, foi tomado cuidado para não realizar alterações no *core*, implementando o descompressor externamente ao processador, dado ser esta uma das premissas deste projeto. A existência do Leon permitiu também implementar todo o conjunto em uma única FPGA, e incluir o descompressor entre o processador/*cache* e a memória conforme mostra a Figura 5.1.

5.1 Ambiente de Prototipagem

O processador utilizado no protótipo foi o Leon [33, 63], que é um *core* escrito em VHDL e sintetizável para várias tecnologias de FPGAs e ASICs. O Leon implementa o conjunto

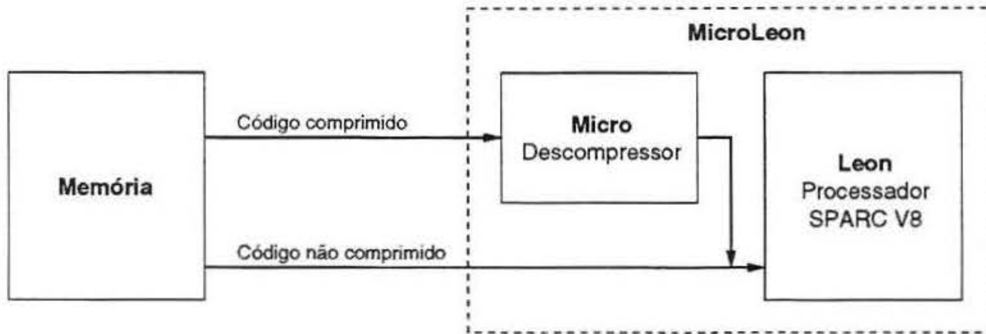


Figura 5.1: Diagrama de blocos do conjunto Processador/Descompressor/Memória utilizado.

de instruções do padrão SPARC V8 [66], e foi desenvolvido inicialmente para aplicações da Agência Espacial Européia, sendo colocado em domínio público posteriormente. O processador Leon possui as seguintes características:

- *Pipeline* de 5 estágios que implementa o conjunto de instruções SPARC V8;
- Unidades de multiplicação, divisão e MAC (*Multiply and Accumulate*) em hardware;
- *Cache* de dados e instruções separadas, ambas *direct mapped*;
- Implementação completa dos barramentos AHB e APB para interligação de periféricos internos seguindo o padrão AMBA-2.0 [5];
- *Cache* de dados capaz de efetuar *snooping* do barramento AHB;
- Controlador programável de memória RAM e ROM externas com barramentos de 8, 16 ou 32 bits;
- Unidade de depuração interna para inspeção do estado do processador sem afetar a execução do programa;
- Diversos periféricos como UART, temporizadores, controlador de interrupção e uma porta de entrada e saída de 16 bits;

- Interface com a unidade de ponto flutuante Meiko e um co-processador definido pelo usuário;
- Unidade de ponto flutuante IEEE-754 própria que permite a soma, subtração e comparação (outras operações têm que ser implementadas por software);
- Modo de economia de energia.

Como *benchmark* foram utilizados os mesmos programas do SPEC CINT'95 dos capítulos anteriores compilados em uma SUN Enterprise E450 com `gcc` versão 2.95 com as opções `-Os -mv8` e cinco novos programas descritos na Tabela 5.1, que fazem parte do conjunto de testes do Leon (aqui denominados de *benchmark* Leon). Estes foram compilados em um PC com gcc 2.95.2 configurado como *cross-compiler* e bibliotecas estáticas para execução direta no *kit* de desenvolvimento. Foram utilizadas opções de compilação padrão para cada um dos testes, além das opções de posicionamento das seções de código e dados na memória (`-Ttext=60000000 -Tdata=40000000`). Destes programas, dois são *benchmarks* sintéticos (`dhry` e `stanford`), dois são programas de testes (`c-irq` e `paranoia`) e o último é o conhecido programa "*Hello World*" (`hello`) que foi utilizado nos testes preliminares. Estes cinco programas foram escolhidos porque não requerem serviços de um RTOS (*Run Time Operating System*), e assim sendo, resultam em um código menor que garante um melhor uso da pequena memória interna da FPGA existente no *kit* de desenvolvimento.

Programa	Descrição
c-irq	Teste de interrupções do processador
dhry	*Benchmark* Dhrystone
hello	Programa *Hello World*
paranoia	Teste da unidade de ponto flutuante
stanford	Pequeno *benchmark* de desempenho criado por John Henessy

Tabela 5.1: Programas específicos para implementação IBC/Leon.

O *benchmark* sintético `stanford` reporta os tempos de execução de cada uma de suas rotinas internas, que na realidade, são programas já conhecidos e estão listados na Tabela 5.2.

Rotina	Descrição
Perm	Permutações. Fortemente recursivo. Calcula 5 vezes as permutações de 7 elementos)
Towers	Soluciona uma Torre de Hanoi de 14 peças
Queens	Problema das 8 rainhas, solucionado 50 vezes
Intmm	Multiplica duas matrizes 40 x 40 de números inteiros
Mm	Multiplica duas matrizes 40 x 40 de números de ponto flutuante
Puzzle	Monta um quebra-cabeça
Quick	Ordenação utilizando Quicksort com 5000 elementos
Bubble	Ordenação utilizando Bubblesort com 500 elementos
Tree	Ordenação utilizando TreeSort com 5000 elementos
FFT	Transformada de Fourier

Tabela 5.2: Rotinas do programa Stanford.

O Leon e o descompressor foram sintetizados para utilização no *kit* de desenvolvimento XESS XSV800 [21, 69] (Figura 5.2) cujos principais componentes são:

- FPGA Virtex XCV800 [70] da Xilinx [38], que é o módulo principal da placa e onde o Leon e o descompressor foram implementados;
- CPLD XC95108 que é utilizado para controles e programação da FPGA Virtex;
- Dois bancos de memória SRAM de 512K x 16 e 16 Mbit de memória Flash;
- Oscilador de 100MHz com divisor programável (entre 1 e 2052);
- Conectores para interface paralela e serial;

Como ferramentas de síntese, foram utilizados o Leonardo Spectrum versão 2001d, e ISE 4.2i como ferramenta específica para a FPGA da Xilinx (*placement* e *routing*). As

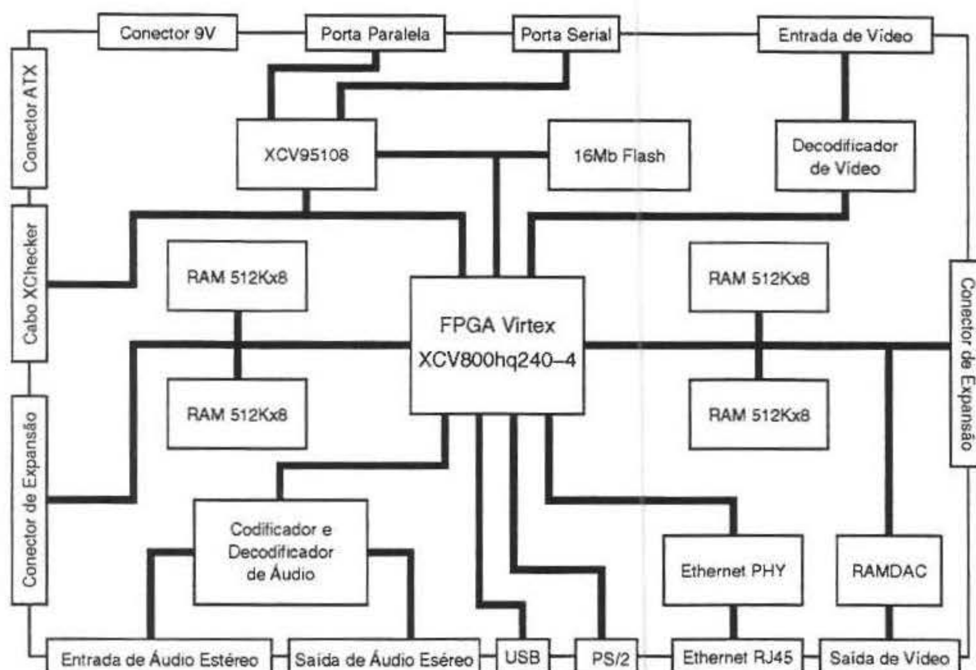


Figura 5.2: Diagrama de blocos da placa XESS XSV800.

sínteses foram efetuadas numa estação SUN Enterprise 450 com quatro processadores e 4GB de RAM e em um PC AMD Athlon 1,4GHz com 1Gb de RAM.

5.2 Análise do Método para SPARC

A mesma análise do tamanho dos programas utilizada para MIPS foi feita para os dois *benchmarks*. A Tabela 5.3 mostra o tamanho de cada um dos programas utilizados e o número de instruções distintas. Em média, esse valor é de 33,5% do programa (30,0% para os programas do SPEC CINT'95 e 38,5% para os programas de teste do Leon). Um fato relevante que deve ser notado é o aumento das instruções únicas da arquitetura SPARC em relação à MIPS (30,0% contra 18,3% para o SPEC CINT'95). Essa maior quantidade de instruções únicas deve-se à maior variedade de tipos diferentes de instruções na arquitetura SPARC em relação à MIPS. Enquanto a arquitetura SPARC V8 possui 214 tipos de instrução em 23 possibilidades de formatos, a arquitetura MIPS possui apenas

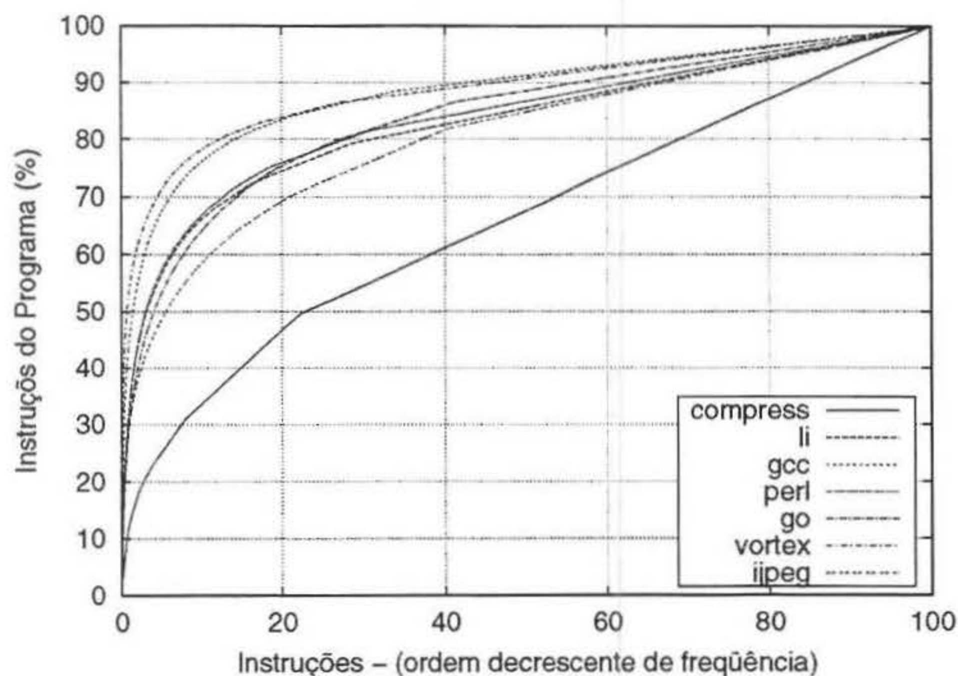
103 tipos de instrução em 17 possibilidades de formatos[1]. Esses números foram obtidos através da contagem das alterações possíveis de cada um dos campos das arquiteturas, excluídos do cálculo os campos de registradores, imediatos, valores que fornecem instruções inválidas e instruções de ponto-flutuante e co-processador.

