Compressão de Programas Usando Árvores de Expressão

Paulo Cesar Centoducatte

Tese de Doutorado

Compressão de Programas Usando Árvores de Expressão

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Paulo Cesar Centoducatte e aprovada pela Banca Examinadora.

Campinas, 6 de março de 2000.

Mario Lúcio Côrtes (Orientador)

Guido Costa Souza de Araujo (Co-orientador)

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

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Centoducatte, Paulo Cesar

C333c Compressão de programas usando árvores de expressão / Paulo Cesar Centoducatte -- Campinas, [S.P. :s.n.], 1999.

Orientador : Mario Lúcio Côrtes

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

1. Arquitetura de computador. 2. Circuitos integrados. 3. VHDL (Linguagem descritiva de hardware). I. Côrtes, Mario Lúcio. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Instituto de Computação
Universidade Estadual de Campinas

Compressão de Programas Usando Árvores de Expressão

Paulo Cesar Centoducatte

Fevereiro de 2000

Banca Examinadora:

- Mario Lúcio Côrtes (Orientador)
- Sergio Bampi
  Instituto de Informática – UFRGS
- Ivanil Sebastião Bonatti
  Faculdade de Engenharia Elétrica e Computação – Unicamp
- Nelson Castro Machado
  Instituto de Computação – Unicamp
- Célio Cardoso Guimarães
  Instituto de Computação – Unicamp
- Furio Damiani (Suplente)
  Faculdade de Engenharia Elétrica e Computação – Unicamp
- Paulo Lício de Geus (Suplente)
  Instituto de Computação – Unicamp

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 14 de fevereiro de 2000, pela Banca Examinadora composta pelos Professores Doutores:

Prof. Dr. Sérgio Bampi
UFRGS

Prof. Dr.Ivanil Sebastião Bonatti
FEEC-UNICAMP

Prof. Dr. Nelson Castro Machado
IC-UNICAMP

Prof. Dr.Célio Cardoso Guimarães
IC -UNICAMP

Prof. Dr. Mário Lúcio Côrtes
IC-UNICAMP

Resumo

A redução no tamanho dos programas tem sido um fator importante no projeto de sistemas embarcados modernos voltados à produção em larga escala. Este problema tem direcionado grandes esforços em projetos de processadores que se utilizam de um conjunto de instruções com formato de tamanho reduzido (ex. ARM Thumb e MIPS16) ou que sejam capazes de executarem códigos comprimidos (ex. CCRP, CodePack, etc). Muitos dos trabalhos publicados na literatura têm sido realizados para arquiteturas RISC. Este trabalho propõe um algoritmo de compressão de programas e uma máquina de descompressão para arquiteturas RISC e DSP. O algoritmo utiliza como símbolos para a compressão as árvores de expressão do programa. Resultados experimentais, baseados em programas do SPECInt95 executando em processador MIPS R4000, mostraram uma razão de compressão média, para os programas, de 27,2% e uma razão de compressão de 60,7% quando a área ocupada pela máquina de descompressão é considerada. Resultados experimentais para programas típicos de aplicações para DSPs, executando em um processador TMS320C25, mostraram uma razão de compressão média, para os programas, de 28% e de 75% quando a área da máquina de descompressão é considerada. As máquinas de descompressão foram sintetizadas usando–se bibliotecas *standard cell* da AMS, para a tecnologia CMOS de $0,6$ μm e 5 *volts*. Simulações da máquina de descompressão mostraram uma freqüência mínima de operação de 90MHz (R4000) e de 130MHz (TMS320C25).

Abstract

Reducing program size has become an important goal in the design of modern embedded systems targeted to mass production. This problem has driven a number of efforts aimed at designing processors with shorter instruction formats (e.g. ARM Thumb and MIPS16), or that are able to execute compressed code (e.g. CCRP, CodePack, etc). Much of the published work has been directed towards RISC architectures. This work proposes a code compression algorithm and a decompression engine for embedded RISC and DSP architectures. In the algorithm, the encoded symbols are the program expression trees. Experimental results, based on SPECInt95 programs running on the MIPS R4000, reveal an average compression ratio of 27.2% to the programs and 60.7% if the area of the decompression engine is considered. Experimental results for typical DSP programs running on the TMS320C25 processor reveal an average compression ratio of 28% to the programs and 75% if the area of the decompression engine is considered. The decompression engines are synthesized using the AMS CMOS standard cell library and a 0.6 μm 5 volts technology. Gate level simulation of the decompression engines reveals minimum operation frequencies of 90MHz (R4000) and 130MHz (TMS320C25).

Agradecimentos

Ao Prof. Dr. Mario Lúcio Côrtes meu agradecimento pelo apoio irrestrito desde o início desta jornada. Agradeço a valiosa orientação, profissionalismo e amizade.

Ao meu co-orientador e amigo Guido Costa Souza de Araújo agradeço pela sua garra e profissionalismo com que se dedicou à este trabalho.

Aos colegas de jornada Ricardo Pannain e Rodolfo Azevedo pela colaboração e amizade.

Ao Instituto de Computação pela oportunidade e apoio constante.

À Fundação de Amparo à Pesquisa do Estado de São Paulo – FAPESP pelo apoio financeiro (processo nº1997/10982-0).

Ao amigo Jorge Stolfi pela companhia em tantas noites no IC, trocas de idéias e pela sua perseverança em tornar o *Fominha* realidade.

À amiga e companheira de sala Anamaria pela amizade e apoio.

Gostaria ainda de agradecer a todos os amigos que direta ou indiretamente contribuíram para a realização deste trabalho: com sugestões técnicas, com uma palavra de incentivo, com demonstração de amizade e pelos valiosos momentos de descontração.

Por último gostaria de agradecer à Rose, Vítor e Gabriel por tudo que representam em minha vida.

Aos meus pais Donato R. Centoducatte e Odette C. G. Centoducatte,
À Roseclea L. O. Melo, Vítor L. Centoducatte e Gabriel L. Centoducatte.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

O avanço na tecnologia CMOS para implementação de circuitos integrados digitais, vem permitindo a integração de milhões de transistores em um único *chip* [2, 10, 61] e com isto a implementação de sistemas completos em um único *chip* [14] capazes de realizar funções muito além da capacidade dos ASICs (*Application Specific Integrated Circuits*) tradicionais. Estes sistemas são tipicamente denominados de sistemas embarcados (*embedded systems*) uma vez que, em geral, fazem parte de um sistema maior e executam funções específicas [17, 68]. Em 1992 foi estimado que o mercado de processadores destinados a sistemas embarcados suplantou o mercado de processadores de propósito geral, ou seja US$ 5.4 bilhões para microcontroladores e DSPs contra US$ 4.9 bilhões para as CPUs de propósito geral [14]. Estes dados não levam em consideração a possível quantidade de CPUs de propósito geral que foram usadas para implementar sistemas embarcados. Fato que tem experimentado um grande avanço nos últimos anos, principalmente o uso de CPUs RISCs (*Reduced Instruction-Set Computer*). Segundo dados mais recentes da International Data Corporation publicados na revista The Economist [21] em 1998 o mercado de sistemas embarcados experimentou um crescimento de 40%.

Os sistemas embarcados têm encontrado aplicações nos mais diversos setores, passando pelo automotivo (controle de freio ABS, navegação), consumo doméstico (televisão,

brinquedos e jogos eletrônicos), telecomunicação (PABX digitais, telefone celular, multiplexadores de canais), medicina (controle de equipamentos de ultrason, tomografia), etc. Devido ao seu uso bastante diversificado, os sistemas embarcados geralmente possuem características específicas da aplicação, porém algumas destas características são comuns à maioria dos sistemas. Em geral, os sistemas embarcados: (a) executam funções dedicadas; (b) devem estar operacionais durante toda a vida útil do sistema ao qual fazem parte; (c) demandam funcionamento correto, uma vez que executam operações de controle e um mal funcionamento pode causar acidentes com graves conseqüências; (d) executam em tempo real e muitas vezes com restrições severas; (e) são implementados parte em *hardware* e parte em *software* com ambos projetados e desenvolvidos especificamente para a aplicação.

O projeto de um sistema embarcado está sujeito a diferentes tipos de restrições que incluem desempenho, tamanho, peso, consumo de energia, confiabilidade, custo e tempo de desenvolvimento (*time-to-market*).

O tempo de desenvolvimento tem sido reduzido com uso de técnicas de projeto de circuito integrado que utilizam linguagens de descrição de *hardware* (HDL — *Hardware Description Languages*), como por exemplo VHDL (*Very High Speed Integrated Circuit Hardware Description Languages*), reuso de descrições de sub-sistemas, síntese automática entre outras técnicas.

Os esforços para a redução de custos têm sido concentrados principalmente na redução do tempo de projeto e na redução da área de silício necessária a implementação do sistema. A redução da área além de reduzir o custo final, uma vez que o custo de um *die* é proporcional à quarta potência da área por ele ocupada [31], pode ter como beneficio a redução do consumo de potência e um aumento no desempenho.

Neste sentido, tem-se procurado implementar o processador, a memória e os circuitos específicos que caracterizam um dado sistema embarcado em um único *chip* (SoC —

System-on-a-Chip) que além de reduzir a área de silício tem influência importante no desempenho final do sistema e na confiabilidade uma vez que todos os módulos estão integrados em um único *chip*. A figura 1.1 mostra os componentes típicos de um sistema embarcado implementado em um SoC.

Devido ao grande número de facilidades encontradas nestes sistemas, que são normalmente implementadas em *software*, a área de memória em um SoC dedicada a armazenar o programa tem ocupado, em muitos casos, de 40% a 70% de toda a área de silício do SoC. Desta forma, a redução da área ocupada pelos programas, mantendo suas funcionalidades, tem um impacto considerável no custo final do sistema.

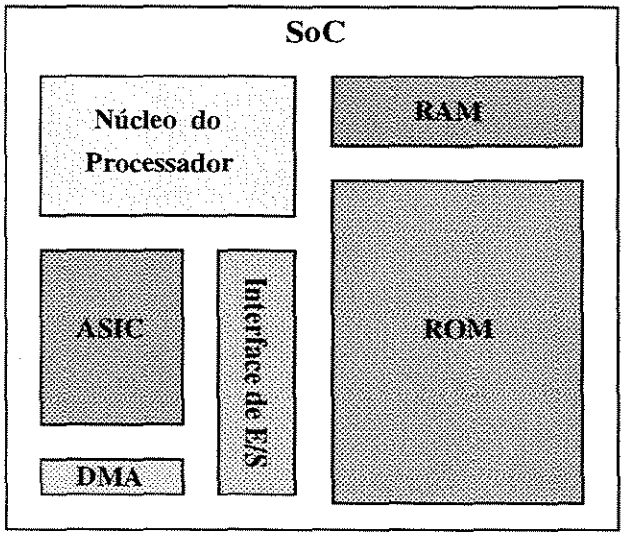

Figura 1.1: Sistema embarcado implementado em um SoC

Duas abordagens têm sido usadas para reduzir a área de memória de programas em sistemas embarcados implementados em um SoC. A primeira é utilizar versões especiais de processadores, principalmente RISCs que utilizam tamanho de palavra reduzido (tipicamente metade do original), e com isso reduzir o tamanho do código do programa. Essa é a abordagem utilizada pelos processadores Thumb e MIPS16 [40]. A segunda abordagem

é implementar sistemas capazes de executar programas comprimidos, com auxilio de um mecanismo que realize a descompressão do código em tempo real. Esta é a abordagem utilizada neste trabalho.

O trabalho aqui apresentado faz parte de um esforço conjunto realizado no Laboratório de Sistemas de Computação (LSC) do Instituto de Computação da Unicamp com objetivo de desenvolver um método de compressão e de descompressão de programas no qual a descompressão é realizada em tempo de execução. O objetivo principal deste esforço é propor técnicas que permitam reduzir a área ocupada por programas em sistemas embarcados implementados em SoCs, sem que isso comprometa o tempo de desenvolvimento (*time-to-market*) e permita que as restrições de desempenho sejam satisfeitas.

Originalmente, este trabalho tinha o objetivo de analisar e propor soluções relativas ao projeto e implementação automatizada do sistema de descompressão e execução do código comprimido pelo método de compressão denominado *Compressão de Programas usando Fatoração de Operandos* desenvolvido por Pannain [60] no LSC. No decorrer deste trabalho, foi proposto um novo método de compressão de programa, baseado no anterior, denominado *Compressão Baseada em Árvores de Expressão* (TBC – *Tree Based Compression*). TBC mostrou-se mais eficiente que o primeiro (ganho médio de 7 a 13 pontos percentuais na razão de compressão dos programas) e que proporciona a implementação de uma máquina de descompressão mais simples e com melhor desempenho (maior freqüência de uso) do que a que seria necessária para o método Fatoração de Operandos. Esta é a contribuição principal deste trabalho.

1.1 Organização deste Trabalho

Sistemas embarcados são usados nos mais diversos segmentos da vida moderna, com uma produção de milhões de unidades anuais e com um crescimento de 40% no ano de 1998.

Estes fatos justificam os esforços que têm sido realizados para a redução do custo final de tais sistemas.

Uma forma de reduzir o custo de um sistema embarcado é reduzindo a área necessária para o armazenamento do código do seu programa mantendo as funcionalidades existentes. Neste capítulo foi dado um panorama do problema e mostrado o enfoque deste trabalho para resolvê–lo.

O capítulo 2 apresenta compressão de dados e de texto e as razões pelas quais os métodos voltados a esse tipo de compressão não podem ser diretamente aplicados à compressão de programas para descompressão em tempo real. Ainda no capítulo 2 são apresentados os principais trabalhos publicados relacionados à compressão de programas.

O capítulo 3 apresenta, com mais detalhes, o método de compressão de programa denominado *Fatoração de Operandos* [60] desenvolvido no Laboratório de Sistemas de Computação do IC/Unicamp, por ser o inspirador direto do método TBC.

O capítulo 4 apresenta o processador MIPS R2000, o método de compressão de programa Baseado em Árvores de Expressão (TBC – *Tree Based Compression*), a aplicação do método TBC em programas executáveis para o processador R2000, a máquina de descompressão e os resultados experimentais obtidos.

No capítulo 5 é apresentado o processador TMS320C25 e o método TBC aplicado a programas executáveis para o processador TMS320C25, que é um processador DSP (*Digital Signal Processor*) muito utilizado em implementações de sistemas embarcados, principalmente nas áreas de telefonia, automotiva e instrumentação. Neste capítulo, também é apresentada a máquina de descompressão e os resultados experimentais obtidos.

O capítulo 6 apresenta os resultados da aplicação do método de compressão TBC a programas executáveis para o processador MIPS R4000 e uma técnica diferente daquela apresentada no capítulo 4 para implementar a máquina de descompressão, que apesar de resultar em uma maior área de silício ocupada torna, para programas grandes, a tarefa

de síntese da máquina de descompressão possível de ser realizada, dada a limitação de memória disponível nas máquinas nas quais os programas de síntese foram executados.

O capítulo 7 apresenta a metodologia usada para a síntese da máquina de descompressão, as dificuldades encontradas durante esse processo e as soluções adotadas para a realização da síntese frente a uma dada realidade (software e máquinas e suas configurações disponíveis).

O capítulo 8 apresenta as considerações finais sobre o trabalho realizado e sugere alguns caminhos de investigações futuras que podem trazer melhorias ao método de compressão TBC.

Capítulo 2

Trabalhos Relacionados

2.1 Introdução

Os primeiros estudos em compressão de programas datam dos anos 70 quando um conjunto de instruções foi projetado para minimizar a utilização de memória que era escassa e cara. Em 1972, projetistas do Borroughs B1700 [66] desenvolveram uma abordagem baseada na utilização das instruções para determinarem o tamanho dos campos das instruções. Instruções mais freqüentes foram associadas a campos menores e as menos freqüentes a campos maiores, usando o método de codificação de Huffman [34].

O primeiro método de compressão de código de programas para uma arquitetura RISC foi proposta em 1992 por Wolfe e Channin [69]. Desde então, diversos métodos vêem sendo propostos.

Este capítulo, em primeiro lugar, apresenta a compressão de dados e de texto (seção 2.2), que apesar de não serem apropriados, formam a base para diversos métodos de compressão de código de programa que tenham como requisito a descompressão em tempo real. Em seguida, apresenta os principais trabalhos relacionados à compressão de código de programas para descompressão em tempo real (seção 2.3).

2.2 Compressão de Dados e de Texto

Compressão é o processo que transforma um conjunto de dados *fonte* em em conjunto de dados com tamanho menor. Futuramente os dados comprimidos deverão ser recuperados, o que é feito pelo processo denominado de descompressão. O processo de compressão e descompressão pode ser irreversível (ou inexato) ou reversível (ou exato). A compressão irreversível é aquela na qual há perda de informação e o dado recuperado não corresponde fielmente ao dado original (*fonte*) e sim a uma aproximação deste. Este tipo de compressão encontra diversas aplicações práticas, porém não pode ser utilizado para a compressão de programas uma vez que estes devem ser recuperados integralmente.

Compressão reversível, também denominada compressão de texto refere–se aos métodos que permitem que o texto comprimido possa ser descomprimido em um texto idêntico ao original. Esta propriedade faz com que compressão de texto possa ser aplicável à compressão de programas.

Compressão de texto é possível devido à redundância de informação existente nos textos. A redundância apresenta–se em diversas formas, tais como: na repetição mais freqüente de certos símbolos, na ocorrência repetida de um dado padrão de símbolos, na forma de codificação dos símbolos, etc [11, 18].

Os métodos de compressão de texto, de uma forma geral, são classificados em duas categorias: estatístico e dicionário [11].

Os métodos de compressão baseados em estatística usam a freqüência dos símbolos para a escolha da nova codificação, atribuindo um código menor para os símbolos mais freqüentes e um maior para os símbolos menos freqüentes. A codificação de um símbolo no processo de compressão é denominado de *codeword*. O método de Huffman para compressão de texto [34] é um exemplo bem conhecido desta categoria.

A compressão baseada em dicionários seleciona padrões de símbolos e os substitui por

uma única *codeword*, que é então usada como índice de entrada do dicionário que contém o padrão de símbolos por ela substituído. A compressão é possível porque em média as *codewords* usam menos *bits* que os padrões de símbolos que elas representam.

O método de compressão de texto (dicionário ou estatístico) a ser utilizado em uma dada aplicação é determinado por diversas características, sendo os fatores mais importantes a eficiência da decodificação e a *razão de compressão*[1] (ou *taxa de compressão*). A eficiência da decodificação é medida em relação ao trabalho necessário para obter–se o texto original a partir do texto comprimido. A razão e a taxa de compressão são definidas como:

$$razão\ de\ compressão\ =\ \frac{(tamanho\ do\ texto\ comprimido)}{(tamanho\ do\ texto\ original)}$$

$$taxa\ de\ compressão\ =\ 1\ -\ (razão\ de\ compressão)$$

Para cada método baseado em dicionário existe um método estatístico que possui razão de compressão igual e que pode ser melhorado, alcançando razão de compressão melhor [11]. Desta forma, métodos estatísticos sempre podem apresentar razões de compressão melhores que os métodos baseados em dicionários, entretanto com um custo computacional adicional para a descompressão. Compressão baseada em dicionário alcança bons resultados em sistemas com restrição de memória e tempo disponível para a descompressão, uma vez que uma *codeword* é expandida em vários símbolos.

Em geral, compressão baseada em dicionários apresenta descompressores mais rápidos e simples, enquanto compressão baseada em estatística produz melhores razões de compressão.

Neste trabalho, procurou–se determinar uma nova técnica de compressão que permita uma implementação eficiente de um descompressor em tempo real. Em um primeiro

[1] Note que quanto menor o valor, melhor será a razão de compressão.

momento pode-se facilmente levar a crer que este problema seja uma extensão natural do problema de compressão de texto, para o qual existe uma literatura extensa [11]. Os algoritmos existentes para compressão de texto formam uma base para a compressão de códigos de programas, porém eles não podem ser aplicados diretamente. Por exemplo, a maioria dos compressores de texto atuais, mais eficientes e que apresentam as melhores razões de compressão são baseados em dicionários (ou um misto entre dicionário e estatístico) e utilizam resultados do trabalho de Lempel e Ziv (LZ) [46] e suas variações [70, 71, 65]. No método de compressão LZ, o dicionário é codificado junto ao texto comprimido e a descompressão só é possível de ser realizada de forma seqüencial e começando no início do texto comprimido. Na descompressão de um programa, que tem que ser realizada em tempo real, essa característica é uma grande restrição para o uso de LZ. Durante a execução de um programa existem desvios (para frente e para trás) o que implica em dizer que o processo de descompressão deve ser possível iniciar-se a partir de cada instrução que seja alvo de alguma instrução de desvio.

A primeira técnica para compressão e descompressão em tempo real de programas foi proposta em 1992 e desde então várias outras têm sido propostas, principalmente nos últimos três anos. O restante deste capítulo é dedicado a apresentar essas técnicas de compressão de programa.

2.3 Compressão de Código de Programas

Os métodos desenvolvidos para compressão de texto não são adequados à compressão de programas que devam ser descomprimidos em tempo real, por dois motivos principais. O primeiro é por não permitirem a descompressão aleatória dos símbolos codificados. O segundo, relaciona-se à quantidade de recursos necessários à sua descompressão, principalmente memória.

A descompressão aleatória é necessária devido a presença de instruções de desvios nos programas. Hennessy e Patterson [31], estudando a freqüência com que ocorrem as instruções em programas, relatam para um conjunto de programas do SPECInt92 (`compress`, `expresso`, `eqnotott`, `gcc` e `li`) que 22% das instruções para o processador 80x86 e 20% para o processador DLX são instruções de desvio (*branch, jump, call e ret*).

Nesta seção são apresentados ainda, trabalhos que têm como objetivo principal a redução do tamanho de programas. Estes trabalhos podem ser classificados em quatro técnicas diferentes. A primeira técnica busca a redução do tamanho do código codificando o conjunto de instruções do processador (seção 2.3.1). A segunda utiliza-se de uma representação intermediária menor que a original e um interpretador é usado para gerar as instruções nativas do processador a partir do código intermediário comprimido (seção 2.3.2) . A terceira técnica utiliza-se de métodos de compressão de texto (Ziv-Lempel, Huffman) para reduzir o tamanho do programa (seção 2.3.3). Estes métodos de compressão não podem ser aplicados ao programa todo devido às instruções de desvios, sendo, então aplicados a trechos do programa (procedimentos, linhas de *cache*). A quarta técnica procura alcançar seu objetivo de uma forma independente do *hardware* utilizando técnicas de otimização em compiladores de modo a produzir códigos menores (seção 2.3.4).

Uma quinta técnica para redução de programa tem como meta principal a redução do tempo de transmissão do programa em uma rede ou de transferência entre a memória e disco. Estas técnicas não são aplicáveis diretamente à redução de código para descompressão em tempo real devido a descompressão ser realizada em todo o programa de uma vez, não reduzindo assim o espaço ocupado pelo mesmo durante a sua execução.

2.3.1 Compressão Codificando o Conjunto Nativo de Instruções

Esta técnica tem como princípio restringir o tamanho das instruções do processador com objetivo de reduzir o tamanho do código gerado. Ela foi utilizada no projeto dos processadores *Thumb* [62, 64] e MIPS16 [40]. *Thumb* e MIPS16 são definidos como um subconjunto das arquiteturas ARM e MIPS-III respectivamente. O subconjunto implementado foi determinado após análise de um grande número de aplicações. As instruções incluídas no subconjunto foram aquelas mais freqüentes, as que não necessitam 32 *bits* para sua codificação e aquelas importantes para que os compiladores gerem código objeto menor. As instruções originais de 32 *bits* foram recodificadas em instruções de 16 *bits*, basicamente reduzindo-se o número de *bits* que codificam os registradores (menos registradores disponíveis) e os imediatos (menor intervalo de valores para os imediatos).

As instruções do *Thumb* e do MIPS16 possuem uma correspondência um-para-um com as instruções da arquitetura original. As instruções de 16 *bits* são buscadas na memória, decodificadas para suas correspondentes de 32 *bits* e passadas ao processador para execução. A redução do campo que codifica os registradores reduziu o número de registradores para 8. Execução condicional não está disponível no *Thumb*, e instruções de ponto flutuante não podem ser utilizadas no MIPS16.

O tamanho do código gerado para o *Thumb* e para o MIPS16 é menor do que o gerado para as arquiteturas originais, porém os novos programas necessitam de mais instruções para executarem a mesma tarefa, o que reduz o seu desempenho. Por exemplo, o código do *Thumb* é 30% a 40% menor (razão de compressão de 70% a 60%) e a execução dos programas é 15% a 20% mais lenta do que no ARM [62].

2.3.2 Compressão por Interpretação de Programas

Esta técnica consiste em descrever os programas executáveis em uma linguagem intermediária (programa comprimido). A execução dos programas descritos nessa linguagem intermediária é realizada por um interpretador (não comprimido) residente. Esta técnica pode ser subdividida em duas classes: execução direta da linguagem de alto nível e conjunto de macro instruções .

Execução Direta da Linguagem de Alto Nível

Esta abordagem usa a linguagem de alto nível como base para definir os novos operadores. Flynn [26] introduz a noção de execução direta da linguagem (DELs)[2]. DEL é uma representação do programa que está entre a linguagem de alto nível e a linguagem de máquina. Os programas em DEL são executados por um interpretador DEL descrito em linguagem de máquina. A representação de um programa em DEL é reduzida, pois os seus operadores são aqueles encontrados tipicamente em linguagens de alto nível; DEL não usa as instruções convencionais *load/store*, referindo–se diretamente aos objetos do programa fonte; todos os operandos e operadores estão alinhados a 1 bit e o tamanho do campo de operandos varia dependendo do número de objetos existentes no escopo corrente. Os campos são de tamanho $log_2 N$ *bits*, onde N é o número de objetos no escopo corrente. Flynn relata que o tamanho dos programas em DEL são de 2.6 a 5.5 vezes menores do que as representações em linguagem de máquina. Flynn não informa qual o tamanho do interpretador.

Conjunto de Macro Instruções

Esta abordagem, apresentada por Fraser [28] cria um conjunto de macro instruções a partir de instruções na representação intermediária dos compiladores (IR). Padrões repetidos nas

[2]Do inglês *Directly Executed Languages.*

árvores IR são transformados em macro instruções no código comprimido. O gerador de código gera *byte code* que são interpretados quando executados. O *overhead* causado pela introdução desses interpretadores é de 4 a 8 KB.

Fraser mostra que este método de compressão reduz em até a metade o tamanho de programas para o processador SPARC, entretanto a velocidade de execução é 20 vezes menor do que na execução do programa original.

Ernst *et al* [22] propuseram BRISC (*Byte-coded RISC*) que é um formato para programas comprimidos e interpretáveis para uma máquina virtual denominada `OmniVM` que adiciona macro instruções ao conjunto de instruções.

A compressão é realizada substituindo-se seqüências de instruções que se repetem por *codewords* (um *byte*) que são mapeadas nas macro instruções. Macro instruções, que só diferem nos argumentos, são representadas pela mesma *codeword* e com argumentos diferentes. As *codewords* são codificadas usando-se um esquema de Markov de primeira ordem. Este esquema permite que mais *opcodes* sejam representados por poucos *bits*, porém a descompressão da instrução corrente é função da instrução anterior e da corrente, tornando o descompressor mais complicado e lento, inviabilizando assim, o uso dessa técnica à descompressão em tempo real. A razão de compressão relatada pelos autores é próxima de 30%.

2.3.3 Compressão Baseada em Freqüência

Compressão baseada em freqüência é a técnica utilizada na maioria dos trabalhos referentes à compressão de programas executáveis para descompressão em tempo real. Ela explora a freqüência com que os símbolos ocorrem no programa para comprimi-los. Os diversos métodos, nesta categoria, têm se diferenciados uns dos outros pela forma como são definidos (escolhidos) os símbolos, como eles são codificados e como é realizado o processo de descompressão.

Compress Code RISC Processor

O *Compress Code* RISC *Processor* (CCRP) [69, 42] utiliza como unidade de compressão uma linha da *cache* de instruções. Em tempo de compilação, as linhas da *cache* são codificadas utilizando–se o método de Huffman. Em tempo de execução as linhas da *cache* comprimidas são lidas da memória, descomprimidas pela máquina de preenchimento de *cache* [3] e colocadas na *cache* de instruções. As instruções lidas, pela CPU, na *cache* possuem os mesmos endereços do que as mesmas no programa original. Desta forma, o processador não necessita de modificações adicionais para permitir a execução do código comprimido. Durante um *cache miss*, a linha de *cache* requerida não está no mesmo endereço original na memória principal. CCRP usa uma tabela para resolver os endereços (LAT, *Line Address Table*) originais (gerados pela CPU) para os endereços do programa comprimido, na memória principal. Acessos freqüentes à LAT reduzem a velocidade de execução do programa comprimido. Para reduzir os acessos à LAT é utilizado um *buffer* auxiliar (CLB, *Cache Line Address Lookaside Buffer*) que armazena as referências mais recentes à *cache*. Para endereços presentes na CLB o mapeamento dos endereços não comprimidos para os comprimidos é realizado sem acesso à LAT tornando o processo mais rápido.

Os autores registram uma razão de compressão de 73% para o MIPS (R2000), porém esse valor não inclui a área de silício destinada à máquina de descompressão de Huffman e não está claro se inclui a LAT e a CLB.

Em trabalhos subseqüentes Benês *et al* [13, 12] descrevem a implementação *full custom* de um descompressor de código de Huffman para o CCRP, que utilizando tecnologia CMOS de $0.8\mu m$ ocupa $0.75mm^2$ e pode descomprimir 560 $Mbits/s$.

[3] Do inglês *Cache Refill Engine*.

Método de Kirovski *et al*

Kirovski *et al* [39] descreve um método de compressão no qual a unidade básica de compressão é um procedimento. Cada procedimento do programa é comprimido utilizando–se o algoritmo de Ziv–Lempel. Um segmento de memória, de tamanho maior ou igual ao maior procedimento do programa original, é reservado para armazenar o procedimento descomprimido em tempo de execução. Durante a execução do programa, em uma chamada de procedimento, um serviço de diretório é usado para localizar o procedimento comprimido, descomprimi–lo e colocá–lo no segmento de memória reservado (*cache de procedimentos*). O mapeamento do espaço de endereços não comprimido é realizado por meio de um diretório de endereços. Neste esquema um pequeno mapa é utilizado, uma entrada por procedimento.

Os autores obtiveram uma razão de compressão de 60% para o processador SPARC, entretanto não fica claro se no cálculo desse valor está contabilizado o *overhead* devido ao diretório, ao software de descompressão e gerenciamento da *cache* de procedimentos.

Usando uma *cache* de procedimentos de 64 KB, foi medida uma penalidade de 166%, em média, no tempo de execução. Uma vantagem deste esquema é que ele não necessita modificações na CPU e requer um mínimo de *hardware* extra (uma RAM *on–chip*) para a *cache* de procedimentos.

Mini Sub–rotinas

Liao *et a* [49, 47] apresentam dois métodos para reduzir o tamanho de código. O primeiro método não utiliza nenhum recurso de *hardware* extra para sua implementação, enquanto o segundo utiliza–se de modificações no *hardware* do sistema para executar a descompressão em tempo real, obtendo com isso uma melhora na razão de compressão.

No primeiro método, um conjunto de instruções do programa que se repete é transformado em uma mini sub–rotina e substituído no programa por uma instrução *call*. A mini

sub–rotina é colocada no programa e terminada por uma instrução de retorno (*ret*) de sub–rotina. O contorno de uma sub–rotina não está limitado à blocos básicos [1] e pode conter instruções de desvios se certos cuidados são tomados. A vantagem deste método consiste em não ser necessário suporte extra em *hardware*. Entretanto, as instruções *call* e *ret* introduzidas reduzem a velocidade de execução e aumentam a razão de compressão. Os autores reportam para este método uma razão de compressão média de 88% para o processador TMS320C25.

O segundo método é idêntico ao primeiro exceto que é introduzida uma modificação no conjunto de instruções original do processador no qual é adicionada uma instrução de chamada de dicionário com dois argumentos (endereço de desvio e número de instruções a serem executadas). Os conjuntos de instruções comuns do programa são armazenados em um dicionário e substituídos, no programa, por uma instrução de chamada de dicionário, não sendo mais necessário o uso da instrução *ret*. Durante a execução, o processador desvia para o endereço do dicionário indicado na instrução, executa tantas instruções quanto for o número de instruções indicado e em seguida retorna ao programa. A vantagem desta solução é que eliminando a instrução *ret* das mini sub–rotinas melhora a razão de compressão. Para este segundo método os autores reportam uma razão de compressão média de 84% para o processador TMS320C25.

Uma desvantagem dos métodos de Liao *et al* é que introduzem muitas instruções de desvio, o que aumenta o tempo de execução dos programas.

Em [48] Liao *et al* propõem modificações nos algoritmos que determinam as mini sub–rotinas, tornando–os mais flexíveis, conseguindo com isso uma razão de compressão de 85% e 82% respectivamente para o primeiro e segundo método.

Dicionário com *Codewords*

Lefurgy *et al* [43] apresentam um método de compressão de código baseado na codificação do programa usando um dicionário de códigos e *codewords*.

Neste método, o programa é analisado e as seqüências de instruções que se repetem freqüentemente são substituídas por uma *codeword*. A seqüência de instruções é então colocada em um dicionário de códigos. Durante a execução do programa as *codewords* são expandidas na seqüência original das instruções que codificam usando–se o dicionário de códigos. As *codewords* durante a fase de compressão são associadas a índices do dicionário. Apenas seqüências freqüentes são comprimidas. Um *bit* de escape é utilizado para diferenciar uma *codeword* de uma instrução não comprimida. Um programa comprimido neste esquema é composto por *codewords* intercaladas por instruções não comprimidas e por um dicionário de instruções.

As *codewords* não são codificadas por um método de tamanho variável (como por exemplo em Huffman) por ser cara a descompressão nestes métodos. Utilizando um método de *codeword* de tamanho fixo foi encontrada uma razão de compressão de 68% e de 81% quando o tamanho da *codeword* é 2 e 4 *bytes* respectivamente. Para melhorar essa razão de compressão foi introduzido um método misto, no qual as *codewords* podem ter tamanho de 8, 12 e 16 *bits*. Com tal esquema a razão de compressão encontrada foi de 61%, 66% e 74% para os processadores PowerPC, ARM e i386 respectivamente. Não está claro se foi incluído no cálculo da razão de compressão o tamanho do dicionário.

No método proposto por Lefurgy *et al* as instruções de desvios relativos não são comprimidas, uma vez que sua codificação pode alterar os endereços–alvo, o que implicaria em uma recodificação, tornando o problema NP–completo [43]. Os desvios indiretos não representam problema uma vez que os endereços–alvo estão em registradores, apenas as palavras de código necessitam ser reescritas e os valores contidos nos registradores devem

levar em consideração a mudança do endereço alvo. Os desvios diretos são resolvidos por uma tabela que mapeia os endereços originais para os endereços comprimidos. Os endereços–alvo também representam problemas devido aos acessos à memória serem alinhados. A primeira solução, para esse problema, é alinhar todas as instruções alvo como definido pelo processador. Essa solução apesar de simples e de fácil implementação reduz drasticamente a compressão. Uma segunda solução é permitir um alinhamento diferente daquele do processador, o que é possível com a introdução de um *hardware* adicional. Esta última foi a adotada no método de Lefurgy. A área ocupada por este *hardware* adicional não está computada no cálculo da razão de compressão final.

Métodos de Lekatsas

Lekatsas *et al* [44] propuseram dois métodos para compressão de programa, um independente e outro dependente do conjunto de instruções.

O primeiro método, denominado de *Semiadaptative Markov Compression* (SAMC) utiliza uma codificação aritmética binária [67] e o modelo de Markov [18, 16]. A instrução é dividida em campos e cada um é considerado um símbolo a ser codificado. Por exemplo, cada instrução do processador R2000 foi dividida em dois campos de 6 *bits* e quatro campos de 5 *bits*. Um modelo de Markov é então construído para cada campo e em seguida é construída a árvore binária de Markov que é utilizada para a compressão. Esta abordagem é independente do conjunto de instruções, uma vez que os campos são escolhidos sem nenhuma relação semântica.

