SPARC16: Uma nova visão de compressão para processadores SPARC

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Leonardo Luiz Ecco e aprovada pela Banca Examinadora.

Campinas, 25 de Novembro de 2010.

Prof. Dr. Rodolfo Jardim de Azevedo
(Orientador)

Prof. Dr. Paulo Cesar Centoducatte
(Coorientador)

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

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

Ecco, Leonardo Luiz
Ec29s SPARC16: uma nova visão de compressão para processadores SPARC/Leonardo Luiz Ecco-- Campinas, [S.P. : s.n.], 2010.

Orientadores : Rodolfo Jardim de Azevedo ; Paulo Cesar Centoducatte.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1.Arquitetura de computador. 2.Sistemas embutidos de computador. 3.Compresssão de dados (Computação). I. Azevedo, Rodolfo Jardim de. II.Centoducatte, Paulo Cesar. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Título em inglês: SPARC16: a new compression approach for SPARC processors

Palavras-chave em inglês (Keywords): 1.Computer architecture. 2.Embedded computer systems. 3.Data compression (Computer science).

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Paulo Cesar Centoducatte (IC – UNICAMP)
Prof. Dr. Antônio Augusto Medeiros Fröhlich (CT – UFSC)
Prof. Dr. Mario Lúcio Côrtes (IC - UNICAMP)

Data da defesa: 25/11/2010

Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 25 de novembro de 2010, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Antônio Augusto Medeiros Fröhlich
Centro Tecnológico / UFSC

Prof. Dr. Mario Lúcio Côrtes
IC / UNICAMP

Prof. Dr. Paulo Cesar Centoducatte
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

SPARC16: Uma nova visão de compressão para processadores SPARC

Leonardo Luiz Ecco[1]

25 de Novembro de 2010

Banca Examinadora:

- Prof. Dr. Paulo Cesar Centoducatte (Coorientador)
- Prof. Dr. Antônio Augusto Medeiros Fröhlich
- Prof. Dr. Mario Lúcio Côrtes
- Prof. Dr. Luiz Cláudio Villar dos Santos (Suplente)
- Prof. Dr. Guido Araújo (Suplente)

[1]Suporte financeiro das seguintes agências de fomento: CNPq (processo 131854/2008-9) e FAPESP (processo 2008/01970-5).

Resumo

Processadores RISC podem ser usados para atender a crescente demanda por desempenho requerida por sistemas embarcados. Entretanto, essas arquiteturas têm como desvantagem uma densidade de código ruim. Recodificações do conjunto de instruções, como o MIPS16 e o Thumb, representam uma abordagem eficiente para lidar com esse problema. Esse trabalho propõe uma codificação alternativa para a arquitetura SPARCv8. A nova codificação, chamada SPARC16, foi projetada com a ajuda de um modelo de programação linear inteira. As novas instruções utilizam 16 bits para serem codificadas e são facilmente traduzidas para suas correspondentes no conjunto de instruções original em tempo de execução, tornando possível posicionar um descompressor antes do estágio de *decode* de um processador SPARC e usar o restante do *pipeline* de forma transparente. O descompressor foi projetado e integrado no processador Leon 3 (SPARCv8) e ocasionou um acréscimo de 24% na área e nenhuma penalização na frequência. Apenas um montador foi implementado para a extensão SPARC16. O descompressor foi validado através de programas que exercitam todas as instruções SPARC16 escritos diretamente em linguagem de montagem. As razões de compressão dos programas dos *benchmarks* Mediabench e Mibench foram obtidas inferindo como código SPARCv8 seria representado com instruções SPARC16. Através desse método, razões de compressão de até 58% foram atingidas (para o programa *cjpeg*) com uma média de 61.27% para os programas do Mediabench e 60.77% para os programas do Mibench. Utilizando a mesma abordagem, uma avaliação da mudança trazida pelo uso de SPARC16 nos padrões de acesso à *cache* de instruções foi feita e mostrou reduções no número de *misses* até superiores a 50%.