Benchmark	Programa	Tamanho (em instruções)	Instruções Distintas (%)
SPEC CINT'95	compress	1375	891 (64,8)
	gcc	268425	46902 (17,5)
	go	55247	12654 (22,9)
	ijpeg	29492	8946 (30,3)
	li	10751	3124 (29,1)
	perl	53232	14219 (26,7)
	vortex	105404	19541 (18,5)
Leon	c-irq	10104	3917 (38,8)
	dhry	8292	3391 (40,9)
	hello	10048	3890 (38,7)
	paranoia	26620	9762 (36,7)
	stanford	13732	5123 (37,3)

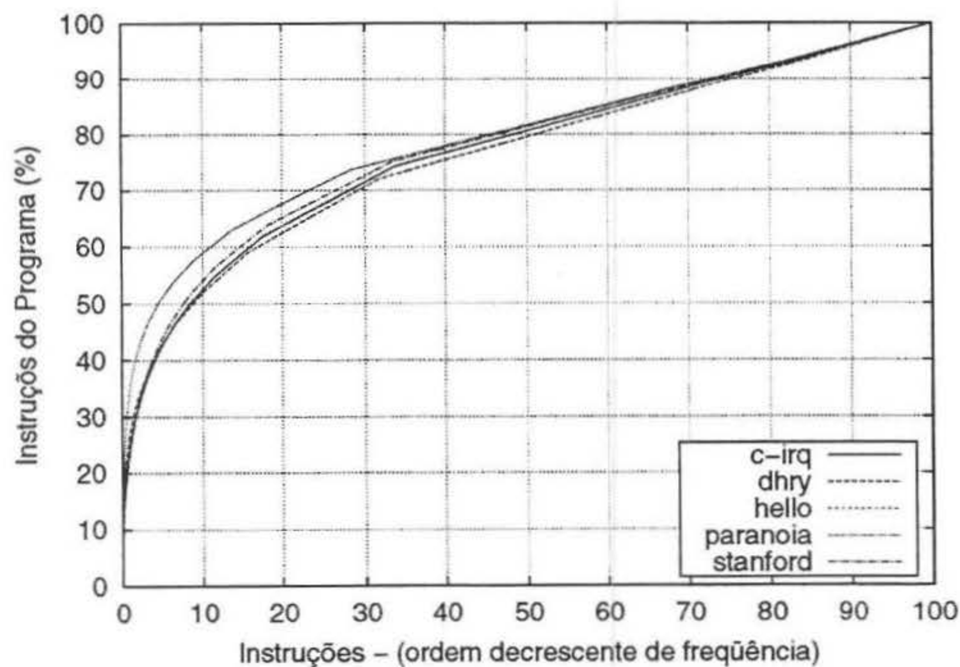
Tabela 5.3: Número de instruções únicas para SPARC.

Os dois gráficos da Figura 5.3 mostram a distribuição das instruções para cada um dos *benchmarks* utilizados. No caso do SPEC (Figura 5.3(a)), mais de 70% do total de instruções do programa é coberto por apenas 20% das instruções únicas. Embora esse número seja significativo, deve ser destacado que para o MIPS a cobertura era superior a 80% (Figura 4.3). Essa diferença deve ser creditada novamente à maior variedade de instruções disponíveis na arquitetura SPARC. O resultado para o *benchmark* Leon é mostrado na Figura 5.3(b), dela temos que 20% das instruções únicas cobrem mais de 60% dos programas. A queda na cobertura deve-se à pequena quantidade de instruções em cada um dos programas, o que diminui as chances de repetições.

[1] O conceito de formato de uma instrução aqui utilizado inclui não somente a divisão dos bits em campos, mas também o significado de cada um dos campos.



(a) *Benchmark* SPEC

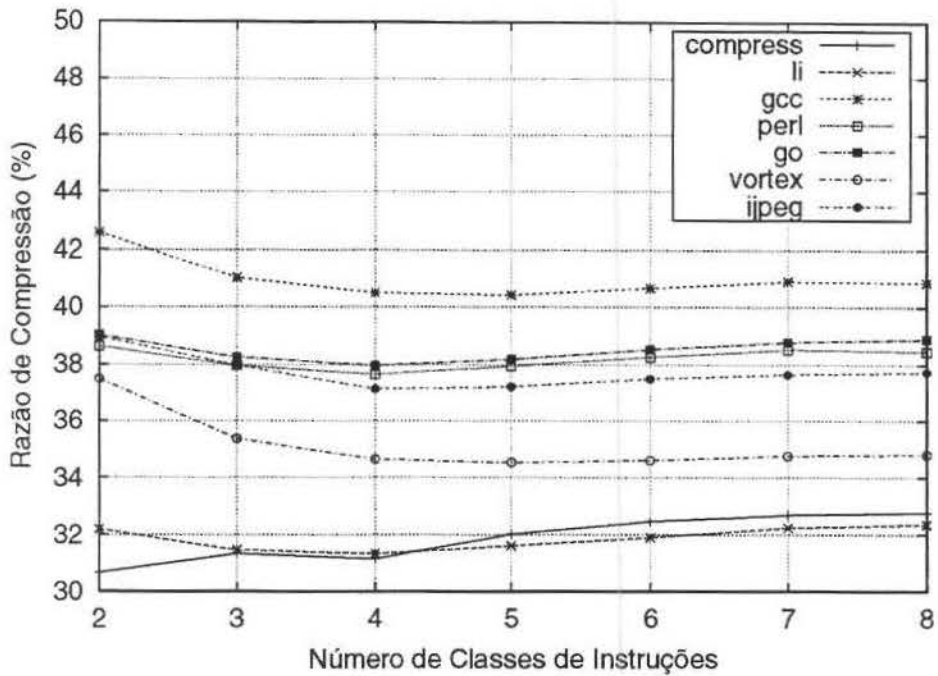


(b) *Benchmark* Leon

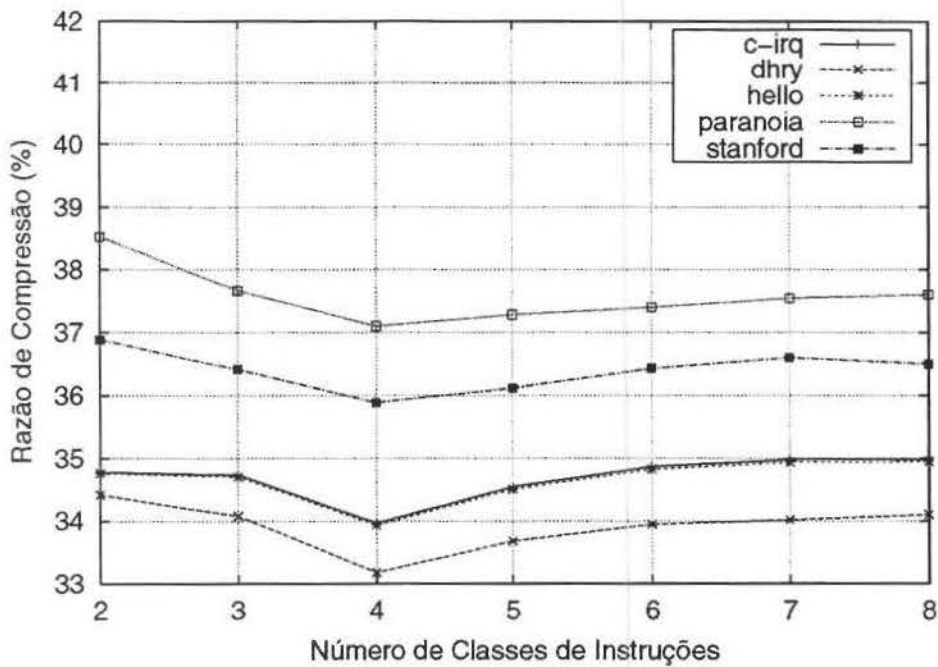
Figura 5.3: Porcentagem do programa coberta por instruções distintas.

O funcionamento do compressor para a arquitetura SPARC segue os mesmos passos descritos na Figura 4.4 que foram utilizados para a arquitetura MIPS. No entanto, o estudo do número ideal de classes passou a utilizar classes sobrepostas (Seção 4.3) e, conforme mostrado na Figura 5.4, resultou em um comportamento semelhante ao do MIPS (Figura 4.5). A melhor combinação ocorre novamente em 4 classes, o que é mais fortemente verificado no *benchmark* Leon (Figura 5.4(b)). A Tabela 5.4 mostra as razões de compressão obtidas para cada um dos programas e os tamanhos em bits de cada uma das quatro classes (os 2 bits de prefixo não estão incluídos). Nesta tabela, também é mostrada a razão de compressão quando não se utiliza classes sobrepostas para as mesmas configurações de classes. Observe que a maior diferença absoluta é de 0,43%, um valor pequeno e que permite então escolher o método de implementação mais simples baseado na sobreposição de classes (Seção 4.3). A razão de compressão média utilizando classes sobrepostas para o *benchmark* SPEC CINT'95 é de 69,45% e para o *benchmark* Leon é de 76,72%. Deve ser observado que o valor para SPEC CINT'95 é significativamente superior ao valor encontrado para MIPS, que é de 53,60%. Esse fato pode ser creditado novamente à maior variedade de instruções da arquitetura SPARC.

Um aspecto importante que não pode ser observado em detalhes na Figura 5.3 é a quantidade de instruções que ocorrem apenas uma vez e que passaram a ter um grande impacto na razão de compressão do IBC implementado para a arquitetura SPARC. Essas instruções, como já dito na Seção 4.3, ocupam mais espaço depois de comprimidas que no programa original uma vez que precisam ser armazenadas na IT e também codificadas no programa comprimido. A Tabela 5.5 mostra que em média 67,97% (68,02% para o *benchmark* SPEC CINT'95 e 67,90% para o *benchmark* Leon) das instruções distintas ocorrem apenas uma vez no programa o que mostra que uma forma alternativa de codificação dessas instruções deve ser buscada. A solução proposta para este problema foi permitir que instruções não compactadas sejam incluídas no código comprimido, reservando um



(a) *Benchmark* SPEC



(b) *Benchmark* Leon

Figura 5.4: Razão de compressão para diferentes quantidades de classes.

Benchmark	Programa	Classes				Razão de Compressão		Diferença Absoluta
		I	II	III	IV	Com Sobreposição	Sem Sobreposição	
SPEC CINT'95	compress	3	6	8	10	98,68%	98,25%	0,43%
	gcc	7	10	13	16	62,27%	62,00%	0,27%
	go	5	8	11	14	64,77%	64,42%	0,35%
	ijpeg	4	8	11	14	71,18%	70,94%	0,24%
	li	3	6	9	12	63,70%	63,41%	0,29%
	perl	5	8	11	14	68,28%	67,96%	0,32%
	vortex	4	7	11	15	57,28%	57,05%	0,23%
Leon	c-irq	2	7	10	12	76,06%	75,86%	0,20%
	dhry	1	7	10	12	77,39%	77,24%	0,15%
	hello	2	7	10	12	75,97%	75,77%	0,20%
	paranoia	5	8	11	14	77,48%	77,14%	0,34%
	stanford	3	7	10	13	76,71%	76,45%	0,26%

Tabela 5.4: Resultado da divisão em classes para SPARC.

dos prefixos para elas. Ao adotar essa estratégia, as instruções da quarta classe passam a ser codificadas com 34 bits (32 bits da instrução e 2 de prefixo) conforme mostrado na Figura 4.10(b)

As Tabelas 5.6 e 5.7 mostram os resultados dos experimentos quando o tamanho da IT é restringido, forçando a codificação das instruções menos freqüentes diretamente no programa comprimido (utilizando 34 bits). A coluna *Limite da IT* expressa o número de bits necessários para endereçar a IT. Para o programa `compress`, por exemplo, foram testadas ITs com 128, 256 e 512 linhas (7, 8 e 9 bits de endereçamento respectivamente) e o melhor resultado foi obtido utilizando 256 linhas (8 bits de endereçamento). Uma vez que o *compress* possui apenas 891 instruções únicas, não faz sentido testar configurações da IT com 1024 ou mais linhas. Os mesmos testes foram feitos para todos os outros programas dos *benchmarks* e, em média, a razão de compressão para o *benchmark* SPEC CINT'95 ficou em 61,82% e para o *benchmark* Leon foi de 67,41%.

Benchmark	Programa	Instruções Distintas	Instruções pouco freqüentes	
			1 ocorrência	2 ocorrências
SPEC CINT'95	compress	891	692 (77,67%)	129 (14,48%)
	gcc	46902	31200 (66,52%)	5630 (12,00%)
	go	12654	7397 (58,46%)	1908 (15,08%)
	ijpeg	8946	5322 (59,49%)	1769 (19,77%)
	li	3124	2240 (71,70%)	381 (12,20%)
	perl	14219	9877 (69,46%)	1707 (12,01%)
	vortex	19541	14240 (72,87%)	1988 (10,17%)
Leon	c-irq	3917	2600 (66,38%)	630 (16,08%)
	dhry	3391	2315 (68,27%)	545 (16,07%)
	hello	3890	2583 (66,40%)	622 (15,99%)
	paranoia	9762	7008 (71,79%)	1399 (14,33%)
	stanford	5123	3416 (66,68%)	779 (15,21%)

Tabela 5.5: Número de instruções que ocorrem uma ou duas vezes no programa.

Dada a similaridade dos programas do *benchmark* Leon tanto na faixa de tamanho quanto nas bibliotecas estáticas utilizadas pelos programas, a razão de compressão final foi obtida sempre com a limitação da IT em 1024 linhas (10 bits de endereçamento) e a razão de compressão ficou sempre em torno de 67% para todos os programas.

A inclusão de instruções não comprimidas diretamente no programa comprimido, e conseqüentemente a limitação do tamanho da IT permitiu uma redução absoluta de 7,63% (melhora de 10,99%) na razão de compressão para o *benchmark* SPEC CINT'95 e 9,31% (melhora de 12,14%) para o *benchmark* Leon.