O segundo método, *Semiadaptative Dictionary Compression* (SADC), utiliza–se de um dicionário para comprimir *opcodes*, *opcodes* e registradores e *opcodes* e imediatos. Aqui também uma instrução é dividida em campos, só que agora a escolha dos campos possui uma relação semântica (*opcodes*, registradores, imediatos e imediatos longos). O dicionário é gerado percorrendo–se o programa e criando uma árvore com todos os *opcodes* e suas

freqüências e todos os grupos de 2 e 3 *opcodes* e suas freqüências, em seguida os *opcodes* são inseridos no dicionário, codificado os grupos de instruções adjacentes ou codificado o *opcode* com um registrador ou um imediato específico.

Utilizando, como conjunto de testes o SPECInt95 foi obtido uma razão de compressão de 57% e 75% para o MIPS e PentiumPro respectivamente para o método SADC. Entretanto esses dados não incluem a máquina de descompressão.

Em [45] os autores apresentam um esquema de descompressão baseado em uma máquina de estados finitos, implementada em uma tabela. A razão de compressão média para o processador MIPS, levando-se em consideração uma estimativa para o tamanho da tabela, é de aproximadamente 60%.

CodePack

CodePack [35] é usado pela IBM no PowerPC para sistemas embarcados. Este esquema lembra o CCRP uma vez que a CPU não sofre alterações e um esquema de mapeamento similar à LAT (página 15) é utilizado para mapear o espaço original de endereços no espaço de endereços comprimido. O descompressor, na presença de um *miss* na *cache*-L1, busca os *bytes* correspondentes na memória, descomprime-os e os coloca na *cache*-L1 de instruções nativas do PowerPC.

CodePack divide cada instrução do PowerPC em dois símbolos de 16 *bits* (meia-palavra) que são codificados em *codewords* de tamanho variável de 2 a 11 *bits*. As meias-palavras possuem distribuição de freqüências muito diferentes, desta forma as mais freqüentes são codificadas com *codewords* menores e as menos freqüentes com *codewords* maiores. As *codewords* são divididas em dois campos: um campo de *tag* com 2 ou 3 *bits* que informa o tamanho da *codeword* e o outro campo é o índice do dicionário. As meias-palavras com valor zero são codificadas com 2 *bits* de *tag* e sem índice por ocorrerem muito freqüentemente. O dicionário é personalizado para cada programa e carregado na

memória em tempo de execução. Instruções pouco freqüentes são codificadas com *tags* de 3 *bits*, identificando–as. O campo seguinte é a própria instrução e não um índice para o dicionário. As instruções são combinadas em grupos de 16 instruções, formando um bloco de compressão e são descomprimidas em bloco, preenchendo completamente uma linha de *cache*.

Esse esquema de compressão resulta em uma razão de compressão média de 60% a 65%, não incluída a área de silício dedicada a máquina de descompressão.

2.3.4 Compressão Usando Técnicas de Otimização de Código

Diferentemente dos trabalhos anteriores, que têm como ponto de partida para a compressão um programa objeto (executável), as técnicas incluídas nesta categoria buscam a compressão de programas por meio da melhoria nos métodos de otimização utilizados por compiladores.

Método de Cooper e McIntosh

Cooper e McIntosh [15] apresentam técnicas baseadas no trabalho de Fraser *et al* [27] para serem aplicadas na fase final do processo de compilação dos programas.

A técnica consiste em aplicar algoritmos de casamento de padrão (*pattern–matching*) identificando seqüências de códigos idênticas e substituindo–as por procedimentos (similar ao método de Liao *et al*) ou *cross–jumping* (rearranjo dos trechos de programas que possuem seu final idênticos de tal forma a escrever uma única vez estes últimos).

Para trechos similares mas não idênticos, uma relocação de registradores é realizada, quando possível, na tentativa de torná–los idênticos e aplicar uma das duas técnicas acima.

Os autores reportam que, para processadores RISC, foi encontrado uma razão de compressão de até 85% e em média de 95% para o conjunto de teste utilizado.

Método de Debray *et al*

Debray em et al [19] apresenta uma técnica também baseada no método de Fraser *et al* [27] e é similar ao de Cooper e McIntosh, ou seja, busca substituir trechos idênticos de instruções por uma única ocorrência. A diferença consiste em que, além de determinar os trechos idênticos, instruções são alteradas de posição no código na tentativa de tornar trechos semelhantes em idênticos para assim aplicar o algoritmo de substituição.

Os autores relatam uma razão de compressão de 78% usando para teste programas do SPECInt95 e MediaBench.

2.4 Conclusões

Diferentes métodos aplicados em diferentes níveis têm sido propostos para reduzir o tamanho de programas. A comparação entre tais métodos não é uma tarefa simples, uma vez que cada estudo usa diferentes compiladores, opções de compilação e conjunto de instruções. Não tem muito significado dizer que tal método tem razão de compressão de 55% e aquele outro de 40% sem saber o que realmente está sendo usado como base de comparação. Por exemplo, códigos originais gerados por compiladores com poucos recursos para otimização de código (ou sem as opções ativadas) podem resultar em razões de compressão muito boas (pequenas), enquanto códigos gerados por compiladores com algoritmos sofisticados de otimização podem não alcançar razões de compressão tão boas quando utilizado o mesmo método de compressão de programa.

As técnicas aqui apresentadas representam um conjunto de soluções possíveis para a compressão de programas. Essas técnicas não são necessariamente exclusivas, podendo serem aplicadas de forma combinada de maneira a tirar vantagem dos diferentes tipos de representação.

Flynn (seção 2.3.2) acredita que uma representação em alto nível é ideal para a com-

primir programas pois seu método encontra uma alta taxa de compressão. Entretanto, para aplicações em sistemas embarcados, nos quais há requisitos severos de desempenho a execução do programa de forma interpretada inviabiliza a utilização direta de seu método. O mesmo é verdade para o método apresentado por Fraser e para o de Ernst *et al* (seção 2.3.2).

Os métodos que usam a codificação LZ em seus algoritmos trabalham bem com dados alinhados em *bytes*, tais como ASCII, entretanto os campos das instruções nativas não são alinhadas a *byte* o que faz com que a aplicação desses métodos perca o significado semântico dos campos das instruções perdendo oportunidades de compressão.

Os métodos baseados em procedimentos apresentados por Fraser (seção 2.3.2), Kirovski e Liao (seção 2.3.3) são úteis nos casos onde repetições não são óbvias aos programadores e/ou quando os compiladores não otimizam o suficiente.

Os métodos baseados na melhoria dos algoritmos de otimização dos compiladores (seção 2.3.4) são úteis para gerarem o programa objeto no qual algum dos outros métodos será aplicado.

Os demais métodos encontram uma maior oportunidade de virem a ser aplicados em sistemas embarcados nos quais a descompressão deva ser realizada em tempo real. Vale lembrar aqui que, com exceção daqueles que não utilizam recursos adicionais de *hardware*, à razão de compressão reportada pelos autores deve ser adicionada a parcela relativa à área de silício necessária para implementar tais recursos.

Os métodos de compressão de programas apresentados na literatura mais adequados para o uso em aplicações nas quais é necessária uma descompressão em tempo real usam como símbolos de compressão informações em um *byte* ([43, 44, 45]), linha de *cache* ([69, 42]) ou conjuntos de instruções que se repetem pelo programa ([49, 47]). Informações essas que não possuem nenhuma (ou quase nenhuma) relação direta com a forma como o compilador gera o código executável para os programas por ele compilados ou mesmo

com o formato do conjunto de instruções do processador para o qual o código é gerado.

Este trabalho teve como motivação inicial a escolha de símbolos de compressão que tenham uma relação forte com a forma com que os compiladores geram o código executável e também com o conjunto de instruções do processador alvo, para assim conseguir melhorar a razão de compressão dos programas.

Os próximos capítulos apresentam dois métodos para a escolha dos símbolos de compressão que satisfazem aos requisitos descritos acima e métodos de compressão que os utilizam bem como os resultados obtidos para a compressão de dois conjuntos distintos de programas e três processadores distintos.

Capítulo 3

Método Fatoração de Operandos

Este capítulo apresenta mais detalhadamente o método de compressão de programas para descompressão em tempo real denominado *Fatoração de Operandos* [60], por ser parte integrante do projeto *Compressão de Código de Programas para uso em Sistemas Embarcados* que está sendo desenvolvido no Laboratório de Sistemas de Computação (LSC) do Instituto de Computação da UNICAMP, no qual se originou esse trabalho.

3.1 Fatoração de Operandos

Fatoração de Operandos é um método de compressão de programas que permite a descompressão em tempo real e usa conceitos dos métodos de compressão de texto baseados em estatística e dos métodos baseados em dicionário. Fatoração de Operandos utiliza árvores de expressão como elemento básico para determinar os símbolos usados na compressão. Árvores de expressão são objetos do programa com grande apelo semântico, ou seja, mantêm uma relação forte com as estruturas da linguagem (de alto nível) na qual foi escrito o programa que está sendo comprimido.

Neste método, cada árvore de expressão (*expression-tree*) [1] do programa é isolada e separada em duas componentes: a primeira denominada padrão de árvore (*tree-pattern*) formada pelas instruções da árvore de expressão e a segunda denominada padrão de ope-

randos (*operand–pattern*) formada pela seqüência dos operandos que ocorrem na árvore de expressão (registradores e imediatos). As árvores de expressão, construídas como definido em [1], não contêm instruções de desvios e não atravessam blocos básicos. Instruções de desvio formam uma árvore em si.

O objetivo em se usar árvores de expressão como elementos básicos de compressão é aproveitar os mecanismos baseados em árvores que são utilizados pelos compiladores na geração do código binário para os programas.

As figuras 3.1 e 3.2[1] mostram uma árvore de expressão e os padrões de árvore e de operandos a ela associados para instruções dos processadores R2000 e TMS320C25 respectivamente. Nas figuras 3.1 e 3.2 em (a) temos a árvore de expressão, em (b) o padrão de árvore resultante da fatoração dos operandos da árvore original e em (c) a lista de operandos associada à árvore de expressão.

```
addiu r4, r4, 1          addiu *, *, *      [r4,r4,1,r1,0,r1,0,r4]
lui   r1, 0              lui   *, *
sw    r1, 0(r4)          sw    *, *(*)

  (a)                        (b)                    (c)
```

Figura 3.1: Fatoração de Árvore de Expressão para o R2000.
(a) Árvore de Expressão; (b) Padrão de árvore; (c) Padrão de Operandos.

Para estudar o impacto do uso desta técnica na compressão de programas foi usado um conjunto de programas do SPECInt95, compilados com o compilador gcc para o processador MIPS R2000 e um conjunto de programas representativos de aplicações que executam em sistemas embarcados e DSPs compilados com o compilador da Texas Instruments para o processador TMS320C25. Em ambas as compilações usou–se a opção

[1]Na figura, parte (a), os símbolos *, + e - fazem parte da sintaxe da linguagem de montagem do TMS320C25 e seu significado será visto na seção 5.2.

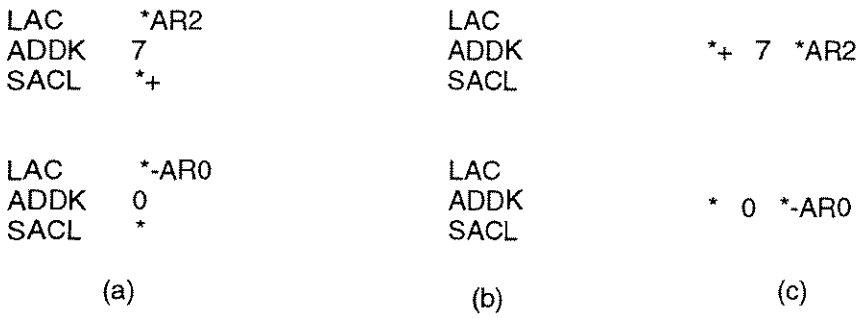

Figura 3.2: Fatoração de Árvore de Expressão para o TMS320C25.
(a) Árvore de Expressão; (b) Padrão de árvore; (c) Padrão de Operandos.

-O2 para otimização de código, visto que esta opção resulta no menor código (número de instruções) para os programas dentre todas as opções de otimização disponíveis nos respectivos compiladores.

O processador R2000 foi utilizado por ser um processador RISC típico que possui diversas características de outros processadores modernos. O TMS320C25 foi também escolhido, por ser um processador largamente utilizado em sistemas embarcados para processamento de sinais.

A análise realizada mostrou que o número de padrões de árvores e de operandos distintos em um programa não é grande e muito menor que o número de árvores de expressão. As tabelas 3.1 e 3.2 apresentam o número de árvores de expressão, o número de padrões de árvores e de operandos para os programas analisados para o R2000 e o TMS320C25 respectivamente.

Note que por exemplo, para o programa gcc, tem–se 311488 árvores de expressão e somente 1547 padrões de árvores distintos, representando menos do que 0,5% de todas as árvores do programa. Isso pode ser explicado devido ao: número limitado de instruções existente para um dado processador; ao tamanho médio reduzido das árvores (possibilitando poucas combinações entre as instruções); e principalmente, pela forma determinista

com que os compiladores geram código para as estruturas das linguagens de alto nível, tais como `if-then-else`, `for` e `while`.

Os padrões de operandos possuem uma distribuição um pouco mais uniforme que as árvores, por exemplo, o gcc possui 311488 seqüências de operandos, que podem ser representados por 41468 padrões distintos, representando 13,3% de todas as seqüências.

Programa	Árvore de Expressão	Padrões de Árvore (%)	Padrões de Operandos (%)
compress	1444	125 (9.0)	731 (51.0)
gcc	311488	1547 (0.5)	41486 (13.3)
go	54651	578 (1.1)	12561 (23.0)
ijpeg	38426	767 (2.0)	9839 (26.0)
li	15761	157 (1.0)	3056 (19.4)
perl	62915	648 (1.0)	11209 (18.0)
vortex	128104	471 (0.4)	16143 (13.0)

Tabela 3.1: Número de padrões distintos de árvores e de operandos em um programa, para o R2000.

Programa	Árvores de Expressão	Padrões de Árvore (%)	Padrões de Operandos (%)
aipint2	1043	90 (8.6)	285 (27.3)
bench	9483	572 (6.0)	2263 (23.9)
gnucrypt	3682	263 (7.1)	778 (21.1)
gzip	10835	582 (5.4)	2354 (21.7)
hill	920	121 (13.2)	279 (30.3)
jpeg	2305	190 (8.2)	563 (24.4)
rx	563	61 (10.8)	114 (20.3)
set	4565	319 (7.0)	1084 (23.7)

Tabela 3.2: Número de padrões distintos de árvores e de operandos em um programa, para o TMS320C25.

A freqüência com que ocorrem cada um dos padrões de árvore e de operandos foi determinada e as listas de padrões (de árvore e de operandos) foram ordenadas em ordem

decrescente de contribuição na cobertura do programa (tamanho vezes freqüência), sendo ainda calculada a percentagem acumulativa para cada um dos padrões. A figura 3.3 mostra a cobertura dos programas pelos padrões de árvores distintos, para o processador R2000. Note que as curvas têm um comportamento exponencial e que 10% dos padrões de árvores são responsáveis pela cobertura de aproximadamente 30% do programa (100% dos padrões de árvores cobrem de 33% a 35% do programa).

A figura 3.4 mostra a cobertura dos programas pelos padrões de operandos distintos, para o processador TMS320C25. Note que as curvas também têm um comportamento exponencial e que 20% dos padrões de operandos cobrem aproximadamente 30% do programa (100% dos padrões de operandos cobrem de 54% a 58% do programa).

As curvas para os padrões de operandos para o processador R2000 e para os padrões de árvores para o processador TMS320C25 (não mostrados no texto) têm comportamento semelhante à sua similar para o processador TMS320C25 e R2000 respectivamente.

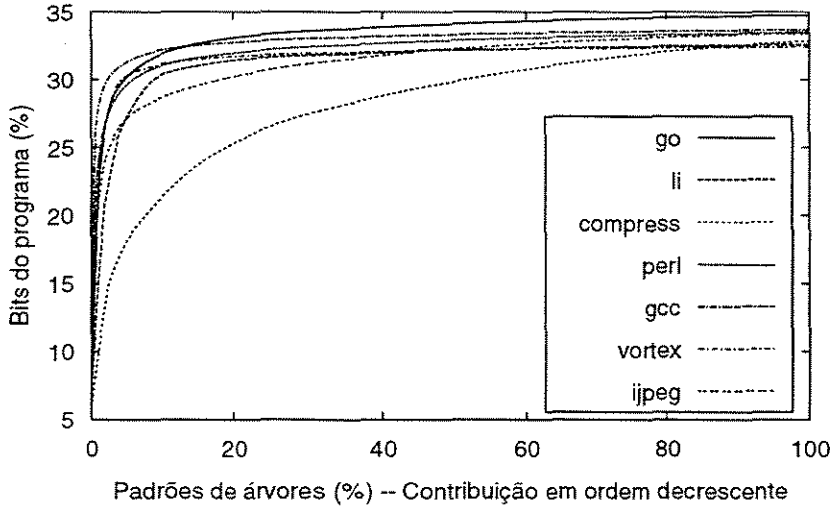

Figura 3.3: Percentagem de bits do programa cobertos pelos padrões de árvores, para o R2000.

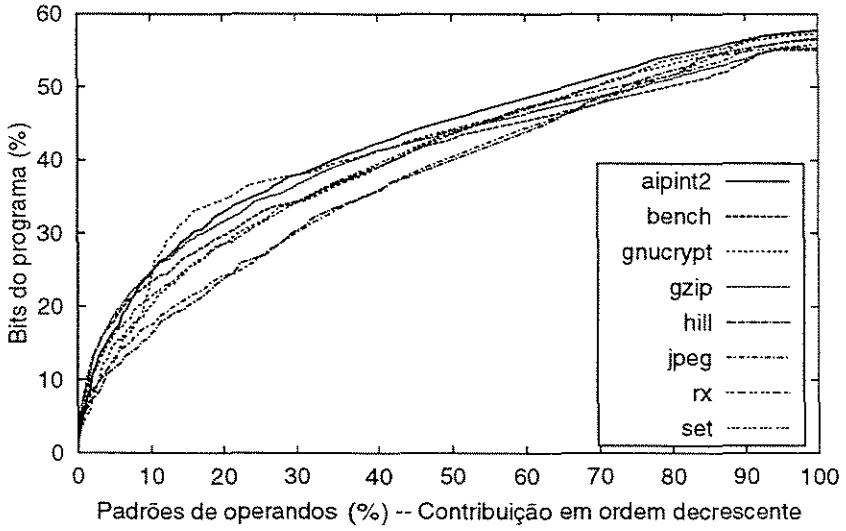

Figura 3.4: Percentagem de bits do programa cobertos pelos padrões de operandos, para o TMS320C25.

Note, nas figuras, um comportamento não uniforme. Tanto a distribuição das árvores quanto a dos operandos apresentam um comportamento exponencial, o que sugere que uma boa razão de compressão deva ser alcançada se forem utilizados algoritmos de compressão baseados em freqüência (estatística).

3.2 Algoritmos de Compressão

Os resultados obtidos mostram que a contribuição dos padrões de árvores e de operandos é não uniforme, o que sugere a aplicação de algoritmos de compressão baseados em freqüência, ou seja, que os padrões sejam codificados com *codewords* de tamanho variável [11]. Padrões com grandes (pequenas) freqüências são codificados usando–se menos (mais) *bits*. Este método de codificação tem como desvantagem o fato que as *codewords* que se referem aos símbolos pouco freqüentes possuem muitos *bits*. A decodificação de *codewords* grandes é mais difícil, mais lenta e custa mais área de silício na máquina de descompressão.

Para evitar *codewords* grandes e difíceis de serem decodificadas, foram realizados diferentes experimentos para determinar uma forma de codificação das *codewords*, com objetivo de se alcançar boas razões de compressão e alto desempenho na descompressão.

Dois métodos de codificação de tamanho variável e um fixo foram investigados, e ainda métodos mistos, onde parte das *codewords* (as mais freqüentes) foram codificados com um método variável e o restante fixo. Isso foi feito para reduzir o tamanho das *codewords* pouco freqüentes visando assim melhorar o desempenho da máquina de descompressão.

Os métodos de tamanho variável escolhidos foram o de Huffman, que sempre apresenta bons resultados de compressão para distribuições não uniforme dos símbolos, e uma variação do VLC (*Variable Lenght Codeword*), método adotado no formato MPEG-2 [29]. A motivação para a utilização do VLC é que apesar de ser um método variável, diferentemente de Huffman, sua *codeword* carrega consigo a informação do seu tamanho, o que torna o processo de identificação da *codeword* muito mais simples no instante da descompressão.

O programa comprimido é montado, substituindo-se cada árvore de expressão por um par de *codewords* <Tp,Op>, onde Tp representa o padrão de árvore e Op o padrão de operandos da árvore de expressão. Tp e Op quando decodificados geram índices para os dicionários de árvore de e de operandos respectivamente.

Um problema com o uso dos métodos de codificação de Huffman e VLC é que o tamanho das *codewords* para os símbolos menos freqüentes podem assumir tamanhos consideráveis (por exemplo maior que uma palavra do processador), o que representa um problema para a implementação da máquina de descompressão tanto no que diz respeito a área de silício ocupada quanto ao desempenho (ter que ler mais de uma palavra de memória para obter uma *codeword*). Limitar o tamanho das *codewords* é um fator importante para a implementação do descompressor. Desta forma, foram realizados experimentos em que um subconjunto dos símbolos (os que mais contribuem na cobertura do programa) foi codificado com Huffman ou VLC e os símbolos restantes (os que menos contribuem)

foram codificados com um formato de codificação de tamanho fixo (codificação binária). Um primeiro experimento foi codificar todos os símbolos com uma codificação fixa e foi-se gradativamente aumentando o tamanho do subconjunto de símbolos codificados com Huffman ou VLC até que todos os símbolos foram codificados desta forma. A figura 3.5 mostra o comportamento da razão de compressão associada aos padrões de árvores quando todos os símbolos foram codificados com a codificação binária até todos símbolos serem codificados com codificação Huffman, para o processador R2000.

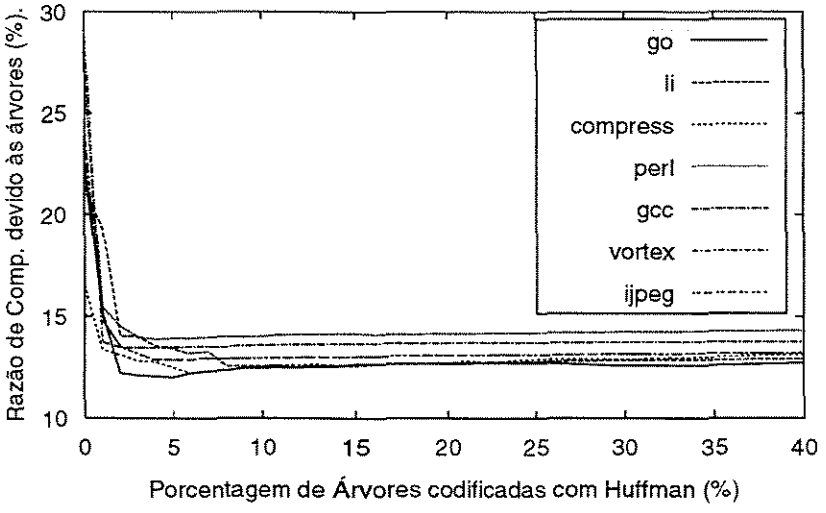

Figura 3.5: Razão de Compressão dos padrões de árvores usando Huffman-fixo e o processador R2000

A figura 3.6 mostra o comportamento da razão de compressão associado aos padrões de operandos quando todos os símbolos foram codificados com a codificação binária até todos símbolos serem codificados com codificação VLC, para o processador TMS320C25.

O mesmo experimento foi realizado para todas as combinações pertinentes, codificação fixa, codificação Huffman ou VLC, padrões de árvores ou de operandos e processador R2000 ou TMS320C25. O resultado final mostrou um comportamento semelhante para os dois

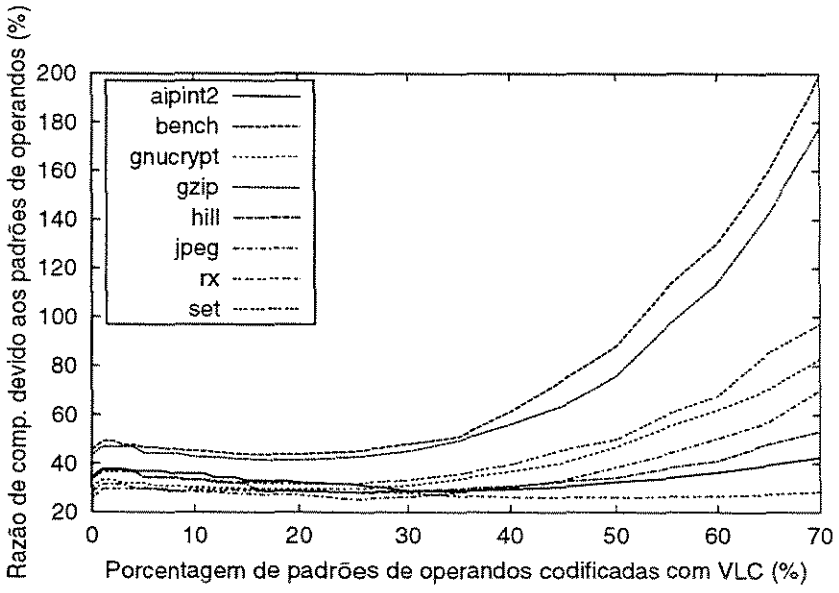

Figura 3.6: Razão de Compressão dos padrões de operandos usando VLC e o processador TMS320C25

processadores para uma mesma combinação padrão–método de codificação.

A tabela 3.3 mostra para o R2000 a razão média de compressão quando os métodos de codificação para as árvores e para os operandos são combinados. Esta tabela nos proporciona um espaço de escolha de soluções de compromisso entre razão de compressão e desempenho da máquina de descompressão. O pior caso de razão de compressão é 57%, quando todas as árvores e operandos são codificados com codificação fixa, e o melhor caso é 35% quando ambos os padrões são todos codificados com Huffman. A desvantagem dessa escolha é que há *codewords* grandes que não carregam consigo informação a respeito do seu tamanho. Usando VLC resulta em uma razão de compressão de 41%, 5,7 pontos percentuais maior que Huffman puro. Este é o preço pago para uma descompressão mais eficiente.

As figuras 3.7 e 3.8 mostram as razões de compressão para o R2000 e TMS320C25 quando uma estimativa do tamanho da máquina de descompressão é considerada. Os *overheads* associados à RGEN, IMD e TPD para o R2000 são devidos ao gerador de registradores e

Método de Codificação	Fixo	Huffman	Huffman-Fixo	VLC-Fixo
Fixo	57.7	46.2	48.1	50.5
Huffman	45.4	35.0	35.8	38.2
Huffman-Fixo	46.0	36.6	37.4	39.8
VLC-Fixo	47.9	37.5	38.3	40.7

Tabela 3.3: Razão de Compressão média após combinar os métodos de codificação para os padrões das árvores (linhas) e dos operandos (colunas) para o processador R2000.

aos dicionários de imediatos e de padrões de árvore da máquina de descompressão (ver seção 4.4). Os *overheads* associados à INGEN, ARGEN e IMGEN são devidos aos análogos para a máquina de descompressão para o TMS320C25.

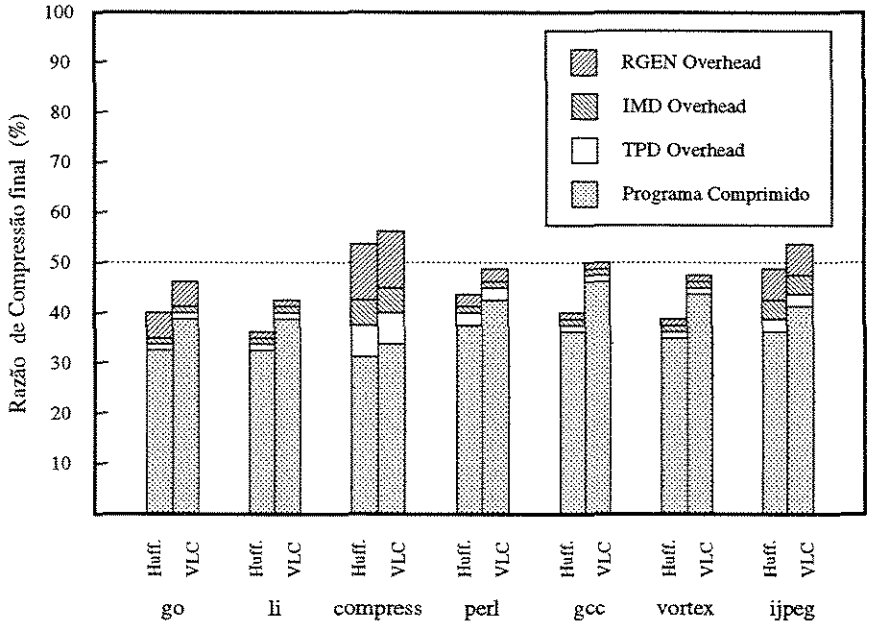

Figura 3.7: Razão de compressão final para o R2000 considerando estimativas para os *overheads*.

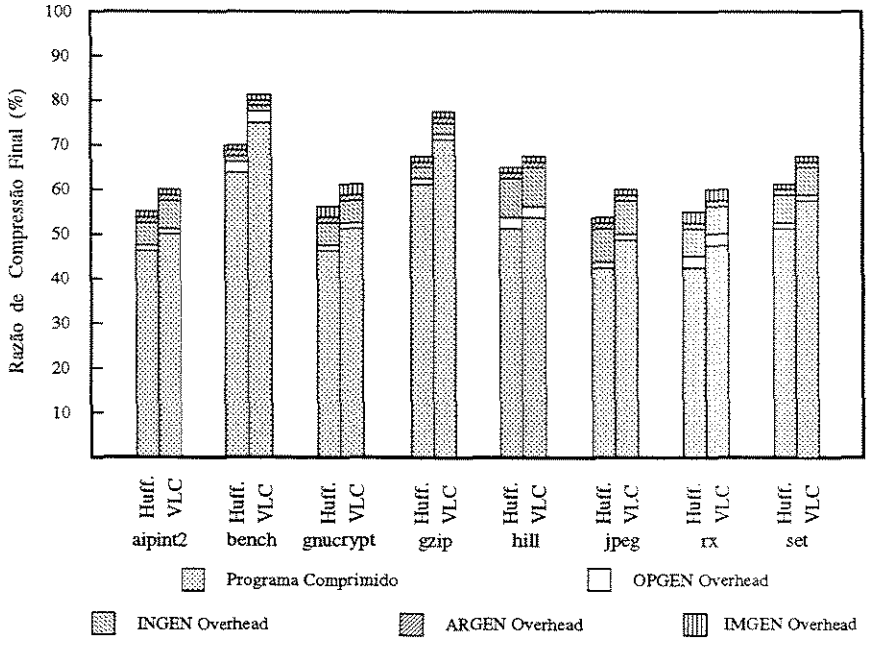

Figura 3.8: Razão de compressão final para o TMS320C25 considerando estimativas para os *overheads*.

3.3 Conclusões

O método Fatoração de Operandos para compressão de programas e descompressão em tempo real apresenta uma razão de compressão muito boa, melhor que qualquer outro método até então publicado. Este método apresenta a sua melhor razão de compressão (em média 35%) quanto os símbolos são codificados com codificação Huffman. Essa forma de codificação tem como desvantagem uma maior complexidade da máquina de descompressão, uma vez que o tamanho da *codeword,* que está sendo decodificada, só é determinado ao término de sua extração da palavra lida da memória. Essa maior complexidade na extração das *codewords* implica em uma maior área de silício necessária ao circuito de extração e um menor desempenho do mesmo. Estas dificuldades na implementação do descompressor são suavizadas quando o método de compressão usado é o VLC, porém a razão de compressão dos programas cai para 40,7%. Como será mostrado no capítulo 4 a principal idéia do método Fatoração de Operandos, ou seja escolha do símbolo de

compressão com significado semântico em relação ao objeto que está sendo comprimido pode ser usada para determinar uma nova técnica de compressão de programas para descompressão em tempo real, melhorando as característica de compressão dos programas e proporcionando uma implementação do descompressor mais eficiente, tanto em área de silício ocupada quanto em desempenho.

Capítulo 4

Compressão Baseada em Árvores de Expressão (TBC)

4.1 Introdução

Este capítulo apresenta uma nova técnica para compressão de código de programas para descompressão em tempo real denominada *Compressão Baseada em Árvores de Expressão* (TBC — *Tree Based Compression*).

A técnica TBC utiliza o mesmo principio básico que o método Fatoração de Operandos [60, 9] na definição dos símbolos de compressão, ou seja, os símbolos de compressão mantêm as informações semânticas do programa. Diferentemente do método Fatoração de Operandos, que divide as árvores de expressão em dois padrões (de árvore e de operandos) e utiliza esses padrões como símbolos para a compressão, o método TBC utiliza a própria árvore de expressão como símbolo para a compressão. Outras otimizações no processo de compressão também são propostas, resultando uma melhora em 13 pontos percentuais na razão de compressão dos programas quando se utiliza a codificação de Huffman para codificar os símbolos dos dois métodos. Uma codificação mais simples para os símbolos é proposta, resultando em uma melhora de 7 pontos percentuais na compressão dos programas quando comparada com a codificação Huffman para o método

fatoração de operandos. Essa codificação mais simples possibilita uma implementação mais eficiente da máquina de descompressão tanto do ponto de vista da área de silício utilizada quanto do desempenho. O método TBC foi aplicado a conjuntos de programas compilados para três processadores distintos, R2000, TMS320C25 e R4000. Neste capítulo, o método TBC é apresentado bem como os resultados de sua aplicação em um conjunto de programas compilados para o processador R2000.

Este capítulo está organizado da seguinte forma: na seção 4.2 é introduzida a arquitetura do conjunto de instruções do processador R2000, na qual está baseada a proposta da máquina de descompressão apresentada na seção 4.4; a seção 4.3 apresenta a técnica de compressão TBC, bem como compara os resultados obtidos com os descritos por Pannain para a razão de compressão usando a codificação das *codewords* em VLC e Huffman; a seção 4.4 propõe a máquina de descompressão e discute o tratamento especial dedicado às instruções de desvio; a seção 4.5 apresenta os resultados obtidos para a área de silício ocupada e desempenho esperado para uma implementação da máquina de descompressão usando a técnica TBC em *standard cell* CMOS de $0,6\ \mu m$; e por fim a seção 4.6 apresenta alguns comentários e conclusões.

4.2 Conjunto de Instruções do R2000

Os processadores têm como objetivo executar uma seqüência de instruções, denominado programa. A execução de uma instrução, de forma simplificada, pode ser dividida em passos ou etapas: busca da instrução na memória (*fetch*); decodificação e obtenção dos operandos; execução da operação especificada pela instrução; e armazenamento do resultado da operação no operando destino. Cada etapa, na execução de uma instrução, pode ser executada em um ou mais períodos de relógio (*período de clock*). A figura 4.1 mostra de forma esquemática o fluxo de execução de uma instrução. Ciclo de instrução

refere–se ao tempo necessário para que uma instrução seja executada por completo, em geral medido em número de períodos de relógio. Período de relógio refere–se à menor unidade de tempo em um processador e em geral é medido em nanosegundos. O ciclo de instrução é um múltiplo do período de relógio. O número e duração de cada passo difere de processador para processador [36, 31, 30].