Abstract

RISC processors can be used to face the ever increasing demand for performance required by embedded systems. Nevertheless, these architectures have as drawback a poor code density. Alternate encodings for instruction sets, such as MIPS16 and Thumb, represent an effective approach to deal with this problem. This work proposes an alternate encoding for the SPARCv8 architecture. The new encoding, called SPARC16, was designed with the aid of an integer linear programming model. The new instructions are 16-bits wide and are easily translated to its 32-bit counterparts during execution time, making it possible to place a decompressor engine before the decode stage of a SPARC processor and use the remaining of the pipeline transparently. The decompressor engine was designed and integrated into the Leon 3 processor (SPARCv8) and caused an increase of 24% in area and no timing overhead. Only an assembler was implemented for the SPARC16 extension. The decompressor engine was validated using programs that cover all the SPARC16 instructions written directly in assembly language. The compression ratios for the programs belonging to the Mediabench and Mibench benchmarks were obtained inferring how SPARCv8 code would be represented with SPARC16 instructions. Through this method, compression ratios as low as 58% were achieved (for the *cjpeg* program) with an average of 61.27% for the Mediabench programs and 60.77% for the Mibench programs. Using the same approach, an evaluation of the change brought by the use of SPARC16 in the instruction *cache* access patterns was performed and showed reductions in the number of misses even greater than 50%.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A crescente capacidade de integrar transistores em uma mesma área de silício permite implementar sistemas computacionais inteiros em um único *chip*. Esses sistemas, conhecidos como *System-on-a-chip*, desempenham funções cada vez mais sofisticadas e envolvem algoritmos complexos, interfaces com o usuário e suporte à aplicações em tempo real [54].

Para suprir todo o poder computacional demandado por esses sistemas, vêm sendo adotados processadores com um conjunto de instruções reduzidas (RISCs), uma vez que eles, em geral, possuem uma frequência de relógio mais alta e melhor desempenho que os CISCs. Em contrapartida, as arquiteturas RISC são amplamente conhecidas por terem uma densidade de código baixa, ou seja, o número de bits necessário para representar um programa nessas arquiteturas é maior do que o observado para arquiteturas CISC. Isso se deve principalmente ao fato de que, em um RISC, todas as instruções ocupam exatamente o mesmo espaço na memória (por exemplo, um `NOP`[1] utiliza o mesmo número de *bits* que uma instrução `LOAD` que utilize imediatos).

Esse fator combinado à *softwares* cada vez mais complexos implica códigos cada vez maiores, tornando necessário destinar uma área maior do sistema à memória, aumentando não só os custos ($custo \approx \text{área}^4$) [23], como também o consumo de energia do produto.

Uma estratégia usada para amenizar esses problemas é o uso de compressão de código. As técnicas baseadas nessa estratégia consistem, basicamente, em comprimir as instruções do programa depois da compilação e descomprimí-las durante a execução. É importante ressaltar que, apesar de ter sido concebida inicialmente para diminuir os requisitos de memória, pesquisadores observaram que essa abordagem também pode melhorar o desempenho e o consumo de energia das aplicações [10, 11]. Isso ocorre quando código comprimido é armazenado na *cache* de instruções, fazendo com que o número de falhas de acesso à mesma seja reduzido. Consequentemente, a memória principal, que é lenta, é acessada menos vezes, de modo que a aplicação execute mais rápido e consuma menos

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

energia (porque ocorre uma redução na atividade de transição do barramento).

Diversos trabalhos nessa área já estão disponíveis na literatura e alguns são descritos na capítulo 2. Dentre os trabalhos realizados no Instituto de Computação da UNICAMP, destacamos [10, 11, 15, 43–45], que utilizam o processador Leon 2, uma implementação GPL do padrão SPARCv8 [50] e métodos baseados em dicionários, onde as instruções que mais se repetem são armazenadas num dicionário e suas ocorrências no código são substituídas por índices para estes dicionários. Esse método, aparentemente simples, produz resultados extremamente eficientes nos três parâmetros desejados: quantidade de memória necessária para armazenar o programa, consumo de energia e desempenho.

As desvantagens dos métodos baseados em dicionário estão no momento da compressão do código, etapa geralmente realizada por um compressor que não faz parte do compilador, exigindo que os endereços de destino dos saltos sejam corrigidos diretamente nos arquivos binários executáveis. Também existem problemas relacionados à manipulação dos dicionários no que diz respeito a como tratar as bibliotecas de carga dinâmica e como fazer troca de contexto. Embora a solução trivial, trocar o dicionário em cada uma destas ocorrências, pareça factível, um simples aumento no tamanho do dicionário gerará um grande *overhead*.