Os resultados da Tabela 4.2 para a arquitetura MIPS e os da Tabela 5.6 para a arquitetura SPARC foram agrupados na Tabela 5.8, que mostra os resultados finais obtidos com o método IBC para os programas do SPEC CINT'95 nestas duas arquiteturas. Como os programas foram compilados com as mesmas otimizações e nenhum dos dois casos incluiu as bibliotecas do sistema operacional, o menor número de instruções nos programas para a arquitetura SPARC comprova o melhor poder de expressão do seu conjunto de instruções.

Programa	Limite da IT	Classes				Razão de Compressão			
		I	II	III	IV	Código	IT	ATT	Total
compress	7	3	5	7	32	73,35%	9,31%	2,93%	85,59%
	8	3	6	8	32	62,90%	18,62%	2,93%	**84,45%**
	9	4	7	9	32	50,15%	37,24%	2,92%	90,31%
gcc	9	4	7	9	32	69,93%	0,19%	4,49%	74,61%
	10	5	8	10	32	63,76%	0,38%	4,49%	68,63%
	11	5	8	11	32	58,08%	0,76%	4,49%	63,33%
	12	6	9	12	32	53,25%	1,53%	4,49%	59,27%
	13	7	10	13	32	49,46%	3,05%	4,48%	57,01%
	14	7	11	14	32	46,52%	6,10%	4,31%	**56,93%**
	15	7	11	15	32	44,00%	12,21%	4,29%	60,50%
go	9	5	7	9	32	67,19%	0,93%	4,10%	72,22%
	10	5	8	10	32	60,33%	1,85%	4,10%	66,28%
	11	5	8	11	32	53,76%	3,71%	3,90%	61,37%
	12	6	9	12	32	47,90%	7,41%	3,91%	**59,22%**
	13	7	10	13	32	43,07%	14,83%	3,91%	61,81%
ijpeg	9	4	7	9	32	66,06%	1,74%	3,90%	71,70%
	10	4	7	10	32	59,70%	3,47%	3,91%	67,08%
	11	4	8	11	32	53,39%	6,94%	3,62%	**63,95%**
	12	4	9	12	32	46,46%	13,89%	3,71%	64,05%
	13	5	10	13	32	39,23%	27,78%	3,71%	70,72%
li	8	3	6	8	32	54,27%	2,38%	3,52%	60,17%
	9	3	6	9	32	48,67%	4,76%	3,52%	56,95%
	10	4	7	10	32	43,61%	9,52%	3,52%	**56,65%**
	11	5	8	11	32	38,09%	19,05%	3,32%	60,46%
perl	9	4	7	9	32	65,12%	0,96%	4,11%	70,19%
	10	5	8	10	32	59,06%	1,92%	3,91%	64,89%
	11	5	8	11	32	53,57%	3,85%	3,90%	61,32%
	12	6	9	12	32	48.74%	7,69%	3,91%	**60,34%**
	13	7	10	13	32	44,69%	15,39%	3,91%	63,99%
vortex	9	3	6	9	32	54,34%	0,49%	4,10%	58,93%
	10	4	7	10	32	49,81%	0,97%	4,11%	54,89%
	11	4	7	11	32	46,00%	1,94%	4,10%	52,04%
	12	4	8	12	32	43,23%	3,89%	4,10%	**51,22%**
	13	5	9	13	32	41,03%	7,77%	4,10%	52,90%
	14	5	9	14	32	37,44%	15,54%	4,10%	57,08%

Tabela 5.6: Razão de compressão para tamanhos diferentes da IT (*benchmark* SPEC CINT'95).

Programa	Limite da IT	Classes				Razão de Compressão			
		I	II	III	IV	Código	IT	ATT	Total
c-irq	8	1	5	8	32	67,95%	2,53%	3,52%	74,00%
	9	1	6	9	32	61,15%	5,07%	3,52%	69,73%
	10	2	7	10	32	53,68%	10,13%	3,52%	**67,33%**
	11	2	8	11	32	46,02%	20,27%	3,52%	69,80%
dhry	8	1	5	8	32	65,15%	3,09%	3,52%	72,75%
	9	1	6	9	32	59,28%	6,17%	3,52%	68,97%
	10	1	7	10	32	51,40%	12,35%	3,52%	**67,27%**
	11	1	7	11	32	43,86%	24.70%	3,52%	71,88%
hello	8	1	5	8	32	67,83%	2,55%	3,52%	73,90%
	9	1	6	9	32	61,02%	5,10%	3,52%	69,63%
	10	2	7	10	32	53,52%	10,19%	3,52%	**67,23%**
	11	2	8	11	32	45,89%	20,38%	3,52%	69,89%
paranoia	8	3	6	8	32	69,72%	0,96%	3,91%	74,59%
	9	3	6	9	32	64,97%	1,92%	3,91%	70,80%
	10	4	7	10	32	60,11%	3,85%	3,71%	**67,67%**
	11	5	8	11	32	54,89%	7,69%	3,71%	66,29%
	12	5	9	12	32	49,11%	15,39%	3,71%	69,21%
	13	5	9	13	32	40,82%	30,77%	3,71%	75,30%
stanford	8	2	6	8	32	70,38%	1,86%	3,71%	75,96%
	9	3	7	9	32	63,62%	3,73%	3,71%	71,06%
	10	3	7	10	32	56,56%	7,46%	3,52%	**67,54%**
	11	3	8	11	32	49,19%	14,91%	3,52%	67,62%
	12	5	9	12	32	40,84%	29,83%	3,52%	74,19%

Tabela 5.7: Razão de compressão para tamanhos diferentes da IT (*benchmark* Leon).

Portanto, era de se esperar uma razão de compressão melhor para a arquitetura MIPS em relação à SPARC, visto que o código gerado para os programas é maior (em média 46,32%). Essa diferença de tamanho cai após a compressão do código, passando o código MIPS a ocupar 25,82% a mais que o SPARC após a compressão. Uma característica que deve ser destacada também é que a razão de compressão obtida para a arquitetura MIPS não utilizou o recurso de limitação da tabela de instruções, que mostrou resultados bastante satisfatórios para a arquitetura SPARC.

Programa	MIPS			SPARC		
	Tamanho (em instr.)	Razão de compressão	Tamanho Final	Tamanho (em instr.)	Razão de compressão	Tamanho Final
compress	2152	65,83%	1417	1375	84,45%	1161
gcc	363560	50,39%	183198	268425	56,93%	152814
go	73908	50,86%	37590	55247	59,22%	32717
ijpeg	47988	60,23%	28903	29492	63,95%	18860
li	18448	46,55%	8588	10751	56,65%	6090
perl	69536	51,43%	35762	53232	60,34%	32120
vortex	151348	45,74%	69227	105404	51,22%	53988

Tabela 5.8: Comparação dos Resultados para as arquiteturas MIPS e SPARC.

5.3 Descompressor IBC *pipelined* para SPARC V8

Um descompressor similar ao projetado para o MIPS foi desenvolvido para a arquitetura SPARC V8. O modelo de processador utilizado foi o Leon e a implementação foi feita para o *kit* XESS XSV800.

O *kit* XESS XSV800 possui 2MB de memória RAM, que foram divididos em dois bancos de memória de 256K x 32 bits. Um dos bancos foi utilizado como região de memória comprimida (que armazena o código comprimido), e o outro banco foi utilizado como região de dados e código não comprimido. O Leon possui sinais de controle para 5 bancos de memória RAM, sendo que os quatro primeiros bancos ficam no intervalo de 40000000_h a $5FFFFFFF_h$ podendo cada um deles ter o tamanho máximo de 512MB. Esses quatro bancos de memória têm sua temporização fixada num registrador de controle do processador tendo um número fixo de *wait-states*. O quinto banco é mapeado no intervalo de 60000000_h a $7FFFFFFF_h$ e ao invés de possuir temporização fixa, ele possui um sinal de entrada que indica quando os dados da memória estão disponíveis no barramento. Como o descompressor não possui temporização fixa (a descompressão de uma instrução pode demorar uma quantidade variável de ciclos), ele foi mapeado para descomprimir

o conteúdo do banco 5 de memória. Dessa forma, 256K x 32 bits de memória do *kit* de desenvolvimento foram mapeados no endereço de 40000000_h a 40100000_h (banco 0 de memória RAM do Leon), utilizados para dados e outros 256K x 32 bits foram mapeados no endereço de 60000000_h a 60100000_h (banco 5 de memória RAM do Leon), utilizados para código comprimido. A Figura 5.5 mostra como ficou o mapeamento da memória do Leon após essa divisão.

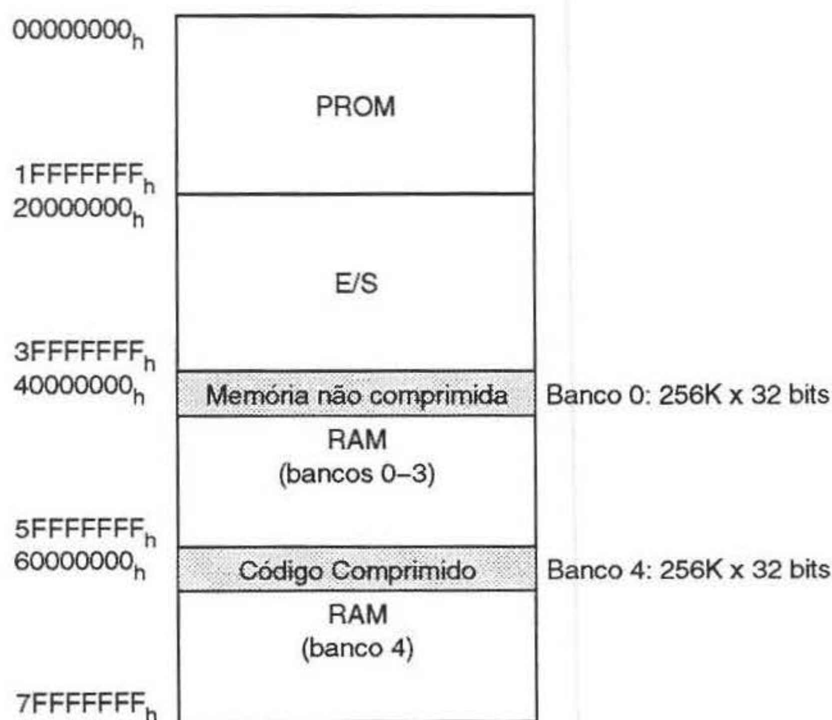


Figura 5.5: Mapa de memória para o Leon incluindo a região de código comprimido.

A Figura 5.6 mostra a implementação *pipelined* de 4 estágios do IBC para o Leon. Ele é similar ao mostrado na Figura 4.14 com os componentes do CWR (Figura 4.13) implementados diretamente no *pipeline*. Os estágios e seus componentes são descritos a seguir:

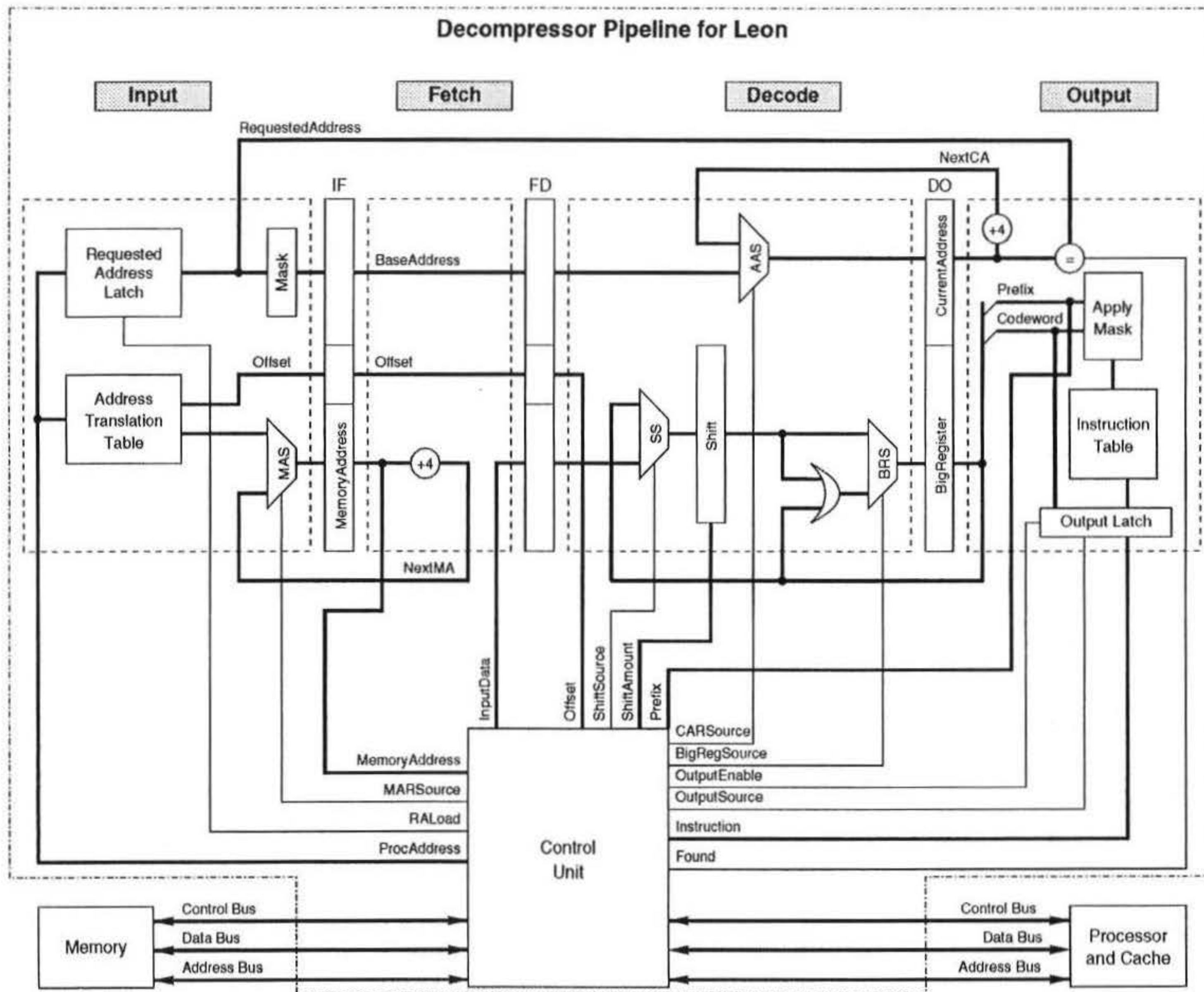


Figura 5.6: Descompressor para Leon.