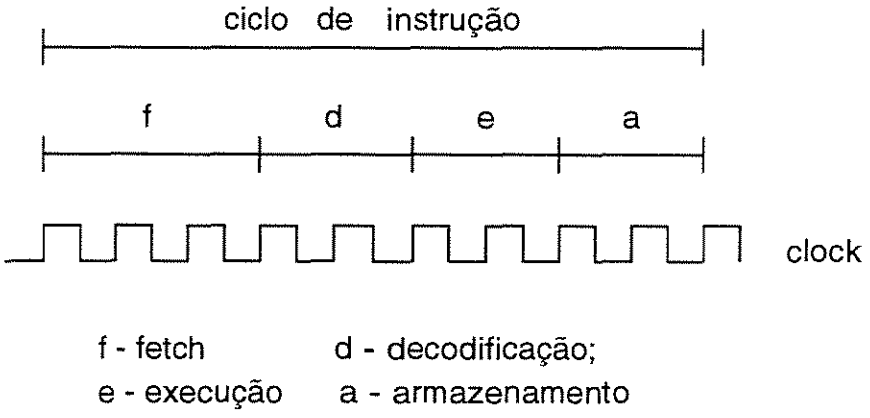

Figura 4.1: Etapas na execução de uma instrução.

O tempo de execução de um programa é determinado pelo tamanho do programa, em número de instruções; pelo número médio de períodos de relógio necessário para execução de uma instrução; e pela duração do período de relógio do processador.

O tamanho do programa é uma função do programa em si e do conjunto de instruções disponível no processador. O número médio de ciclos para a execução de uma instrução é função do conjunto de instruções e da arquitetura do processador. O período de relógio é função da tecnologia utilizada na implementação do processador. O desempenho de um processador pode ser melhorado otimizando um ou mais dos fatores citados acima. A técnica denominada *pipelining* foi introduzida para melhorar o desempenho de processadores, aumentando o número médio de instruções executadas por unidade de tempo (embora o número de períodos para a execução de uma dada instrução isoladamente seja

o mesmo). A técnica consiste na execução das diferentes etapas do ciclo de execução de uma instrução por unidades independentes, comunicando–se entre si por meio de registradores síncronos (sincronizados pelo sinal de *clock*). Essas unidades são denominadas de estágios do *pipelining*. Esta técnica permite um paralelismo no nível de instrução, o que leva a um menor número médio de ciclos para a execução de uma instrução (aumentando o *throughput* de instruções), melhorando assim o desempenho final do processador. Um efeito colateral que pode ser obtido com uso de *pipelining* é que, se os estágios do mesmo forem simples, pode–se ter, para uma mesma tecnologia, um período de *clock* menor.

A primeira máquina de propósito geral a utilizar *pipelining* foi Stretch (IBM7030) e diversas inovações foram introduzidas no CDC 6600 no início da década de 1960 [31]. A técnica de *pipelining* tem sido utilizada na maioria dos atuais processadores de propósito geral.

Quando os primeiros processadores foram produzidos, a etapa de busca de instruções (*fetch*) e de dados na memória levavam mais tempo (devido a baixa velocidade de resposta das memórias da época) que as demais fases. Segundo Kogge [41], essa foi a principal razão do surgimento das arquiteturas denominadas CISC (*Complex Instruction Set Computer*). A idéia básica era aproveitar ao máximo o tempo gasto com o acesso à memória. Desta forma projetaram–se instruções que executavam o máximo de trabalho possível. Isto aumentou a complexidade e o tempo necessário à execução de outras etapas no ciclo de instrução (decodificação e execução) o que torna mais difícil a implementação eficiente de *pipelining* neste tipo de arquitetura.

No fim dos anos 70 e início dos 80 a tecnologia de implementação de memórias experimentou avanços consideráveis, diminuindo o tempo necessário ao acesso a dados nelas armazenados. Com o surgimento das memórias *caches*, propostas por Smith [41], que são memórias de alta velocidade localizadas próximas à CPU (em alguns casos encapsulada no mesmo *chip*), o tempo de busca de instruções diminuiu consideravelmente. Isso motivou

o aparecimento das arquiteturas denominadas RISC (*Reduced Instruction Set Computer*) que caracterizam-se por um conjunto reduzido de instruções uniformes, possibilitando o uso eficiente da técnica pipelining. A desvantagem neste tipo de arquitetura é que o tamanho dos programas, em número de instruções, cresce. Jonhson [36] cita que, quando comparados com os processadores CISC, os processadores RISC produzem um número médio de ciclos por instrução reduzido de um fator de 2 a 5, enquanto o tamanho do código dos programas cresce de 30% a 50%. Além disso, a simplificação do *hardware* permite o uso de freqüências de relógio mais elevadas, proporcionando um ganho no desempenho.

4.2.1 O Processador MIPS R2000

O processador MIPS R2000 é um processador implementado com tecnologia RISC [37] clássica e possui muitas das características dos processadores atuais.

O processador R2000 possui uma arquitetura *load/store* com instruções de 32 *bits* e três tipos diferentes de formato. Tipo R para instruções aritméticas e lógicas que envolvem somente registradores. Tipo J para instruções de desvios e tipo I para aquelas instruções com valores imediatos e instruções *load/store*. A figura 4.2 mostra os três formatos das instruções do R2000 e seus campos.

As instruções do processador R2000 podem ser classificadas em:

Load/Store

Instruções para movimentar dados entre registradores e memória. São do tipo I e o endereço efetivo de memória é dado pelo conteúdo do registrador base (rs) mais o imediato (*offset*) de 16 *bits*.

Lógica e Aritmética

Instruções que executam operações lógicas, aritméticas ou de deslocamento. Podem

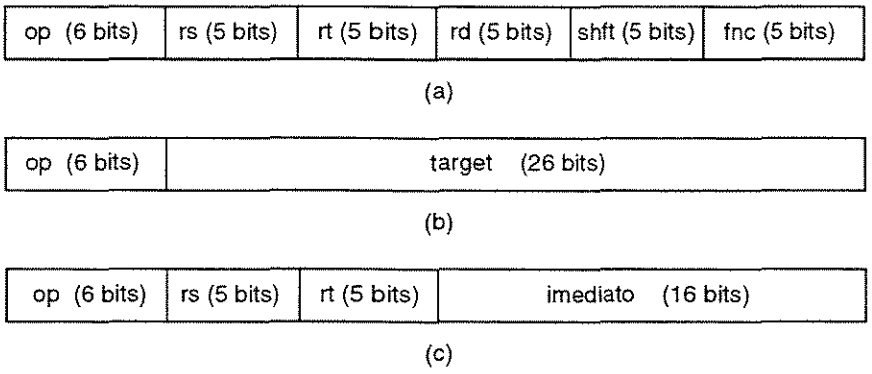

Figura 4.2: Formato das instruções do R2000.
(a) tipo R; (b) tipo J; e (c) tipo I.
rs – registrador fonte; rt – registrador alvo; rd – registrador destino;
shtf – número de deslocamentos; e fnc – especializa a instrução.

ser do tipo R, quando todos os operandos estão em registradores ou do tipo I, quando um dos operandos é um imediato.

Jump e Branch

São instruções de desvios. As instruções *jump* podem ser do tipo J (endereço destino é formado pela concatenação dos 6 *bits* de mais alta ordem do PC com os 26 bits do target) ou do tipo R (endereço destino de 32 bits armazenado em um registrador). As instruções de *branch*, que têm sempre um *offset* de 16 *bits* relativo ao PC corrente e dado pelo imediato, são sempre do tipo I.

Co–processador

Instruções de ponto–flutuante que usam o coprocessador e formato próprio.

Instruções Especiais

São instruções do tipo chamada de sistema (*syscall*) e retorno de tratamento de exceção. São todas do tipo R.

4.3 Compressão de Programas no R2000

Pannain em [60, 9] propõe uma técnica para compressão de código denominada Fatoração de Operandos (seção 3) que baseia-se na fatoração dos operandos de uma árvore de expressão, obtendo um padrão de árvore e um padrão de operandos. Pannain *et al* mostram que estes padrões repetem-se com freqüência no código do programa e que possuem uma distribuição exponencial.

A abordagem adotada por Pannain *et al* (ver resumo no capítulo 3) para codificar as *codewords* pode ser melhorada, pelo menos, de duas formas. A primeira é imediata e diz respeito ao critério adotado para a ordenação das listas de padrões de árvore e de operandos. A segunda, decorre de observações sobre o comportamento das combinações padrão de árvore com padrão de operandos para formarem, novamente, as árvores de expressão. A seguir são examinados estes dois casos e é proposto um novo método para codificar os programas comprimidos que melhora significativamente a razão de compressão média apresentada em [60, 9].

Critério de Ordenação dos Padrões de Árvore e de Operandos

Suponha que temos dois padrões de árvore Tp_1 e Tp_2 com tamanhos de 6 e de 32 *bits* respectivamente e Tp_1 ocorre 20 vezes no programa e Tp_2 5 vezes. Pelo método descrito por Pannain [60, 9] (seção 3.2) , à Tp_1 é atribuída uma *codeword* maior que a atribuída à Tp_2, suponha que sejam 7 e 3 *bits* respectivamente para as *codewords* associadas à Tp_1 e Tp_2. Desta forma a contribuição total, em *bits*, devido a Tp_1 ao programa comprimido será de 140 *bits* (7 x 20) e a devido à Tp_2 de 15 *bits* (3 x 5). Os dois padrões de árvore contribuirão em conjunto com 155 *bits* para o tamanho final do programa comprimido. Tp_1 ainda usará 6 *bits* no dicionário de árvores e Tp_2 32 *bits* neste mesmo dicionário.

Ordenando os padrões de árvores somente pela freqüência com que ocorrem no pro-

grama (e não pela contribuição em *bits* no programa original) à Tp_1 será associado uma *codeword* menor que a associada à Tp_2, suponha que sejam 3 e 7 *bits* respectivamente. A contribuição em *bits* de Tp_1 ao tamanho do programa comprimido será de 60 *bits* (3 x 20) e a contribuição devido à Tp_2 será de 35 *bits* (7 x 5). A contribuição total devido aos dois padrões será de 95 *bits* (60 + 35) que é 60 *bits* menor que na abordagem original. O mesmo ocorre com os padrões de operandos.

A nova forma de ordenação dos padrões, por freqüência e não por contribuição em *bits* ao tamanho do programa original, produz um ganho na razão de compressão. Esse ganho varia entre 2% a 5% para os programas do conjunto de teste. Isso ocorre devido ao fato de que os padrões de árvore e de operandos mais freqüentes possuem o mesmo tamanho ou a diferença ser mínima. Como resultado, a mudança de critério para ordenar as listas de padrões não altera demasiadamente a ordem dos padrões na lista ordenada por um ou outro critério.

Combinação dos Padrões de Árvore e de Operando

A análise do comportamento das combinações $< Tp, Op >$ revela que somente uma parcela muito pequena das combinações possíveis são realmente formadas. Este número é muito próximo do número de padrões de operandos distintos (Op), sugerindo que os padrões de operandos sozinhos sejam quase que suficientes para codificar o programa, funcionando como uma espécie de "impressão digital" do programa.

Cada combinação $< Tp, Op >$ representa uma árvore de expressão e todas as combinações que ocorrem para um dado programa formam o conjunto de árvores de expressão distintas do programa. A tabela 4.1 mostra para cada um dos programas do SPECInt95 o número de árvores de expressão, o número de padrões de árvore distintos, o número de padrões de operandos distintos e o número de árvores de expressão distintas. A coluna (VI) da tabela 4.1 mostra a diferença percentual entre o número total de árvores distin-

tas e o número de padrões de operandos distintos, onde pode–se notar que o número de árvores distintas é, em média, só 3,9 pontos percentuais maior que o número de padrões de operandos distintos. Também é possível observar que a diferença entre o número de árvores de expressão distintas e o número de padrões de operandos distintos é bem menor que o número de padrões de árvores distintas. Estas observações nos levam a concluir que no método de compressão apresentado em [60, 9] ainda existe redundância de informação nos programas comprimidos. Ou seja, ainda é possível representar os programas utilizando menos *bits*, uma vez que essa diferença entre os padrões de operandos e as árvores de expressão está sendo codificada pelos padrões de árvores que são em número muito maior do que o realmente necessário.

Programa (I)	Árvores (II)	Árvores Distintas (III)	Padrões de Árvores (IV)	Padrões de Operandos (V)	(III) - (V) (VI (%))
compress	1444	744	125 (8,6)	731 (50,6)	1,9
gcc	311488	44226	1547 (0,5)	41486 (13,3)	6,6
go	54651	13025	578 (1,1)	12561 (23,0)	3,7
ijpeg	38426	10169	767 (2,0)	9839 (25,6)	3,4
li	15761	3135	157 (1,0)	3056 (19,4)	2,9
perl	62915	11769	648 (1,0)	11209 (17,8)	5,0
vortex	128104	16733	471 (0,4)	16143 (12,6	3,7
média	87541	14257	613 (2,1)	13575 (23,2)	3,9

Tabela 4.1: Número de árvores (II) e de árvores distintas (III) e o número de padrões de árvores (IV) e de operandos (V) nos programas. Os valores entre parênteses são percentagens com respeito ao número total de árvores de expressão

A figura 4.3 mostra o gráfico obtido quando as árvores de expressão distintas são ordenadas em ordem decrescente de freqüência e é computada a contribuição cumulativa das árvores de expressão distintas em relação ao total de árvores de expressão do programa. O comportamento exponencial também é observado aqui, sugerindo fortemente uma forma de codificação que utilize *codewords* de tamanho variável.

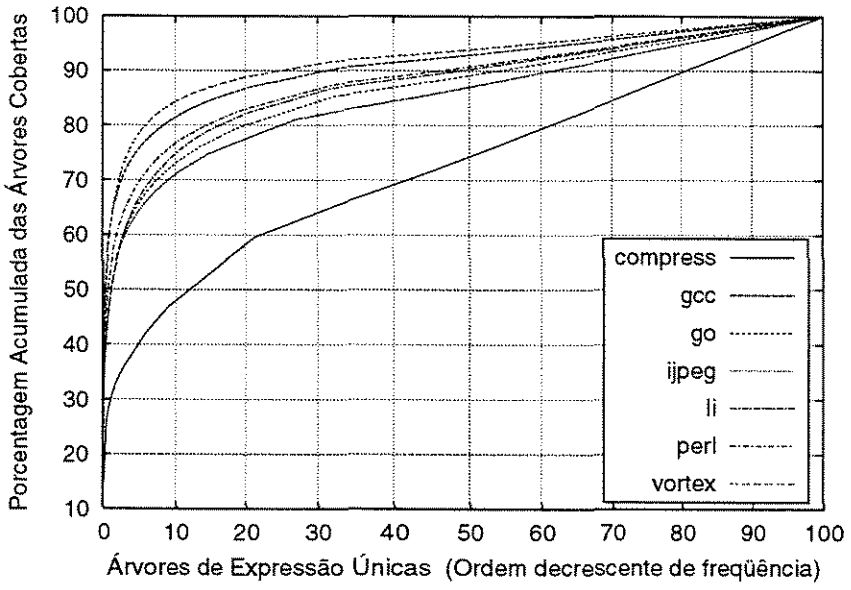

Figura 4.3: Porcentagem de árvores do programa cobertas pelas árvores distintas.

A implementação de uma máquina de descompressão utilizando codificação de Huffman, VLC ou um método misto (Huffman–fixo e VLC–fixo) requer um circuito para alinhamento, identificação e extração das *codewords* do programa comprimido complexo, exigindo uma grande área de silício e com baixo desempenho.

Desta forma, optou–se por uma solução de compromisso, na qual as *codewords* foram codificadas com um número fixo de tamanhos diferentes. *Bits* de escape foram usados para a identificação do tamanho da *codeword* corrente, facilitando sobremaneira o trabalho do circuito de extração.

Um experimento foi realizado, usando dois tamanhos de *codewords*: 8 *bits* e $\lceil \log_2(N - 127) \rceil + 1$ *bits*, onde N é o número de árvores distintas do programa e 127 o número de árvores distintas codificadas com o primeiro tipo de *codeword*. A figura 4.4 mostra o formato das *codewords* utilizadas no experimento.

A tabela 4.2 mostra para o R2000 a razão de compressão encontrada para cada programa e a média quando utilizado os métodos apresentados em [60, 9] com codificação

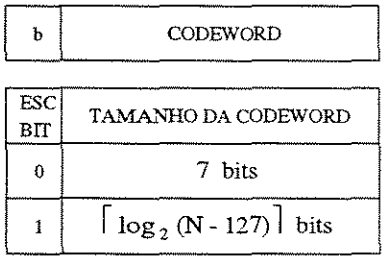

Figura 4.4: Formato das *codewords*

Huffman (melhor caso) e VLC e usando o método apresentado nesta seção, denominado compressão baseada em árvores de expressão (TBC – *Tree Based Compression*).

	Fatoração de Operandos		TBC	
Programa (I)	Huffman (%) (II)	VLC (%) (III)	(%) (IV)	Huffman (%) (V)
compress	31,5	33,4	21,5	6,4
gcc	38,1	45,5	33,0	26,8
go	33,4	37,4	27,6	23,7
ijpeg	36,3	41,3	27,6	24,1
li	32,7	37,6	26,3	22,3
perl	37,0	46,2	30,3	25,0
vortex	36,0	43,6	30,4	25,2
média	35,0	40,7	28,1	21,9

Tabela 4.2: Comparação entre as razões de compressão para o R2000

Pode–se observar que, mesmo usando uma forma de codificação simples para as *codewords*, o TBC obtém resultado, em média 6,9 (II - IV) pontos percentuais melhor que os obtidos para Fatoração de Operandos usando o método de codificação de Huffman. A utilização do método de codificação de Huffman em associação ao método TBC resultaria razões de compressão ainda melhores, em média, 13,1 (II - V) pontos percentuais melhores que aquelas obtidos em [60, 9], porém com reflexos indesejáveis na máquina de descompressão, tornando–a mais complexa, maior e mais lenta.

A razão de compressão pode ser melhorada um pouco aumentando o número de tamanhos das *codewords*. Estudos neste sentido foram realizados para a CPU R4000 e revelou que 4 ou 5 tipos de tamanhos dão as melhores razões de compressão. Isso será analisado com mais detalhes no capítulo 6 que apresenta os estudos para a CPU R4000.

4.4 Máquina de Descompressão

Esta seção apresenta a máquina de descompressão implementada em hardware, capaz de descomprimir um programa utilizando o método de compressão TBC (seção 4.3).

A máquina de descompressão aqui apresentada trabalha em duas fases. Na primeira, uma *codeword*, representando uma árvore de expressão, é extraída do programa comprimido. A segunda fase consiste em mapear a *codeword* na seqüência de *opcodes* e operandos (registradores e/ou imediatos). Estas informações são, então, usadas para montar as instruções originais, que são entregues à CPU (ou colocadas na *cache*).

A figura 4.5 mostra a máquina de descompressão e como ela interliga-se com a CPU e a memória. O bloco `extrator` é responsável por identificar e isolar uma *codeword* das palavras lidas da memória, que contêm o programa comprimido. Uma vez extraída uma *codeword*, ela é então passada ao descompressor que a decodifica em um conjunto de *opcodes* e de operandos (registradores e/ou imediatos) e monta as instruções originais que são colocadas em um *buffer* de saída e são entregues à CPU quando da realização de um *fetch* de instrução.

Uma unidade de controle é responsável por controlar e sincronizar todo este processo.

A seguir os módulos `extrator` e `descompressor` são detalhados. A unidade de controle é uma máquina de estados finitos que monitora o *status* da máquina de descompressão e gera os sinais de controles necessários à realização das tarefas.

Os acessos a dados na memória, durante a execução de uma instrução (por exemplo

load) são realizados de forma transparente à máquina de descompressão, ou seja a unidade de controle identifica que o acesso à memória não é um *fetch* de instrução e abre um canal direto entre a CPU e a memória (de dados) retirando do circuito os módulos extrator e descompressor.

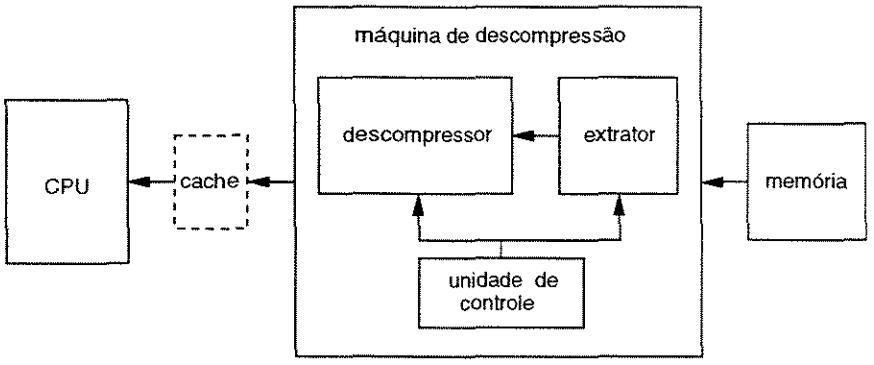

Figura 4.5: Máquina de Descompressão. R2000

4.4.1 Módulo Extrator de Codewords

A extração de uma *codeword*, a partir do programa comprimido, é uma das tarefas críticas executada pela máquina de descompressão. Para melhorar a razão de compressão, as *codeword* não precisam estar alinhadas na memória (nem a palavra, nem a *byte*), ou seja, elas podem ter seu início em qualquer um dos 32 *bits* de uma palavra de memória. Desta forma, a localização de uma *codeword* em uma palavra depende do tamanho e da localização da *codeword* anterior. Um caso a parte são as *codewords* que são alvo de uma instrução de desvio. Trataremos destes casos na seção 4.4.3, na qual discutiremos os problemas e as soluções para as instruções de desvios.

Para reduzir a quantidade de *hardware* foi usado um processo de extração em dois passos [13]. No primeiro, uma palavra é lida da memória e armazenada em um banco de registradores de forma que o primeiro *bit* da próxima *codeword* fique armazenado no

registrador mais a esquerda. O segundo passo consiste em extrair a *codeword* por meio de uma rede de alinhamento *barrel shifter* de três níveis (maior desalinhamento possível é de sete *bits*).

A figura 4.6 mostra o extrator, que é composto por um *Buffer* de entrada constituído por 7 registradores. Quatro registradores de 9 *bits*, onde 8 *bits* são usados para armazenar parte ($\frac{1}{4}$) da palavra lida da memória e 1 *bit* de *status*. Os *bits* de *status* são carregados com o valor "1" (automaticamente) no momento em que a palavra lida da memória é carregada nos registradores. As palavras de memórias são lidas e carregadas nos 4 registradores mais à direita (figura 4.6 (a)) de forma assíncrona em relação às outras atividades do extrator e do descompressor, toda vez que estes 4 registradores estão livres (*bits* de *status* iguais a "0"). Essa abordagem para a leitura das palavras de memória contendo o programa comprimido (*codewords*) implementa o equivalente a um *prefetch*, aumentando o desempenho do extrator, em particular, e do descompressor em geral. Os dados lidos, nos 4 registradores mais à direita são então deslocados para os registradores à esquerda para serem processados. Durante tais deslocamentos os *bits* de *status*, gradativamente, vão sendo preenchidos por "0". Quando o sinal *full* torna–se "0" a unidade de controle dispara uma nova leitura de memória, preenchendo os 4 registradores à direita com uma nova palavra. A unidade de controle é também responsável por determinar quantos deslocamentos são necessários a fim de que o primeiro *bit* da *codeword* a ser extraída esteja no registrador mais à esquerda. Por exemplo, no início da execução de um programa (execução da primeira instrução do programa) a unidade de controle tem que determinar que sejam dados 3 deslocamentos à esquerda, colocando o início da primeira *codeword* do programa no registrador mais a esquerda (neste caso, teremos também que o primeiro *bit* da *codeword* coincidirá com o primeiro *bit* do registrador). A quantidade de deslocamentos necessários ao posicionamento da *codeword* é computada pela unidade de controle, com base no tamanho e localização da última *codeword* extraída.

Uma vez que cada deslocamento realiza um movimento de 8 *bits* à esquerda, uma *codeword* pode estar alinhada ou desalinhada de no máximo 7 *bits*. Esse possível desalinhamento é corrigido pela rede de alinhamento, implementada por uma estrutura *barrel shifter* mostrada na figura 4.6 (b). Os sinais de controle S_0, S_1 e S_2 do *barrel shifter* são gerados pela unidade de controle com base no desalinhamento da *codeword* anterior, que é determinado a partir do tamanho e do início da *codeword* anterior.

O tamanho da *codeword* corrente é determinado pelo primeiro *bit* da *codeword* (ESC BIT, figura 4.4). Se "0" a *codeword* possui 8 *bits* (BIT ESC + 7 *bits*) e se "1" ela possui 1 + $\lceil \log_2(N-127) \rceil$ *bits*, que é a quantidade de *bits* necessária para representar o ESC BIT e a codificação binária dos $N - 127$ *codewords* menos freqüentes, onde N é o número total de árvores de expressão distintas do programa e 127 o número de árvores de expressão distintas (as mais freqüentes) codificadas com 8 *bits*.

Note que, a leitura de uma nova palavra de memória pode ser realizada em paralelo com a extração da *codeword* corrente, assim que o *buffer* de entrada esteja liberado (sinal *full* = "0"). Desta forma a máquina de descompressão pode entregar à CPU a instrução solicitada sem que seja necessário esperar uma leitura da memória, extração e decodificação da instrução. Durante a execução, pela CPU, de uma instrução todo o processo de identificação da *codeword* seguinte está acontecendo em paralelo.

A figura 4.6 mostra a rede de alinhamento e o conjunto de registradores para *codeword* de 8 e 16 *bits*. Programas que usem por exemplo 8 e 14 *bits*, a diferença seria na rede de alinhamento que teria 14 *bits* de saída e 20 *bits* de entrada, mantendo-se a estrutura. Programas para os quais a maior *codeword* tenha no máximo 9 *bits*, são necessários somente 6 registradores (os 4 mais a direita e dois dos mais a esquerda). Este esquema de extração foi originalmente proposto por Benes *et al* [13] e é usado no descompressor de código de Huffman para o CCRP.

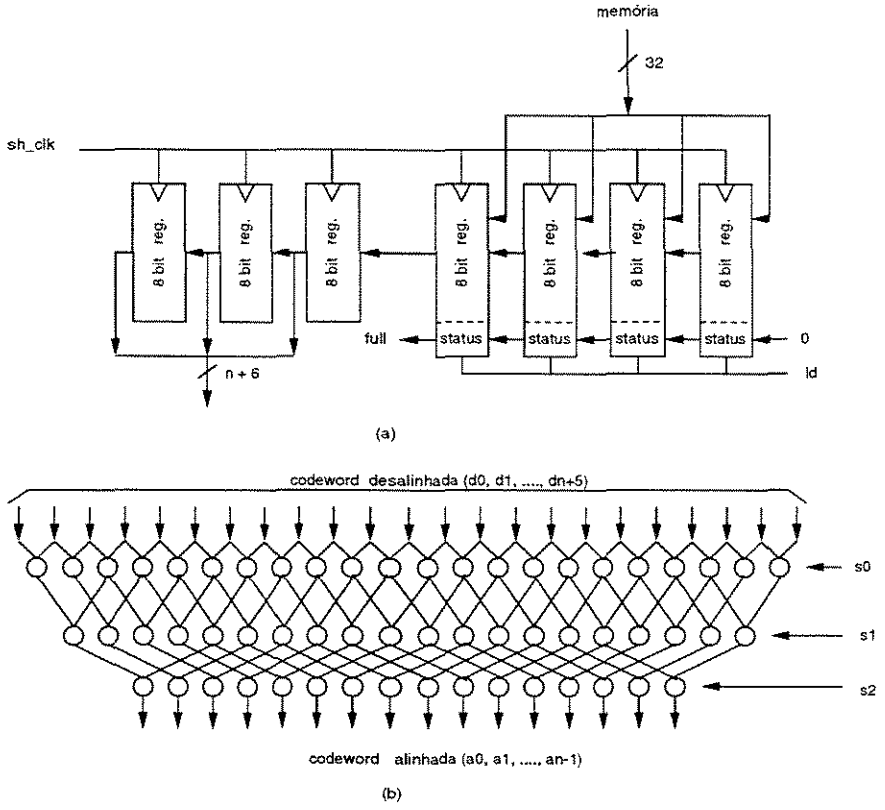

Figura 4.6: Extrator: *Buffer* de entrada e rede de alinhamento das *codewords*

4.4.2 Módulo Descompressor

A figura 4.7 mostra o `descompressor` que é composto por três módulos básicos. O dicionário de padrões de árvores (*Tree Patterns Dictionary* – TPD) que armazena, para cada *codeword*, os *opcodes* das instruções que compõem a árvore de expressão associada. O gerador de padrão de árvore e de operandos (*Tree Patterns, Registers and Immediate Generator* – TRIGen) é uma máquina de estados finitos (FSM) que gera as informações necessárias ao módulo IAB (*Instruction Assembler Buffer*) que por sua vez monta as instruções (uma a uma) que compõem a árvore de expressão e as entrega à CPU.

O dicionário TPD possui três campos: OPCODE, ITYPE e END. O campo OPCODE contém o código de operação da instrução que está sendo montada. O campo ITYPE

codifica o tipo, ou seja, o formato da instrução e é usado para orientar o módulo IAB na montagem da instrução. O campo END informa ao módulo TRIGen e à unidade de controle que a instrução sendo montada é a última da árvore de expressão, e determina que uma nova *codeword* deve ser entregue ao módulo TRIGen pelo extrator enquanto a máquina de estado finito TRIGen deve migrar para o seu estado inicial ($S0$).

A máquina TRIGen é uma FSM síncrona. Inicialmente, está no estado inicial $S0$ e quando recebe uma *codeword* gera o endereço *tpaddr*, correspondente ao início de um padrão de árvore no dicionário TPD e os registradores e/ou imediato necessários à montagem da instrução. Os registradores e/ou imediato gerados são colocados nos barramentos RS1, RS2, RD e IMD respectivamente. RS1 e RS2 correspondem aos campos associados aos registradores fontes da instrução enquanto RD ao registrador destino e IMD ao campo imediato. Caso a instrução não possua um ou mais destes campos, no barramento correspondente é colocado um valor irrelevante (*don't care*). Nos estados seguintes TRIGen incrementa o valor de *tpaddr* e gera os novos valores para RS1, RS2, RD e IMD. O número de estados de TRIGen é limitado pelo número de instruções da maior árvore de expressão do programa.

4.4.3 Instruções e Endereços de Desvios

De forma geral, em compressão de programas existem dois problemas associados às instruções e endereços de desvios. O primeiro diz respeito à determinação do endereço alvo de desvios incondicionais em instruções do tipo *jmp* (instrução tipo J) que usam endereço absoluto durante a compressão. Em implementações de outros autores, normalmente, este tipo de instrução não é comprimida, para evitar a necessidade de reescrever as palavras de código que representam estas instruções, o que pode alterar o endereço alvo, tornando necessário a reescrita da palavra de código entrando em um processo ciclico. Lefurgy *et al* [43] adotam esta solução para as instruções de desvio.

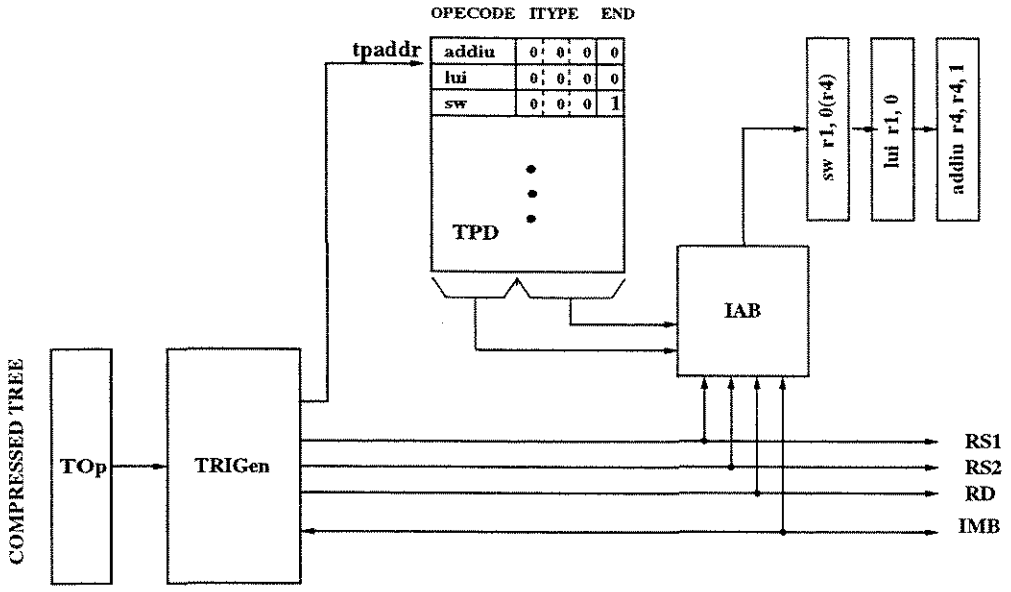

Figura 4.7: Descompressor para o R2000

Para programas gerados por compiladores os desvios indiretos podem ser codificados normalmente, pois como seu endereço alvo está em registrador, apenas a palavra de código necessita ser reescrita. Neste caso é necessário uma tabela para mapear os endereços originais armazenados nos registradores para os endereços comprimidos.

O segundo problema relaciona-se com o alinhamento das instruções. Existem duas soluções possíveis para esse problema. A primeira é alinhar todas as *codewords* que representam instruções que são endereços alvo, como definido pela arquitetura do conjunto de instruções (ISA – *Instruction Set Architecture*) do processador. Ou seja, sempre no início de uma palavra de memória. Esta solução é simples de ser implementada, porém inaceitável do ponto de vista da razão de compressão. Uma segunda solução é aceitar o desalinhamento e modificar a unidade de controle do processador para tratar *offsets* com alinhamento segundo o tamanho das *codewords*. Neste caso um endereço do programa original tem que ser mapeado em dois endereços. O primeiro indicando em qual palavra

de memória está localizada a *codeword* e o segundo (*offset*) para indicar o deslocamento em relação ao início da palavra. Esta foi a solução adotada por Lefurgy *et al* [43].

A solução apresentada neste trabalho aproveita idéias aplicadas no trabalho de Lefurgy *et al* [43], dividindo o endereço alvo de uma instrução de desvio em dois endereços *addr* com 21 *bits* e um *offset* com 5 *bits*. Apenas uma quantidade muito pequena de endereços nos programas utilizam mais do que 21 *bits*. Para esses casos especiais foi utilizada uma tabela de desvios para armazenar os novos endereços alvo, abordagem semelhante à adotada em outros trabalhos [43, 44].

Diferentemente da solução adotada por Lefurgy [43] que impõe modificações no *core* da CPU, a solução apresentada neste trabalho é implementada pela máquina de descompressão. Durante a operação de descompressão os valores de *addr* e do *offset* são gerados pelo TRIGen e colocadas no barramento IMD (26 *bits*). Somente *addr* é passado à CPU junto a instrução de desvio e o *offset* é armazenado. Quando a CPU termina a execução da instrução os 21 *bits* do barramento de endereço da CPU correspondentes são comparados com *addr* para saber se o desvio foi tomado. Caso ele não seja tomado, a próxima *codeword* é extraída e decodificada. Caso contrário, a palavra de memória armazenada em *addr* é lida e a unidade de controle, usando o valor do *offset*, determina tantos deslocamentos no extrator quantos forem necessários para posicionar o início da *codeword* no registrador mais a esquerda da figura 4.6 (a). O número de deslocamento será igual a 3 mais o valor dado pelos dois *bits* mais significativos do *offset*. Os três *bits* menos significativos do *offset* determinam o desalinhamento com o qual o *barrel shifter* deve ser acionado para extrair corretamente a *codeword*.

Para esta solução, como são utilizados somente dois tamanhos de *codewords*, pode-se comprimir os desvios absolutos, pois caso haja necessidade de reescrever a *codeword* associada à instrução de desvio e o endereço alvo modificar-se, provocando uma reescrita da *codeword*, esse processo converge em uma ou duas iterações, pois a *codeword* mudará

de tamanho no máximo uma vez.

4.5 Resultados Experimentais

Para testar a máquina de descompressão os módulos extrator, unidade de controle e descompressor foram descritos em VHDL. Os módulos extrator e unidade de controle são praticamente os mesmos para todo o conjunto de programas de teste, diferindo-se no tamanho do *buffer* de saída do extrator, que tem o tamanho (em *bits*) da maior *codeword* usada pelo programa. Os *buffers* foram descritos uma única vez e o seu tamanho é passado como parâmetro no momento em que são instanciados. Estes módulos, quando comparados com o módulo descompressor, utilizam uma área de silício desprezível. Desta forma será dada atenção, nesta seção, a apresentação do módulo descompressor.