Visando atacar essas deficiências, este projeto propõe uma solução alternativa de compressão de código para a arquitetura SPARC, denominada SPARC16. SPARC16 é um novo conjunto de instruções de 16 bits, projetado para ser traduzido facilmente para o conjunto SPARCv8 de 32 bits em tempo de execução, de forma similar ao MIPS16 e ao Thumb 1 e 2. Neste trabalho, foram estudados os problemas ocasionados pela criação de um conjunto de instruções de tamanho reduzido, desde a modelagem do conjunto de instruções até sua implementação final em *hardware*.

O grande atrativo dessa abordagem é a facilidade no projeto das ferramentas de software que comporão o sistema, uma vez que um mecanismo uniforme para representação das instruções facilita bastante o desenvolvimento de novas versões destas ferramentas, além de manter um fluxo de projeto similar ao anterior.

1.1 Fundamentação teórica

O uso de compressão de dados é tão corriqueiro hoje em dia, que, geralmente, passa despercebido pelas pessoas. Um arquivo MP3 (compressão de áudio) ou um arquivo MPEG (compressão de vídeo) podem ser citados para ilustrar essa situação. Esses formatos, em geral, partem de uma ideia simples: eliminar a redundância de informação. Basicamente, há duas formas para se obter tal feito:

Estatística: Nessa abordagem, leva-se em consideração a frequência com que os símbolos

aparecem no objeto a ser comprimido. Os mais frequentes são substituídos por *codewords* de tamanho reduzido. Os menos frequentes, por outro lado, são codificados com *codewords* de tamanho maior. A codificação de Huffman [25] é um exemplo clássico.

Dicionário: Nessa abordagem, conjuntos inteiros de símbolos que aparecem com frequência no objeto a ser comprimido são substituídos por índices para um dicionário. O dicionário contém as sequências de símbolos. Sendo os índices significativamente menores do que as sequências, obtém-se um objeto de tamanho reduzido.

Entretanto, há uma diferença significativa entre compressão de dados e compressão de código. No caso do último, é necessário que possamos ter acesso a partes aleatórias do programa alvo da compressão. Isso se deve à natureza procedimental do mesmo, que inclui saltos para rotinas em endereços distantes. Portanto, técnicas que descomprimem um objeto do início ao fim não são aplicáveis para essa finalidade. Obviamente, outras abordagens surgiram para lidar com esse problema e são apresentadas na seção 2.

O termo razão de compressão é utilizado para mensurar a eficácia das abordagens já existentes e também do método proposto neste trabalho. Ele é definido como:

$$\text{razão de compressão} = \frac{\text{tamanho do código comprimido}}{\text{tamanho do código original}} \times 100$$

Alguns métodos requerem que, além do programa comprimido, informação extra seja armazenada. É o caso dos métodos baseados em dicionário (para os quais o dicionário precisa ser armazenado). Nesses casos, a razão de compressão é dada por:

$$\text{razão de compressão} = \frac{\text{tamanho do código comprimido + informação extra}}{\text{tamanho do código original}} \times 100$$

Observe que quanto menor for a razão de compressão, melhor é o resultado.

1.2 Contribuições deste trabalho

Neste trabalho, é proposto um modelo de programação linear inteira para auxiliar na recodificação de conjuntos de instruções visando compressão de código. O modelo é genérico o suficiente pra ser aplicado em qualquer arquitetura. Nesse trabalho, ele foi aplicado à arquitetura SPARCv8 e ajudou a definir os formatos e instruções da extensão SPARC16.

As instruções SPARC16 podem ser facilmente traduzidas para instruções SPARCv8 equivalentes. Portanto, é possível executar código SPARC16 em um núcleo SPARCv8,

bastando que um descompressor (responsável pelo processo de tradução) seja inserido no processador. O descompressor foi projetado e integrado no processador Leon 3, chegando a uma implementação funcional em uma FPGA.