***Input*:** Esse estágio é equivalente ao `Address Decoder` do *pipeline* do descompressor para MIPS. Ele é responsável por decodificar os endereços solicitados pelo processador e efetuar a conversão entre endereço da instrução não comprimida e o endereço da *codeword* correspondente na memória. O módulo `Address Mask` do descompressor para MIPS não é mais necessário, pois o Leon fornece um sinal de habilitação para cada um dos bancos de memória. Dessa forma, o sinal de habilitação do banco 4 indica uma leitura a instrução comprimida. Os módulos do estágio *Input* são:

Requested Address Latch **(`RAL`):** É responsável por armazenar o endereço solicitado pelo processador. Toda vez que a `Control Unit` detecta uma solicitação de instrução comprimida, o sinal `RALoad` é ativado para que o RAL armazene o endereço desejado.

Address Translation Table **(`ATT`):** É responsável por converter os endereços das instruções. A `ATT` está descrita em detalhes na Seção 4.2.

Memory Address Selector **(`MAS`):** Dependendo do valor do sinal `MARSource` (gerado na `Control Unit`), seleciona para o próximo valor do registrador `MemoryAdddress` o resultado da busca na `ATT` ou o valor anterior de `MemoryAddress` incrementado (através do sinal `NextMA`).

***Fetch*:** Esse estágio efetua as leituras das instruções comprimidas na memória do processador. Para isso, o sinal `MemoryAddress` é gerado e o sinal `InputData` é lido e repassado para o próximo estágio do *pipeline*. Toda vez que uma leitura da memória é efetuada, o valor do registrador `MemoryAddress` é incrementado para apontar para a próxima palavra da memória. Os sinais `BaseAddress` e `Offset` são apenas transportados para o estágio `Decode` do *pipeline*.

Decode: Esse é o estágio principal do *pipeline*. É nele que as palavras lidas da memória são agrupadas e as *codewords* são decodificadas. Os módulos desse estágio são:

Shift Selector **(SS)**: Dependendo do valor do sinal `ShiftSource` (gerado na `Control Unit`), seleciona para deslocamento a palavra lida da memória ou os bits restantes do valor que está sendo processado no momento, proveniente do `BigRegister`.

Shift: Efetua o deslocamento do valor de entrada em `ShiftAmount` bits. É utilizado para alinhar as palavras lidas da memória, permitindo seu alinhamento com a palavra que está sendo processada (`BigRegister`) e também efetuando o deslocamento para a remoção da *codeword* atual do `BigRegister`.

Big Register Selector **(BRS)**: Dependendo do valor do sinal `BigRegSource` (gerado na `Control Unit`), seleciona para próximo valor o resultado do deslocamento ou então o resultado do deslocamento mesclado com o valor anterior do `BigRegister`.

Current Address Selector **(AAS)**: O `CurrentAddress` é um registrador que armazena a posição de memória que está sendo decodificada no momento. Após fornecer uma instrução ao processador ou descartá-la no processamento, o valor selecionado é o `NextAA` que é o valor do `CurrentAddress` incrementado para indicar o endereço da próxima instrução. Caso uma carga tenha sido efetuada e a `ATT` tenha sido utilizada para alterar o fluxo de leitura de memória, o valor selecionado é o do `BaseAddress`.

Output: Este estágio é responsável por fornecer ao processador a instrução solicitada, assim que ela for lida da memória e separada em *codewords* através dos estágios anteriores. Os módulos desse estágio são:

Apply Mask **(`APM`):** É o mesmo módulo APM utilizado pelo descompressor para MIPS (Seção 4.4). Ele é responsável por aplicar uma máscara na `codeword` de forma a descartar os bits não utilizados pela classe indicada pelo campo `prefix`.

Instruction Table **(`IT`):** É a tabela de instruções que foi descrita em detalhes na Seção 4.3. Ela é responsável por decodificar as *codewords* em instruções. No caso de *codewords* que representam instruções completas (34 bits), o resultado do acesso à IT é descartado e o valor da instrução é fornecido ao processador.

Output Latch **(`OL`):** Armazena o valor de saída para o processador. É utilizado para permitir que o *pipeline* realize mais um ciclo de trabalho enquanto fornece a instrução descomprimida para o processador. O sinal `OutputSource` é usado para indicar se o valor a ser armazenado pelo *latch* virá da IT ou se será a *codeword* diretamente.

***Control Unit*:** É a unidade de controle do descompressor. Além de receber as informações sobre o estado de cada um dos componentes do descompressor, ela também é responsável pela temporização e interface tanto com o processador (e *cache*) quanto com a memória. Durante o processamento de uma instrução comprimida, a unidade de controle é capaz de forçar uma divisão do barramento de dados e controle permitindo que as leituras à memória sejam realizadas de forma transparente ao processador. Também é responsável por efetuar *pre-fetch* da memória.

5.4 Um Exemplo do Funcionamento do Descompressor

Nessa seção, será mostrado um exemplo de como o descompressor funciona através da decodificação de um bloco de instruções comprimidas da memória seguindo a implementação mostrada na Figura 5.6. O código original é mostrado na Tabela 5.9 e o código comprimido na Figura 5.7.

Endereço (hexadecimal)	Instrução	*Codeword* (binário)	Prefixo (binário)
60000000_h	`sethi 281, %o0`	001011101110_b	10_b
60000004_h	`or %o0, 184, %o0`	01011101101_b	01_b
60000008_h	`call 92113`	$01000000000000101100111111010001_b$	11_b
$6000000C_h$	`nop`	001110_b	00_b

Tabela 5.9: Exemplo de código a ser comprimido.

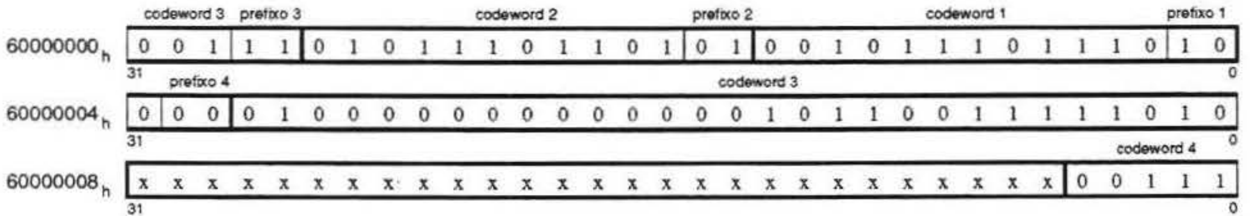


Figura 5.7: Trecho de código comprimido na memória.

As páginas a seguir mostram um exemplo do funcionamento do descompressor quando o fluxo de execução do processador for transferido para o endereço 60000000_h para a leitura das 4 instruções comprimidas:

1. O processador ativa os sinais de controle solicitando uma leitura do endereço 60000000_h. A unidade de controle observa o pedido de leitura para o banco 4 de memória, indicando que se trata de um acesso à região de código comprimido; A unidade de controle então ativa o sinal de `Load` do `RAL`, define que o sinal `MARSource` selecione a saída da `ATT` e também ativa o sinal de `Load` do registrador de *pipeline* `IF`. Nenhum dos outros registradores de *pipeline* carregam dados nesse ciclo;

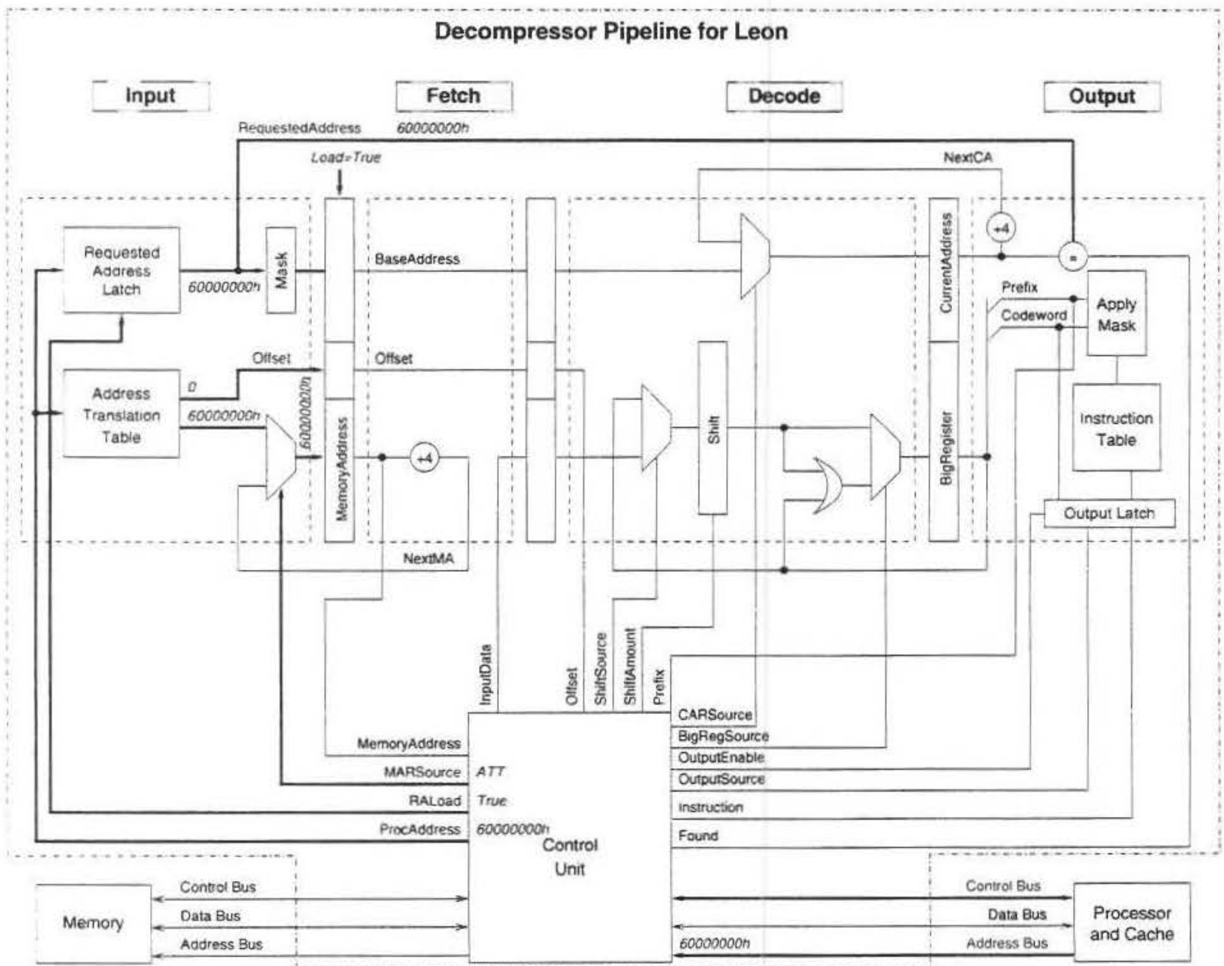


Figura 5.8: Exemplo do IBC para SPARC: Passo 1.