O descompressor é composto pelo dicionário de padrões de árvore TPD, IAB e pelo TRIGen. O dicionário TPD é implementado em memória cuja largura de palavra é 10 *bits*: 6 *bits* para armazenar o *opcode*; 3 *bits* para codificar a informação necessária ao IAB montar a instrução e 1 *bit* para indicar fim da árvore corrente. A tabela 4.3 mostra o tamanho dos programas de teste e o *overhead* devido ao dicionário TPD em *bits*.

Programa	Tamanho Bits	Overhead (%) (Bits)
compress	61088	3840 (6,2)
gcc	11737248	73260 (0,6)
go	2408864	24080 (1,0)
ijpeg	1596224	36580 (2,3)
li	591168	3580 (0,6)
perl	2324960	20420 (0,9)
vortex	4765952	16500 (0,4)
média	3355072	25465 (1,7)

Tabela 4.3: *Overhead* devido ao Dicionário TPD

O IAB *Instruction Assembler Buffer* é um um conjunto de *multiplexadores* que tem como sinais de controle os *bits* de ITYPE, sendo mostrado na figura 4.8

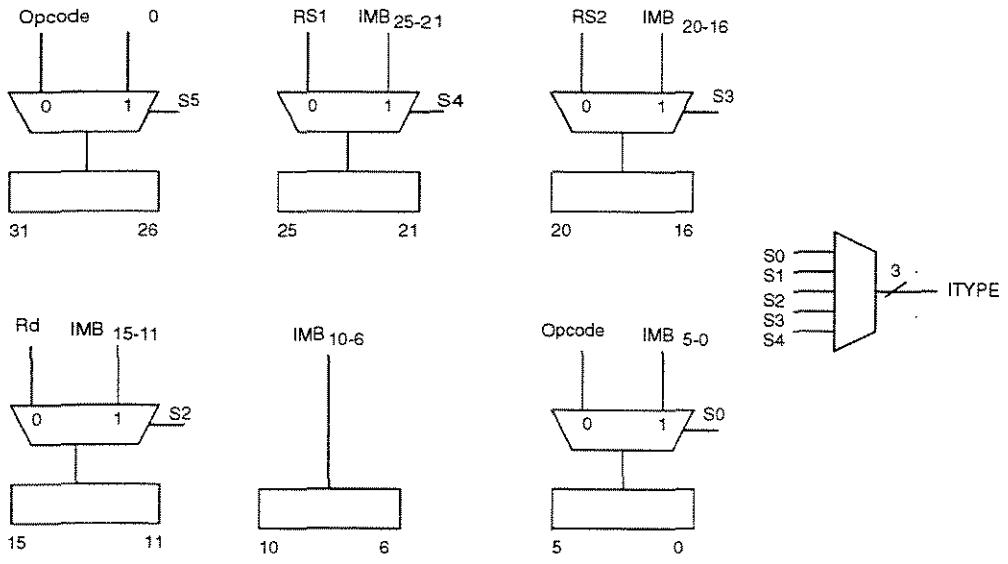

Figura 4.8: *IAB – Instruction Assembler Buffer*

O TRIGen é uma máquina de estados finitos síncrona que tem como entrada de controle uma *codeword* e, como exposto na seção 4.4.2, gera o endereço inicial do padrão de árvore em TPD e os valores dos registradores e/ou imediato que formam a instrução. TRIGen tem tantos estados quanto o número de instruções na maior árvore de expressão.

O código VHDL para a máquina de estados finitos TRIGen é gerado automaticamente por um programa escrito especificamente para esse fim que usa como entrada da máquina de estados as atribuições das *codewords* durante a compressão e gera:

- o endereço inicial da árvore de expressão no TPD,
- os registradores e/ou imediato para a primeira instrução,
- a cada transição de estado:

- gera os registradores e/ou imediato para as outras instruções (uma instrução por estado) da árvore de expressão associada à *codeword*,
- incrementa o endereço gerado no primeiro estado.

O conteúdo da memória TPD também é gerado por programa. TRIGen foi sintetizado usando a ferramenta de sínteses Leonardo Spectrum e a biblioteca *standard cell* de $0,6\mu m$ e 5 volts da AMS. Para cada TRIGen sintetizado, foi usada otimização por área e estimadas a área ocupada e a freqüência máxima de uso. Também foram medidas a área de memória ocupada e o tempo de CPU gastos na sínteses, em uma E450 da SUN com 4 processadores de 400 MHz e 4 GB de RAM e 4 GB de área de *swap*. Na tabela 4.4 são mostrados esses resultados.

Programa	No. de Instruções	Área (mm^2)	Freqüência (MHz)
compress	1909	4,0	146,4
li	18474	15,1	107,2

Tabela 4.4: Área de silício (mm^2) e freqüência máxima de operação (MHz) estimada para a máquina de descompressão.

4.6 Conclusões

O método de compressão de programas para descompressão em tempo real Fatoração de Operandos apresenta uma razão de compressão para um conjunto de programas do SPECInt95 de 35% para a codificação Huffman e de 40% para codificação VLC que podem ser consideradas muito boas. Mostrou–se que a razão de compressão dos programas pode ser melhorada, usando–se os mesmos princípios básicos que norteiam a escolha dos símbolos de compressão no método Fatoração de Operandos e usando–se outra política para a ordenação dos símbolos. O método de compressão de programas Compressão

Baseada em Árvores de Expressão (TBC) foi proposto. O método TBC usa como símbolo básico de compressão as árvores de expressão (e não os padrões de árvores e de operandos). Os símbolos são ordenados por ordem decrescente de freqüência (e não de contribuição, em *bits* na cobertura do programa). A razão de compressão dos programas para o conjunto de programas do SPECInt95 encontrada foi de 21,9% para codificação Huffman e de 28,1% para uma codificação, bem mais simples, que utiliza somente dois tamanhos de *codewords* e que proporciona uma implementação mais eficiente da máquina de descompressão tanto para a área de silício ocupada quanto para o desempenho. Uma proposta de máquina de descompressão foi apresentada bem como os resultados de sua síntese em uma tecnologia *standard cell*, CMOS de $0,6\ \mu m$, 5 volts da AMS.

Capítulo 5

Compressão e Descompressão Usando TBC e o TMS320C25

5.1 Introdução

Neste capítulo é analisado o comportamento do método de compressão de programas para descompressão em tempo real (TBC) em programas compilados para o processador TMS320C25.

O processador TMS320C25 foi escolhido para avaliar o método TBC por ser um processador específico para aplicações de processamento digital de sinais (DSP – *Digital Signal Processors*) de larga utilização em sistemas embarcados nas áreas de telecomunicações, instrumentação, eletrônica de consumo, etc.

Este capítulo está organizado da seguinte forma: a seção 5.2 introduz o processador TMS320C25 e a sua arquitetura do conjunto de instruções; a seção 5.3 apresenta o método de compressão TBC aplicado a programas compilados para o processador TMS320C25; a seção 5.4 descreve a máquina de descompressão personalizada para o processador TMS320C25, bem como o tratamento dispensado às instruções de desvio; a seção 5.5 apresenta os resultados da síntese (área de silício e desempenho) da máquina de descompressão para um conjunto de programas compilados para o processador TMS320C25,

utilizando–se uma biblioteca *standard cell*, CMOS de $0,6\ \mu m$, 5 volts da AMS; por fim a seção 5.6 apresenta algumas considerações e conclusões.

5.2 O Processador TMS320C25

O processador TMS320C25 pertence à família de processadores TMS320 que são processadores especializados para processamento digital de sinais (DSP – *Digital Signal Processors*) de 16/32 *bits*, interface com o exterior e registradores de 16 *bits* e operações na unidade lógica aritmética (ALU – *Arithmetic Logic Unit*) de 32 *bits*. Os processadores da família TMS320 têm sido utilizados em um grande número de aplicações [7, 6, 63]:

Processamento de sinais — filtros digitais, convolução, transformadas rápidas de Fourier, transformadas de Hilbert, geração de formas de ondas, etc

Imagens e gráficos — rotações 3–D, visão de robô, transmissão e compressão de imagens, animação, etc

Instrumentação — análise espectral, geração de funções, análise de transientes, etc

Voz — reconhecimento de voz, tradução texto–voz, síntese de voz, etc

Controle — disco, robô, impressora laser, etc

Comunicação — PABX digitais, multiplexadores de canais, modens, criptografia de dados, voz digital, vídeo conferência, FAX, telefone celular, etc

Automotivo — análise de vibrações, navegação, comando de voz, radio digital, etc

Consumo doméstico — TV, audio digital, brinquedos, jogos eletrônicos, etc

Indústria — robôs, controle numérico, acesso seguro, monitores de potência, etc

Medicina — monitoramento de pacientes, equipamentos de ultra–som, ferramentas de diagnósticos, monitoramento de feto, aparelhos auditivos, etc

Uma característica normalmente encontrada nas arquiteturas DSPs é a execução de instruções de multiplicação e adição em um único ciclo. Os processadores da família TMS320 foram projetados de forma que um ciclo de instrução seja igual a um ciclo de *hardware*. Isso faz com que as etapas de execução de uma instrução (busca da instrução, decodificação, busca dos operandos e execução da instrução) sejam realizadas em um único ciclo de máquina.

Esses processadores são do tipo CISC com instruções, de certa forma, comprimidas. O número de tarefas executadas por uma instrução em um processador DSP é grande, quando comparado com as instruções dos processadores RISCs.

Uma desvantagem desse tipo de processador é que a geração de código, pelos compiladores, é menos regular e isso faz com que as técnicas de otimização sejam menos eficientes.

Outra característica da maioria dos processadores DSPs é não possuírem unidades de ponto flutuante. Isso é devido às unidades de ponto flutuante ocuparem mais área de silício, consumirem mais potência, necessitarem mais ciclos de relógio e serem mais lentas que as unidades de ponto fixo [7].

Devido à necessidade de uma instrução ser executada em um único ciclo de relógio a memória é dividida em duas partes: memória de programa e memória de dados. O acesso a estas duas memórias pode ser realizado simultaneamente, por barramentos independentes. Desta forma permite–se a busca de uma instrução e dos operandos de outra instrução (que está executando no *pipeline*) simultaneamente. Arquiteturas com duas memórias distintas (uma para dados e e outra para programas) são conhecidas como "*arquiteturas Harvard*". Os DSPs que utilizam essa abordagem para o sistema de memória possuem uma memória

de dados do tipo RAM (*fast static* RAM – *Random Access Memory*) e uma memória de programas do tipo ROM (*Read Only Memory*), ambas trabalhando com ciclo de acesso igual a um ciclo de máquina. Vários processadores DSPs são baseados em arquiteturas memória–registradores, como os da família TMS320 da Texas Instruments [63, 37], onde os operandos estão um em memória e o outro em registrador e o resultado é sempre armazenado em um acumulador [63, 7].

5.2.1 Descrição do Conjunto de Instruções do TMS320C25

Os processadores DSPs da família TMS320 [63] são largamente utilizados pela indústria de aplicações em processamento digital de sinais. Seu conjunto de instruções possui três modos de endereçamento distintos: direto, indireto e imediato. Os dois primeiros são usados em acesso a dados na memória. Estes modos têm como vantagem reduzir o número de bits usado na codificação da instrução, diminuindo o número de bits necessários para representar um endereço. No modo direto, os 7 bits menos significativos da palavra de instrução são concatenados com os 9 bits de um registrador especial denominado apontador de página da memória de dados (DP – *Data memory page Pointer*) que mantém o endereço da página de dados corrente, formando os 16 bits de endereço de memória. Todas as instruções podem utilizar este modo de endereçamento, com exceção das instruções de desvio (`jmp`, `call` e `ret`), as instruções que operam com imediatos e as sem operandos. Instruções que usam esse modo de endereçamento têm tamanho de 16 bits (uma palavra) e são dos tipos 4,5 e 6, como mostra a figura 5.2.

No modo endereçamento indireto, o endereço de memória é dado por um dos registradores auxiliares (AR_0 – AR_7 – *Auxiliary Registers*) de 16 *bits* cada. A seleção de qual registrador auxiliar será usado é feita pelo registrador ARP (*Auxiliary Register Pointer*) de 3 *bits*. O conteúdo dos registradores AR pode ser alterado por meio de uma unidade aritmética de registradores auxiliares (ARAU – *Auxiliar Register Arithmetic Unit*), cujas

operações são executadas no mesmo ciclo da instrução e após o conteúdo do registrador AR ter sido usado pela instrução (figura 5.1). Este esquema permite aumentar a velocidade de acesso à localizações vizinhas de memória, operação muito utilizada em processamento digital de sinais. As operações realizadas no registrador ARP e nos registradores AR_{0-7} (incremento, decremento e carga de um novo valor) são indicadas nas instruções pelos caracteres especiais *,+ e - como pode ser visto nas figuras 3.2 (a) e 5.3 (a). Todas as instruções, exceto as instruções que operam com imediatos e aquelas sem operandos, podem usar este modo de endereçamento.

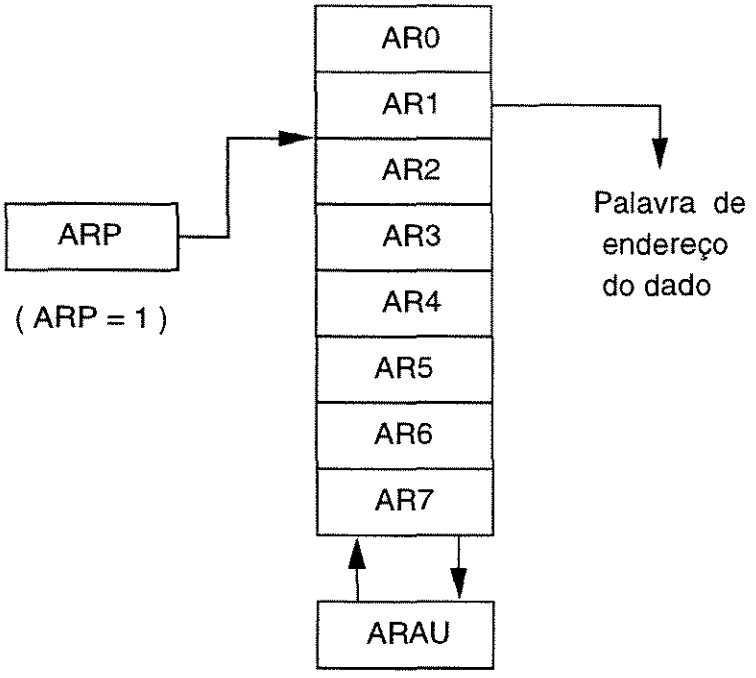

Figura 5.1: Diagrama do modo de endereçamento indireto

A figura 5.2 mostra os formatos das instruções do processador TMS320C25, que possui 15 formatos diferentes de instruções. As instruções possuem *opcodes* de diferentes tamanhos (11) que podem ter seus *bits* contíguos ou intercalados por outro campo da instrução (instruções tipo 14 e 15). As instruções podem ser de 16 *bits* (uma palavra)

ou de 32 *bits* (duas palavras, formatos 3, 14 e 15 na figura 5.2). As instruções usam um de três modos de endereçamento diferentes (direto, indireto e imediato). No modo direto de endereçamento o endereço é codificado em um campo da instrução. No modo indireto o formato do operando é (`ind,next`), onde `ind` é uma operação a ser realizada no registrador AR corrente, ou seja incremento (*+) ou decremento (*-) e `next` é o próximo registrador AR que será usado como corrente (a partir da execução da próxima instrução). No modo imediato o valor do operando (endereço) é codificado na instrução. Essa diversidade no formato das instruções impõe mais uma dificuldade s ser vencida no processo de compressão e descompressão de programas que utilizam esse conjunto de instruções.

5.3 Compressão de Programas

Para a compressão de programas para o processador TMS320C25 foi usado o mesmo algoritmo descrito no capítulo 4, seção 4.3 para a compressão de programas para o processador MIPS R2000.

O algoritmo de compressão utiliza como símbolo para a codificação um conjunto de instruções agrupadas nas árvores de expressão. As instruções que compõem uma árvore de expressão são aquelas compreendidas entre a instrução raiz da árvore de expressão anterior até (inclusive) a próxima instrução raiz de árvore de expressão. Uma instrução é raiz de uma árvore de expressão se pelo menos uma das seguintes condições ocorre:

- a instrução armazena uma informação na memória,
- o operando destino da instrução é usado como operando fonte em mais do que uma instrução no mesmo bloco básico,
- o operando destino da instrução é usado como operando fonte em pelo menos uma instrução fora do bloco básico,

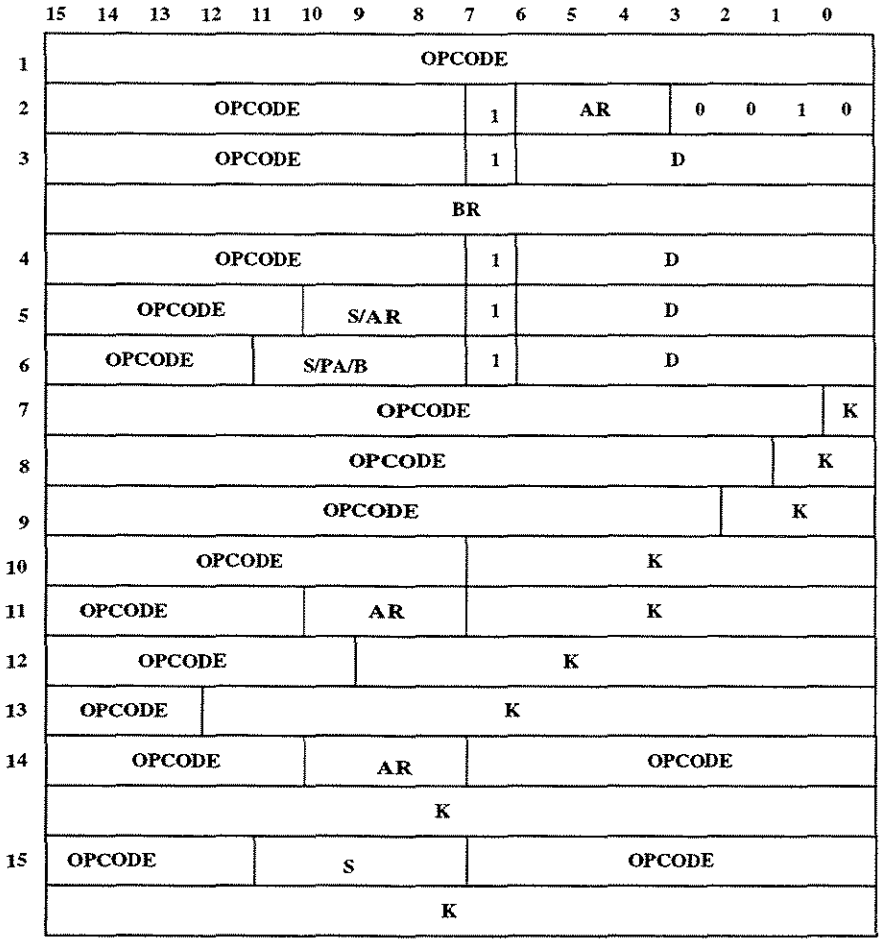

Figura 5.2: Formato das instruções do processador TMS320C25

- a instrução é uma instrução de desvio, e
- a instrução é a última instrução do bloco básico.

Neste trabalho utilizou–se a definição de bloco básico apresentada por Aho *et al* em [1]: a primeira instrução do programa define o início de um bloco básico; instruções alvo de instruções de desvio iniciam um novo bloco básico, instruções de desvio terminam um bloco básico.

A figura 5.3 mostra em (a) um trecho de programa em linguagem de montagem (*as-*

sembler) do TMS320C25, em (b) os blocos básicos e em (c) árvores de expressão.

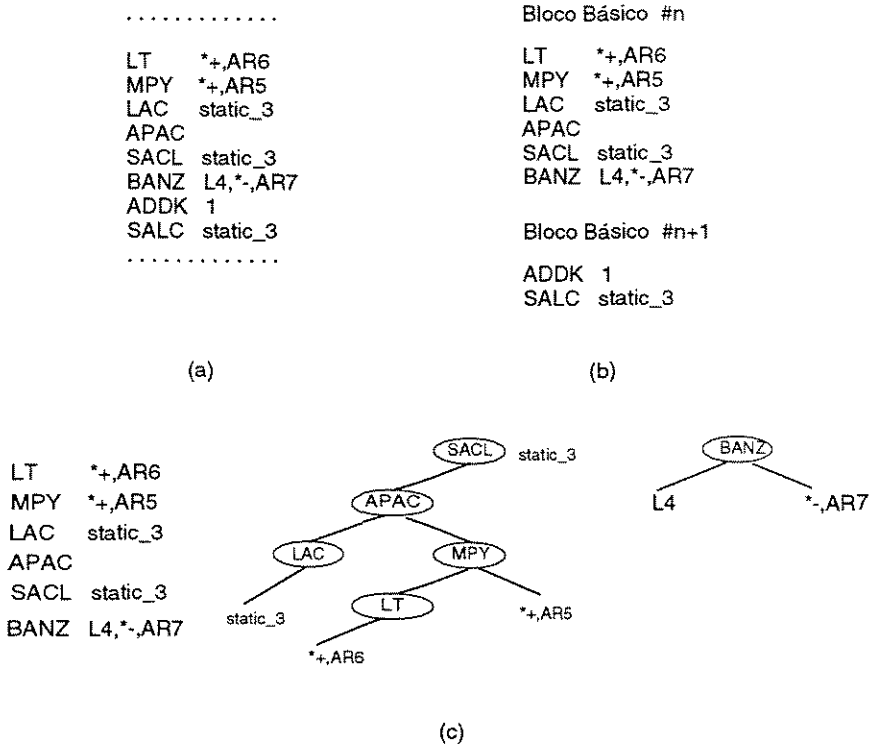

Figura 5.3: Árvores de expressão para o processador TMS320C25.
(a) trecho de programa; (b) blocos básicos; e (c) árvores de expressão.

As árvores de expressão são utilizadas como símbolos básicos da compressão porque os compiladores tendem a gerar árvores de expressão similares durante a geração de código, para estruturas tais como `if-then-else`, `for`, `while`, `repeat` existentes e muito utilizadas pelos programadores de linguagem de alto nível.

Para mostrar o método TBC, foi utilizado um conjunto de programas representativos de aplicações comumente encontradas em sistemas dedicados que utilizam algum tipo de processador DSP. O programa `jpeg` é uma implementação do algoritmo de compressão de imagens JPEG; `bench` é um controlador de *cache* de disco; `gzip` é um compressor de textos; `set` é uma coleção de rotinas de manipulação de *bits* para aplicações em DSP; os

programas `hill` e `gnucrypt` são programas de criptografia de dados e `rx` é uma máquina de estados de um controlador.

Os programas do conjunto de testes foram compilados com opção de otimização (-O2), usando o compilador da Texas Instruments para o processador TMS320C25. O número de instruções, de árvores de expressão e de árvore de expressão distintas são mostrados na tabela 5.1. Em média as árvores distintas representam 37% de todas as árvores do programa.

Programa	Total de Instruções	Total de Árvores	Árvores distintas (%)
aipint2	1403	928	295 (32)
bench	9483	6693	2089 (31)
gnucrypt	3683	1976	750 (38)
gzip	10835	7171	2146 (30)
hill	920	627	292 (47)
jpeg	2305	1124	572 (51)
rx	563	360	121 (34)
set	4565	2798	969 (35)

Tabela 5.1: Número de árvores distintas nos programas.
Os valores entre parênteses são porcentagens com respeito ao número total de árvores de expressão.

Duas árvores de expressão são distintas se elas diferem em pelo menos uma instrução (*opcode* e/ou operandos). Por exemplo, as instruções `ADD *+,AR3` e `ADD *+,AR3,0` são distintas devido a primeira não possuir o operando `0`.

Para efeito de comparação dos resultados obtidos para o método TBC com aqueles da literatura os padrões de árvores e de operandos distintos foram determinados. A tabela 5.2 mostra para cada programa, e para a média dos programas, o número de padrões de árvores e de operandos distintos, o número de árvores de expressão distintas e a diferença, em porcentagem, entre o número de árvores de expressão distintas e o número de padrões

de operandos distintos. Mostrando mais uma vez que essa diferença é pequena, em média 4,2%, e que o código comprimido usando a estratégia definida em [60] contém redundância de informação.

Programa (I)	Seqüências de opcodes (II)	Seqüências de operandos(III)	Árvores (IV)	Diferença.(%) (V)
aipint2	62	291	295	1.4
bench	372	1997	2089	4.6
gnucrypt	193	718	750	4.5
gzip	403	2058	2146	4.3
hill	92	273	292	6.9
jpeg	146	560	572	2.1
rx	50	115	121	4.3
set	218	927	969	4.5
Total	1246	6940	7234	4.2

Tabela 5.2: Número de opcodes (II) e seqüências de operandos (III) distintos, quando comparados com o número de árvores (IV) e (V) é a diferença em porcentagem entre (IV) e (III).

A seleção da forma como os símbolos (árvores de expressão) são codificados para gerar o programa comprimido é semelhante à utilizada para o processador R2000 (capítulo 4). As árvores de expressão foram ordenadas em ordem decrescente pela freqüência com que aparecem no programa. A contribuição acumulada das árvores de expressão para cobertura do programa foi calculada e é mostrada na figura 5.4. O eixo horizontal representa as árvores de expressão distintas ordenadas pela freqüência em ordem decrescente. O eixo vertical a porção do programa coberta pelas árvores de expressão distintas.

Note que, também para o TMS320C25, a distribuição das árvores de expressão distintas tem um comportamento não uniforme e exponencial, como a encontrada para o processador R2000 (capítulo 4) e para os processadores PowerPC, ARM e i386 mostrado por Lefurgy *et al* [43]. Note ainda que em média 70% de todo o programa é coberto por 30% das árvores de expressão distintas (as mais freqüentes).

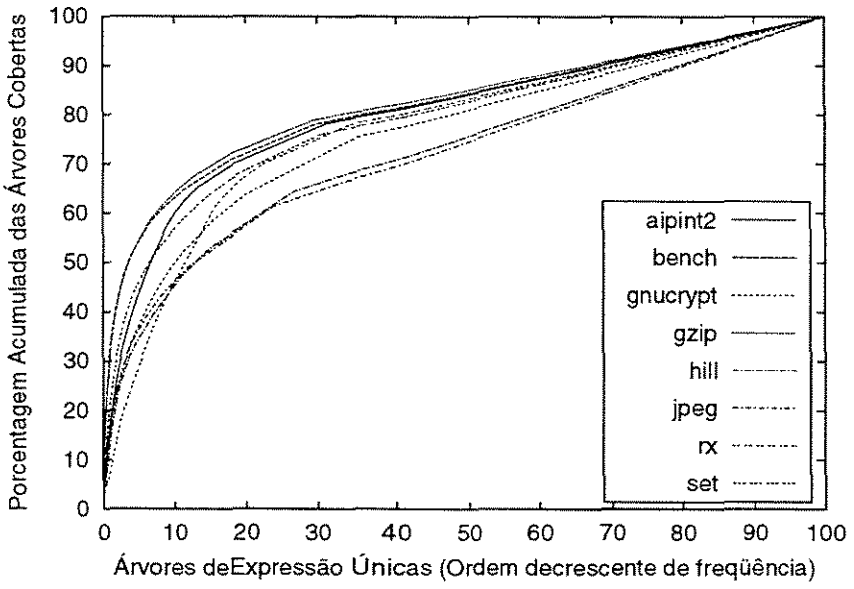

Figura 5.4: Porcentagem das árvores do programa cobertas pelas árvores distintas.

Estes resultados apontam para um esquema de codificação dos símbolos por *codewords* de tamanho variável, como por exemplo o método de Huffman. Porém, como discutido para o caso do R2000, o método de codificação de Huffman não foi usado por exigir uma máquina de descompressão mais complexa, que requer uma maior área de silício e uma grande latência. Desta forma, adotou–se uma codificação mais simples para representar as árvores de expressão no programa comprimido, utilizando *codewords* de dois tamanhos e um *bit* denominado de ESC *bit* para identificá–las. A figura 5.5 mostra o esquema de codificação dos símbolos utilizado. Para o ESC *bit* igual a `zero`, os k *bits* seguintes codificam as 2^k árvores de expressão mais freqüentes, onde k é o número mínimo de *bits* necessários para codificar 30% das árvores de expressão (joelho da exponencial), ou seja $k = \lceil log_2 0.3N \rceil$, onde N é o número de árvores de expressão distintas do programa que está sendo comprimido. Quando o ESC *bit* é `um`, os *bits* seguintes codificam as árvores de expressão menos freqüentes e o número de *bits* necessários é $\lceil \log_2(N - 2^k) \rceil$.

O algoritmo de compressão substitui cada árvore de expressão do programa pela *codeword* correspondente. O código resultante é compactado de forma que todos os *bits* de uma palavra de memória são usados. Isso melhora a razão de compressão, porém tem como conseqüência dois problemas. O primeiro, uma palavra de memória pode ter mais que uma *codeword*. O segundo, uma *codeword* pode ocupar *bits* em até duas palavras consecutivas de memória.

No primeiro caso a máquina de descompressão deve ser capaz de identificar o início de uma *codeword* em qualquer posição (*bit*) de uma palavra de memória. No segundo caso, a máquina de descompressão deve ser capaz de montar a *codeword* capturando parte desta em uma palavra e outra parte na palavra de memória seguinte.

b	CODEWORD

ESC BIT	CODEWORD LENGTH
0	$K = \lceil \log_2 0.3*N \rceil$
1	$\lceil \log_2 (N - 2^K) \rceil$

Figura 5.5: Codificação das árvores de expressão

A tabela 5.3 mostra, para o TMS320C25, a razão de compressão encontrada para cada programa e a média quando utilizados os métodos apresentados em [60] com codificação Huffman e VLC e o método apresentado nesta seção.

Pode–se observar que mesmo usando uma forma de codificação simples para as *codewords* o método TBC produz resultados melhores que os obtidos pelo método Fatoração de Operandos. Em média a compressão usando TBC é 6,9 pontos percentuais melhor do

	Fatoração de Operandos		TBC
Programa (I)	Huffman (%) (II)	VLC (%) (III)	(%) (IV)
aipint2	31,5	33,4	21,5
bench	38,1	45,5	33,0
gnucrypt	33,4	37,4	27,6
gzip	36,3	41,3	27,6
hill	32,7	37,6	26,3
jpeg	37,0	46,2	30,3
rx	36,0	43,6	30,4
set	36,0	43,6	30,4
média	35,0	40,7	28,1

Tabela 5.3: Comparação das razões de compressão para o TMS320C25.

que quando é usado o método de Huffman para codificação dos símbolos, e 12,6 pontos percentuais melhor do que quando é usado o método de codificação VLC. A utilização do método de codificação de Huffman em associação ao método TBC resulta em razões de compressão ainda melhores, porém com reflexos indesejáveis na máquina de descompressão, tornando-a mais complexa, maior e mais lenta.

5.4 Máquina de Descompressão

A máquina de descompressão de programas TMS320C25 é similar à apresentada na seção 4.4 para o processador R2000. A figura 5.6 mostra a máquina de descompressão e como ela é inserida no sistema. Da mesma forma que para o R2000 ela trabalha em duas fases. Na primeira, uma *codeword*, representando uma árvore de expressão, é extraída do programa comprimido. A segunda fase consiste em mapear a *codeword* na seqüência de *opcodes* e operandos (registradores e/ou imediatos). Estas informações são, então, usadas para montar as instruções originais, que são entregues à CPU (ou colocadas na *cache*).

O bloco `extrator` é responsável por identificar e isolar uma *codeword* das palavras

lidas da memória, que contém o programa comprimido. Uma vez extraída uma *codeword*, ela é então passada ao descompressor que a decodifica em um conjunto de *opcodes* e de operandos (registradores e/ou imediatos) e monta as instruções originais que são colocadas em um *buffer* de saída e são entregues à CPU quando da realização de um *fetch* de instrução. Uma unidade de controle é responsável por controlar e sincronizar todo este processo.

A unidade de controle é uma máquina de estados finitos que monitora o *status* da máquina de descompressão e gera os sinais de controles necessários à realização das tarefas.

O bloco extrator e a unidade de controle são "idênticos" aos correspondentes para a máquina de descompressão para o R2000 apresentados na seção 4.4.1 e não serão apresentados aqui. O módulo *descompressor*, que é semelhante ao apresentado na seção 4.4.2, é descrito a seguir.

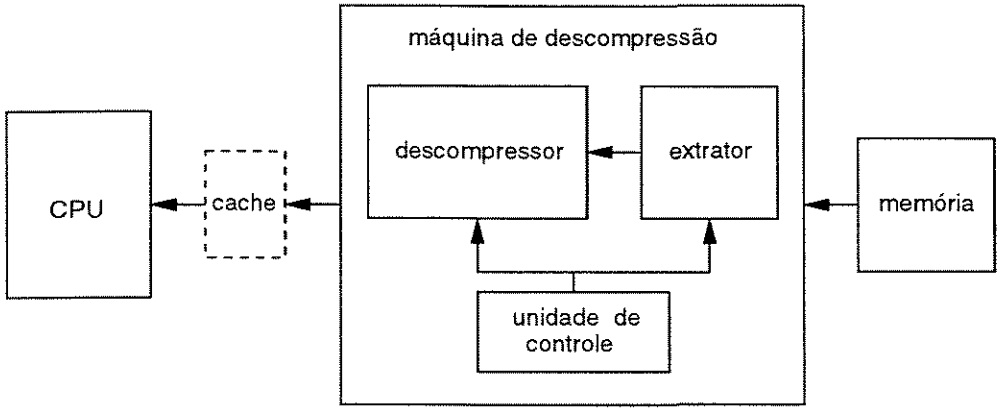

Figura 5.6: Máquina de Descompressão. TMS320C25

5.4.1 Módulo Descompressor

Para o módulo descompressor, foram avaliadas duas alternativas de implementação. A primeira é baseada na solução apresentada para o processador R2000 (seção 4.4.2) que

utiliza um dicionário TPD de padrões de árvores (*opcodes*).

Devido à complexidade do conjunto de instruções do processador TMS320C25, que apresenta instruções com *opcodes* de tamanhos variados (desde 3 *bits* a 16 *bits* como mostrado na figura 5.2), a implementação de um dicionário de árvores para o TMS320C25, que seja eficiente (na área ocupada), deve ser organizada em bancos, onde cada banco armazena um conjunto de *opcodes* de mesmo tamanho. A figura 5.7 mostra o esquema do dicionário. O descompressor então deve gerar o endereço do *opcode* da instrução no dicionário TPD (OPADDR) e a seleção do banco a qual ele pertence (OPSEL). Note que com este esquema os *opcodes* das instruções de uma mesma árvore de expressão não mais estão armazenados em endereços consecutivos no dicionário (como no caso para o R2000) e o descompressor tem que gerar endereços aleatórios para as instruções em uma mesma árvore de expressão. Esta alternativa mostrou-se, em termos da área de silício ocupada, comparável à segunda alternativa de implementação do descompressor que consiste em gerar diretamente os *opcodes* das instruções da árvore de expressão, com a vantagem de não utilizar área de silício para implementar o dicionário TPD e de não ser necessário um acesso à memória (dicionário) para montar as instruções, o que torna a máquina de descompressão mais rápida. Esta última alternativa foi a adotada para implementar a máquina de descompressão para o processador TMS320C25.