1.3 Organização

Este trabalho foi estruturado da seguinte forma:

- No capítulo 2 são apresentadas as técnicas de compressão de código disponíveis na literatura.
- No capítulo 3 é justificada a escolha da arquitetura SPARCv8 como alvo dos experimentos de compressão desse trabalho. Isso foi feito através de uma avaliação do tamanho de código do *benchmark* SPEC 2006 compilado para 15 variações de 7 arquiteturas. Além disso, uma visão geral da arquitetura SPARCv8 é apresentada.
- No capítulo 4 a extensão SPARC16 é descrita.
- No capítulo 5, o projeto, integração e validação do descompressor no processador Leon 3 são detalhados.
- No capítulo 6, é feita uma avaliação de como a integração do descompressor afetou o processador alvo no que diz respeito à área ocupada e frequência do *clock*. As estimativas de razão de compressão e a estimativa de mudança nos padrões de acesso à *cache* também são apresentadas nesse capítulo. As razões de compressão precisaram ser estimadas porque, até o presente momento, só um montador SPARC16 está disponível (permitindo a codificação de programas pequenos). O compilador está em desenvolvimento e faz parte do escopo do trabalho de doutorado de Bruno Cardoso Lopes, aluno do Laboratório de Sistemas de Computação da Unicamp.
- No capítulo 7 são apresentadas as conclusões e trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Uma estratégia óbvia para se obter tamanho de código reduzido é projetar o processador já levando em consideração esse aspecto, como foi o caso do VAX [1] e do Borroughs B1700 [53], que utilizam campos menores para codificar as instruções mais frequentes e campos maiores para as menos frequentes. Além disso, outras abordagens foram desenvolvidas e, nesse capítulo, são apresentadas em 3 seções: técnicas baseadas exclusivamente no uso de software, técnicas baseadas no uso de hardware específico e recodificação do conjunto de instruções.

2.1 Técnicas baseadas em software

A compressão de código utilizando exclusivamente software pode ser feita através do uso de módulos responsáveis pela descompressão em tempo de execução, da interpretação de código comprimido ou até mesmo através de alterações no compilador de forma a gerar menos código, como impedir o uso de *loop unrolling* ou *procedure inlining*.

Fraser e Proebsting [19] sugerem alterações no compilador *lcc* [18] para que, ao invés desse gerar código executável, fossem gerados simultaneamente uma representação intermediária compacta e um interpretador para esta. Os autores relatam uma razão de compressão média de 50%, entretanto, o tempo de execução é em torno de 20 vezes mais lento. Em uma continuação desse trabalho por Ernst [17], é proposto o formato *wire code* que é direcionado a sistemas onde o gargalo está na transferência dos dados e não na execução. O autor relata uma razão de compressão média de 59%. Liao [38, 39] propõe um método baseado em dicionário para reduzir o código de DSPs, onde depois da compilação, os blocos básicos do código são varridos para identificar sequências comuns de instruções. Essas sequências são, então, armazenadas em um dicionário e suas ocorrências no código são substituídas por chamadas a mini-subrotinas (chamadas a procedimentos sem argumentos). Os resultados indicam razões de compressão até 88%. Kirovski [28]

propõe um método cuja unidade de compressão é um procedimento. Os procedimentos são comprimidos utilizando-se o algoritmo Ziv-Lempel [37] e recebem um identificador. Uma área da memória, chamada procedure *cache* (*pcache*), é reservada para guardar os procedimentos descomprimidos. As chamadas a procedimentos no código compilado são substituídas por acessos a um serviço de diretório, que recebe o identificador de um procedimento, o localiza e o descomprime na *pcache*. O diretório fica responsável por manter um mapeamento entre os procedimentos no espaço comprimido e descomprimido. Os autores relatam uma razão de compressão em torno de 60% e uma perda de desempenho de 10%.

A grande desvantagem dessas técnicas é o impacto no tempo de execução do código. Nenhum dos métodos acima consegue executar o código comprimido na mesma faixa de desempenho que o original.

2.2 Técnicas baseadas no uso de hardware específico

Nesse tipo de abordagem, o programa é comprimido após a compilação e é descomprimido em tempo de execução com o auxílio de um descompressor em *hardware*. A posição do descompressor no sistema determina o tipo da arquitetura de descompressão:

***Cache-Decompressor-Memory* (CDM):** Indica que o descompressor está posicionado entre a *cache* e a memória principal. Permite esquemas de decompressão mais sofisticados, dado que o tempo gasto descomprimindo instruções é mascarado pela latência da memória. A desvantagem consiste em não manter instruções comprimidas na *cache*. A subseção 2.2.1 apresenta os trabalhos que utilizam esse esquema de implementação.