2. No próximo ciclo, o sinal de `Found` do estágio `Output` é observado para determinar se a instrução solicitada encontra-se em decodificação. Como ela não está disponível, tem-se início um ciclo de leitura da memória, com o `MAR` fornecendo o endereço de busca na memória e o resultado da leitura sendo transferido através do sinal `InputData` que será carregado no registrador de *pipeline* `FD`. A leitura da memória pode demorar vários ciclos;

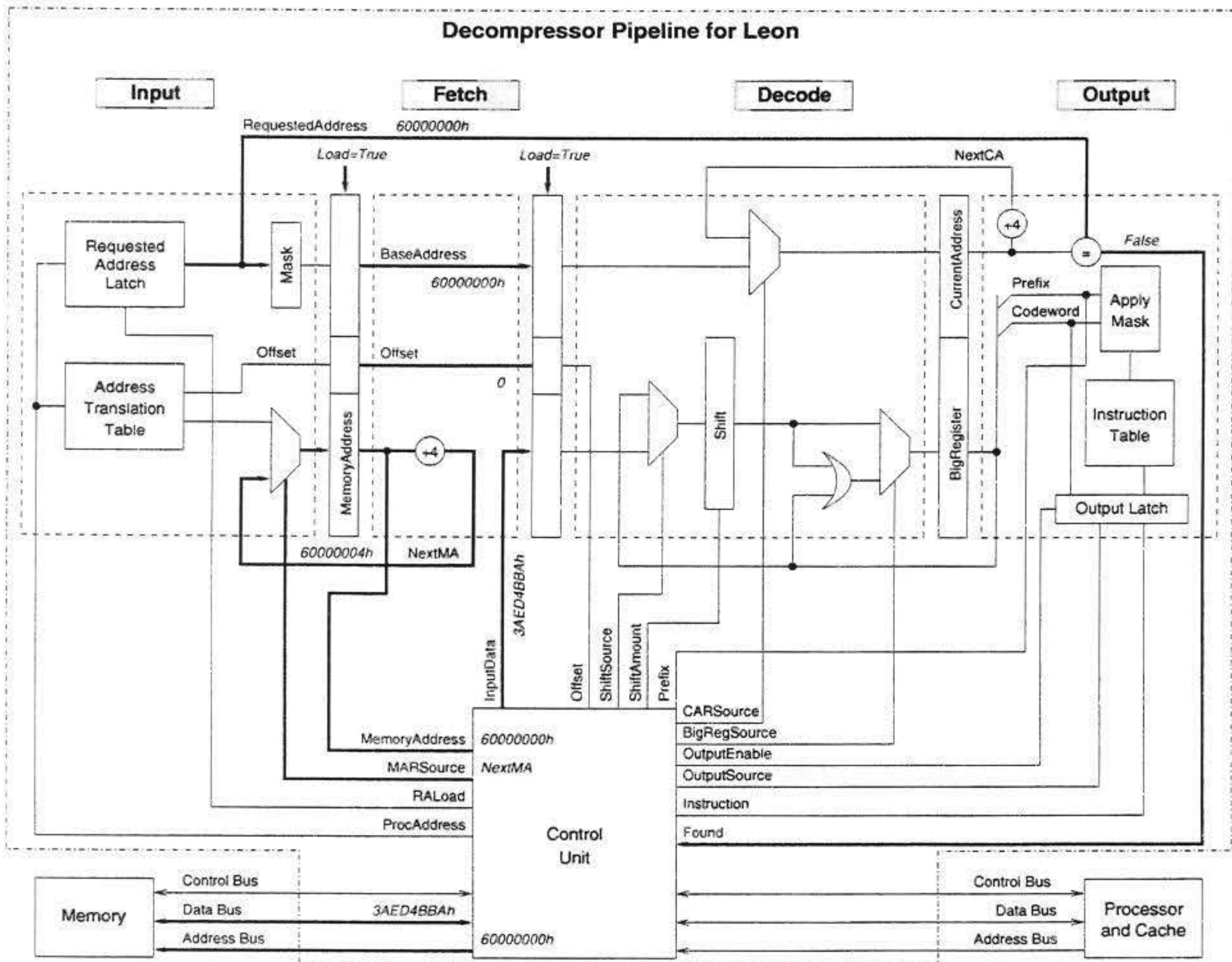


Figura 5.9: Exemplo do IBC para SPARC: Passo 2.

3. No ciclo seguinte ao término da leitura da memória, o sinal `Found` volta a ser observado, e como não está ativo ainda (a instrução solicitada não está no estágio `Output`), o procedimento de descompressão continua. O valor do sinal `ShiftSource` permite a passagem da palavra lida da memória (`InputData`) para o `Shift`, que recebe 0 como `ShiftAmount` (esse valor foi gerado pela `ATT` e foi transferido através do sinal `Offset`). O sinal `BigRegSource` permite a passagem do valor do `Shift` para o registrador `BigRegister` e o sinal `CARSource` permite a passagem do sinal `BaseAddress` para o registrador `CurrentAddress`. O sinal de `Load` do registrador de *pipeline* `DO` é ativado para armazenar os dados processados;

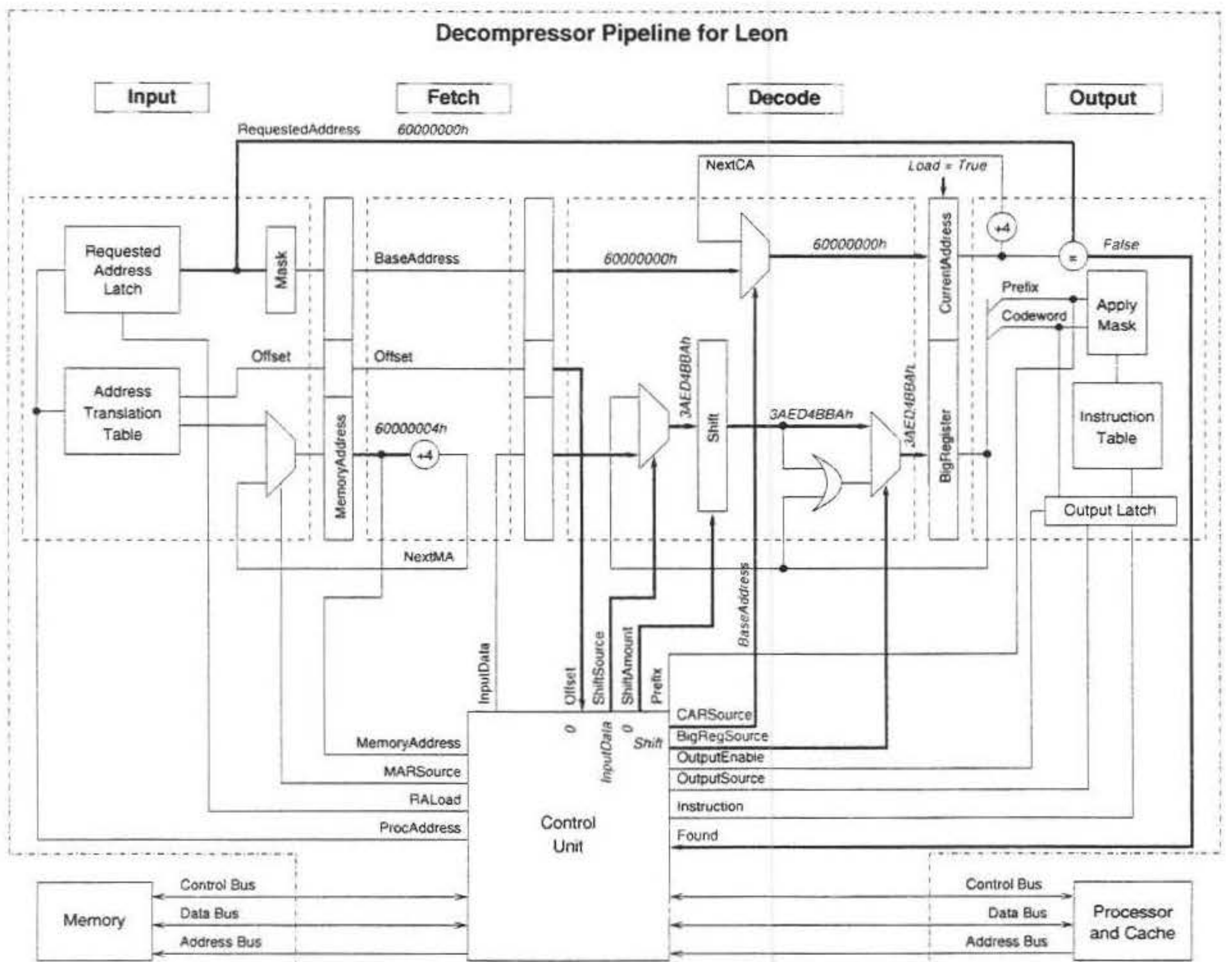


Figura 5.10: Exemplo do IBC para SPARC: Passo 3.

4. Após a palavra lida da memória chegar ao registrador `BigRegister` (que passa a conter 32 bits), a unidade de controle detecta o prefixo 10_b e verifica que existem bits suficientes para decodificar uma instrução. A *codeword* é utilizada como índice na IT e o sinal `OutputSource` força a carga do valor da IT. Neste mesmo momento, os sinais `ShiftSource`, `ShiftAmount`, `CARSource` e `BigRegSource` são ativados para efetuar um deslocamento de 14 bits, produzindo um novo valor de 18 bits para o `BigRegister`. Ainda nesse ciclo, a unidade de controle inicia a resposta ao processador;

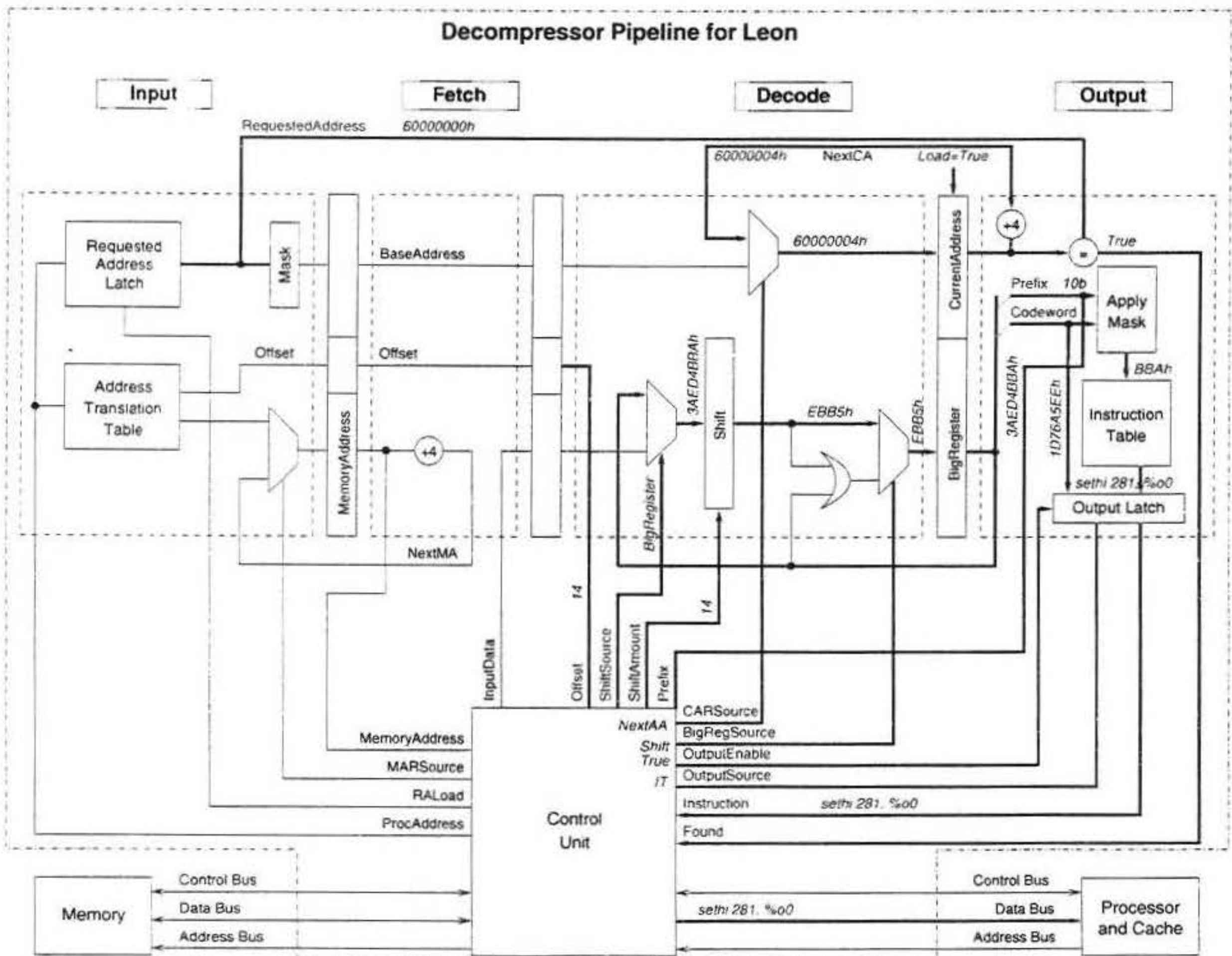


Figura 5.11: Exemplo do IBC para SPARC: Passo 4.