A figura 5.8 mostra o descompressor proposto para o processador TMS320C25. O trabalho de descompressão está quase todo concentrado na máquina de estados Gerador de Instruções (IGEN – *Instruction Generator*). Durante um ciclo de máquina, IGEN gera os *bits* dos diversos campos de uma instrução da árvore de expressão associada à *codeword* que está sendo decodificada e os sinais de controle. Os campos e sinais de controle gerados por IGEN são: *opcode* (`opcode`), imediato (`immed`), indireto (`ind`), próximo (`next`), endereço de desvio comprimido (`caddr`) e endereço de desvio não comprimido (`uaddr`). Os diversos sinais são então usados pelo bloco IAB (*Instruction Assembly Buffer*) para montar a

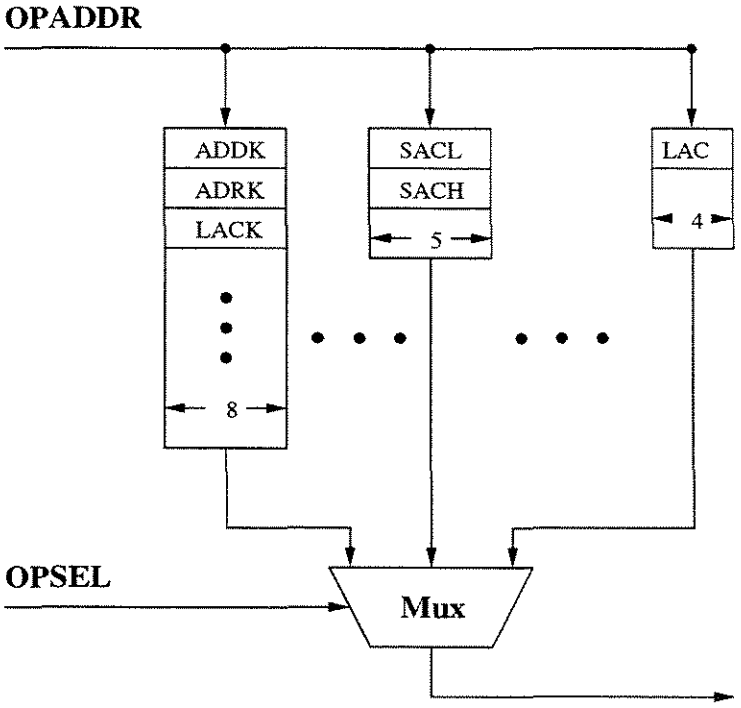

Figura 5.7: Dicionário de árvore de expressão.

instrução e entrega-la à CPU, a cada ciclo de máquina. Uma árvore de expressão pode conter mais que uma instrução. Neste caso, enquanto a CPU executa uma instrução, uma nova instrução está sendo descomprimida sem que seja necessário um novo acesso à memória. O módulo IAB possui um *buffer* de instruções de modo que, se o fluxo de descompressão de instruções for mais rápido que o fluxo de execução de instruções pela CPU, não é necessário interromper o ciclo de descompressão para esperar pela CPU.

Na figura 5.8 o módulo FETCH é responsável pela busca de uma nova *codeword* (contida em uma palavra de memória). Este módulo engloba o `extrator` e o tratamento dos endereços de desvio e será analisado na seção 5.4.2

O tamanho do módulo IGEN não cresce de forma explosiva quando o tamanho do programa cresce (seção 5.5), como poderia ser esperado. A razão disso é que o número de estados da máquina IGEN é limitado pelo número de instruções da maior árvore de

expressão e muitas árvores de expressão distintas têm instruções com campos similares, o que eventualmente compartilham o mesmo pedaço de lógica na máquina IGEN. Por exemplo, as seguintes instruções `ADD *,AR3` e `ADD *-,AR3` são distintas, mas sua decodificação pode compartilhar lógica uma vez que possuem o mesmo *opcode* (`ADD`) e o mesmo campo *next* (`AR3`). A máquina de estados IGEN para cada programa em nosso *benchmark* foi sintetizada e os resultados são apresentados na seção 5.5.

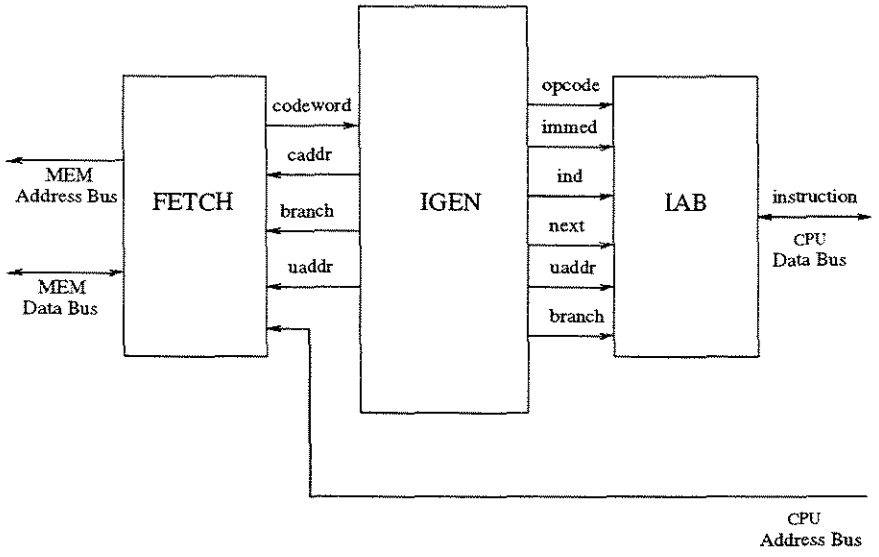

Figura 5.8: Descompressor para o TMS320C25.

5.4.2 Busca de Codewords

O módulo FETCH é composto pelo `extrator`, circuito responsável por isolar uma *codeword* e passá–la para o módulo IGEN, e o circuito que trata as instruções e endereços de desvio.

O `extrator` é idêntico ao usado no R2000 (figura 4.6 (b)) e não será tratado aqui. A leitura de uma palavra da memória de programa é realizada com a ajuda do registrador de dados de memória (MDR – Memory Data Register) e do registrador de endereços de memória (MAR – Memory Address Register), mostrados na figura 5.9. O registrador MDR

é usado para armazenar a(s) palavra(s) de memória que contém a *codeword* associada à árvore de expressão a ser descomprimida (equivale ao *buffer* de entrada da figura 4.6 (a)) e o registrador MAR é usado para armazenar o endereço da próxima palavra a ser lida da memória de programa.

O algoritmo de compressão TBC permite que o início de uma *codeword* possa estar em qualquer *bit* da palavra de memória. Desta forma, o endereço alvo de uma instrução de desvio pode ser qualquer posição dentro de uma palavra de memória (endereços desalinhados). Por outro lado, a CPU só pode manusear endereços descomprimidos (alinhados). Para solucionar esse conflito, IGEN gera dois endereços para cada instrução de desvio que é descomprimida. O endereço descomprimido `uaddr` e o endereço comprimido `caddr`.

Durante a execução do programa, quando uma instrução de desvio é passada (pelo descompressor) para a CPU, o descompressor na próxima requisição de instrução pela CPU compara o endereço gerado pela CPU (endereço do *fetch*) com o endereço `uaddr`. Caso o desvio for tomado os endereços comparados serão iguais e o descompressor faz um acesso à memória usando o endereço `caddr` e extrai, da palavra lida, a próxima *codeword*. Caso contrário, a próxima *codeword* a ser extraída será a seguinte à última (a que gerou a instrução de desvio).

No processador TMS320C25 uma instrução de desvio ocupa duas palavras de memória, onde a segunda é o endereço alvo da instrução de desvio. Assim que um desvio é detectado, IGEN ativa o sinal `branch` que se mantém desta forma até que a próxima instrução seja descomprimida. O módulo IAB adiciona os *bits* em `uaddr` a instrução gerada por por IGEN, montando assim a instrução de desvio que é então entregue à CPU. Ao mesmo tempo o módulo de extração usa os *bits* de `caddr` para determinar: (a) o endereço de memória onde está a *codeword* associada a árvore de expressão alvo do desvio; (b) o *offset* que determina o início da *codeword* (desalinhamento) dentro da palavra. O endereço efetivo da próxima *codeword* depende do resultado da execução da instrução de desvio

passada à CPU. O módulo `extrator` usa o sinal `taken` (figura 5.9) para verificar o *status* do desvio. Se o sinal `taken` = 1, então o desvio foi tomado e se `taken` = 0 a instrução não era de desvio ou o desvio não foi tomado.

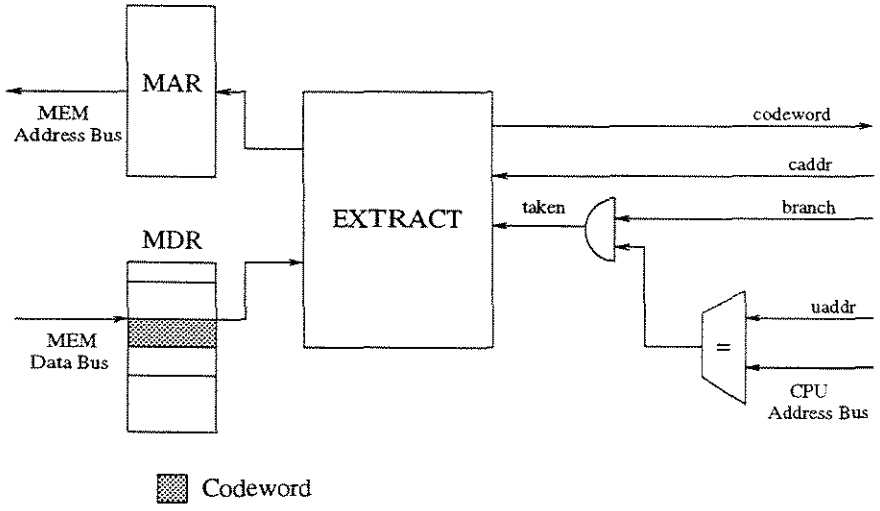

Figura 5.9: Extrator para o TMS320C25.

O sinal `taken` é gerado monitorando–se os endereços gerados pela CPU. Se durante o próximo ciclo de leitura de instrução o conteúdo do barramento de endereços da CPU for diferente do endereço alvo passado para a CPU (ou seja `uaddr` $\neq$ CPU address bus), `taken` = 0 e a próxima *codeword* está em MDR ou ela é a primeira *codeword* da próxima palavra de memória. No primeiro caso ela é extraída do MDR pelo `extrator` e passada a IGEN. No segundo caso MAR é incrementado, é realizada uma leitura de memória e a primeira *codeword* da palavra é extraída e passada para IGEN. Se `taken` = 1 `caddr` é carregado em MAR, é realizada uma leitura de memória, a *codeword* é extraída pelo `extrator` e é entregue a IGEN. Esta abordagem permite que a CPU trabalhe com os endereços do programa original (descomprimido), enquanto o acesso à memória de programa é realizado com os endereços comprimidos (`caddr`). Com isso não é necessária nenhuma modificação na unidade de geração de endereços do processador, como em [43].

5.5 Resultados da Síntese de IGEN

O algoritmo de compressão TBC foi testado com um conjunto de programas típicos de aplicações DSP. A razão de compressão foi calculada. Um código VHDL que descreve a máquina de descompressão foi gerado automaticamente (capítulo 7), usando as *codewords* atribuídas durante a fase de compressão. A máquina de descompressão foi sintetizada com a ferramenta Leonardo Spectrum da Exemplar/Mentor Graphics e biblioteca *standard cell* da AMS, tecnologia CMOS de $0,6\mu m$ e $5,0\ Volts$. A tabela 5.4 mostra, para cada máquina de descompressão, a área (em mm^2) e a freqüência máxima de operação (em MHz) para a síntese orientada para otimização da área. A figura 5.10 mostra esses mesmos dados quando plotados em função do tamanho dos programas (número de instruções). A área média obtida foi de $3,3mm^2$ e a freqüência de operação média de 167 MHz. Essa freqüência de operação é suficiente para operar com a maioria dos DSPs, que operam em freqüências entre 33 MHz e 50 MHz [63]. Isso mostra que a latência do decodificador não terá impacto significativo no desempenho final do sistema. A máquina de descompressão também foi sintetizada com otimização orientada a melhorar a freqüência de operação e em média obteve-se uma freqüência de operação 40% maior e uma área de silício 31% maior do que aquelas obtidas com a síntese orientada a otimizar a área.

A razão final de compressão, considerando a área da máquina de descompressão é apresentada na figura 5.11. A razão de compressão para cada programa é mostrada pelas barras escuras no gráfico da figura 5.11. A razão de compressão média dos programas é 28% contra a razão média do método Fatoração de Operandos de 50% (Huffman) e 58% (VLC). A área da máquina de descompressão foi medida e normalizada com respeito à área necessária para acomodar uma memória ROM do tamanho do programa (em *bits*) quando implementada em uma tecnologia de $0.6\mu m$. A área da máquina de descompressão normalizada está representada na figura 5.11 pelas barras em branco.

Programa	Número de Instruções	Área (mm^2)	Freqüência (MHz)
rx	563	0.5	195
hill	920	0.8	165
aipint2	1403	1.1	179
jpeg	2305	2.1	150
gnucrypt	3683	2.9	157
set	4565	3.5	182
bench	9483	7.5	130
gzip	10835	8.1	137

Tabela 5.4: Área de silício(mm^2) e freqüência máxima de operação (MHz) estimadas para a máquina de descompressão.

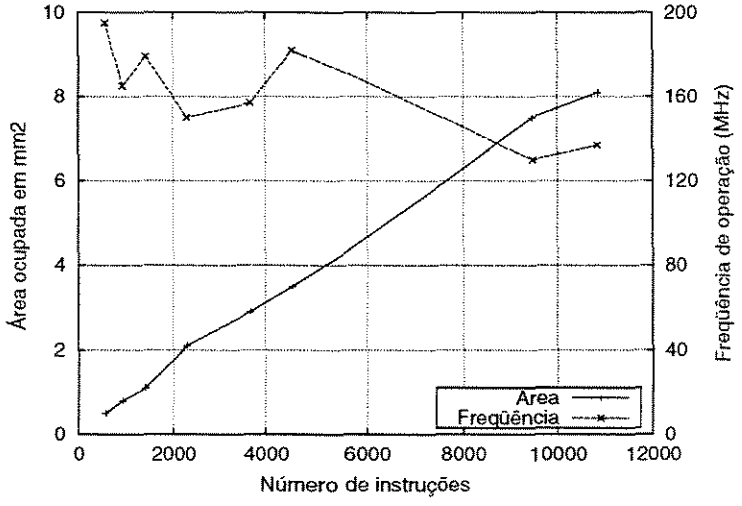

Figura 5.10: Área de silício(mm^2) e freqüência máxima de operação (MHz) estimadas para a máquina de descompressão.

Em média a máquina de descompressão contribui com 47% da razão final de compressão. Quando esse valor é levado em consideração obtém-se uma razão média de compressão de 75%, que é melhor que o valor encontrado por Liao *et al* [48] (82%).

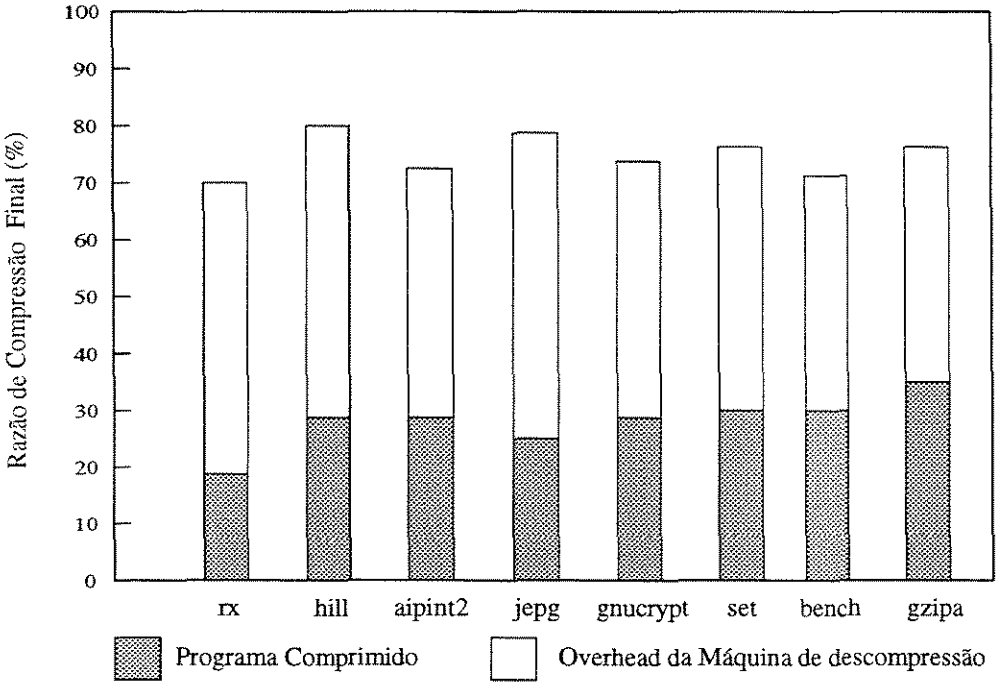

Figura 5.11: Razão de compressão final.

5.6 Conclusões

O método de compressão de programas TBC foi aplicado a um processador dedicado a aplicações de processamento de sinais digitais (DSP). O processador escolhido para os testes foi o processador da Texas Instruments TMS320C25. Esse processador foi o escolhido por se tratar de um DSP muito utilizado pela indústria.

O método TBC foi aplicado a um conjunto de oito programas representativos de aplicações típicas de sistemas embarcados que usam processadores DSP. Uma razão de compressão média de 28% foi encontrada para os programas do conjunto de teste, melhor

que as razões de compressão produzidas pelo método Fatoração de Operandos, 35% para codificação usando Huffman e 40% para codificação usando VLC. Uma proposta para a máquina de descompressão foi apresentada e sintetizada, obtendo-se freqüências de operação superiores a 150 MHz. A razão de compressão média, quando a ela é agregada a área de silício ocupada pela máquina de descompressão sintetizada em *standard cell*, tecnologia CMOS de $0,6\mu m$ e $5,0$ $Volts$ da AMS é de 75%, melhor que a razão de 82% obitida por Liao [48] para o mesmo conjunto de programas e processador.

Capítulo 6

Compressão e Descompressão de Código R4000

6.1 Introdução

Este capítulo apresenta os resultados do estudo do método de compressão TBC quando aplicado a um conjunto de programas para o processador R4000. O processador R4000 foi escolhido para testar o método TBC por ser um processador RISC cuja utilização em sistemas embarcados tem experimentado um grande crescimento. Para efeito de comparação, também são apresentados os principais resultados da aplicação do método Fatoração de Operandos em programas para o processador R4000.

Uma nova técnica para implementação da máquina de descompressão é introduzida com objetivo principal de minimizar o tempo e uso de memória durante o processo de síntese, possibilitando assim a síntese da máquina de descompressão para outros programas além do `compress` e `li` (únicos casos em que a síntese pode ser completada, para o processador R2000).

Este capítulo está organizado da seguinte forma: a seção 6.2 apresenta de forma resumida as características do processador R4000 e o ambiente usado para testar os métodos Fatoração de Operandos e TBC; a seção 6.3 apresenta os dados referentes a compressão de

programas R4000, obtidos usado–se o método Fatoração de Operandos, que é apresentado em Araújo *et al* [8]. O método TBC para o processador R4000 é apresentado na seção 6.4 (compressão) e na seção 6.5 (descompressão). A seção 6.6 apresenta os resultados (área de silício e desempenho) obtidos com a síntese do bloco TGen e a seção 6.7 apresenta algumas considerações e conclusões.

6.2 Programas de Teste e o Processador R4000

Para testar os algoritmos de compressão foi usado um conjunto de programas do *benchmark* SPECInt95 compilados usando–se o compilador *gcc* versão 2.8.1 executando em uma máquina Sun Enterprise E450, gerando código para o processador MIPS R4000 [37]. O processador R4000 possui uma arquitetura RISC clássica com a maioria das características encontradas em processadores RISC modernos, além de ser um processador com crescente uso em sistemas dedicados. O formato do seu conjunto de instruções é idêntico ao formato das instruções do R2000 e o seu conjunto de instruções difere em algumas poucas instruções a mais que possui. A principal diferença em relação ao R2000 está em possuir características de *hardware* mais avançadas no controle do *pipeline* permitindo aos compiladores, quando acionada as opções de otimização de código, gerarem códigos menores e mais eficientes. Por exemplo, nos processadores MIPS as instruções de desvio (*jump* e *branches*) e carga de registradores (*load*) possuem atraso de um ciclo (*delay slot*) para tomar o desvio ou tornar disponível o dado para uso por outra instrução, em relação à execução das outras instruções [37]. O R4000 possui *load interlock* que permite que o atraso na execução dos *loads* seja tratado pelo *hardware*, enquanto que no R2000 (não possui *load interlock*) esse atraso, quando há dependência de dados (*data hazard*), é tratado pelo compilador, reorganizando as instruções para evitar os problemas decorrentes da dependência de dados. Os compiladores também reorganizam as instruções para tratar

os desvios, preenchendo o *delay slot* com instruções que sempre serão executadas (sendo o desvio tomado ou não) e que não alteram o resultado final da execução do programa. Quando uma reorganização de instruções não é possível o compilador preenche o *delay slot* com uma instrução NOP (*No OPeration*). No R2000 a ocorrência de NOPs, introduzidos pelo compilador, é maior que no R4000, pois neste último somente os *delay slots* devidos à desvios, que não foram resolvidos por rearranjo de instruções, precisam ser preenchidos pelo compilador com NOPs, uma vez que a dependência de dados que não pode ser tratada com rearranjo de instruções é tratada pelo *hardware* por meio do mecanismo de *data interlock* [31, 37]

A tabela 6.1 mostra o tamanho do código gerado pelo compilador *gcc* (em número de instruções) para o conjunto de programas de teste compilados para a arquitetura R2000 (*mips1*) e para o R4000 (*mips2*), quando usado as chaves de otimização -O2 e -Os. A chave -O2 gera código otimizado para desempenho e tamanho (inclui a maioria das opções de otimização do compilador). A chave -Os é similar à -O2, porém só utiliza as chaves de otimização que não aumentam o tamanho do código. Neste trabalho foram utilizados os códigos dos programas de testes gerados pela compilação que produziu os menores códigos (chave de compilação -Os).

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 6.1: Parâmetros de compilação e número de instruções geradas; mips1 (R2000) e mips2 (R4000).

6.3 Fatoração de Operandos Aplicado ao R4000

A técnica Fatoração de Operandos usa como símbolos para a compressão os padrões de árvore e de operandos, obtidos após a fatoração dos operandos das árvores de expressão. Para maiores detalhes como isso é realizado refira–se ao capítulo 3 e [60, 9].

A tabela 6.2 mostra, para os programas do conjunto de teste, o número total de árvores de expressão do programa, o número de árvores de expressão distintas e o número de padrões de árvore e de operandos. Os valores dados entre parênteses são a percentagem de padrões e árvores de expressões distintos em relação ao número total de árvores de expressão do programa. Note que para os programas maiores (exemplo, *gcc*) as percentagens de padrões e árvores distintas são menores que os mesmos percentuais para os programas menores (exemplo, *compress*), o que mostra que os programas maiores possuem mais redundância, apresentando maiores oportunidades de compressão.

Programa (I)	Árvores (II)	Árvores Distintas (III)	Padrões Árvore (IV)	Padrões Operandos (V)	(III) - (V) (VI) (%)
compress	1844	832 (45,1)	107 (5.8)	767 (41.6)	8.5
gcc	291758	51186 (17,5)	921 (0.3)	45469 (15.6)	12.6
go	62423	12460 (19,9)	256 (0.4)	11373 (18.2)	9.6
ijpeg	40621	11264 (27,7)	348 (0.9)	9907 (24.4)	13.7
li	15509	3072 (19,8)	169 (1.1)	2840 (18.3)	8.2
perl	57276	12793 (22,3)	547 (1.0)	11579 (20.2)	10.5
vortex	130336	17493 (13,4)	324 (0.2)	15592 (12.0)	12.2
média	85681	15585 (23,2)	382 (1.4)	13932 (17.4)	10.8

Tabela 6.2: Número de padrões de árvore e de operandos em um programa. Os valores em parênteses são percentagens com respeito ao número total de árvores de expressão.

Como visto na seção 4.3 a ordenação em ordem decrescente de freqüência de ocorrência traz mais benefícios para a compressão (menor razão de compressão) do que a ordenação em ordem decrescente da contribuição em *bits* (freqüência x tamanho) na cobertura do

programa. Desta forma os padrões de árvores e de operandos foram ordenados em ordem decrescente de freqüência de ocorrência e as figuras 6.1 e 6.2 apresentam a percentagem acumulada da cobertura de todos os padrões de árvores e de operandos do programa. Em média 20% dos padrões de árvores cobrem quase a totalidade das árvores do programa e 20% dos padrões de operandos cobrem aproximadamente 80% de todas as seqüências de operandos do programa.

Os padrões de árvore e de operandos foram, então codificados separadamente, usando-se um algoritmo de atribuição de *codewords* similar ao usado no método TBC para o R2000 (seção 4.3), porém mais eficiente. A *codeword* possui dois campos, um prefixo que indica a classe a qual ela pertence e o outro é a codificação do símbolo (seção 6.4, figura 6.4). O campo classe codifica o tamanho da *codeword* em *bits*, agora permitindo mais do que dois tamanhos para as *codewords*. Este algoritmo de atribuição de *codewords* aos símbolos também foi utilizado no método TBC aplicado ao R4000 e será objeto de análise na seção 6.4. Após encontrada a melhor codificação para os símbolos (a que produz menor razão de compressão), os padrões de árvore e de operandos são codificados e o programa comprimido é gerado com os pares $< Tp, Op >$. A tabela 6.3 mostra as razões de compressão dos padrões de árvore e de operandos e a razão de compressão final para cada programa e a razão de compressão média de todos os programas (39,8%), quando usado 4 classes (tamanhos) de *codewords*.

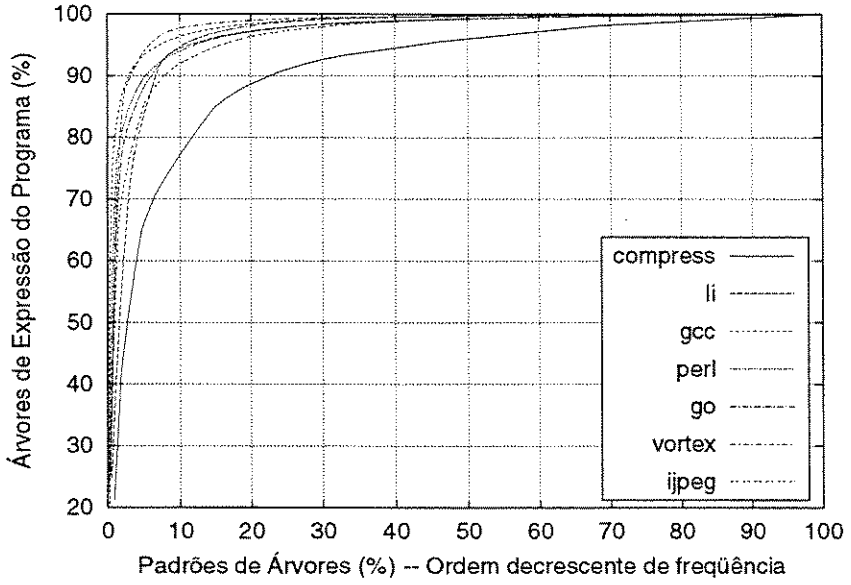

Figura 6.1: Percentagem acumulada do programa coberto pelos padrões de árvores. Fatoração de Operandos.

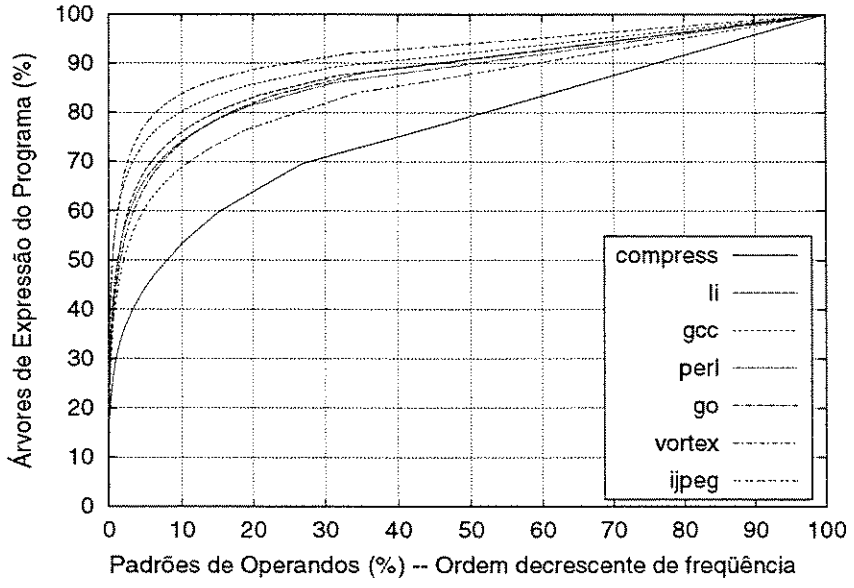

Figura 6.2: Percentagem acumulada do programa coberto pelos padrões de operandos. Fatoração de Operandos.

Programa	Razão de Compr. – Tp	Razão de Compr. – Op	Razão Composta (%)
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
média	26,7	13,0	39,8

Tabela 6.3: Razão de compressão composta quando combinados os padrões de árvores e de operandos. Fatoração de Operandos.

6.4 Compressão Baseada em TBC

Como visto anteriormente, o método de compressão TBC usa como símbolo básico para a compressão as árvores de expressões únicas. As árvores de expressão são obtidas, do programa que está sendo objeto de compressão, analisando o código do programa e aplicando a definição de árvore de expressão apresentada por Aho *et al* [1].

A tabela 6.2 mostra para cada programa o número total de árvores de expressão e o número de árvores de expressão distintas (únicas). A partir da tabela 6.2 nota–se que o número de árvores de expressões únicas em um programa é somente 23,2% do número total de árvores de expressão.

A seleção do melhor método para codificar uma árvore (símbolo) depende de sua contribuição na cobertura total do programa. Desta forma as árvores de expressão foram ordenadas em ordem decrescente pela freqüência com que ocorrem no programa. A distribuição acumulada das árvores de expressão distintas no programa foi computada e a figura 6.3 mostra este resultado. No eixo horizontal do gráfico temos as árvores de expressão distintas ordenadas em ordem decrescente pela freqüência e no eixo vertical a

contribuição acumulada das árvores distintas na cobertura do programa. Note que a distribuição por freqüência das árvores de expressão distintas nos programas é não uniforme e que em média 80% de todo o programa é coberto por somente 20% (as mais freqüentes) das árvores de expressões únicas.

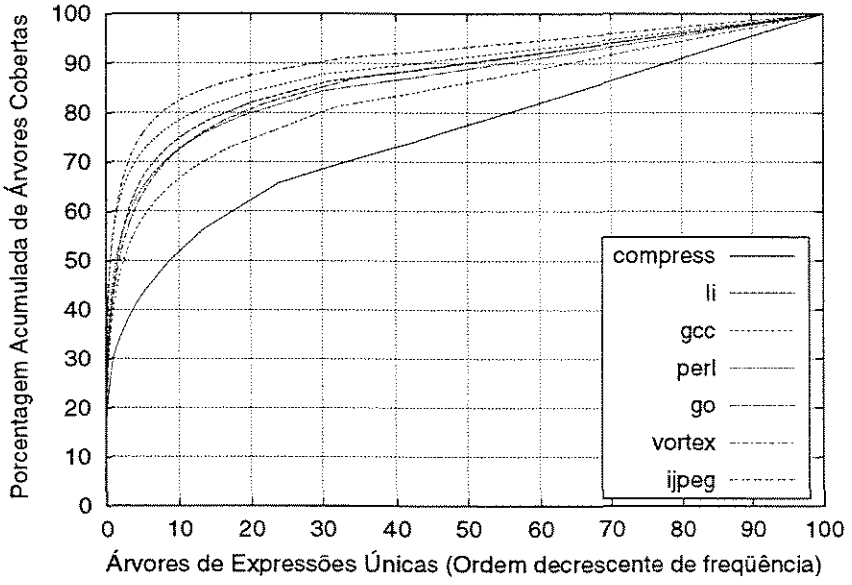

Figura 6.3: Percentagem do programa coberto pelas árvores de expressão. TBC

Novamente, esse fato sugere um método de codificação das *codewords* associadas a cada árvore de expressão tal que aquelas mais freqüentes recebam uma codificação menor e as menos freqüentes uma codificação maior. O método de Huffman [34] poderia ser utilizado. Porém, pelos motivos já expostos na seção 4.3 o método de Huffman não foi utilizado para codificar as árvores de expressão, uma vez que a máquina de descompressão seria mais lenta e ocuparia maior área de silício.

Com objetivo de simplificar a máquina de descompressão, foi utilizado um outro algoritmo para codificação das *codewords* que também explora a natureza exponencial da distribuição das árvores de expressão distintas no programa e que permite uma implementação mais simples da máquina de descompressão. O algoritmo utilizado aqui é similar ao

utilizado nos capítulos 4 e 5, porém a escolha da codificação é mais elaborada permitindo uma melhor razão de compressão.

O algoritmo de codificação das *codewords* divide o conjunto de árvores distintas em n_c classes (e não somente em duas classes como para o R2000 e para o TMS320C25), com cada classe k tendo n_k árvores. O número de classes n_c é determinado, por um programa, de forma exaustiva, explorando todas as possíveis partições para um número de classes entre 2 a 8 classes. Para cada número de classes, novamente de forma exaustiva, são determinadas todas as possíveis combinações de tamanho de cada classe e determinada a sua razão de compressão. Após determinadas todas as razões de compressão é então selecionada para codificar as *codewords* aquela partição que produz a menor razão de compressão.

Para cada árvore em uma classe k é atribuída uma codificação de tamanho fixo de $\lceil log_2 n_k \rceil$ *bits* e anexado à *codeword* um prefixo de tamanho $\lceil log_2 n_c \rceil$ *bits*, que é usado pela máquina de descompressão para identificar a qual classe pertence a *codeword*. O formato final de uma *codeword* é mostrado na figura 6.4.

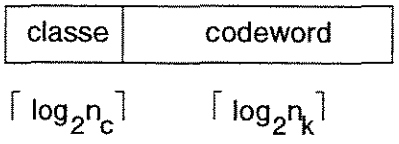

Figura 6.4: Codificação das árvores de expressão.

Considere, por exemplo o programa `li` e uma partição das árvores de expressão distintas com quatro classes. A tabela 6.4 mostra as possíveis combinações de tamanho para as *codewords*, usando quatro classes e a razão de compressão (está incluído no cálculo o prefixo) associada a cada combinação. A melhor razão de compressão (23,4%) é alcançada quando é atribuído 1/5/8/12 *bits* para as classes I/II/III/IV respectivamente.

Tamanho da *Codeword*				Razão de
I	II	III	IV	Compressão (%)
1	1	1	12	30.3
1	1	2	12	29.2
.	.	.	.	.
1	**5**	**8**	**12**	**23.0**
1	5	9	12	23.5
.	.	.	.	.
9	9	8	12	31.2
9	9	9	12	30.5

Tabela 6.4: Possíveis combinações de tamanhos das *codeword* para o programa `li` usando quatro classes (I a IV). TBC.

Uma vez determinada a melhor razão de compressão para uma dada partição, repetimos o algoritmo para outra partição. A figura 6.5 mostra as melhores razões de compressão obtidas pelo método descrito acima para os programas do conjunto de teste, quando particionados de duas a oito classes. Note que a razão de compressão diminui quando o número de classes aumenta até atingir um mínimo, a partir do qual começa a crescer quando o número de classes cresce. Isto ocorre devido ao algoritmo associar *codewords* menores a árvores mais freqüentes e *codewords* maiores a árvores menos freqüentes. Quando o número de classes aumenta, os novos prefixos necessários à identificação das classes crescem e o *overhead* associado ao prefixo passa a ser maior que os benefícios alcançados com o aumento do número de classes, piorando a razão de compressão. Um fato interessante que pode ser notado é que, para a maioria dos programas do conjunto de teste, a melhor razão de compressão foi encontrada para uma partição com quatro classes. Nos outros casos (programa `go`) a melhor razão de compressão ocorre para cinco classes, no entanto a diferença entre as razões de compressão para quatro e cinco classes é de apenas 0,09%.