***Processor-Decompressor-Cache* (PDC):** Indica que o descompressor está posicionado entre o processador e a *cache*. As instruções são mantidas comprimidas na *cache*, diminuindo o número de *misses*. Entretanto, o método de descompressão precisa ser simples, sob pena de aumentar o *cycle-time* do processador. A subseção 2.2.2 apresenta os trabalhos que utilizam esse esquema de implementação.

A grande desvantagem do modelo de implementação usando *hardware* específico é que, na busca por simplicidade de *hardware*, transferiu-se a complexidade para as ferramentas que gerarão o software. Fazer trocas de contexto exige modificação no sistema operacional e gerar código que utilize bibliotecas demanda o uso de ferramentas extras que precisam ser inseridas no fluxo de desenvolvimento.

2.2.1 Arquiteturas CDM

Em 1992, Wolfe propôs o *Compressed Code RISC Processor* [55] (CCRP), que utiliza linhas da *cache* como unidade de compressão. As linhas são comprimidas, em tempo de compilação, utilizando o código de Huffman. Durante a execução, *cache misses* disparam o funcionamento do descompressor, que localiza a linha correta na memória principal, a descomprime e envia à *cache* de instruções. Uma vez que o mapeamento de endereços entre a *cache* e a memória principal é afetado, o descompressor utiliza uma tabela de tradução de endereços, batizada de LAT (*Line Address Table*). O CCRP mantém código descomprimido na *cache* de instruções, de modo que a compressão é transparente ao processador, ou seja, não é necessário utilizar *hardware* extra para tratar saltos ou ajustar o endereço de destino dos saltos no código original. A técnica mostrou razões de compressão de 73%, na média. Em um trabalho posterior, Beneš projetou um circuito decodificador Huffman específico para o CCRP [6, 7] que é capaz de decodificar 32 bits em 25ns.

A IBM também lançou um mecanismo para execução de código comprimido chamado CodePack [26, 27], disponível em processadores PowerPC. O CodePack é um método baseado em dicionário. Os projetistas chegaram a conclusão que se as instruções fossem divididas em duas partes de 16 bits (a "alta" e a "baixa") e fossem utilizados dicionários diferentes para cada uma delas, uma razão de compressão maior seria alcançada. A codificação usada é mostrada na Figura 2.1. Cada formato de símbolo consiste de uma *tag* de dois ou três bits seguida pelo valor do símbolo (cujos bits são representados por n). Para a parte alta, os oito símbolos mais frequentes recebem a codificação mais densa, 00_2. Os próximos 32 recebem a codificação 01_2, e assim por diante, até os menos frequentes, que recebem sua codificação original, 16 bits não comprimidos precedidos pela *tag* indicando o formato (111_2).

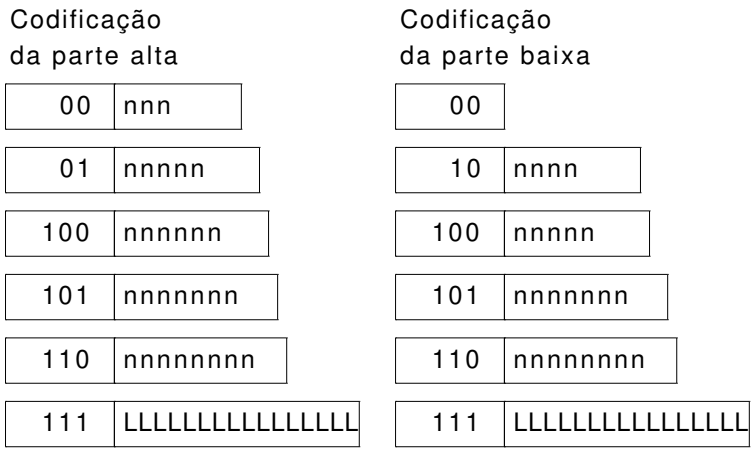


Figura 2.1: Codificação de símbolos do CodePack

O descompressor está posicionado entre a memória e o processador, conforme pode ser visto na figura 2.2. Isso permite que o processador continue utilizando os endereços originais das instruções (como se o código não estivesse comprimido) para acessar a memória. O descompressor fica responsável por encontrar a instrução correspondente no código comprimido. Para tanto, ele utiliza uma tabela que mapeia os endereços do espaço não comprimido para os endereços do espaço comprimido.