5. O processador solicita agora a instrução do endereço 60000004_h. O descompressor novamente ativa o sinal de Load do RAL, e aguarda o final do ciclo para estabilizar o sinal de entrada;

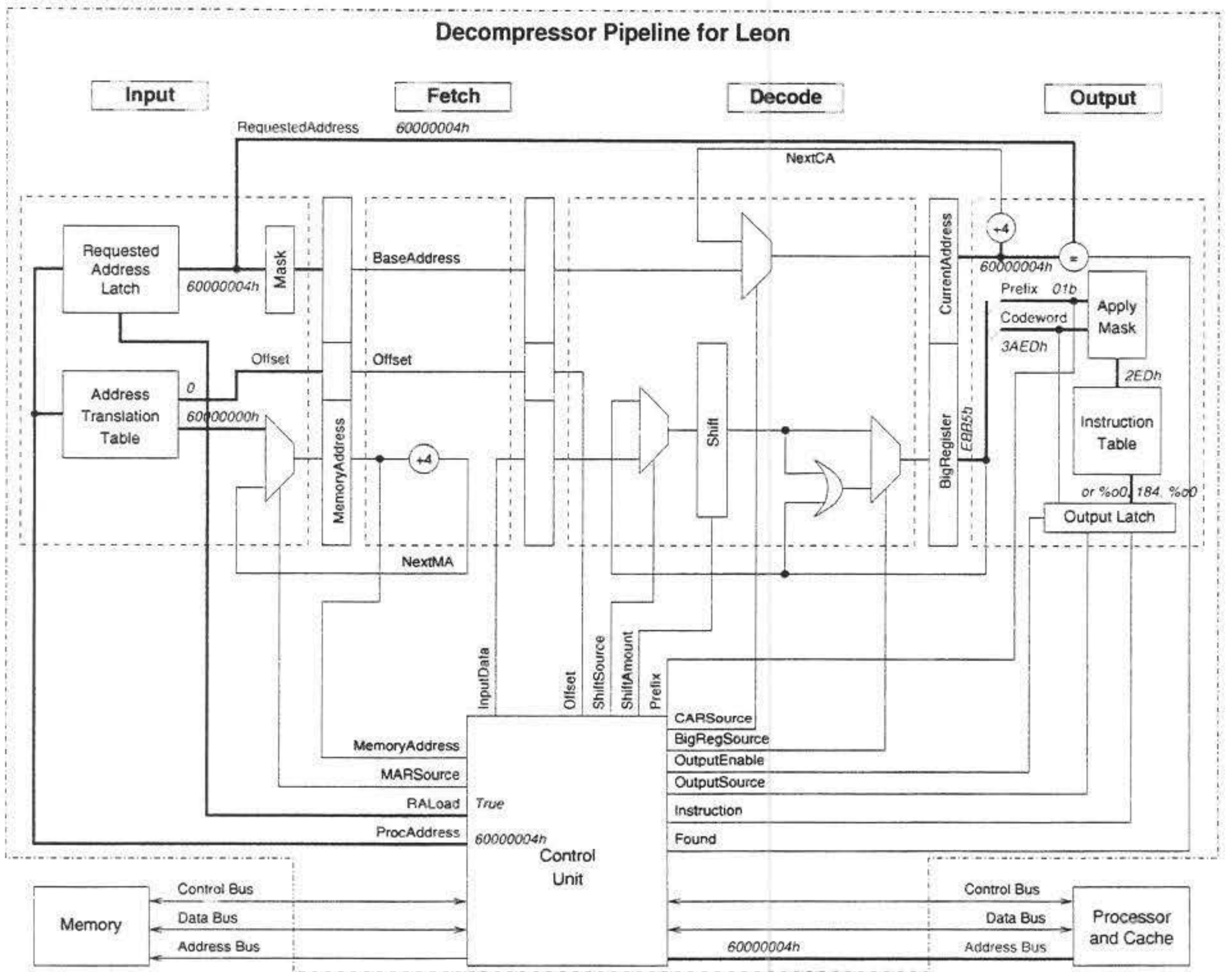


Figura 5.12: Exemplo do IBC para SPARC: Passo 5.

6. No ciclo seguinte, o sinal `Found` é ativado pelo fato da instrução solicitada estar no estágio `Output`. Isso faz com que a unidade de controle já comece a responder ao processador. Ao mesmo tempo, uma leitura da memória é iniciada para transferir a palavra do endereço 60000004_h para o sinal `InputData` e também um deslocamento de 13 bits é efetuado no estágio `Decode`, fazendo com que restem apenas 5 bits disponíveis no `BigRegister`;

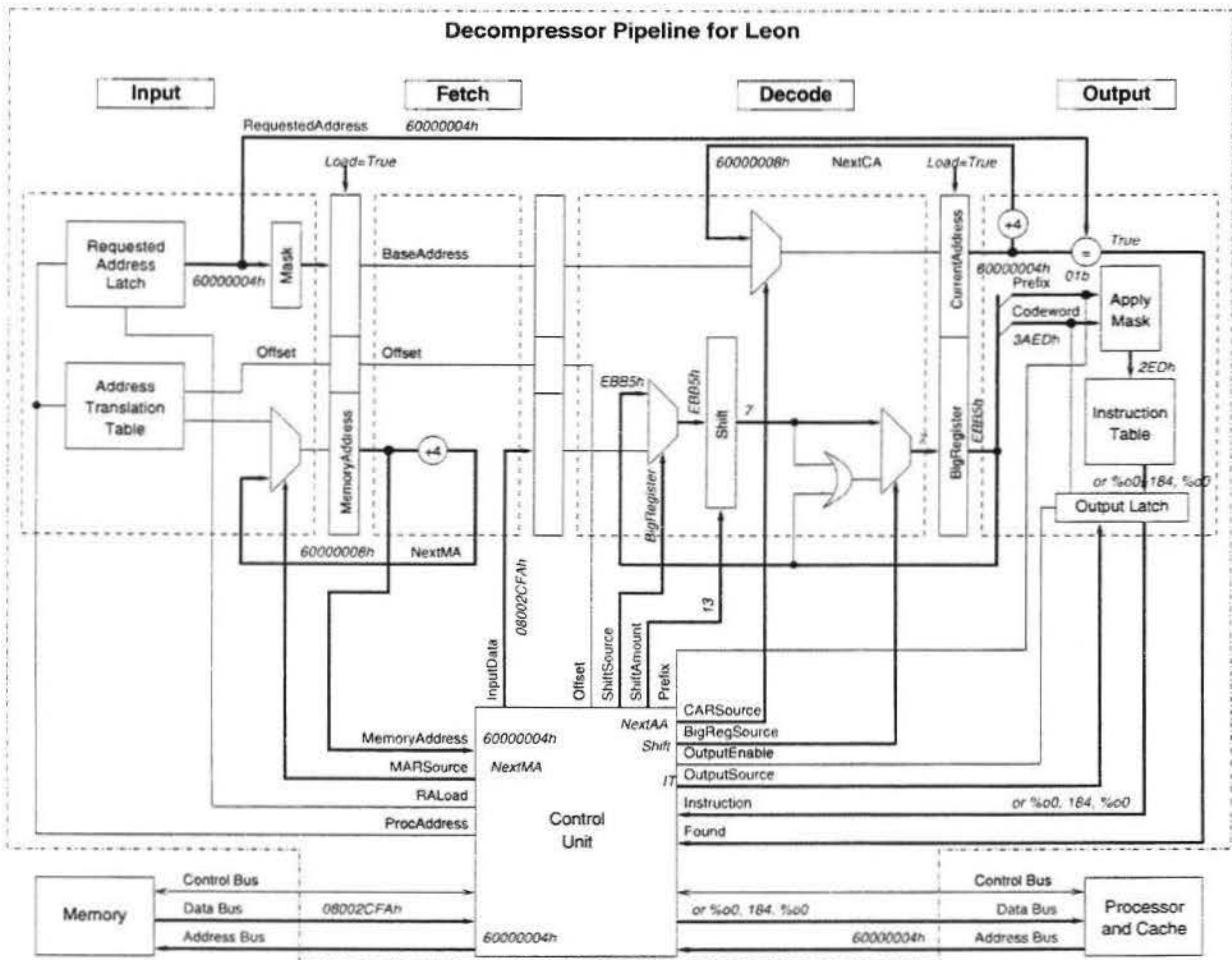


Figura 5.13: Exemplo do IBC para SPARC: Passo 6.

7. A unidade de controle ainda realiza um ciclo interno para agrupar a palavra lida da memória com os 5 bits disponíveis internamente. Isso faz com que o `BigRegister` fique com 37 bits e a instrução do endereço 60000008_h fique no estágio `Output`;

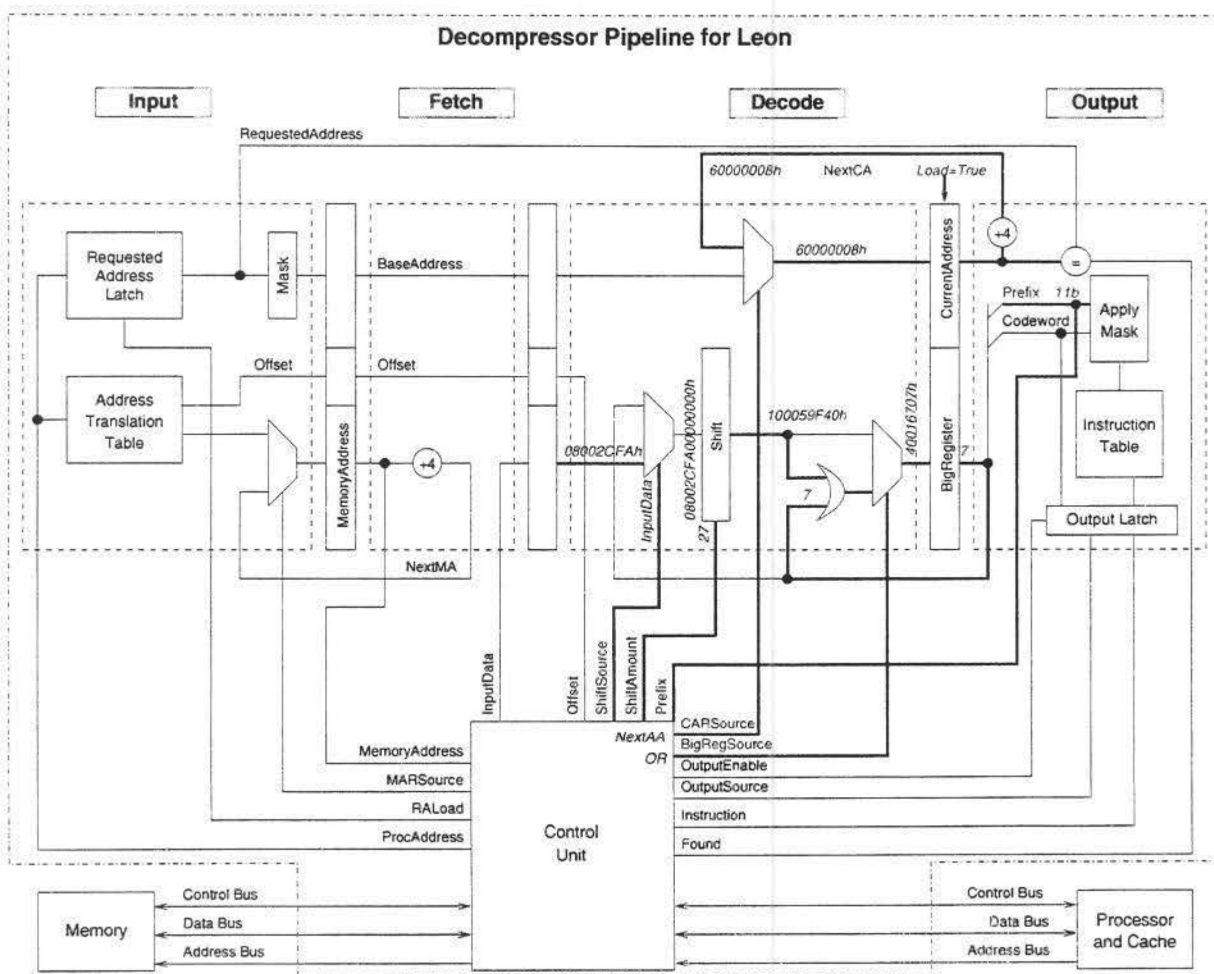


Figura 5.14: Exemplo do IBC para SPARC: Passo 7.