Usando o algoritmo acima, foram determinadas as melhores razões de compressão para cada programa e a tabela 6.5 mostra estas razões bem como o número de bits usado pelas

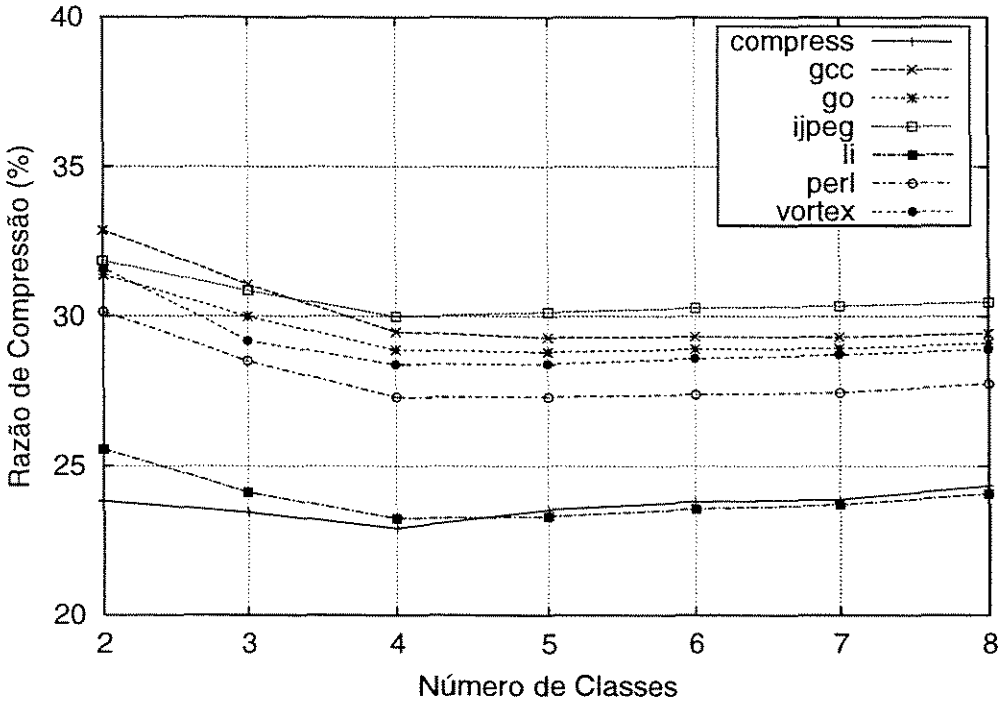

Figura 6.5: Razão de compressão para diferentes partições. TBC.

codewords em cada classe. O cálculo das razões leva em consideração o número de *bits* do prefixo e para o programa go foi usado quatro classes. A média entre as razões de compressão para os programas estudados foi de 27,2%.

Como nos casos apresentados anteriormente (R2000 e TMS320C25) foi permitido que as *codewords* estejam alinhadas a *bit* e que uma *codeword* possa ter uma parte em uma palavra de memória e a outra na palavra de memória seguinte.

6.5 Máquina de Descompressão

Esta seção apresenta a máquina de descompressão utilizada para descomprimir os programas R4000. Como nas demais máquinas de descompressão apresentadas (seções 4.4.2 e 5.4.1) ela trabalha em duas fases. Uma fase é dedicada a extração das *codewords* (`Tc`) das palavras de memória e é realizada da mesma forma que para o R2000 e para o TMS320C25.

Programa	Tamanho da *Codeword*				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 6.5: Partição das classes que resulta na melhor razão de compressão para quatro classes (I a IV). TBC.

Na outra fase (figura 6.6), `Tc` é decodificada pela lógica combinacional `TGEN` (diferentemente que nos casos para o R2000 e TMS320C25, onde é usado uma máquina de estados finitos) e convertida no endereço `tdaddr`. O endereço `tdaddr` aponta para a entrada no dicionário de árvores de expressão `TD` (*Tree Dictionary*) que contém a primeira instrução da árvore de expressão correspondente à Tc. Cada entrada do dicionário `TD` é composta por dois campos: `INSTR` e `END` (no caso do R2000 uma entrada do dicionário possui `opcode`, `itype` e `end` e no TMS320C25 o dicionário inexiste). O campo `INSTR` possui 32 *bits* e contém uma instrução da árvore de expressão descomprimida. O campo `END` é de 1 *bit* e indica se a instrução é a última instrução da árvore de expressão corrente ("1") ou não ("0"). O *bit* `END` é usado para carregar um novo valor no contador `INC`. Se a instrução corrente não é a última instrução da árvore de expressão corrente, `INC` é incrementado e aponta para a próxima instrução da árvore de expressão, caso contrário é carregado um novo valor de `tdaddr` em INC, iniciando a execução de uma nova árvore de expressão. Na figura 6.6, o trecho de programa que aparece no dicionário TD é uma árvore de expressão com três instruções , sendo a instrução `store word` (`sw`) a última instrução da árvore.

Para o R4000 não foi utilizada uma máquina de estados finitos para realizar a descompressão dos programas devido terem sido encontradas algumas dificuldades para síntese

de tais máquinas, principalmente no que diz respeito à memória utilizada e o tempo de execução do programa de síntese. Desta forma optou-se por implementar uma solução que, embora não resulte na melhor razão de compressão (quando considerado a área de silício ocupada pela máquina de descompressão), tornou possível a síntese, com o *software* e máquina disponíveis a sua realização. O capítulo 7 aborda a metodologia de síntese e os problemas e soluções encontrados durante a síntese das máquinas de descompressão.

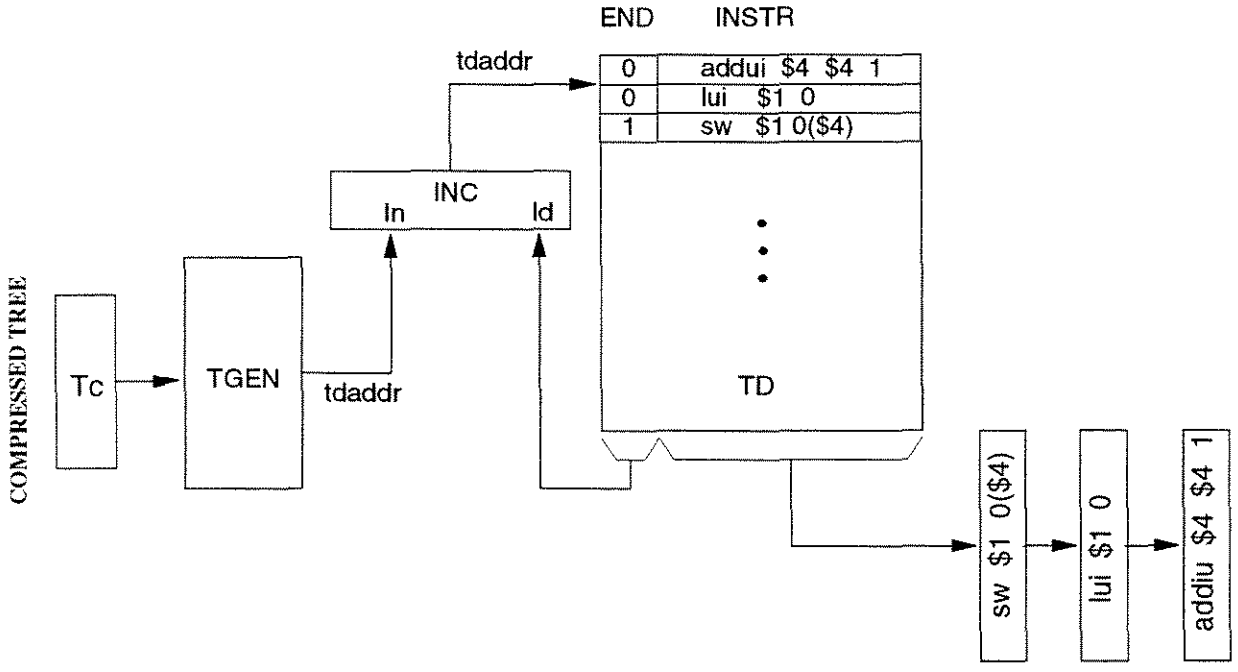

Figura 6.6: Máquina de descompressão para a Compressão baseada em árvore de expressão (TBC).

6.5.1 Instruções e Endereços de Desvio

No modelo proposto, o processador executa instruções descomprimidas que geram endereços também descomprimidos, enquanto na memória estão armazenadas instruções comprimidas (*codewords* representando as árvores de expressão). Durante a execução de uma instrução de desvio (*branch e jump*) o endereço de memória requisitado pela CPU difere do endereço real da próxima instrução (endereço comprimido). Para resolver esse

problema é necessário que a máquina de descompressão seja capaz de mapear o endereço descomprimido fornecido pela CPU no endereço comprimido. Esse mapeamento é realizado pelo módulo mostrado na figura 6.7, onde o mapeamento é realizado usando-se uma tabela de mapeamento de endereços denominada de ATT (*Address Translation Table*). O bloco SHIFT (figura 6.7) é implementado da mesma forma que o extrator para o R2000 (figura 4.6).

Quando o processador busca por uma instrução, primeiro é verificado se a instrução já está na *cache* de instruções. Caso a instrução esteja presente na *cache* ela é entregue ao processador pelo controlador da *cache*. Na presença de um *miss*, será requisitado da memória uma linha de *cache* (*cache-line*). A memória contém *codewords* (que serão descomprimidas em uma árvore de expressão) e estão em endereços diferentes daqueles gerados pela CPU. Neste ponto a máquina de descompressão é acionada. Os *bits* mais significativos do endereço gerado pela CPU (A_{5-n}, onde $n - 5$ é a quantidade de *bits* necessária para fazer acesso a todas as entradas da tabela ATT) são usados para fazer acesso a uma entrada na tabela ATT. As entradas da tabela ATT possuem dois campos ADDR e OFFSET. O campo ADDR é o endereço, na memória comprimida, da árvore de expressão comprimida (*codeword*) que contém a primeira instrução da linha de *cache* que está sendo buscada pelo controlador da *cache* e o campo OFFSET determina qual é essa instrução. A máquina de descompressão obtém a *codeword* da palavra lida da memória comprimida (bloco SHIFT da figura 6.7) e a descomprime fazendo o mesmo com as *codewords* seguintes até preencher por completo a linha de *cache* que é entregue ao controlador da *cache*. Na figura 6.7 são descartados os 5 *bits* menos significativos do endereço gerado pela CPU por ter sido considerado *caches* com linhas de 8 palavras (32 *bytes*). Para *caches* com tamanho de linha de *cache* diferente de 8 palavras a quantidade de *bits* a serem descartados devem ser ajustados para o novo tamanho da linha de *cache*. O tamanho da linha da *cache* influencia diretamente o tamanho (em número de entradas)

da tabela ATT que por sua vez influencia a razão de compressão final (quando considerado o programa comprimido e a máquina de descompressão), porém o peso relativo da tabela ATT na razão de compressão final é pequeno como pode ser visto na figura 6.9.

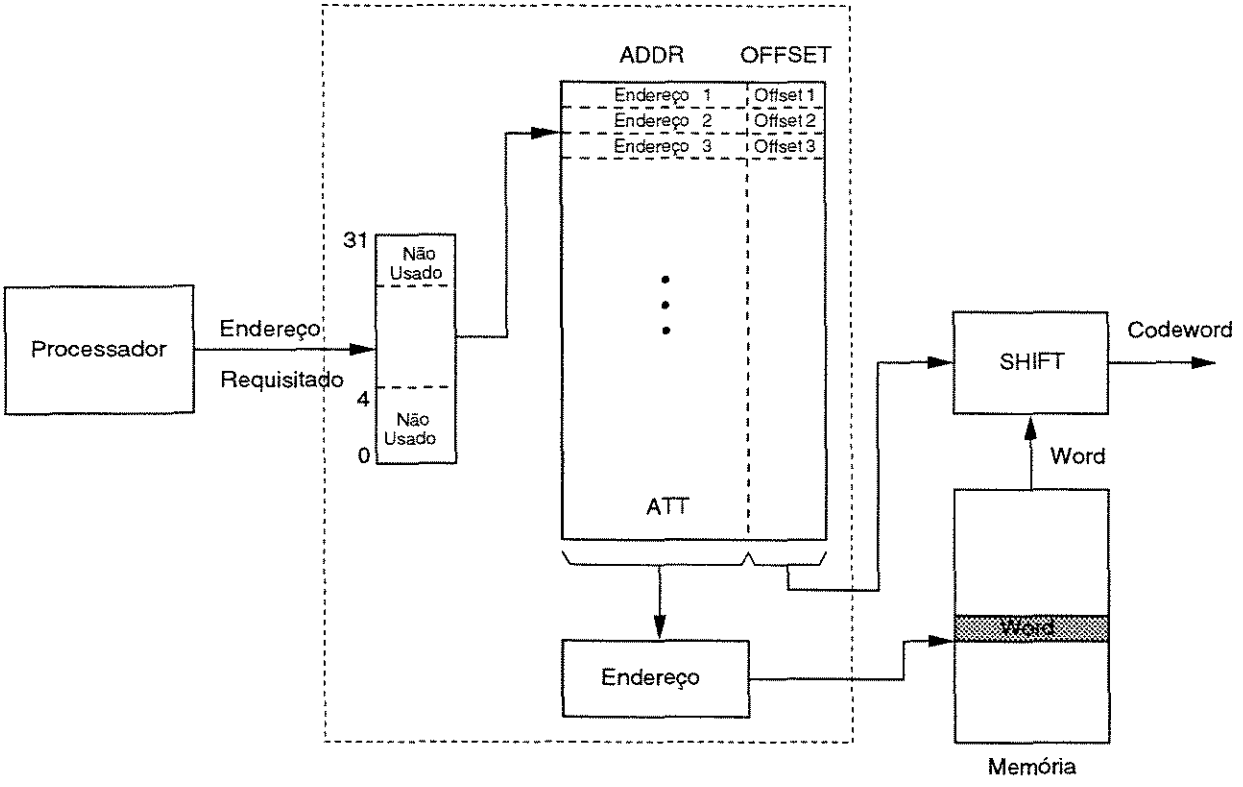

Figura 6.7: Address Translation Table (ATT).

A latência desta abordagem é principalmente determinada pelo tempo de busca na memória da seqüência de palavras na qual está a *codeword* requisitada mais o tempo para decodificá–la. Após este passo, a máquina de descompressão usa um contador interno para buscar as demais palavras que são necessárias para completar a linha de *cache* descomprimida. Pode–se ter um ganho em desempenho se a máquina de descompressão for mais rápida que a memória, o que permite uma entrega de instruções à *cache* mais rápida, uma vez que as *codewords* são menores que as instruções e pode existir mais do que uma *codeword* em cada palavra de memória.

6.6 Síntese de TGen

O algoritmo de compressão TBC apresentado neste capítulo foi testado em um conjunto de programas do SPECInt95 compilados com o compilador gcc para o processador R4000. A máquina de descompressão foi descrita em VHDL e é composta pelos seguintes módulos:

TGen – gera a partir de uma *codeword* o endereço inicial da árvore de expressão associada a ela

TD – dicionário (memória) que contém as instruções do programa agrupadas em árvores de expressões únicas

ATT – tabela, em memória, usada para a conversão dos endereços gerados pela CPU (não comprimidos) nos endereços comprimidos

SHIFT – responsável pelo alinhamento e extração das *codewords* de uma palavra de memória. Composto por um *buffer* de entrada e uma rede de alinhamento (*barrel shifter*), figura 4.6

O módulo TGen, diferentemente do seu similar para o R2000 (TRIGen, figura 4.7) e para o TMS320C25 (IGen, figura 5.8), é um circuito combinacional e não uma máquina de estados finitos. TGen foi sintetizado usando–se a ferramenta `Leonardo Spectrum` da Exemplar/Mentor Graphics utilizando–se uma biblioteca *standard cell* da AMS, tecnologia CMOS de $0,6\mu m$ e $5,0\ Volts$. A tabela 6.6 mostra, para cada programa, a área (em mm^2) ocupada e a freqüência (em MHz) máxima de operação quando a síntese é orientada a minimizar a área de silício. A figura 6.8 mostra esses mesmos dados quando plotados em função do tamanho dos programas (número de instruções).

Para o programa gcc não foi possível realizar a síntese, uma vez que a ferramenta `Leonardo` aborta a execução por problemas no uso da memória quando a memória

alocada para o programa ultrapassa 2 G*Bytes* (a pesar da máquina utilizada possuir 4 G*Bytes*). Como mencionado na seção 6.5 essa solução para a máquina de descompressão foi apresentada para contornar a limitação do uso de memória pelas ferramentas de síntese utilizadas, porém o problema persiste com o programa `gcc`. Por outro lado este programa não representa uma aplicação típica de sistemas embarcados e seu uso no conjunto de testes foi motivado pelo fato deste programa ter sido utilizado para testar métodos de compressão de programas apresentados em outros trabalhos (que usam o SPECInt95 como *benchmark*). Esse problema apresentado pelas ferramentas no uso de memórias muito grandes deverá ser resolvido com as versões das ferramentas para sistemas que usam 64 *bits* para endereços, como por exemplo o `Solaris 7`.

Programa	Número de Instruções	Área (mm^2)	Clk. Freqüência (MHz)
compress	2152	1,04	197,2
go	72516	7,95	99,6
ijpeg	47988	6,73	113,4
li	18448	2,87	149,3
perl	69536	8,29	101,0
vortex	151348	12,71	89,7

Tabela 6.6: Área de silício (mm^2) e freqüência máxima de operação (MHz) estimadas para a máquina de descompressão.

A razão de compressão final para cada programa é mostrada na figura 6.9 onde está discriminada a razão de compressão devida ao programa comprimido, à tabela ATT e à máquina de descompressão (TGen e TD). A razão média de compressão dos programas é de 27,2%, a da tabela ATT é de 3,8% e a razão de compressão média das máquinas de descompressão (IGen e TD) é de 29,7%.

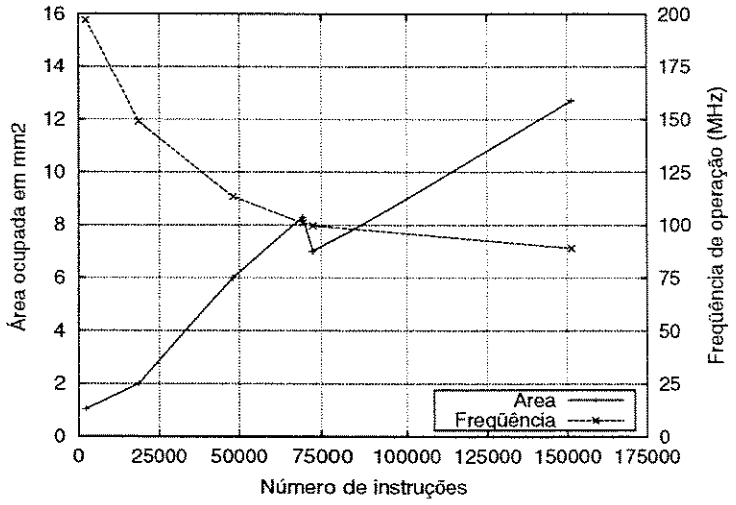

Figura 6.8: Área de silício(mm^2) e freqüência máxima de operação (MHz) estimadas para a máquina de descompressão.

6.7 Conclusões

Este capítulo apresentou os resultados do método de compressão de programas Fatoração de Operandos quando aplicado a um conjunto de programas compilados para o processador R4000. Neste caso os símbolos de compressão (padrões de árvores e de operandos) foram codificados com *codewords* com quatro tamanhos distintos e foi encontrada uma razão de compressão média para os programas de 39,8% (tabela 6.3). Também foram apresentados os resultados do método de compressão de programas para descompressão em tempo real TBC quando aplicado ao mesmo conjunto de programas de teste, tendo como resultado para a razão de compressão média para os programas de 27,2%, 12,6 pontos percentuais melhor que o método Fatoração de Operandos.

Uma implementação alternativa para a máquina de descompressão que não faz uso de uma máquina de estados finitos foi proposta. Esta nova maneira de implementar a máquina de descompressão possibilita que a tarefa de síntese seja terminada com sucesso para programas bem maiores (em número de instruções) que aqueles implementas pela

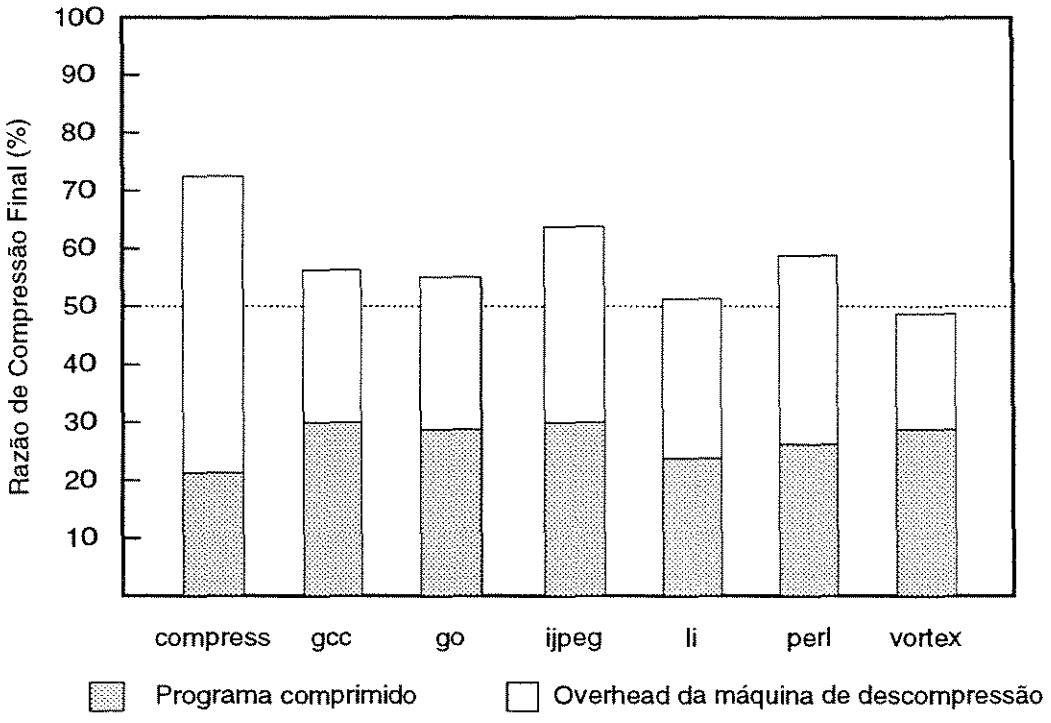

Figura 6.9: Razão de compressão Final para o TBC.

solução apresentada no capítulo 4. A dificuldade na síntese decorre de limitações das máquinas e principalmente das ferramentas utilizadas para essa tarefa e é estudada no capítulo 7. O preço pago para tornar a síntese possível é uma pior razão de compressão final (quando a área ocupada pela máquina de descompressão é levada em consideração). Uma razão de compressão final média de 60,7% foi obtida para o processador R4000.

Capítulo 7

Descrição e Síntese do Descompressor

Este capítulo descreve o processo usado para gerar a descrição VHDL e a síntese dos descompressores a partir do código executável dos programas, bem como os problemas encontrados durante esse processo (especialmente a síntese) com o uso das ferramentas disponíveis. Com objetivo de reduzir o tempo de projeto e tornar o processo automatizado foram escritos programas que tratam os dados disponíveis (seção 7.1) e geram automaticamente a descrição VHDL do descompressor (seção 7.2) que depende do programa que está sendo comprimido. Os programas foram escritos na linguagem C e compilados com o compilador *gcc*.

7.1 Dados Inicialmente Disponíveis

Os programas do SPECInt95 usados para testar os métodos de compressão e descompressão apresentados neste trabalho foram compilados com o compilador *gcc* (projeto GNU), gerando-se código executável otimizado (opção -O2). Utilizando-se o programa `objdump` foi então gerado, a partir do código executável, o código `assembler` que foi analisado, identificando-se os blocos básicos, árvores de expressão, padrões de árvores e de operandos. Diversos dados estatísticos (freqüência de ocorrência dos padrões de árvores e

de operandos, contribuição acumulada dos padrões de árvores e de operandos) e diversos arquivos contendo a descrição do programa com base nos padrões de árvores e de operandos foram criados. Esse tratamento inicial dos programas foi realizado por Pannain e apresentado em [60].

Para gerar os dados necessários a implementação do método TBC (descrição VHDL do descompressor, dicionário e programa comprimido), apresentados neste trabalho (capítulos 4, 5 e 6) foram usados os arquivos `*.sequence`, `*.dictionary` e `*.dictionary-.sequence`[1] provenientes do trabalho realizado por Pannain. As figuras 7.1, 7.2 e 7.3 apresentam o formato e os dados[2] contidos nesses arquivos.

```
Code Sequence Tree

1909

 N.    Tree    Sequence  Target
 0       4        27       -1
 1       1        15       -1
 2       2        43       -1
 ...
 33      2        662      -1
 34      127      589      0
 35      15       27       -1
 ...
 100     8        188      -1
 101     12       558      110
 103     1        578      -1
```

Figura 7.1: Trechos do arquivo compress.sequence

A figura 7.1) mostra um trecho do arquivo `compress.sequence` que descreve o pro-

[1] Aqui, * deve ser substituído pelo nome de um dos programa de teste. Por exemplo, `compress`, `go`, `aipint2`, `rx`, etc.

[2] Estes dados foram retirados dos arquivos referentes ao programa de teste *compress* compilado para o processador R2000

grama executável por meio da seqüência de ocorrência das árvores de expressão, expressas por meio dos padrões de árvores e de operandos no programa.

A coluna `N.` é um inteiro e determina, no programa, a ordem em que um par de padrões (de árvore e de operandos) ocorre no programa. As colunas `Tree` e `Sequence` determinam qual é o padrão de árvore e o de operando que ocorrem na posição dada pela coluna `N.`, respectivamente. Cada padrão (árvore ou operando) é representado por um inteiro que determina a sua posição, em relação aos seus pares, quando estes são ordenados por ordem decrescente de sua contribuição (em *bits*) na cobertura do programa. A coluna *Target* é igual a -1 quando a árvore de expressão, formada pelos padrões de árvore e de operandos correspondentes, não termina com uma instrução de desvio e é um valor inteiro maior ou igual a zero quando a árvore de expressão termina com uma instrução de desvio. O valor informado, na coluna *Target*, corresponde à posição (dada pela coluna *N*) para onde o fluxo de execução será desviado caso o desvio seja tomado. Essa informação é usada durante a compressão do programa para determinar o endereço dos desvios (palavra de memória e deslocamento na palavra) no programa comprimido.

Por exemplo, a linha cujo valor é 33 para a coluna `N.` informa que na seqüência crescente de endereços do programa a 33ª árvore de expressão (não a 33ª instrução, uma árvore de expressão pode ser constituída por mais de uma instrução) é composta pela combinação do padrão de árvore (instruções) número 2 (o 3º que mais contribui para a cobertura do programa) e do padrão de operandos 662 (o 663º que mais contribui para a cobertura do programa) e essa árvore de expressão não termina com uma instrução de desvio, ou seja após a sua execução a árvore de expressão (posição N. 34) determinada pelos padrões 127 (árvore) e 589 (operandos) será executada. Após a execução dessa última árvore de expressão (posição 34) poderá ser executada a árvore de expressão determinada pelos padrões 4 (árvore) e 27 (operandos), posição 0, caso o desvio seja tomado, caso contrário será executada a árvore de expressão determinada pelos padrões 15 e 27 (posição

35).

```
0       0000000
         OPCODE          TYPE    RANDS    NAME
         100101           11      11       or

...

8       0001000
         OPCODE          TYPE    RANDS    NAME
         100011           01      10       lw
         101011           01      10       sw

9       0001001
         OPCODE          TYPE    RANDS    NAME
         001111           01      10       lui
         001001           01      11       addiu
         100001           11      11       addu
         101011           01      10       sw
...
```

Figura 7.2: Trecho do arquivo compress.dictionary

A figura 7.2 apresenta trechos do arquivo `compress.dictionary` com os padrões de árvores, ordenados em ordem decrescente de sua contribuição (em *bits*) na cobertura do programa. Para cada padrão está associado um valor inteiro, iniciando em 0 (zero), que determina a posição ocupada pelo padrão na ordenação. Na mesma linha é mostrado a codificação do padrão, neste caso a codificação usada é a binária. Na linha seguinte tem-se a descrição das próximas informações que estão divididas em 4 colunas e são: o *opcode* da instrução (OPCODE), o tipo da instrução (TYPE), um código (RANDS) que associado com o tipo da instrução informa o número de operandos da instrução e o mnemônico da instrução (NAME). Existe uma linha para cada instrução pertencente ao padrão de árvore.

```
SEQCODE    SEQUENCE

   0       0000000000      r0 r0 r0

   1       0000000001      16(r29) r28

   2       0000000010      r31 r25

   ...

   5       0000000101      0(r2) r2 0(r3) r3 r2 r3 r2

   6       0000000110      r0 511 r2 -29792(r28) r1 r2 r1 0(r1)
```

Figura 7.3: Trecho do arquivo compress.dictionary.sequence

A figura 7.3 apresenta trechos do arquivo com os padrões de operandos, ordenados em ordem decrescente de sua contribuição (em *bits*) na cobertura do programa. Para cada padrão de operandos está associado um inteiro (coluna SEQCODE) que representa a posição que o padrão ocupa na ordenação, uma codificação para o padrão (neste caso a codificação usada é a binária de tamanho fixo) e a lista dos operandos que formam o padrão.

Os programas usados para testar a compressão e descompressão de programas para o TMS320C25 foram fornecidos por Stan Y. Liao (Synopsys). Os programas fornecidos por Liao já estavam no formato `assembler` e foram compilados usando-se o compilador da Texas Instruments para o processador TMS320C25. Estes programas passaram por um tratamento semelhante aos programas do SPECInt95 (realizado por Pannain), e foram gerados arquivos com o mesmo formato que os apresentados nas figuras 7.1, 7.2 e 7.3. Estes arquivos foram utilizados neste trabalho para gerar a descrição VHDL do descompressor e o conteúdo da ROM (programa comprimido) por meio de programas.

7.2 Método TBC e os Programas de Teste

O método TBC difere do método apresentado em [60] em dois aspectos básicos. Primeiro, utiliza as árvores de expressão únicas como símbolo básico para a compressão no lugar do par de padrões de árvores e de operandos. O segundo aspecto refere–se à forma usada para ordenar os símbolos. No método TBC os símbolos são ordenados em ordem decrescente de sua freqüência no programa e não pela sua contribuição (em *bits*) na cobertura do programa (4.3). Desta forma, os programas provenientes do trabalho realizado por Pannain [60] não podem ser utilizados diretamente para gerar a descrição do descompressor. Assim, foi desenvolvido um conjunto de programas que, a partir das informações contidas no arquivo **.sequence*:

- reconstrói as árvores de expressão únicas (um par de padrões de árvore e de operandos determina uma árvore de expressão),
- determina a freqüência de cada árvore de expressão única no programa,
- atribui a cada árvore de expressão única uma codificação (dois tamanho de código para o R2000 e para o TMS320C25 e quatro tamanhos para o R4000) baseada na freqüência de ocorrência de cada uma delas no programa,
- cria tabelas que permitem que uma árvore de expressão possa ser mapeada nos padrões de árvore e de operandos correspondentes e vice e versa, e
- cria arquivos com dados estatísticos (usados para plotar as novas curvas de cobertura dos programas pelas árvores de expressão).

O programa comprimido, seqüência de códigos de árvores de expressão, foi gerado a partir destes novos dados e do arquivo `*.sequence` original onde cada par padrão de

árvore, padrão de operandos foi substituído pelo código da árvore de expressão correspondente. Foi gerada uma tabela que mostra a localização de cada *codeword* no programa comprimido (palavra onde ocorre e o deslocamento dentro da palavra onde ela inicia). Essa informação, em conjunto com o *Target* do arquivo `*.sequence` (seção 7.1), permitem a geração correta dos endereços de desvio, no programa comprimido, pela máquina de descompressão.

Para gerar o código VHDL do TRIGen para o R2000 (figura 4.7), do IGen para o TMS320C25 (figura 5.8) e do TGen para o 4000 (figura 6.6) foi montada uma estrutura de dados em memória, usando–se as informações que mapeiam uma ocorrência de um padrão de árvore de expressão em um par de padrões de árvore e de operandos (e vice e versa) e os arquivos `*.dictionary.tree` e `*.dictionary.sequence`. A estrutura mantém para cada árvore de expressão as informações necessárias à construção do código VHDL, que são:

- endereço inicial da árvore no dicionário TPD (R2000) ou no dicionário TD (R4000),
- número de instruções na árvore,
- número de operandos,
- tipo de cada instrução e
- os operandos associados a cada instrução.

No caso do R2000 e do TMS320C25, para os quais a máquina de descompressão é uma máquina de estados finitos (seções 4.4 e 5.4), a estrutura é percorrida uma vez para cada instrução. Para o R2000, na primeira instrução é montado o código VHDL que gera o endereço inicial da árvore de expressão no dicionário TPD, os operandos da primeira instrução e o valor do próximo estado. Para as demais instruções é montado o código que

incrementa de um o endereço do dicionário TPD e gera os operandos da instrução corrente. Para o TMS320C25, em todos os estados é gerado o código e os operandos necessários a montagem da instrução corrente. Em todos os estados e para ambos os processadores é gerado o novo estado (estado zero para a última instrução e estado corrente incrementado de um para os demais estados).

No caso do R4000, é gerado somente o endereço inicial do dicionário TD, uma vez que o dicionário contém toda a instrução e os endereços seguintes do dicionário TD de uma mesma árvore são gerados por um contador (seção 6.5).

Em todos os três casos, as definições das bibliotecas utilizadas, da entidade e da arquitetura da descrição VHDL são produzidas a partir de um arquivo gabarito (`trigen.vhd`). Este arquivo é mostrado no apêndice A (R2000 A.1, TMS320C25 A.2 e R4000 A.3). Os valores que dependem do programa que está sendo comprimido são definidos em um *package* personalizado para cada programa (o que é realizado com a edição de umas poucas linhas do *package*). No caso do R2000 e do TMS320C25, o tipo e os valores possíveis para a variável que define o estado da máquina de estados finitos são gerados pelo programa. Os arquivos *package* também são mostrados no apêndice A.

Os códigos VHDL foram escritos e compilados usando-se o padrão ANSI/IEEE Std 1076-1993 [20, 33] e seguindo as recomendações das ferramentas de síntese utilizadas quanto ao estilo da descrição com objetivo de se obter um código sintetizável e cuja síntese fosse realizada obtendo-se um melhor resultado para a área ocupada pelo circuito final [54, 23, 32, 24, 59]. Procurou-se também, nas descrições do código VHDL, obedecer a uma metodologia que permita que o módulo descrito possa ser integrado junto com outros módulos em um `SoC` (*System-on-a-Chip*) [38]. O apêndice B mostra trechos do código VHDL que gera o TRIGen para o programa `compress` para o processador R2000.

A figura 7.4 mostra, em forma de diagrama, o fluxo das atividades executadas para cada programa, desde a geração dos dados por Pannain [60], passando pela simulação,

síntese e geração dos *layouts*.