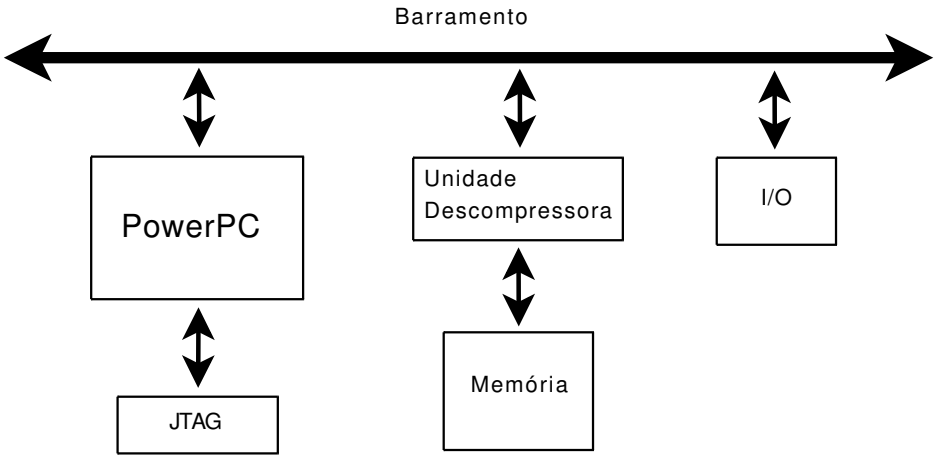


Figura 2.2: Configuração de um PowerPC com suporte a CodePack

Caso seja assumida uma memória de um ciclo, é fácil perceber que uma implementação ingênua desse esquema conseguiria trazer apenas uma instrução a cada três ciclos de relógio (um ciclo para indexar a tabela de mapeamentos e encontrar o endereço no "espaço descomprimido", um ciclo para buscar a instrução comprimida e finalmente outro ciclo para descomprimir a instrução). Para melhorar essa latência, duas otimizações foram implementadas. Primeiramente, a tabela de mapeamentos pode apontar para blocos de instruções (16 instruções por bloco) em vez de manter uma entrada para cada instrução. Com isso, o descompressor pode criar estágios de descompressão na forma de um *pipeline*, o que faz com que seja possível fornecer uma instrução por ciclo, desconsiderando o tempo para encher o *pipeline*. A outra otimização parte da observação que o processador passa uma grande parte do tempo executando trechos de códigos sequenciais, ou seja, sem instruções de salto. Portanto, o descompressor conhece com antecedência a próxima linha da *cache* a ser buscada na memória, eliminando a necessidade de consultar a tabela de mapeamentos e economizando um ciclo do relógio. Uma razão de compressão de 60% é reportada. Não há dados sobre o impacto no desempenho e no consumo de energia.

Dos projetos realizados no Instituto de Computação e na Faculdade de Eng. Elétrica e Computação da Unicamp, podemos destacar as abordagens de Pannain, Centoducatte e Azevedo. Pannain [4, 46] propõe a compressão baseada em fatoração de operandos

(PBC)[1]. Essa estratégia consiste em separar as árvores de expressão com instruções do processador em dois componentes: os padrões de árvore, que contêm fragmentos de *op-codes* da expressão e os padrões de operandos, que contêm os registradores e imediatos. Depois que os dois componentes são comprimidos, o programa passa a ser uma sequência de pares [Tp, Op], onde Tp é um padrão de árvore e Op é um padrão de operandos. Foram usados diferentes esquemas para comprimir os padrões, dentre eles Huffman e VLC[2], obtendo razões de compressão médias de 43% e 48%, respectivamente. Posteriormente, Centoducatte [3, 13] propõe a compressão baseada em árvores (TBC)[3]. Nesse método, as árvores de expressão não são decompostas, mas comprimidas inteiras. As árvores de expressão são agrupadas em n_c classes, cada classe k possuindo n_k árvores. O número de classes e o tamanho das mesmas é determinado exaustivamente de modo a obter a maior razão de compressão. Cada árvore de expressão no código é substituída por um par [prefixo, *codeword*], onde o prefixo identifica a classe da árvore. A razão de compressão média obtida é de 60,7%, levando em consideração o tamanho de estruturas de dados auxiliares utilizadas pelo método, além do *hardware* de descompressão. Finalmente, Azevedo [15] sugere a compressão baseada em instruções (IBC)[4], onde as instruções do processador são agrupadas em classes. As ocorrências das instruções no código original são substituídas por pares no formato [prefixo, *codeword*], onde o prefixo indica a classe da instrução e a *codeword* serve como um índice em uma tabela de instruções. Foram desenvolvidas implementações para MIPS e SPARC que obtiveram razões de compressão médias de 56,4% e 61,4%, respectivamente. O autor adiciona que a implementação para a arquitetura SPARC executa os programas com um acréscimo de tempo de, na média, 5,89%.