8. A próxima leitura do processador novamente encontra a instrução solicitada no estágio `Output`, o que faz com que o descompressor responda imediatamente. Só que nesse caso, como a instrução está codificada diretamente na *codeword*, o sinal `OutputSource` selecionará a *codeword* como entrada do `OutputLatch`. O deslocamento efetuado no estágio `Decode` será de 34 bits, restando então 3 bits válidos a serem carregados no `BigRegister` no próximo ciclo

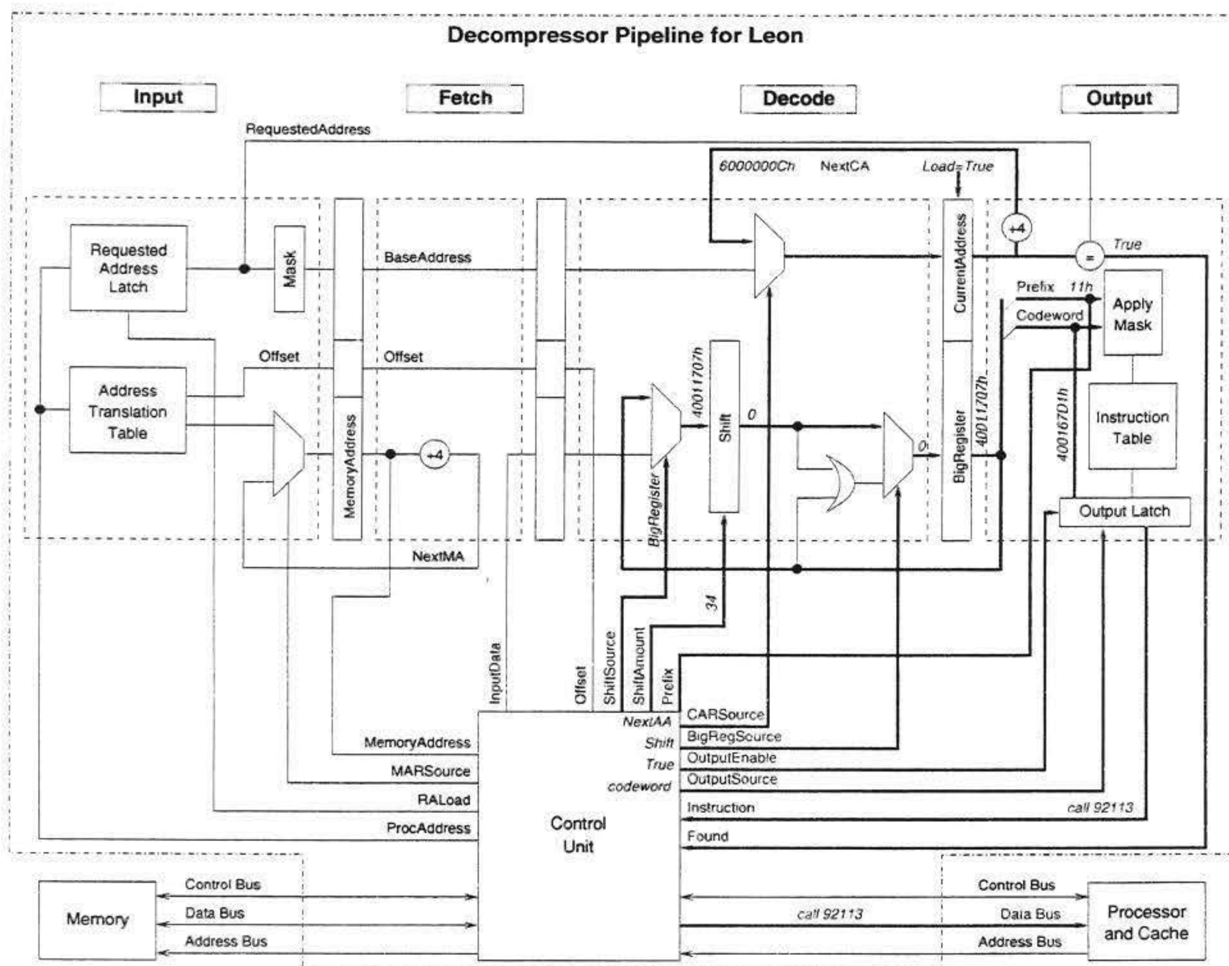


Figura 5.15: Exemplo do IBC para SPARC: Passo 8.

9. A descompressão continua quando o processador solicita o próximo endereço ($6000000C_h$). Desta vez, o descompressor tem que efetuar uma leitura da memória e agrupar a palavra lida com os 3 bits não processados internamente, para ser capaz de fornecer a próxima instrução para o processador.

Caso a transferência inicial de fluxo fosse para o endereço $6000000C_h$ (que não é o endereço da primeira instrução de uma *cache-line* base) ao invés do endereço 60000000_h, todos os passos anteriores teriam que ser executados pois a política de preenchimento da *cache* exigiria todas as instruções para preencher uma *cache-line*.

5.5 Análise de Desempenho do Descompressor

Dos programas do *benchmark* Leon, três foram executados no *kit* de desenvolvimento: `hello`, `c-irq` e `stanford`. Todas as execuções foram baseadas em ITs de 256 instruções. O `hello` foi utilizado nos testes iniciais durante a estabilização do modelo do descompressor. A execução do `c-irq` serviu para comprovar a eficácia da `ATT` e a correta transferência do fluxo do programa, mesmo em casos de interrupções geradas por hardware (o `c-irq` simula interrupções de hardware através da escrita em registradores internos do Leon). Para o programa `stanford`, os resultados de cada uma de suas rotinas estão mostrados na Tabela 5.10 (as colunas de tempo da tabela indicam o número de ciclos de um dos temporizadores internos do Leon). Os resultado de `Mm` para o código comprimido e o de FFT para código não comprimido foram afetados por *overflow* no contador utilizado para armazenar o tempo e não foram mostrados na tabela. A coluna *Comparação* da tabela mostra o resultado da divisão do tempo de execução do programa comprimido pelo tempo de execução do programa não comprimido. Os resultados mostram um bom desempenho do hardware descompressor, tornando a execução apenas 5,89% mais lenta em média. Um fator que deve ser considerado no desempenho é que o tempo de acesso à memória do *kit* de

desenvolvimento (10ns) é muito menor que o período do *clock* da FPGA (120ns) e, como em circuitos reais as memórias são mais lentas que os processadores, pode-se esperar uma melhora no desempenho do descompressor quando implementado num circuito mais rápido em relação à memória. Outro fator que pode melhorar o desempenho do descompressor é a utilização de uma IT maior, fazendo com que um menor número de leituras de memória seja necessário em média para fornecer as instruções para o processador.

Programa	Tempo (não comprimido)	Tempo (comprimido)	Comparação
Perm	150	150	100,00%
Towers	217	234	107,83%
Queens	150	166	110,67%
Intmm	516	517	100,19%
Mm	9534	-	-
Puzzle	1516	1500	98,94%
Quick	184	183	99,46%
Bubble	250	267	106,80%
Tree	1433	1766	123,24%
FFT	-	9384	-
		Média	105,89%

Tabela 5.10: Resultados do programa `stanford`

Os programas `paranoia` e `dhry` não foram executados por não ter sido possível contornar a alocação feita pelo *gcc* da pilha e do *heap* na área de código do programa. Esta característica do *gcc* é incomum e não recomendável pela prática usual em projetos de compiladores. Por serem mais simples, os outros programas não foram afetados por este problema.

5.6 Comparação entre IBC e Outros Métodos

A Tabela 5.11 mostra como fica o quadro comparativo da Tabela 4.4 após a inclusão dos métodos PBC, TBC e IBC.

Seção	Modelo	Arquitetura Base	Razão de Compressão	Tempo de Execução[l]
2.2.1	Thumb[a b]	ARM	55%~70%	70%~120%
	MIPS16[a]	MIPS	60%	n/d
2.2.2	Fraser[c]	SPARC	50%	2000%
	Ernst[c]	SPARC	59%	1260%
	Liao[d]	TMS320C25	84%~88%	115%~117%
	Kirovski[c e]	SPARC	60%	111%
2.2.3	Wolfe[a f]	MIPS	70%	~100%
	Lefurgy[a]	PowerPC, ARM, i386	61%, 66%, 75%	n/d
	Lefurgy[a g h]	SHARC	42%~64%	n/d
	CodePack[a]	PowerPC	60%~65%	90%~110%
	Lekatsas[a i]	MIPS, i386	50%, 65%	n/d
	Lekatsas[a j]	SHARC, ARM	48%, 55%	n/d
	Lekatsas[a]	SPARC	54%	n/d
3.1	TBC	MIPS II, TMS320C25	60,7%,75%	n/d
3.2	PBC	MIPS II	61,3%	n/d
4	IBC[k]	MIPS II	53,6%	88%~184%
5	IBC	SPARC V8	63,83%	105,89%

[a] Implementado em Hardware.
[b] Perda de desempenho com barramento de 32 bits e ganho com barramento de 16 bits.
[c] Totalmente implementado em Software.
[d] Hardware opcional pode acelerar o desempenho.
[e] Números estimados.
[f] Desempenho estimado.
[g] Foi utilizada otimização `-O1` apenas.
[h] O processador SHARC utiliza instruções de 48 bits.
[i] Apenas experimentos sobre a forma de comprimir os programas.
[j] O descompressor é capaz de descomprimir em média 6,84 bits por ciclo.
[k] Desempenho estimado através de simulação de leitura da memória, e não de *traces* de execução do programa.
[l] Considerando como 100% o tempo de execução do programa original sem utilizar compressão.

Tabela 5.11: Quadro comparativo dos métodos de compressão de código

Apenas os métodos que utilizaram a arquitetura SPARC podem ser comparados com essa implementação do IBC. Os métodos de Fraser [32], Ernst [26] geram uma razão de compressão melhor que a do IBC, mas o desempenho é muito ruim nos dois métodos pois eles não foram projetados para descompressão em tempo de execução.

A razão de compressão obtida por Kirovski [39] (60%) supera por pouco a do IBC, mas ela foi obtida por estimativa e além disto não foi informado se o tamanho da *cache* de procedimentos (*pcache*) foi incluído nesta estimativa.

O trabalho de Lekatsas [51] consegue uma razão de compressão (54%) melhor que a do IBC, mas não ficou claro se nessa razão de compressão estão incluídas as tabelas necessárias para a descompressão pelo método. Cabe destacar que sem as tabelas, o IBC obtém uma razão de compressão de 35,77% para os programas do SPEC CINT'95 e 34,81% para o *benchmark* Leon.

Quanto ao desempenho, a implementação do IBC para SPARC supera o resultado de Kirovski (estimativa de 111%) mostrando um desempenho muito similar ao do código não comprimido e que pode ser melhorado com configurações de memória mais realistas e também com o aumento do tamanho da IT. O IBC é o primeiro método de compressão com implementação em hardware para a arquitetura SPARC.

Capítulo 6

Conclusões e Trabalhos Futuros

O resultado central deste trabalho foi o desenvolvido de um algoritmo e uma arquitetura de Compressão Baseada em Instruções (IBC) direcionado para arquiteturas RISC. O método foi testado na arquitetura MIPS II obtendo uma razão de compressão média de 53,1% para um conjunto de programas do *benchmark* SPEC CINT'95, e na arquitetura SPARC V8 obtendo uma razão de compressão de 61,82% para o mesmo conjunto de programas do SPEC CINT'95 e 67,41% para um *benchmark* formado por programas de teste do Leon.

A implementação do método IBC para MIPS II produz a melhor razão de compressão total entre os métodos disponíveis na literatura. A solução para SPARC encontra concorrentes com razão de compressão próximas, mas nenhum dos trabalhos cita a implementação de um protótipo funcional para o modelo, como no caso do IBC e por isso mesmo, não fornecem medidas de desempenho real, apenas estimativas. A perda de desempenho de 5,89% em média, embora seja resultado de um *benchmark* apenas, mostra que o método é bastante viável. Outro fato que deve ser destacado é a inexistência de um conjunto de instruções de 16 bits para a arquitetura SPARC, como ocorre com o MIPS16 e o Thumb, fazendo com que o ganho do método seja grande se comparado com as alternativas de mercado.

Conforme comprovado por nossos experimentos, o uso de técnicas de compressão de

código que utilizam uma implementação em hardware mostrou-se viável para a arquitetura SPARC. O descompressor de código pode tornar-se um componente necessário em projetos de sistemas dedicados futuros, permitindo que a arquitetura SPARC ocupe parte do mercado que hoje é dividido entre MIPS16 (MIPS), Thumb (ARM) e CodePack (PowerPC). Com a utilização de técnicas de compressão de código, as arquiteturas RISC conseguem minimizar um de seus maiores problemas, que é a quantidade de memória necessária para armazenar os programas.

Juntamente com o método IBC, foi desenvolvida uma ferramenta de análise de código, com capacidade para leitura e decodificação de arquivos binários executáveis ELF tanto para MIPS quanto para SPARC, compressão do código utilizando os métodos PBC, TBC e IBC e escrita do programa comprimido. Essa ferramenta é capaz de detectar os limites dos blocos básicos, as árvores de expressões e decodificar cada uma das instruções da arquitetura, além de gerar relatórios com estatísticas de compressão. Ela foi desenvolvida como um conjunto de classes em C++, totalizando mais de 15.000 linhas de código.