7.3 Fluxo de Projeto

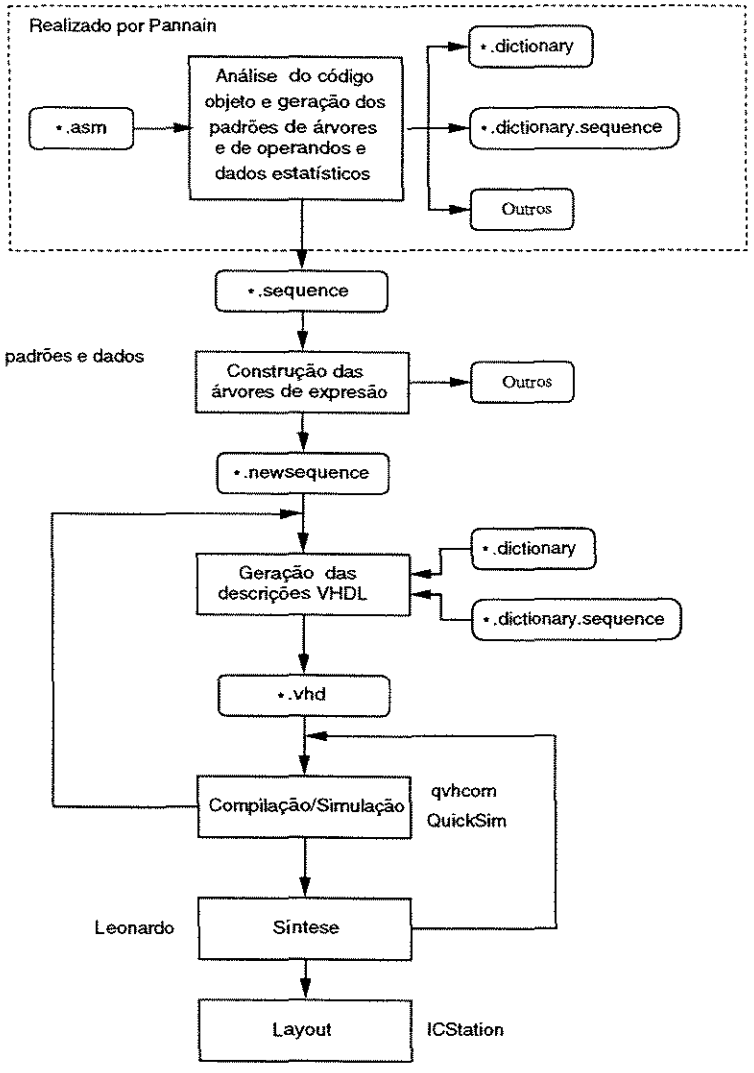

Figura 7.4: Fluxo de desenvolvimento da máquina de descompressão

As descrições VHDL das máquinas de descompressão foram compiladas com o compilador `qvhcom` VHDL da Mentor Graphics e simulado o seu comportamento com a ferramenta `qhsim` [57, 52], também da Mentor Graphics.

Com a simulação comportamental da máquina de descompressão resultando o esperado, o código VHDL foi novamente compilado com o compilador `qvhcom`, com opções de compilação para síntese e o resultado dessa compilação foi utilizado como entrada da ferramenta de síntese `autologicII` [53, 51]. A síntese, usando o `autologic` é realizada em dois passos. No primeiro passo, é realizada uma síntese usando–se uma biblioteca genérica. No segundo passo a síntese é realizada usando–se a biblioteca da tecnologia alvo. A síntese usando–se o `autologic` é realizada em dois passos e é muito lenta quando comparada com a realizada pela ferramenta *LeonardoSpectrum* [25] e os resultados obtidos, com respeito à área ocupada, foram equivalentes com as duas ferramentas. A seção 7.4.1 apresenta os tempos de CPU e quantidade de memória gastos na síntese da máquina de descompressão (TRIGen) para o processador R2000 usando o `autologicII` e o `leonardo`. Com base nestes resultados resolveu–se utilizar, para a síntese de todas as máquinas de descompressão a ferramenta `leonardo`.

Esta ferramenta não é totalmente integrada no sistema de projeto de circuitos integrados da Mentor Graphics. A saída desta ferramenta não pode ser gerada no formato aceito pela ferramenta de roteamento `ICStation`, desta forma o formato escolhido para a saída foi o formato EDIF (*Electronic Design Interchange Format*). A descrição da síntese, no formato EDIF, foi convertida para o formato aceito pela ferramenta de roteamento usando–se a ferramenta `enread` [56], o editor de `ViewPoint` [55] e o editor de esquemático `sg`, todos da Mentor Graphics. Uma vez convertida a saída do `leonardo`, foi utilizada a ferramenta `ICStation` [50, 58] para criar uma célula, realizar o auto *floorplan*, realizar o posicionamento das células da biblioteca na célula criada e fazer o roteamento das interligações entre as células.

7.3.1 Biblioteca Usada para a Síntese

Para a síntese das máquinas de descompressão foi utilizado a biblioteca *standard cell cub* da Austria Mikro Systeme (AMS). A biblioteca *cub* é uma biblioteca para a tecnologia CMOS de $0,6\mu m$ e foi usada a versão para 5 *volts*. A AMS fornece um kit de projeto (*AMS Hit-Kit*) [3, 4, 5] para ser usado com as ferramentas Mentor Graphics, que é constituído de um conjunto de `scripts` utilizados para invocar as ferramentas com os parâmetros ajustados para o uso das bibliotecas *standard cell* da AMS, que estão descritas no formato utilizado pelas diversas ferramentas Mentor Graphics.

7.4 Síntese das Máquinas de Descompressão

As máquina de descompressão para os processadores R2000 (capítulo 4), TMS320C25 (capítulo 5) e R4000 (capítulo 6) possuem o módulo `extrator` comum a todas elas. Este módulo consiste em um *buffer* de entrada (conjunto de registradores, figura 4.6(a)) e uma rede de alinhamento das *codewords* (*barrel shifter*, figura 4.6(b)). As possíveis diferenças entre esses módulos são devidas às diferenças entre os processadores (tamanho da palavra) e ao tamanho do programa que será descomprimido. Estas diferenças consistem na quantidade de registradores a serem usados no *buffer* de entrada e na quantidade de *bits* de entrada do *barrel shifter*. A síntese desses blocos não apresenta maiores dificuldades e eles representam um pequena fração da área total do descompressor. Podem, inclusive, serem projetados em *full custom* (diversos tamanhos), formando uma biblioteca de células a serem instanciadas no projeto final do SoC no lugar de serem implementadas usando–se *standard cell*, o que reduziria a área de silício ocupada por eles.

7.4.1 Síntese do Descompressor para o R2000

O descompressor para o R2000 além dos módulos citados acima possuem o dicionário TPD, o montador de instruções IAB e a máquina de estados finitos TRIGen.

O dicionário TPD é uma memória ROM e pode ser gerada automaticamente pelo gerador de memórias ROM fornecido pela AMS. O módulo IAB é um conjunto de multiplexadores (figura 4.8) que recebe valores do dicionário TPD e do módulo TRIGen, montando os campos da instrução corrente. Este módulo depende somente do processador (independente dos programas), podendo ser projetado em *full custom* para reduzir a área de silício por ele ocupada.

O módulo TRIGen, responsável por gerar os endereços do dicionário TPD (onde encontram-se os códigos das instruções da árvore de expressão correntemente sendo executada) e os operandos das instruções é uma máquina de estados finitos, na qual o número de estados é determinado pelo número de instruções da maior árvore de expressão do programa a ser descomprimido e tem como variável de entrada as *codewords*. A tabela 7.1 mostra, para cada programa usado para testar o sistema de compressão de código de programas para o processador R2000, o número de instruções do programa executável original, o número de linhas de código da descrição VHDL da máquina TRIGen, o número de estados de TRIGen, o tamanho (em *bits*) da variável de entrada de TRIGen e o número de *codewords* distintas. A máquina de estados TRIGen foi descrita como uma máquina de estados finitos de Mealy, para a qual a saída é função do estado atual e da variável de entrada.

Inicialmente, a máquina de estados TRIGen foi sintetizada com a ferramenta `autologicII` executando em uma estação `sparc Ultra-Enterprise` E3000[3] com dois processadores de 280 MHz e 512 MBytes de RAM executando Solaris 5. A tabela 7.2 mostra os tempos e memória gastos em cada uma das duas fases da síntese do módulo TRIGen relativo aos

[3]Esta era a máquina com a maior memória RAM disponível no IC, na época em que se realizou a síntese para o R2000.

Programa	Número de Instruções	Número de Linhas no VHDL	Número de Estados	Tamanho das *Codewords* (*bits*)	*Codewords* distintas
compress	1909	4622	36	11	744
gcc	366789	478317	29	17	44226
go	75277	133488	15	15	13025
ijpeg	49882	109982	58	15	10169
li	18474	32183	11	13	3135
perl	72655	115772	12	15	11769
vortex	148936	168846	34	15	16733

Tabela 7.1: R2000 – Tamanho dos programas de teste (em nº de instruções) e da descrição da máquina de estados (em nº de linhas) e o nº de estados.

programas `compress` e `li`. Para os demais programas de teste não foi possível realizar a síntese do módulo TRIGen, uma vez que o programa `autologicII` acusava erro por falta de memória RAM e abortava a execução. Para os programas sintetizados não foi possível gerar o *layout* final. Durante o uso do `ic-station` com o TRIGen do programa `compress` só foi possível realizar o *placement*, durante o roteamento o programa `ic-station` acusava erro por falta de memória e abortava. Nesta fase de síntese e geração (tentativa) do *layout* foi utilizado a versão B.4 do *software* da Mentor Graphics.

Programa	Primeira Fase		Segunda Fase	
	Tempo CPU	Memória	Tempo CPU	Memória
compress	4 h	150 MB	6,2 h	70 MB
li	57 h	490 MB	71 h	220 MB

Tabela 7.2: R2000/Autologic – Tempo de CPU e memória utilizada na Síntese.

Após as tentativas de sintetizar a máquina de descompressão para os programas R2000 com o *software* `autologicII` foi instalado no IC o *software* `LeonardoSpectrum`, o novo *software* de síntese da Mentor Graphics. Este *software* é mais rápido que o `autologicII` porém com o mesmo problema, ou seja acusa erro por falta de memória e aborta para os

programas de teste maiores. A tabela 7.3 apresenta os dados de tempo e memória gastos na síntese para os programas `compress` e `li`.

Programa	Tempo CPU	Memória
compress	1,1 h	171 MB
li [a]	14,7 h	649 MB

Tabela 7.3: R2000/Leonardo – Tempo de CPU e memória utilizada na Síntese.
[a] O programa `li` foi sintetizado usando-se uma E450 com 4 GBytes de RAM.

7.4.2 Síntese do Descompressor para o TMS320C25

Para o processador TMS320C25, o descompressor não utiliza um dicionário para armazenar o código das instruções, todos os campos das instruções são gerados pelo módulo IGen. Desta forma, nenhuma memória ROM é sintetizada. Como todos os campos necessários à montagem das instruções são gerados no módulo IGen, o módulo IAB também não existe (fisicamente) e sua função é sintetizada no módulo IGen.

O módulo IGen, responsável por gerar os códigos e os operandos das instruções da árvore de expressão correntemente sendo executada e os operandos das instruções, é uma máquina de estados finitos, na qual o número de estados é determinado pelo número de instruções da maior árvore de expressão do programa a ser descomprimido e tem como variável de entrada as *codewords*. A tabela 7.4 mostra, para cada programa usado para testar o sistema de compressão de código de programas para o processador TMS320C25, o número de instruções do programa executável original, o número de linhas de código da descrição VHDL da máquina IGen, o número de estados de IGen, o tamanho (em *bits*) da variável de entrada de IGen e o número de *codewords* distintas. A máquina de estados IGen foi descrita como uma máquina de estados finitos de Mealy, para a qual a saída é função do estado atual e da variável de entrada.

Programa	Número de Instruções	Número de Linhas de VHDL	Número de Estados	Tamanho das *Codewords* (*bits*)	*Codewords* distintas
aipint2	1403	2472	5	9	295
bench	9483	17827	8	13	2089
gnucrypt	3683	7424	8	11	750
gzip	10835	18733	7	13	2146
hill	920	2240	6	9	292
jpeg	2305	5664	7	10	572
rx	563	1042	5	8	121
set	4565	8931	7	11	969

Tabela 7.4: TMS320C25 – Tamanho dos programas de teste (em nº de instruções) e da descrição da máquina de estados (em nº de linhas) e o nº de estados.

A máquina de estados IGen foi sintetizada com a ferramenta `leonardo` executando em uma estação `sparc Ultra` E450 com quatro processadores de 400 MHz e 4GBytes de RAM, rodando Solaris 7. A tabela 7.5 mostra os tempos e memória gastos na síntese do módulo IGen relativo aos programas do conjunto de testes do processador TMS320C25. A figura 7.5 mostra esses mesmos dados quando plotados em função do tamanho (número de linhas) do programa VHDL que descreve as máquinas de estados finitos para cada um dos programas usados nos testes.

Programa	Tempo CPU	Memória
aipint2	3,15 min	85 MB
bench	2,43 h	483 MB
gnucrypt	18 min	174 MB
gzipa	3,6 h	489 MB
hill	2,14 min	81 MB
jpeg	10 min	128 MB
rx	1 min	60 MB
set	29 min	199 MB

Tabela 7.5: TMS320C25/Leonardo –Tempo de CPU e memória utilizada na Síntese.

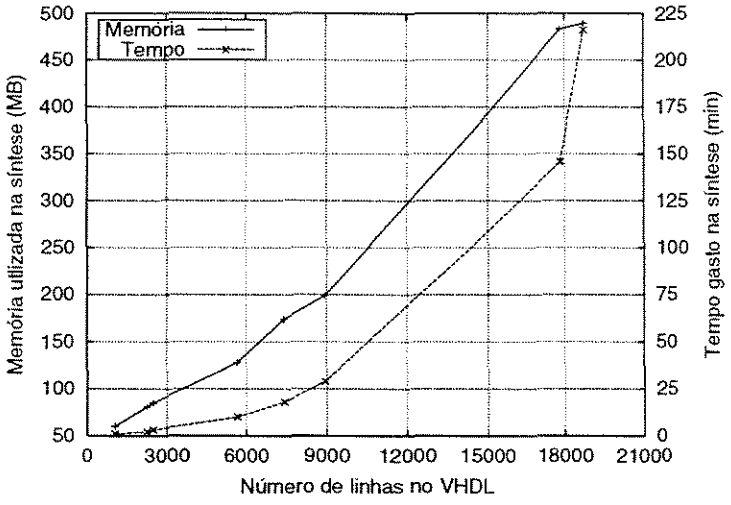

Figura 7.5: TMS320C25/Leonardo –Tempo de CPU e memória utilizada na Síntese.

7.4.3 Síntese do Descompressor para o R4000

O descompressor para o processador R4000 além do módulo `extrator` possui o dicionário TD e o circuito combinacional TGen. O módulo IAB inexiste para o processador R4000 uma vez que o dicionário TD contém todos os campos das instruções (as instruções já estão montadas).

O dicionário TD é uma memória ROM e pode ser gerada automaticamente pelo gerador de memórias ROM fornecido pela AMS e não apresentam nenhum problema para serem sintetizadas.

O módulo TGen, responsável por gerar os endereços do dicionário TD (onde encontram-se as instruções da árvore de expressão correntemente sendo executada) é um circuito combinacional que tem como entrada as *codewords* e como saída o endereço inicial da árvore de expressão no dicionário TD. A tabela 7.6 mostra, para cada programa usado para testar o sistema de compressão de código de programas para o processador R4000, o número de instruções do programa executável original, o número de linhas de código da descrição VHDL do circuito combinacional TGen, o tamanho (em *bits*) da variável de

entrada de TGen e o número de *codewords* distintas.

Programa	Número de Instruções	Número de Linhas de VHDL	Tamanho das *Codewords* (*bits*)	*Codewords* distintas
compress	2152	894	10	832
gcc	363560	51248	16	51186
go	72516	12522	14	12460
ijpeg	47688	11327	14	11624
li	18448	3134	12	3072
perl	69536	12857	14	12793
vortex	151348	17525	14	17493

Tabela 7.6: R4000 – Tamanho dos programas de teste (em nº de instruções) e da descrição da máquina de descompressão (em nº de linhas).

O circuito combinacional TGen foi sintetizado com a ferramenta `leonardo` executando em uma estação `sparc Ultra` E450 com quatro processadores de 400 MHz e 4GBytes de RAM, rodando Solaris 7. A tabela 7.5 mostra os tempos e memória gastos na síntese do módulo TGen relativo aos programas do conjunto de testes do processador R4000. A figura 7.5 mostra esses mesmos dados quando plotados em função do tamanho (número de linhas) do programa VHDL que descreve as máquinas de estados finitos para cada um dos programas usados nos testes.

Não foi possível a realização da síntese do TGen relativo ao programa `gcc`, por essa síntese usar mais do que 2 Gbytes de memória RAM e a ferramenta `leonardo` não ser capaz de tratar memórias maiores que 2 GBytes, uma vez que ele foi gerado para executar em `solaris 5` e esse sistema operacional manipula endereços de 32 *bits*, o que limita o tamanho da memória principal a 2Gbytes.

Programa	Tempo CPU	Memória
compress	7,4 min	90 MB
go	15,5 h	617 MB
ijpeg	16,6 h	684 MB
li	4,2 h	284 MB
perl	13,7 h	648 MB
vortex	45,5 h	1087 MB

Tabela 7.7: R4000/Leonardo –Tempo de CPU e memória utilizada na Síntese.

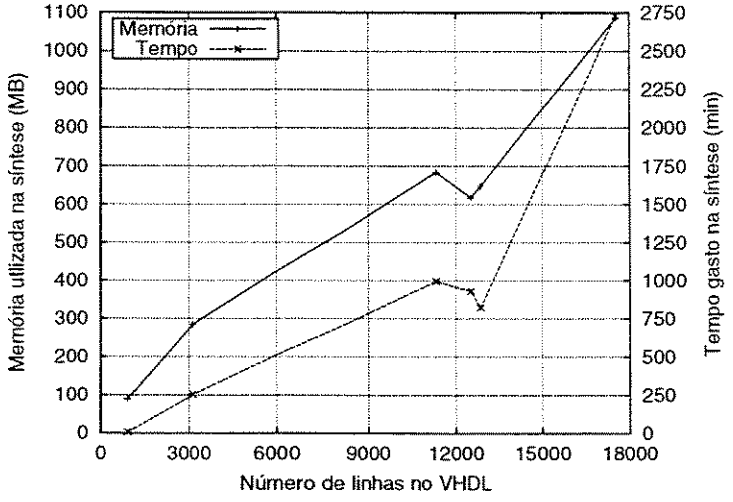

Figura 7.6: R4000/Leonardo –Tempo de CPU e memória utilizada na Síntese.

7.5 Conclusões

Neste capítulo foram apresentados os resultados da síntese das máquinas de descompressão para os processadores R2000, TMS320C25 e R4000.

A máquina de descompressão para o processador R2000 faz uso de uma máquina de estados finitos (TRIGen), que gera o endereço do dicionário TPD onde encontra-se o *opcode* da instrução corrente e os seus operandos.

A síntese da máquina de descompressão, usando essa abordagem, encontrou problemas devidos ao tamanho dos programas de testes (só foi possível sintetizar os dois menores),

que por serem grandes geram uma descrição em VHDL também grande. A síntese dessas descrições leva muito tempo de CPU e utilizam muita memória o que provocou o aborto da síntese. Estes problemas podem ser resolvidos utilizando–se máquinas adequadamente configuradas e com versões das ferramentas (ainda não estão disponíveis) para sistemas operacionais capazes de manipularem grandes quantidades de memórias (maiores que 2 *GBytes*.

A máquina de descompressão para o processador TMS320C25 também foi implementado baseado em uma máquina de estados finitos, similar à solução apresentada para o processador R2000, tendo como diferença principal a geração de todos os campos da instrução corrente (inclusive o *opcode*). Neste caso não houve problema com o uso das ferramentas de síntese, no que diz respeito ao uso de memória. Os programas usados para testes são bem menores que os usados para testar a solução para o processador R2000.

Para o processador R4000 foi usada uma abordagem alternativa para a implementação da máquina de descompressão que permitiu que as máquinas de descompressão para todos os programas de testes, exceto para o `gcc`, fossem sintetizadas. Essa nova abordagem tem como resultado uma razão de compressão pior àquela apresentada pela abordagem adotada para o processador R2000. Ambas as abordagens podem ser utilizadas para os processadores R2000 e R4000.

Essa limitação nas presentes versões das ferramentas de síntese não chega a ser um fator de preocupação por já existirem sistemas operacionais que manipulam memórias grandes (exemplo, solaris 7) e é só uma questão de tempo para aparecerem as versões das ferramentas para esses sistemas. Além disso, os programs usados para testar as soluções para os processadores R2000 e R4000 (programs do SPECInt95) são programs grandes e não representam aplicações típicas de sistemas dedicados embarcados. Eles foram utilizados neste trabalho para permitir uma melhor comparação dos resultados com os de métodos apresentados na literatura. O conjunto mais representativos de aplicações

para sistemas dedicados embarcados foi todo sintetizado em uma máquina E450 com $4 GBytes$ de memória e a memória máxima utilizada foi de $498 MBytes$ (`gzipa`).

Capítulo 8

Conclusões e Trabalhos Futuros

As principais contribuições desse trabalho foram a proposição de um novo método de compressão de programas para descompressão em tempo real (TBC) e da máquina de descompressão e a síntese da máquina de descompressão. O método de compressão TBC e a máquina de descompressão foram avaliados para três processadores distintos.

O método TBC quando aplicado a um conjunto de programas para o processador R2000 alcançou uma razão de compressão média dos programas de 28,1%. Quando o método é aplicado a conjuntos de programas para os processadores TMS320C25 e R4000 a razão de compressão média dos programas é de 28% e de 27,2% respectivamente.

As máquinas de descompressão foram sintetizadas usando–se bibliotecas *standard cell* da AMS, para a tecnologia CMOS de $0.6\mu m$ e 5 volts. Quando a área de silício ocupada pela máquina de descompressão é considerada na razão de compressão, obtém–se uma razão de compressão média dos programas de 75% e de 60,7% respectivamente para os processadores TMS320C25 e R4000. Para o processador R2000 só foi possível a síntese de dois programas do conjunto de programas utilizados nos testes. Isso se deu devido às restrições nas configurações da máquina na qual foi executada os programas de síntese e dos próprios programas de síntese utilizados (`autologicII` e `leonardo spectrum`).

Simulações das máquinas de descompressão mostraram uma freqüência mínima de ope-

ração de 90MHz para o processador R4000 e de 130MHz para o processador TMS320C25.

A tabela 8.1 resume o quadro atual, em termos da razão de compressão, dos trabalhos publicados até o presente momento. Como a maioria dos trabalhos (exceção ao trabalho de Liao), no cálculo da razão de compressão não incluem a área de silício necessária à implementação do *hardware* extra, tabelas e/ou dicionários, na tabela para o método TBC também não foi incluída a área necessária a implementação da máquina de descompressão.

Método	Processador							
	R2000	R4000	TMS320C25	PowerPC	ARM	i386	Pentium	Outros
TBC	28%	27%	28%	—	—	—	—	—
Fat. de Op.	35%	39%	35%	—	—	—	—	—
CCRP	73%	—	—	—	—	—	—	—
Kirovski	—	—	—	—	—	—	—	60%
Liao	—	—	82%	—	—	—	—	—
Lefurgy	—	—	—	63%	66%	74%	—	—
Lekatsas	57%	—	—	—	—	—	75%	—
CodePack	—	—	—	60%	—	—	—	—
Cooper	—	—	—	—	—	—	—	95%
Debray	—	—	—	—	—	—	—	78%

Tabela 8.1: Comparação de diversos métodos de compressão de programas.

Uma comparação semelhante a realizada para a razão de compressão não foi feita para a área ocupada e velocidade do *hardware* necessário à descompressão por estes dados não terem sido relatados na literatura.

8.1 Trabalhos Futuros

O trabalho apresentado aqui não esgota o tema por si só e pode ser complementado com a realização de diversas tarefas e investigações. Um primeiro trabalho a ser realizado é a fabricação de um circuito integrado, utilizando por exemplo o Projeto Multi–Usuário da Fapesp (PMU), que implemente a máquina de descompressão para um determinado

programa e a confecção de uma placa na qual estará integrado um pequeno sistema constituído da máquina de descompressão e de um processador, além de memória e outros dispositivos. Esse sistema seria usado para se fazer uma avaliação do desempenho real do sistema de descompressão.

Estudar modificações na forma como os compiladores geram códigos, de forma a gerarem código cuja compressão, usando–se o método TBC, apresentem uma razão de compressão melhor. Exemplo de modificações na geração de código seria fixar um conjunto de registradores para serem usados como registradores temporários nas árvores de expressão. Essa abordagem reduziria o número de árvores de expressão distintas, uma vez que várias delas diferenciam entre si nos registradores temporários que utilizam. A redução do número de árvores de expressão únicas reduz o número de *codewords*, possivelmente o tamanho da maior *codeword*, reduzindo o tamanho dos circuitos descompressores TRIGen, TGen e IGen e o tamanho dos dicionários TPD e TP. Cabe aqui uma observação: as modificações realizadas nos compiladores não devem aumentar, artificialmente, o tamanho do código gerado, pois assim facilmente se consegue uma razão de compressão menor (melhor), no entanto a área ocupada em silício pode ser maior.

Um outro trabalho a ser feito é realizar um estudo de um conjunto de programas que seja representativo da área de aplicação de interesse, determinando as árvores de expressões únicas para o conjunto de programas (e não mais para cada programa isoladamente) e incorporar ao conjunto de instruções do processador novas instruções que executem toda uma árvore de expressão (as mais freqüentes). Ao gerar o código executável, o compilador, no lugar de gerar o código que implementa uma dessas árvores, gera a nova instrução.

Referências Bibliográficas

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison Wesley, Boston, 1988.

[2] D. Alpert and D. Avnon. Architecture of the Pentium Microprocessor. *IEEE Micro*, pages 11–21, June 1993.

[3] AMS – Austria Mikro Systeme International AG. *AMS HIT-Kit. Mentor User's Guide*, 1996. Version 2.40.

[4] AMS – Austria Mikro Systeme International AG. *IC Station – User's Guide*, 1996. Version 2.40.

[5] AMS – Austria Mikro Systeme International AG. *AMS Continuum QHDL Tutorial*, 1997. Rev E.

[6] G. Araújo and Malik S. Optimal Code Generation for Embedded Memory Non Homogeneous Register Architecture. In *8th International Symposium on Systems Synthesis*. IEEE, September 1995.

[7] Guido Araújo. *Code Generation Algorithms for Digital Signal Processors.* PhD thesis, Princeton University – USA, 1997.

[8] Guido Araújo, Paulo Centoducatte, Rodolfo Azevedo, and Ricardo Pannain. Expression Tree Based Algorithms for Code Compression on Embedded RISC Architectures. *Submetido à IEEE Transactions on VLSI Systems*, 1999.

[9] Guido Araújo, Paulo Centoducatte, Mario Côrtes, and Ricardo Pannain. Code Compression Based on Operand Factorization. In *Proceedings of MICRO-31: The 31th Annual International Symposium on Microarchitecture*, pages 194–201, December 1998.

[10] M. C. Becker and et al. The PowerPC 601 Microprocessor. *IEEE Micro*, 13(5):54–68, Oct 1993.

[11] Timothy C. Bell, Jhon G. Cleary, and Ian H. Witten. *Text Compression.* Advanced Reference Series. Prentice Hall, New Jersey, 1990.

[12] Martin Beneš, Steven M. Nowick, and Andrew Wolfe. A Fast Asynchronous Huffman Decoder for Decompressed–Code Embedded Processors. In *Async98.* ACM, 1998.

[13] Martin Beneš, Andrew Wolfe, and Steven M. Nowick. A High–Speed Asynchronous Decompression Circuit for Embedded Processors. In *Proceedings of 17th Conference on Advanced Research in VLSI*, Los Alamitos, CA, 1997. IEEE Society Press.

[14] R. Camposano and J. Wilberg. *Embedded System Design*, chapter 1, pages 5–50. Kluwer Academic Publishers, Boston, 1996. Design Automation for Embedded Systems.

[15] Keith D. Cooper and Nathaniel McIntosh. Enhanced Code Compression for Embedded RISC Processors. In *Proc. Conf. on Programming Languages Design and Implementation*, May 1999.

[16] G. Cormak and R. Horspool. Data Compression Using Dynamic Markov Modelling. *Computer Journal*, 30(6), 1987.

[17] G. De Micheli. Computer–Aided Hardware–Software Codesign. *IEEE Micro*, 14(4):10–16, August 1994.

[18] Fabíola Gonçalves Pereira de Souza. Métodos Universais de Compressão de Dados. Master's thesis, IMECC–Unicamp, dezembro 1991.

[19] S. Debray, W. Evans, and Muth R. Compiler Techniques for Code Compression. In *In Workshop on Compiler Support for System Software*, May 1999.

[20] Design Automation Standards Committee of the IEEE Computer Society and IEEE Standards Coordinating Committee. *IEEE Standard VHDL Language Reference Manual*, 1994. ANSI/IEEE Std 1076–1993.

[21] Editorial. The Future of Computing. *The Economist*, pages 79–81, Set. 1998.

[22] Jean Ernst, William Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code Compression. In *SIGPLAN Programming Languages Design and Implementation*, 1997.

[23] Exemplar Logic. *Leonardo Spectrum Synthesis and Technology*, 1998. V1998.2.

[24] Exemplar Logic. *LeonardoSpectrum HDL Synthesis*, 1998. 1998.2.

[25] Exemplar Logic. *LeonardoSpectrum User's Guide*, 1998. 1998.2.

[26] M. J. Flynn and L. W. Hoevel. Execution Architecture: The DELtran Experiment. *IEEE Transactions on Computers*, C-32(2), February 1983.

[27] C. Fraser, E. Myers, and A. Wendt. Analyzing and Compressing Assembly Code. In *Proc. of the ACM SIGPLAN Symposium on Compiler Construction*, volume 19, pages 117–121, June 1984.

[28] C. W. Fraser and T. A. Proebsting. Custom Instructions Sets for Code Compression. *Not published*, October 1995. http://www.cs.arizona.edu/people/todd/papers/pldi2.ps.

[29] Barry G. Haskell, Atul Puri, and Arun N. Netravali. *Digital Video: an Introduction to MPEG–2.* Chapman & Hall, 1991.

[30] J. P. Hayes. *Computer Architecture and Organization.* McGraw–Hill, 1988. Second Edition.

[31] J.L. Hennessy and D.A. Patterson. *Computer Architectures: A Quantitative Approach.* Morgan Kaufmann, 1996.

[32] Tom Hill. *Leonardo Spectrum ASIC Design Methodology.* Exemplar Logic, 1998. Applications Note, Version 1.0.

[33] Yu-Chin Hsu, Kevin F. Tsai, Jessie T. Liu, and Eric s. Lin. *VHDL Modeling for Digital Design Synthesis.* Kluwer Academic Publishers, 1995.

[34] D. A. Huffman. A Method for the Construction of Minimum–redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952.

[35] IBM Corporarion. *CodePack PowerPC Code Compression Utility User's Manual*, 1998. Version 3.0.

[36] M. Jonshon. *Superscalar Microprocessor Design.* Printice–Hall, 1981.

[37] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Prentice Hall, New Jersey, 1992.

[38] Michael Keating and Pierre Bricaud. *Reuse Methodology Manual for System-on-a-Chip Designs.* Kluwer Academic Publishers, 1998.

[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. D. Kissell. MIPS16: High-density MIPS for the Embedded Market,. In *Real Time System'97*, 1997.

[41] P. M. Kogge. *The Architecture of Pipelined Computer.* McGraw-Hill, 1981.

[42] Michael Kozuch and Andrew Wolfe. Compression of Embedded System Programs. In *Proceedings of the IEEE International Conference on Computer Design*, pages 270–277, October 1994.

[43] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. Improving Code Density Using Compression Techniques. In *Proceedings of MICRO-30: The 30th Annual International Symposium on Microarchitecture*, pages 194–203, December 1997.

[44] H. Lekatsas and W Wolf. Code Compression for Embedded Systems. In *In Proc. ACM/IEEE Design Automation Conference*, pages 516–521, 1998.

[45] H. Lekatsas and W Wolf. Random Access Decompression Using Binary Arithmetic Coding. In *In Proc. Data Compression Conference*, pages 306–315, March 1999.

[46] A. Lempel and J. Ziv. On the Complexity of Finite Sequences. *IEEE Transaction on Information Theory*, IT–22(1):75–81, January 1976.

[47] S. Liao, S. Devadas, and K. Keutzer. Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques. In *Chapel Hill Conference on Advanced Research in VLSI*, 1995.

[48] S. Liao, S. Devadas, and K. Keutzer. A Text-Compression-Based Method for Code Size minimization in embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 4(1):12–38, 1999.

[49] Stan Y. Liao, Srinivas Devadas, and Kurt Keutzer. Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques. In *Proceedings of 16th Conference on Advanced Research in VLSI*, pages 272–285, Los Alamitos, CA, 1995. IEEE Society Press.

[50] Mentor Graphics. *IC Station Design Manual*, 1993. Software Version 8.2-5.

[51] Mentor Graphics. *AutoLogic II Reference Manual*, 1995. Software Version 8.4-3.

[52] Mentor Graphics. *QuickVHDL User's and Reference Manual*, 1995. Software Version 8.4-4.

[53] Mentor Graphics. *Synthesizing with AutoLogic II*, 1995. Software Version 8.5-1.

[54] Mentor Graphics. *VHDL Style Guide for Autologic II*, 1995. Software Version 8.4-3.

[55] Mentor Graphics. *Design ViewPpoint Editor User's and Reference Manual*, 1996. Software Version 8.5-2.

[56] Mentor Graphics. *EDIF Netlist Interface User's and Reference Manual*, 1996. Software Version 8.5-3.

[57] Mentor Graphics. *Getting Started with QuickHDL and VHDLwrite*, 1996. Software Version 8.5.

[58] Mentor Graphics. *IC Station User's Manual*, 1996. Software Version 8.6-2.

[59] Model Technology, a Mentor Graphics Company. *Large Device Design Methodology*, 1998. Applications Note, Revision 2.1.

[60] Ricardo Pannain. *Compressão de Código de Programa Usando Fatoração de Operandos.* PhD thesis, FEEC–Unicamp, Junho 1999.

[61] B. Ryan. Alpha Rides High. *BYTE*, 19(10):197–198, Oct. 1994.

[62] A. van Someren and A. Atack. *The ARM RISC Chip: A Programmer's Guide.* Addison-Wesley, 1994.

[63] Texas Instruments. *TMS320C2x User's Guide*, 1990.

[64] J.L. Turley. Thumb Squeezes ARM Code Size. *Microprocessor Report*, 9(4), March 1995.

[65] Welch. A Technique for High-performance Data Compression. *IEE Computer*, 17(6):8–19, July 1984.

[66] W. T. Wilner. Burroughs B1700 Memory Utilization. In *Fall Joint Computer Conference*, pages 579–586, 1972.

[67] I. Witten, R. Neal, and J. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(6):520–540, Jun 1987.

[68] W. Wolf. Hardware–Software Co–Design of Embedded Systems. *Procedings of IEEE*, 82(7):967–989, July 1994.

[69] Andrew Wolfe and Alex Channin. Executing Compressed Programs on an Embedded RISC Architecture. In *Proceedings of MICRO–25: The 25th Annual International Symposium on Microarchitecture*, pages 81–91, December 1992.

[70] J. Ziv and Lempel A. A Universal Algorithm for Sequential Data Compression. *IEEE Transaction on Information Theory*, 23(3):337–343, May 1977.

[71] J. Ziv and Lempel A. Compression of Individual Sequences via Variable–rate Coding. *IEEE Transaction on Information Theory*, 24(5):337–343, September 1978.

Apêndice A

Este apêndice apresenta os arquivos gabarito e os package utilizados para gerar e sintetizar os descompressores para os diversos programas, para os processadores R2000, TMS320C25 e R4000.

O arquivo `decengine-package.vhd` personaliza, para cada processador e programas, os dados que dependem do processador (tipo das instruções, tamanho da palavra, etc) e do programa (tamanho das *codewords*, do barraamento de endereços do dicionário, etc).

O arquivo `trigen.vhd` é usado, como gabarito, na geração das descrições VHDL do descompressor. Ele é copiado para o arquivo `*.trigen.vhd` e as linhas marcadas por um * seguido de um inteiro, indicam onde o programa, que gera a descrição, deve atuar. Por exemplo, para o R2000 e TMS320C25 na linha marcada com * 01 deve ser gerado o tipo da variável de estados (o apêndice B mostra exemplos de arquivos *.trigen.vhd).

A.1 R2000

```
-------------------------------------------------------------------------
-- Title        : Decompression  Engine Package
--
-- Project      : Code Compression    --    R2000
-------------------------------------------------------------------------
-- File         : decengine_package.vhd
-- Author       : Paulo C. Centoducatte  <ducatte@dcc.unicamp.br>
-- Company      : IC -- Unicamp
-- Last update : 1999/11/20
-- Platform     : Mentor B4
-------------------------------------------------------------------------
-- Description : Definicao de constantes, tipos e subtipos utilizados
--               pela maquina de descompressao
-------------------------------------------------------------------------
-- Modification history :
-- 1999/02/09 : created
-------------------------------------------------------------------------

LIBRARY IEEE;
USE ieee.std_logic_1164.ALL;
-- USE ieee.std_logic_arith.ALL;

PACKAGE DecEngine_package IS

-------------------------------------------------------------------------
--
--                      DECOMPRESSION ENGINE
--
-- Width
--
    CONSTANT word_width  : positive := 32;  -- Processor Dependent (PD)
    CONSTANT r_width     : positive := 5;   -- PD -- RS1, RS2 and RD width
    CONSTANT imb_width   : positive := 26;  -- PD
--    CONSTANT tpd_high_addr : positive :=  383;   --
--    CONSTANT tpaddr_width  : positive :=  9;     -- Compress
--    CONSTANT tpd_high_addr : positive := 7326;   --
--    CONSTANT tpaddr_width  : positive := 13;     -- Gcc
--    CONSTANT tpd_high_addr : positive := 2032;   --
--    CONSTANT tpaddr_width  : positive := 11;     -- Go
--    CONSTANT tpd_high_addr : positive := 3658;   --
--    CONSTANT tpaddr_width  : positive := 12;     -- Ijpeg
--    CONSTANT tpd_high_addr : positive :=  358;   --
--    CONSTANT tpaddr_width  : positive :=  9;     -- Li
--    CONSTANT tpd_high_addr : positive := 2042;   --
--    CONSTANT tpaddr_width  : positive := 11;     -- Perl
--    CONSTANT tpd_high_addr : positive := 1650;   --
--    CONSTANT tpaddr_width  : positive := 11;     -- Vortex
--
--  Subtypes
```

```
--
     SUBTYPE T_word   IS  STD_LOGIC_VECTOR(word_width - 1 DOWNTO 0);
     SUBTYPE T_R      IS  STD_LOGIC_VECTOR(r_width - 1 DOWNTO 0);
     SUBTYPE T_IMB    IS  STD_LOGIC_VECTOR(imb_width - 1 DOWNTO 0);
     SUBTYPE T_tpaddr IS  natural RANGE 2**tpaddr_width - 1 DOWNTO 0;

-----------------------------------------------------------------
--
--                              TPD
-- Width
--
    CONSTANT tpd_opcode_width : positive := 6;   -- PD
    CONSTANT tpd_type_width   : positive := 3;   -- PD
    CONSTANT tpd_D_width      : positive := 10;  -- PD
--
-- Subtypes
--
    SUBTYPE T_opcode    IS STD_LOGIC_VECTOR(tpd_opcode_width - 1 DOWNTO 0);
    SUBTYPE T_type      IS STD_LOGIC_VECTOR(tpd_type_width - 1 DOWNTO 0);
    SUBTYPE T_tpd_D_bus IS STD_LOGIC_vector(tpd_D_width - 1 DOWNTO 0);

-----------------------------------------------------------------
--
--      Codeword width
--
--    CONSTANT top_width : positive := 11;  Compress
--    CONSTANT top_width : positive := 17;  Gcc
--    CONSTANT top_width : positive := 15;  Go
--    CONSTANT top_width : positive := 15;  Ijpeg
--    CONSTANT top_width : positive := 13;  Li
--    CONSTANT top_width : positive := 15;  Perl
--    CONSTANT top_width : positive := 15;  Vortex

-----------------------------------------------------------------
--
--     Tree-Patterns and Operand_Patterns Subtypes
--
    SUBTYPE T_top IS STD_LOGIC_VECTOR(top_width - 1 DOWNTO 0);

-----------------------------------------------------------------
--
--   Registers Alias
--
    CONSTANT r0,  f0  : T_R := "00000";
    CONSTANT r1,  f1  : T_R := "00001";
    CONSTANT r2,  f2  : T_R := "00010";
    CONSTANT r3,  f3  : T_R := "00011";
    CONSTANT r4,  f4  : T_R := "00100";
    CONSTANT r5,  f5  : T_R := "00101";
    CONSTANT r6,  f6  : T_R := "00110";
    CONSTANT r7,  f7  : T_R := "00111";
```

```
    CONSTANT r8,  f8  : T_R := "01000";
    CONSTANT r9,  f9  : T_R := "01001";
    CONSTANT r10, f10 : T_R := "01010";
    CONSTANT r11, f11 : T_R := "01011";
    CONSTANT r12, f12 : T_R := "01100";
    CONSTANT r13, f13 : T_R := "01101";
    CONSTANT r14, f14 : T_R := "01110";
    CONSTANT r15, f15 : T_R := "01111";
    CONSTANT r16, f16 : T_R := "10000";
    CONSTANT r17, f17 : T_R := "10001";
    CONSTANT r18, f18 : T_R := "10010";
    CONSTANT r19, f19 : T_R := "10011";
    CONSTANT r20, f20 : T_R := "10100";
    CONSTANT r21, f21 : T_R := "10101";
    CONSTANT r22, f22 : T_R := "10110";
    CONSTANT r23, f23 : T_R := "10111";
    CONSTANT r24, f24 : T_R := "11000";
    CONSTANT r25, f25 : T_R := "11001";
    CONSTANT r26, f26 : T_R := "11010";
    CONSTANT r27, f27 : T_R := "11011";
    CONSTANT r28, f28 : T_R := "11100";
    CONSTANT r29, f29 : T_R := "11101";
    CONSTANT r30, f30 : T_R := "11110";
    CONSTANT r31, f31 : T_R := "11111";
    CONSTANT rX,  fx  : T_R := "-----";
END DecEngine_package;

PACKAGE BODY DecEngine_package IS

END DecEngine_package;
```

```
-------------------------------------------------------------------------
-- Title        : Tpd address, registers and operands generator
--
-- Project      : Code Compression   --   DecEngine   --  R2000
-------------------------------------------------------------------------
-- File         : trigen.vhd
-- Author       : Paulo C. Centoducatte  <ducatte@dcc.unicamp.br>
-- Company      : IC -- Unicamp
-- Last update  : 1999/11/20
-- Platform     : Mentor B4 -- Solaris
-------------------------------------------------------------------------
-- Description :
--
-------------------------------------------------------------------------
-- Modification history :
-- 1999/02/09 : created
-------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY work;
USE work.decengine_package.ALL;

ENTITY trigen IS

  PORT (
    top         : IN  T_top;
    tpd_end     : IN  std_logic;
    trigen_clk  : IN  std_logic;
    rst         : IN  std_logic;
    tpaddr      : OUT T_tpaddr;
    rs, rt, rd  : OUT T_R;
    immed       : OUT T_imb);
END trigen;

ARCHITECTURE rtl OF trigen IS

-- Number of states = Max. number of instructions in trees
* 01   TYPE T_state IS (S0
--
   SIGNAL state : T_state := S0;
   SIGNAL tpaddr_aux : T_tpaddr;
   ALIAS  re    : std_logic_vector(4 DOWNTO 0) IS immed(10 DOWNTO 6);
   ALIAS  func  : std_logic_vector(5 DOWNTO 0) IS immed_aux(5 DOWNTO 0);

BEGIN  -- rtl

trigen_proc : PROCESS (trigen_clk, rst, tpd_end)
```

```
  BEGIN  -- PROCESS trigen_proc
    IF rst = '0' THEN
       state <= S0;
       tpaddr_aux <= 0;
    ELSIF (trigen_clk'event AND trigen_clk = '1') THEN
        CASE state IS
* 02
* 03
          WHEN OTHERS => NULL;
        END CASE;
     END IF;
  END PROCESS trigen_proc;

  tpaddr <= tpaddr_aux;

END rtl;
```

A.2 Tms320C25

```
------------------------------------------------------------------
-- Title        : Decompression   Engine  Package
--
-- Project      : Code Compression  --  TMS320C25
------------------------------------------------------------------
-- File         : decengine_package.vhd
-- Author       : Paulo C. Centoducatte  <ducatte@dcc.unicamp.br>
-- Company      :
-- Last update  : 1999/11/20
-- Platform     :
------------------------------------------------------------------
-- Description : Definicao de constantes, tipos e subtipos utilizados
--               pela maquina de descompressao
------------------------------------------------------------------
-- Modification history :
-- 1999/04/21 : created
------------------------------------------------------------------

LIBRARY IEEE;
USE ieee.std_logic_1164.ALL;

PACKAGE DecEngine_package IS

------------------------------------------------------------------
--
--                    DECOMPRESSION ENGINE
--
-- Width
--
    CONSTANT word_width : positive := 16; -- Processor Dependent (PD)
--
--  Subtypes
--
     SUBTYPE T_word   IS  STD_LOGIC_VECTOR(word_width - 1 DOWNTO 0);

------------------------------------------------------------------
--
--      Codeword  width --  Program  Dependent
--
--    CONSTANT top_width : positive := 9;    -- Aipint2
--    CONSTANT top_width : positive := 12;   -- Bench
--    CONSTANT top_width : positive := 11;   -- Gnucrypt
--    CONSTANT top_width : positive := 12;   -- Gzipa
--    CONSTANT top_width : positive := 9;    -- Hill
--    CONSTANT top_width : positive := 10;   -- Jpeg
--    CONSTANT top_width : positive := 8 ;   -- Rx
--    CONSTANT top_width : positive := 11;   -- Set

------------------------------------------------------------------
```

```
--
--      Codeword Subtypes
--
    SUBTYPE T_top IS STD_LOGIC_VECTOR(top_width - 1 DOWNTO 0);

-----------------------------------------------------------------------
--
    SUBTYPE T_R      IS  STD_LOGIC_VECTOR(3 DOWNTO 0);

    CONSTANT \*\      : T_R := "0000";
    CONSTANT \*-\     : T_R := "0001";
    CONSTANT \*+\     : T_R := "0010";
    CONSTANT Notused  : T_R := "0011";
    CONSTANT \BR0-\   : T_R := "0100";
    CONSTANT \*0-\    : T_R := "0101";
    CONSTANT \*0+\    : T_R := "0110";
    CONSTANT \BR0+\   : T_R := "0111";

    CONSTANT \AR0\    : T_R := "0000";
    CONSTANT \AR1\    : T_R := "0001";
    CONSTANT \AR2\    : T_R := "0010";
    CONSTANT \AR3\    : T_R := "0011";
    CONSTANT \AR4\    : T_R := "0100";
    CONSTANT \AR5\    : T_R := "0101";
    CONSTANT \AR6\    : T_R := "0110";
    CONSTANT \AR7\    : T_R := "0111";

    CONSTANT \*AR0\,    \*+AR0\, \*-AR0\ : T_R := "0000";
    CONSTANT \*AR1\,    \*+AR1\, \*-AR1\ : T_R := "0001";
    CONSTANT \*AR2\,    \*+AR2\, \*-AR2\ : T_R := "0010";
    CONSTANT \*AR3\,    \*+AR3\, \*-AR3\ : T_R := "0011";
    CONSTANT \*AR4\,    \*+AR4\, \*-AR4\ : T_R := "0100";
    CONSTANT \*AR5\,    \*+AR5\, \*-AR5\ : T_R := "0101";
    CONSTANT \*AR6\,    \*+AR6\, \*-AR6\ : T_R := "0110";
    CONSTANT \*AR7\,    \*+AR7\, \*-AR7\ : T_R := "0111";
    CONSTANT \*0+AR0\, \*0-AR0\          : T_R := "1000";
    CONSTANT \*0+AR1\, \*0-AR1\          : T_R := "1001";
    CONSTANT \*0+AR2\, \*0-AR2\          : T_R := "1010";
    CONSTANT \*0+AR3\, \*0-AR3\          : T_R := "1011";
    CONSTANT \*0+AR4\, \*0-AR4\          : T_R := "1100";
    CONSTANT \*0+AR5\, \*0-AR5\          : T_R := "1101";
    CONSTANT \*0+AR6\, \*0-AR6\          : T_R := "1110";
    CONSTANT \*0+AR7\, \*0-AR7\          : T_R := "1111";

    CONSTANT \ARP0\   : T_R := "0000";
    CONSTANT \ARP1\   : T_R := "0001";
    CONSTANT \ARP2\   : T_R := "0010";
    CONSTANT \ARP3\   : T_R := "0011";
    CONSTANT \ARP4\   : T_R := "0100";
    CONSTANT \ARP5\   : T_R := "0101";
    CONSTANT \ARP6\   : T_R := "0110";
```

```
    CONSTANT \ARP7\   : T_R := "0111";
    CONSTANT \0\      : T_R := "0000";
    CONSTANT \1\      : T_R := "0001";
    CONSTANT \2\      : T_R := "0010";
    CONSTANT \3\      : T_R := "0011";
    CONSTANT \4\      : T_R := "0100";
    CONSTANT \5\      : T_R := "0101";
    CONSTANT \6\      : T_R := "0110";
    CONSTANT \7\      : T_R := "0111";
    CONSTANT \8\      : T_R := "1000";
    CONSTANT \9\      : T_R := "1001";
    CONSTANT \10\     : T_R := "1010";
    CONSTANT \11\     : T_R := "1011";
    CONSTANT \12\     : T_R := "1100";
    CONSTANT \13\     : T_R := "1011";
    CONSTANT \14\     : T_R := "1110";
    CONSTANT \15\     : T_R := "1111";

END DecEngine_package;

PACKAGE BODY DecEngine_package IS

END DecEngine_package;
```

```
------------------------------------------------------------------------
--
-- Title        : Instruction Generator (opcode, registers and
--                operands)  generator
--
-- Project      : Code Compression -- DecEngine  --  TMS320C25
------------------------------------------------------------------------
-- File         : trigen.vhd
-- Author       : Paulo C. Centoducatte  <ducatte@dcc.unicamp.br>
-- Company      : IC -- Unicamp
-- Last update  : 1999/11/20
-- Platform     : Mentor B4 -- Solaris
------------------------------------------------------------------------
-- Description :
--
------------------------------------------------------------------------
-- Modification history :
-- 1999/02/09 : created
------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY work;
USE work.decengine_package.ALL;

ENTITY trigen IS
    PORT (
      top         : IN  T_top;
      trigen_clk  : IN  STD_LOGIC;
      rst         : IN  STD_LOGIC;
      instr       : OUT T_word;
      br_k        : OUT T_word;
      two_word    : OUT STD_LOGIC;
      tpd_end     : OUT STD_LOGIC);
END trigen;

ARCHITECTURE rtl OF trigen IS

-- Number of states = Max. number of instructions in trees
* 01   TYPE T_state IS (S0
--
  SIGNAL state : T_state;

  ALIAS    AR  : std_logic_vector(3 DOWNTO 0) IS instr(7 DOWNTO 4);
  ALIAS     S  : std_logic_vector(3 DOWNTO 0) IS instr(10 DOWNTO 7);
  ALIAS    PA  : std_logic_vector(3 DOWNTO 0) IS instr(11 DOWNTO 8);
  ALIAS     D  : std_logic_vector(6 DOWNTO 0) IS instr(6 DOWNTO 0);
  ALIAS    K1  : std_logic                    IS instr(0) ;
  ALIAS    K2  : std_logic_vector(1 DOWNTO 0) IS instr(1 DOWNTO 0);
```

```
  ALIAS    K3  : std_logic_vector(2 DOWNTO 0) IS instr(2 DOWNTO 0);
  ALIAS    K8  : std_logic_vector(7 DOWNTO 0) IS instr(7 DOWNTO 0);
  ALIAS    K9  : std_logic_vector(8 DOWNTO 0) IS instr(8 DOWNTO 0);
  ALIAS   K12  : std_logic_vector(11 DOWNTO 0) IS instr(11 DOWNTO 0);
  ALIAS   NAR  : std_logic  IS instr(4);

BEGIN  -- rtl

     trigen_proc : PROCESS (trigen_clk, rst)

   BEGIN  -- PROCESS trigen_proc
     IF rst = '0' THEN
       state <= S0; tpd_end <= '0'; two_word <= '0';
     ELSIF (trigen_clk'event AND trigen_clk = '1') THEN
       CASE state IS
* 02
* 03
       WHEN OTHERS => NULL;
       END CASE;

     END IF;

   END PROCESS trigen_proc;

END rtl;
```

A.3 R4000

```
------------------------------------------------------------------------
--
-- Title        : Decompression  Engine  Package
--
-- Project      : Code Compression    --    R4000
------------------------------------------------------------------------
-- File         : decengine_package.vhd
-- Author       : Paulo C. Centoducatte  <ducatte@dcc.unicamp.br>
-- Company      :
-- Last update  : 1999/11/20
-- Platform     :
------------------------------------------------------------------------
-- Description : Definicao de constantes, tipos e subtipos utilizados
--               pela maquina de descompressao
------------------------------------------------------------------------
-- Modification history :
-- 1999/02/09 : created
------------------------------------------------------------------------

LIBRARY IEEE;
USE ieee.std_logic_1164.ALL;

PACKAGE DecEngine_package IS

------------------------------------------------------------------------
--
--                        DECOMPRESSION ENGINE
-- Width
--
    CONSTANT word_width : positive := 32;  -- Processor Dependent (PD)
    CONSTANT r_width    : positive := 5;   -- PD -- RS1, RS2 and RD width
    CONSTANT imb_width  : positive := 26;  -- PD
--    CONSTANT tpaddr_width  : positive :=  8;     -- Compress
--    CONSTANT tpaddr_width  : positive := 17;     -- Gcc
--    CONSTANT tpaddr_width  : positive := 15;     -- Go
--    CONSTANT tpaddr_width  : positive := 14;     -- Ijpeg
--    CONSTANT tpaddr_width  : positive := 13;     -- Li
--    CONSTANT tpaddr_width  : positive := 15;     -- Perl
--    CONSTANT tpaddr_width  : positive := 15;     -- Vortex
--
--  Subtypes
--
     SUBTYPE T_word   IS  STD_LOGIC_VECTOR(word_width - 1 DOWNTO 0);
     SUBTYPE T_R      IS  STD_LOGIC_VECTOR(r_width - 1 DOWNTO 0);
     SUBTYPE T_imb    IS  STD_LOGIC_VECTOR(imb_width - 1 DOWNTO 0);
     SUBTYPE T_tpaddr IS  natural RANGE 2**tpaddr_width - 1 DOWNTO 0;

------------------------------------------------------------------------
--
```

```
--                              TD
-- Width
--
    CONSTANT tpd_D_width      : positive := 33;  -- PD
--
-- Subtypes
--
    SUBTYPE T_tpd_D_bus IS STD_LOGIC_vector(tpd_D_width - 1 DOWNTO 0);

-------------------------------------------------------------------
--
--      Codeword width
--
--    CONSTANT top_width : positive := 10;  -- Compress
--    CONSTANT top_width : positive := 16;  -- Gcc
--    CONSTANT top_width : positive := 14;  -- Go
--    CONSTANT top_width : positive := 14;  -- Ijpeg
--    CONSTANT top_width : positive := 12;  -- Li
--    CONSTANT top_width : positive := 14;  -- Perl
--    CONSTANT top_width : positive := 14;  -- Vortex

-------------------------------------------------------------------
--
--     Codeword Subtype
--
--
    SUBTYPE T_top IS STD_LOGIC_VECTOR(top_width - 1 DOWNTO 0);

-------------------------------------------------------------------
--
--   Registers Alias
--
    CONSTANT r0,  f0  : T_R := "00000";
    CONSTANT r1,  f1  : T_R := "00001";
    CONSTANT r2,  f2  : T_R := "00010";
    CONSTANT r3,  f3  : T_R := "00011";
    CONSTANT r4,  f4  : T_R := "00100";
    CONSTANT r5,  f5  : T_R := "00101";
    CONSTANT r6,  f6  : T_R := "00110";
    CONSTANT r7,  f7  : T_R := "00111";
    CONSTANT r8,  f8  : T_R := "01000";
    CONSTANT r9,  f9  : T_R := "01001";
    CONSTANT r10, f10 : T_R := "01010";
    CONSTANT r11, f11 : T_R := "01011";
    CONSTANT r12, f12 : T_R := "01100";
    CONSTANT r13, f13 : T_R := "01101";
    CONSTANT r14, f14 : T_R := "01110";
    CONSTANT r15, f15 : T_R := "01111";
    CONSTANT r16, f16 : T_R := "10000";
    CONSTANT r17, f17 : T_R := "10001";
    CONSTANT r18, f18 : T_R := "10010";
```

```
    CONSTANT r19, f19 : T_R := "10011";
    CONSTANT r20, f20 : T_R := "10100";
    CONSTANT r21, f21 : T_R := "10101";
    CONSTANT r22, f22 : T_R := "10110";
    CONSTANT r23, f23 : T_R := "10111";
    CONSTANT r24, f24 : T_R := "11000";
    CONSTANT r25, f25 : T_R := "11001";
    CONSTANT r26, f26 : T_R := "11010";
    CONSTANT r27, f27 : T_R := "11011";
    CONSTANT r28, f28 : T_R := "11100";
    CONSTANT r29, f29 : T_R := "11101";
    CONSTANT r30, f30 : T_R := "11110";
    CONSTANT r31, f31 : T_R := "11111";
    CONSTANT rX,  fx  : T_R := "-----";

END DecEngine_package;

PACKAGE BODY DecEngine_package IS

END DecEngine_package;
```

```
------------------------------------------------------------------------
--
-- Title        : TD address generator
--
-- Project      : Code Compression    --    DecEngine    --   R4000
------------------------------------------------------------------------
-- File         : trigen.vhd
-- Author       : Paulo C. Centoducatte  <ducatte@dcc.unicamp.br>
-- Company      : IC -- Unicamp
-- Last update : 1999/11/20
-- Platform     : Mentor B4 -- Solaris
------------------------------------------------------------------------
-- Description :
--
------------------------------------------------------------------------
-- Modification history :
-- 1999/02/09 : created
------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
LIBRARY work;
USE work.decengine_package.ALL;

ENTITY trigen IS
    PORT (
      top        : IN  T_top;
      trigen_clk : IN  std_logic;
      rst        : IN  std_logic;
      tpaddr     : OUT T_tpaddr);
END trigen;

ARCHITECTURE rtl OF trigen IS
BEGIN  -- rtl

trigen_proc : PROCESS (trigen_clk, rst)

  BEGIN  -- PROCESS trigen_proc
    IF rst = '0' THEN
       tpaddr <= 0;
    ELSIF (trigen_clk'event AND trigen_clk = '1') THEN
      CASE top IS
* 02
        WHEN OTHERS => NULL;
      END CASE;
    END IF;
  END PROCESS trigen_proc;
END rtl;
```

Apêndice B

Este apêndice apresenta trechos dos arquivos `compress.trigen.vhd` para os processadores R2000 e R4000 e do arquivo `rx.trigen.vhd` para o processador TMS320C25. Estes arquivos foram gerados, por programa, usando-se os gabaritos apresentados no apêndice A.

B.1 compress.trigen.vhd para o R2000

```
-------------------------------------------------------------------------
-- Title       : Tpd address, registers and operands generator
-- Project     : Code Compression       --     DecEngine
-------------------------------------------------------------------------
-- File        : compress.trigen.vhd
-- Author      : Paulo C. Centoducatte  <ducatte@dcc.unicamp.br>
-- Company     : IC -- Unicamp
-- Last update : 1999/10/14
-- Platform    : Mentor B4 -- Solaris
-------------------------------------------------------------------------
-- Description :
--
-------------------------------------------------------------------------
-- Modification history :
-- 1999/02/09 : created
-------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY work;
USE work.decengine_package.ALL;

ENTITY trigen IS
```

```
    PORT (
    top         : IN  T_top;
    tpd_end     : IN  std_logic;
    trigen_clk  : IN  std_logic;
    rst         : IN  std_logic;
    tpaddr      : OUT T_tpaddr;
    rs, rt, rd  : OUT T_R;
    immed       : OUT T_imb);
END trigen;

ARCHITECTURE rtl OF trigen IS

-- Number of states = Max. number of instructions in trees
   TYPE T_state IS (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10,S11,S12,S13,
                    S14,S15,S16,S17,S18,S19,S20,S21,S22,S23,S24,S25,S26,S27,
                    S28,S29,S30,S31,S32,S33,S34,S35);
--
   SIGNAL state : T_state := S0;
   SIGNAL tpaddr_aux : T_tpaddr;
   ALIAS  re   : std_logic_vector(4 DOWNTO 0) IS immed_aux(10 DOWNTO 6);
   ALIAS  func : std_logic_vector(5 DOWNTO 0) IS immed_aux(5 DOWNTO 0);

BEGIN  -- rtl

trigen_proc : PROCESS (trigen_clk, rst, tpd_end)

  BEGIN  -- PROCESS trigen_proc
    IF rst = '0' THEN
       state <= S0;
       tpaddr_aux <= 0;
    ELSIF (trigen_clk'event AND trigen_clk = '1') THEN
        CASE state IS
          WHEN   S0  =>
            CASE top IS
              WHEN  "00000000000" =>
                tpaddr_aux <= 0; state <= S0;
                rs <= r0; rt <= r0; rd <= r0; re <= r0;
              WHEN  "00000001000" =>
                tpaddr_aux <= 3; state <= S0;
                rs <= r25; rt <= r0; rd <= r31; re <= r0;
              WHEN  "00000010000" =>
                tpaddr_aux <= 1; state <= S0;
                rs <= r29; rt <= r28;
                immed <= conv_std_logic_vector(16,26);
              WHEN  "00000011000" =>
                tpaddr_aux <= 4; state <= S0;
                rs <= r31; immed <= conv_std_logic_vector(0,26);

          .........
```

```
            WHEN  "11001010001" =>
              tpaddr_aux <= 77; state <= S1;
              rs <= r0; rt <= r2;
              immed <= conv_std_logic_vector(257,26);
            WHEN  OTHERS => state <= S0;
          END CASE;
        WHEN   S1  =>
          CASE top IS
            WHEN  "00001111000" =>
              tpaddr_aux <= tpaddr_aux + 1; state <= S2;
              rs <= r3; rt <= r3;
              immed <= conv_std_logic_vector(0,26);
            WHEN  "00011010000" =>
              tpaddr_aux <= tpaddr_aux + 1; state <= S2;
              rs <= r28; rt <= r1;
              immed <= conv_std_logic_vector(-29792,26);
            WHEN  "00100110000" =>
              tpaddr_aux <= tpaddr_aux + 1; state <= S0;
              rs <= r1; rt <= r1;
              immed <= conv_std_logic_vector(-1,26);

      .........

        WHEN   S33  =>
          CASE top IS
            WHEN  "10111010111" =>
              tpaddr_aux <= tpaddr_aux + 1; state <= S34;
              rs <= r2; rt <= r3; rd <= r2; re <= r0;
            WHEN  OTHERS => state <= S0;
          END CASE;
        WHEN   S34  =>
          CASE top IS
            WHEN  "10111010111" =>
              tpaddr_aux <= tpaddr_aux + 1; state <= S35;
              rs <= r2; rt <= r4; rd <= r2; re <= r0;
            WHEN  OTHERS => state <= S0;
          END CASE;
        WHEN   S35  =>
          CASE top IS
            WHEN  "10111010111" =>
              tpaddr_aux <= tpaddr_aux + 1; state <= S0;
              rs <= r5; rt <= r2; rd <= r5; re <= r0;
            WHEN  OTHERS => state <= S0;
          END CASE;
        WHEN OTHERS => NULL;
      END CASE;
    END IF;
  END PROCESS trigen_proc;

  tpaddr <= tpaddr_aux;
END rtl;
```

B.2 rx.trigen.vhd para o TMS320C25

```
-------------------------------------------------------------------------------
-- Title       : TMS
--               Instruction Generator (opcode, registers and
--               operands)  generator
--
-- Project     : Code Compression       --     DecEngine
-------------------------------------------------------------------------------
-- File        : rx.trigen.vhd
-- Author      : Paulo C. Centoducatte  <ducatte@dcc.unicamp.br>
-- Company     : IC -- Unicamp
-- Last update : 1999/04/22
-- Platform    : Mentor B4 -- Solaris
-------------------------------------------------------------------------------
-- Description :
--
-------------------------------------------------------------------------------
-- Modification history :
-- 1999/02/09 : created
-------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY work;
USE work.decengine_package.ALL;

ENTITY trigen IS
    PORT (
      top         : IN  T_top;
      trigen_clk  : IN  STD_LOGIC;
      rst         : IN  STD_LOGIC;
      instr       : OUT T_word;
      br_k        : OUT T_word;
      two_word    : OUT STD_LOGIC;
      tpd_end     : OUT STD_LOGIC);
END trigen;

ARCHITECTURE rtl OF trigen IS

-- Number of states = Max. number of instructions in trees
   TYPE T_state IS (S0,S1,S2,S3,S4);
--
  SIGNAL state : T_state;

  ALIAS    AR        : std_logic_vector(3 DOWNTO 0) IS instr(7 DOWNTO 4);
  ALIAS     S        : std_logic_vector(3 DOWNTO 0) IS instr(10 DOWNTO 7);
  ALIAS    PA        : std_logic_vector(3 DOWNTO 0) IS instr(11 DOWNTO 8);
  ALIAS     D        : std_logic_vector(6 DOWNTO 0) IS instr(6 DOWNTO 0);
```

```
  ALIAS    K1        : std_logic                      IS instr(0) ;
  ALIAS    K2        : std_logic_vector(1 DOWNTO 0) IS instr(1 DOWNTO 0);
  ALIAS    K3        : std_logic_vector(2 DOWNTO 0) IS instr(2 DOWNTO 0);
  ALIAS    K8        : std_logic_vector(7 DOWNTO 0) IS instr(7 DOWNTO 0);
  ALIAS    K9        : std_logic_vector(8 DOWNTO 0) IS instr(8 DOWNTO 0);
  ALIAS   K12        : std_logic_vector(11 DOWNTO 0) IS instr(11 DOWNTO 0);
  ALIAS   NAR        : std_logic  IS instr(4);

BEGIN  -- rtl

  trigen_proc : PROCESS (trigen_clk, rst)

   BEGIN  -- PROCESS trigen_proc
     IF rst = '0' THEN
       state <= S0; tpd_end <= '0'; two_word <= '0';
     ELSIF (trigen_clk'event AND trigen_clk = '1') THEN
       CASE state IS
          WHEN   S0  =>
            CASE top IS
              WHEN  "00000000" =>
                state <= S0;    tpd_end <= '1';
                instr <= "0110000010000000";
                AR  <= \*AR6\;
              WHEN  "00000001" =>
                state <= S0;    tpd_end <= '1';
                instr <= "0101010110000000";
                D <= conv_std_logic_vector(5571,7);
              WHEN  "00000010" =>
                state <= S0;    tpd_end <= '1';
                instr <= "0001100010000000";
                D  <= conv_std_logic_vector(-9999,7);
                S  <= \AR4\;
              WHEN  "00000011" =>
                state <= S0;    tpd_end <= '1';
                instr <= "1111111110000000";
                br_k  <= conv_std_logic_vector(552,16);

     .........

              WHEN  "01111100" =>
                state <= S1;  tpd_end <= '0';
                instr <= "0101010110000000";
                D <= conv_std_logic_vector(552,7);
              WHEN  OTHERS => state <=  S0;
            END CASE;
          WHEN   S1  =>
            CASE top IS
              WHEN  "00001010" =>
                state <= S2;  tpd_end <= '0';
                instr <= "0000000010000000";
                AR  <= \*AR0\;
```

```
              WHEN  "00001011" =>
                state <= S2;  tpd_end <= '0';
                instr <= "0011110010000000";
                D <= conv_std_logic_vector(552,7);

      .........

              WHEN  "01100001" =>
                state <= S0;    tpd_end <= '1';
                instr <= "0001100010000000";
                D  <= conv_std_logic_vector(-9999,7);
                S  <= \AR3\;
              WHEN  "01101001" =>
                state <= S4;  tpd_end <= '0';
                instr <= "0101010110000000";
                D <= conv_std_logic_vector(552,7);
              WHEN  OTHERS => state <=  S0;
            END CASE;
          WHEN   S4  =>
            CASE top IS
              WHEN  "01101001" =>
                state <= S0;    tpd_end <= '1';
                instr <= "0001100010000000";
                D  <= conv_std_logic_vector(-9999,7);
                S  <= \AR3\;
              WHEN  OTHERS => state <=  S0;
            END CASE;
        WHEN OTHERS => NULL;
        END CASE;

      END IF;

    END PROCESS trigen_proc;

END rtl;
```

B.3 compress.trigen.vhd para o R4000

```
-------------------------------------------------------------------------------
-- Title       : Tpd address, registers and operands generator
-- Project     : Code Compression       --     DecEngine
-------------------------------------------------------------------------------
-- File        : compress.trigen.vhd
-- Author      : Paulo C. Centoducatte  <ducatte@dcc.unicamp.br>
-- Company     : IC -- Unicamp
-- Last update : 1999/09/03
-- Platform    : Mentor B4 -- Solaris
-------------------------------------------------------------------------------
-- Description :
--
-------------------------------------------------------------------------------
-- Modification history :
-- 1999/02/09 : created
-------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
--USE ieee.std_logic_arith.ALL;

LIBRARY work;
USE work.decengine_package.ALL;

ENTITY trigen IS

  PORT (
    top        : IN  T_top;
    trigen_clk : IN  std_logic;
    rst        : IN  std_logic;
    tpaddr     : OUT T_tpaddr);
END trigen;

ARCHITECTURE rtl OF trigen IS

BEGIN  -- rtl

trigen_proc : PROCESS (trigen_clk, rst)

  BEGIN  -- PROCESS trigen_proc
    IF rst = '0' THEN
       tpaddr <= 0;
    ELSIF (trigen_clk'event AND trigen_clk = '1') THEN
      CASE top IS
          WHEN  "00000000000" =>     tpaddr <= 0;
          WHEN  "00000001000" =>     tpaddr <= 1;
          WHEN  "00000010000" =>     tpaddr <= 1;
          WHEN  "00000011000" =>     tpaddr <= 1;
          WHEN  "00000100000" =>     tpaddr <= 15;
```

```
            WHEN  "00000101000" =>    tpaddr <= 7;
            WHEN  "00000110000" =>    tpaddr <= 1;
            WHEN  "00000111000" =>    tpaddr <= 4;
            WHEN  "00001000000" =>    tpaddr <= 1;
            WHEN  "00001001000" =>    tpaddr <= 24;
            WHEN  "00001010000" =>    tpaddr <= 1;
            WHEN  "00001011000" =>    tpaddr <= 16;
            WHEN  "00001100000" =>    tpaddr <= 1;

            ..........

            WHEN  "10001010100" =>    tpaddr <= 147;
            WHEN  "10001010101" =>    tpaddr <= 1;
            WHEN  "10001010110" =>    tpaddr <= 26;
            WHEN  "10001010111" =>    tpaddr <= 25;
            WHEN  "10001011000" =>    tpaddr <= 83;
            WHEN  "10001011001" =>    tpaddr <= 83;
            WHEN  "10001011010" =>    tpaddr <= 108;
            WHEN  "10001011011" =>    tpaddr <= 30;
            WHEN  "10001011100" =>    tpaddr <= 41;
            WHEN  "10001011101" =>    tpaddr <= 41;
            WHEN  "10001011110" =>    tpaddr <= 1;
            WHEN  "10001011111" =>    tpaddr <= 1;
            WHEN  "10001100000" =>    tpaddr <= 20;
            WHEN  "10001100001" =>    tpaddr <= 35;
            WHEN  "10001100010" =>    tpaddr <= 35;
            WHEN  "10001100011" =>    tpaddr <= 146;
            WHEN  "10001100100" =>    tpaddr <= 1;

            ..........

            WHEN  "11010110101" =>    tpaddr <= 26;
            WHEN  "11010110110" =>    tpaddr <= 4;
            WHEN  "11010110111" =>    tpaddr <= 20;
            WHEN  "11010111000" =>    tpaddr <= 31;
            WHEN  "11010111001" =>    tpaddr <= 18;
            WHEN  "11010111010" =>    tpaddr <= 148;
            WHEN  "11010111011" =>    tpaddr <= 31;
            WHEN  "11010111100" =>    tpaddr <= 149;
            WHEN  "11010111101" =>    tpaddr <= 26;

          WHEN OTHERS => NULL;
        END CASE;
      END IF;
    END PROCESS trigen_proc;
END rtl;
```