O modelo do descompressor IBC *pipelined* implementado para SPARC possui mais de 3.500 linhas de código VHDL sintetizável, não considerando neste tamanho o módulo de configuração que é gerado automaticamente pelo compressor. O modelo para o MIPS possui mais de 3.000 linhas de código VHDL sintetizável, também não considerando o tamanho do módulo de configuração.

6.1 Trabalhos Futuros

Embora o método IBC já obtenha uma das melhores razões de compressão para SPARC e a melhor para MIPS (*circa maio 2002*) disponíveis atualmente, alguns aspectos ainda devem ser melhor estudados, podendo vir a serem melhorados futuramente:

- Uma forma melhor de codificação para as instruções que ocorrem apenas uma vez no programa pode levar a uma melhor razão de compressão e também um melhor apro-

veitamento da largura da memória, uma vez que na implementação para SPARC, essas instruções são codificadas com 34 bits, podendo no pior caso exigir 3 leituras da memória para completar a decodificação. Uma alternativa de codificação pode estar na utilização dos bits não utilizados por algumas dessas instruções;

- Uma forma de codificação de duas instruções em uma, aproveitando os bits não utilizados por alguns dos formatos e também tirando proveito dos registradores temporários que não precisam ser codificados explicitamente pode reduzir o número de instruções no programa e como efeito colateral retirar os temporários que tornam duas instruções diferentes sob o ponto de vista do IBC;

- Uma forma direta de resolução de endereços, que não utilize uma tabela pode proporcionar um ganho na razão de compressão do tamanho da ATT (em torno de 3% a 5% da razão de compressão final). O método de Breternitz [15] mostrou-se bastante interessante e pode ser codificado juntamente com o IBC sem muitas modificações no hardware (as alterações maiores devem ser feitas em software);

- O uso de um *buffer* na saída do *pipeline* do descompressor pode permitir o uso de mais bits extras na conversão de endereços (Seção 4.2) e assim diminuir o tamanho da ATT. Esse *buffer* também permitirá o uso de outras políticas de preenchimento da *cache*;

- A inclusão do descompressor dentro do Leon pode permitir o ganho de ciclos extras durante o acesso à memória. Uma vez que o Leon possui registradores em todos os seus sinais de saída, a simples leitura desses sinais diretamente para os registradores do descompressor permitiria o ganho de um ciclo para cada leitura efetuada;

- Alteração no compilador utilizado (*gcc*) para que ele siga corretamente a prática de separação entre regiões de código e de dados e os problemas enfrentados com a pilha e o *heap* possam ser resolvidos definitivamente.

Referências Bibliográficas

[1] *VAX Architecture Handbook*. Digital Equipment Corp., 1979.

[2] *System V Application Binary Interface*. UNIX Press/Prentice-Hall Inc., third edition, 1990.

[3] *System V Application Binary Interface: MIPS processor supplement*. UNIX Press/Prentice-Hall Inc., 1991.

[4] *System V Application Binary Interface, SPARC Architecture Processor Supplement*. UNIX Press/Prentice-Hall Inc, third edition, 1993.

[5] Advanced RISC Machines Ltd. *Advanced Microcontroller Bus Architecture*, 5 1999.

[6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison Wesley, 1988.

[7] Guido Araújo, Paulo Centoducatte, Rodolfo Azevedo, and Ricardo Pannain. Expression tree based algorithms for code compression on embedded RISC architectures. *IEEE Transactions on VLSI Systems*, March 2000.

[8] Guido Araújo, Paulo Centoducatte, Rodolfo Azevedo, and Ricardo Pannain. Expression tree based algorithms for code compression on embedded RISC architectures. Technical Report IC-00-01, Instituto de Computação - UNICAMP, January 2000.

[9] Guido Araújo, Paulo Centoducatte, Mario Côrtes, and Ricardo Pannain. Code compression based on operand factorization. In *Proc. Int'l Symp. on Microarchitecture*, pages 194–201, December 1998.

[10] ARM. *An Introduction to Thumb*. Advanced RISC Machines Ltd., March 1995.

[11] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of the Int'l. Conf. on Software Maintenance*, 1998.

[12] Martin Beneš, Steven M. Nowick, and Andrew Wolfe. A fast asynchronous Huffman decoder for compressed-code embedded processors. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, September 1998.

[13] Martin Beneš, Andrew Wolfe, and Steven M. Nowick. A high-speed asynchronous decompression circuit for embedded processors. In *Proc. Conf. on Advanced Research in VLSI*, September 1997.

[14] Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):520–540, June 1986.

[15] Mauricio Breternitz Jr. and Roger Smith. Enhanced compression techniques to symplify program decompression and execution. In *Proc. Int'l Conf. on Computer Design*, pages 170–176, October 1997.

[16] John D. Bunda, Donald S. Fussell, Roy M. Jenevein, and W. C. Athas. 16-bit vs. 32-bit instructions for pipelined microprocessors. Technical Report TR-92-39, The University of Texas at Austin, Department of Computer Sciences, 1992.

[17] Paulo Centoducatte, Ricardo Pannain, and Guido Araújo. Compressed code execution on DSP architectures. In *Proc. ACM/IEEE Int'l Symp. on System Synthesis*, November 1999.

[18] Paulo Cesar Centoducatte. *Compressão de Programas Usando Árvores de Expressão*. PhD thesis, Instituto de Computação, Universidade Estadual de Campinas, 1999.

[19] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java bytecode compression for embedded systems. Technical Report No 1213, Institut de Recherche en Informatique et Systemes Aleatoires, 1998.

[20] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. Conf. on Programming Languages Design and Implementation*, May 1999.

[21] XESS Corporation. http://www.xess.com.

[22] Saumya Debray, William Evans, and Robert Muth. Compiler techniques for code compression. In *Workshop on Compiler Support for System Software*, May 1999. Note: Preliminary version of article in ACM Transactions on Programming Languages and Systems.

[23] Saumya Debray, William Evans, Robert Muth, and Bjorn de Sutter. Compiler techniques for code compaction. Technical Report TR00-04, Department of Computer Science, The University of Arizona, March 2000.

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

[25] IBM Microeletronics Division. *The PowerPC 405 Core.* International Business Machines (IBM) Corporation, 1998.

[26] Jens Ernst, Christopher W. Fraser, William Evans, Steven Lucco, and Todd A. Proebsting. Code compression. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 358–365, June 1997.

[27] Will Evans and Christopher W. Fraser. Bytecode compression via profiled grammar rewriting. In *Proc. Conf. on Programming Languages Design and Implementation*, June 2001.

[28] M. Franz. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile-object systems. In Jan Vitek and Christian Tschudi, editors, *Mobile Object Systems: Towards the Programmable Internet,*, Lecture Notes in Computer Science, pages 263–276. Springer-Verlag, 1997.

[29] M. Franz and T. Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, December 1997.

[30] Christopher W. Fraser. Automatic inference of models for statistical code compression. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 242–246, May 1999.

[31] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Addison-Wesley, 1995.

[32] Christopher W. Fraser and Todd A. Proebsting. Custom instruction sets for code compression. *Unpublished manuscript.*`http://research.microsoft.com/~toddpro/papers/pldi2.ps`, 1995.

[33] Jiri Gaisler. *The LEON Processor User's Manual.* Gaisler Research, november 2001.

[34] Mark Game and Alan Booker. *CodePack: Code Compression for PowerPC Processors.* International Business Machines (IBM) Corporation, 1998.

[35] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann, 2 edition, 1996.

[36] D.A. Huffman. A method for construction of minimum redundancy codes. In *Proc. of the IEEE*, volume 40, pages 1098–1101, 1952.

[37] IBM. *CodePack: PowerPC Code Compression Utility User's Manual. Version 3.0.* International Business Machines (IBM) Corporation, 1998.

[38] Xilinx Inc. http://www.xilinx.com.

[39] Darko Kirovski, Johnson Kin, and William H. Mangione-Smith. Procedure based program compression. In *Proc. Int'l Symp. on Microarchitecture*, December 1997.

[40] K. Kissell. *MIPS16: High-density MIPS for the Embedded Market.* Silicon Graphics MIPS Group, 1997.

[41] M. Kozuch and A. Wolfe. Compression of embedded system programs. In *Proc. Int'l Conf. on Computer Design*, 1994.

[42] Michael Kozuch and Andrew Wolfe. Performance analysis of the compressed code RISC processor. Technical Report CE-A95-2, Princeton University Computer Engineering, 1995.

[43] Young-Jun Kwon, Danny Parker, and Hyuk Jae Lee. TOE: Instruction set architecture for code size reduction and two operations execution. *Presented at the International Workshop on Compiler and Architecture Support for Embedded Systems. Not published yet.*, October 1999.

[44] Charles Lefurgy. *Efficient Execution of Compressed Programs.* PhD thesis, University of Michigan, June 2000.

[45] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. Improving code density using compression techniques. In *Proc. Int'l Symp. on Microarchitecture*, December 1997.

[46] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. Improving code density using compression techniques. Technical Report CSE-TR-342-97, EECS Department, University of Michigan, 1997.

[47] Charles Lefurgy and Trevor Mudge. Code compression for DSP. Technical Report CSE-TR-380-98, EECS Department, University of Michigan, November 1998.

[48] Charles Lefurgy and Trevor Mudge. Fast software-managed code decompression. *Presented at the International Workshop on Compiler and Architecture Support for Embedded Systems. Not published yet.*, July 1999.

[49] H. Lekatsas, Joerg Henkel, and Wayne Wolf. Arithmetic coding for low power embedded system design. In *Proc. Data Compression Conference*, pages 430–439, 2000.

[50] H. Lekatsas, Joerg Henkel, and Wayne Wolf. Code compression as a variable in hardware/software co-design. In *International Workshop on Hardware/Software Co-Design*, 2000.

[51] H. Lekatsas, Joerg Henkel, and Wayne Wolf. Code compression for low power embedded system design. In *Proc. ACM/IEEE Design Automation Conference*, 2000.

[52] H. Lekatsas and Wayne Wolf. SAMC: A code compression algorithm for embedded processors. *IEEE Transactions on CAD*, 18(12):1689–1701, December 1999.

[53] Haris Lekatsas and Wayne Wolf. Code compression for embedded systems. In *Proc. ACM/IEEE Design Automation Conference*, 1998.

[54] Haris Lekatsas and Wayne Wolf. Random access decompression using binary arithmetic coding. In *Proc. Data Compression Conference*, pages 306–315, March 1999.

[55] A. Lempel and J. Ziv. On the complexity of finite sequences. In *IEEE Transactions on Information Theory*, volume 20, pages 75–81, 1976.

[56] Stan Liao. *Code Generation and Optimization for Embedded Digital Signal Processors*. PhD thesis, Massachusetts Institute of Technology, June 1996.

[57] Stan Liao, S. Devadas, and K Keutzer. Code density optimization for embedded DSP processors using data compression techniques. In *Proc. Conf. on Advanced Research in VLSI*, 1995.

[58] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steve Tjiang, and Albert Wang. Storage assignment to decrease code size. *SIGPLAN Notices*, 30(6):186–195, June 1995.

[59] Advanced RISC Machines Ltd. http://www.arm.com.

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

[61] T. Okuma, H. Tomiyama, A. Inoue, E. Fajar, and H. Yasuura. Instruction encoding techniques for area minimization of instruction rom. In *Proc. ACM/IEEE Int'l Symp. on System Synthesis*, pages 125–130, 1998.

[62] Ricardo Pannain. *Compressão de Código de Programa Usando Fatoração de Operandos*. PhD thesis, Faculdade de Engenharia Elétrica e Computação, Universidade Estadual de Campinas, 1999.

[63] Gaisler Research. http://www.gaisler.com.

[64] J. Runeson, S. Nyström, and J. Sjödin. Code compression techniques for embedded systems. In *Proc. Workshop on Languages, Compilers, and Tools for Embedded Systems*, 2000.

[65] Johan Runeson. Code compression through procedural abstraction before register allocation. Master's thesis, University of Uppsala, March 2000. Note from author: In this version not all experimental results are correct; this will be corrected in a later version.

[66] SPARC International Inc. *SPARC V8 Architecture Manual.*

[67] W. T. Wilner. Burroughs B1700 memory utilization. In *Fall Joint Computer Conference*, pages 579–586, 1972.

[68] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *Proc. Int'l Symp. on Microarchitecture*, 1992.

[69] XESS Corporation. *XSV Board V1.1 Manual*, september 2001.

[70] Xilinx inc. *Virtex Field Programmable Gate Arrays Specification*, 4 2001.

[71] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. In *IEEE Transactions on Information Theory*, volume 23, pages 337–343, 1977.

[72] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. In *IEEE Transactions on Information Theory*, volume 24, pages 530–536, 1978.