2.2.2 Arquiteturas PDC

Lefurgy [31–33] propõe um método baseado em um dicionário similar àquele usado no método de Liao, mas ao invés de tratar as sequências de código repetitivas como mini subrotinas, ele atribui *codewords* a cada uma delas e mistura código comprimido com código não comprimido na memória. Ao encontrar uma *codeword*, a lógica de decodificação de *codeword* gera um índice para o dicionário, juntamente com a quantidade de instruções que ela representa e o dicionário fornece ao processador as instruções referentes à *codeword*. As razões de compressão obtidas para os processadores PowerPC, ARM e i386 são de 61%, 66% e 75%, respectivamente. Entretanto, não são fornecidos detalhes a respeito da integração do descompressor nos processadores mencionados, nem o impacto causado pelo

[1]Do Inglês, *Pattern Based Compression*.
[2]Do inglês, *Variable length code*
[3]Do Inglês, *Tree Based Compression*.
[4]Do Inglês, *Instruction Based Compression*.

uso do mecanismo no desempenho e no consumo de energia.

Lekatsas [34, 35] propõe um esquema de compressão para a arquitetura SPARCv8 onde as instruções são separadas em quatro grupos: Instruções que fazem uso de imediatos (grupo 1, identificado pela sequência de bits 0_2), instruções de salto (grupo 2, identificado pela sequência 11_2), instruções de acesso rápido (grupo 3, identificado pela sequência 100_2), e instruções não comprimidas (grupo 4, identificado pela sequência 101_2). Para alcançar melhor compressão, diferentes esquemas foram utilizados para cada um dos grupos. O grupo 1 foi comprimido utilizando codificação aritmética (mais vantajosa em relação a Huffman). No grupo 2 se utilizou o menor número possível de bits para codificar o campo de deslocamento. Finalmente, as instruções do grupo 3 foram substituídas por índices para um dicionário de acesso rápido. O descompressor foi integrado ao *pipeline* do processador. Cada grupo de instrução passa por um *pipeline* diferente. As instruções do grupo 1, por exemplo, precisam de três ciclos para serem descomprimidas, ao passo que as instruções do grupo 4 simplesmente contornam o descompressor. Isso implica a necessidade de *hardware* extra para impedir que as instruções sejam entregues fora de ordem para o processador. São reportadas, na média, uma razão de compressão de 65%, uma redução no consumo de energia de 28% e um ganho de desempenho de 25%.

Em um outro trabalho, utilizando como alvo dos experimentos o Xtensa-1040, Lekatsas [36] propõe outra arquitetura de descompressão, essa capaz de descomprimir instruções em apenas um ciclo sem impactar o *cycle-time* do processador. Para conseguir isso, um esquema baseado somente em dicionários, mais simples que o proposto anteriormente, é utilizado. *Codewords* de 8 e 16 bits são utilizados para comprimir as instruções de 24 bits do Xtensa. O artigo relata uma média de 25% de aumento de desempenho e uma razão de compressão de 65%.

Benini [8] propõe um método de compressão baseado em um dicionário de 256 palavras de 32 bits. O processador alvo é o DLX. Visando uma redução no consumo de energia, a construção do dicionário é feita levando em consideração estatísticas de execução (número de vezes que cada instrução foi executada). A linha de *cache* é utilizada como unidade de compressão (sem perda de generalidade, o autor realiza experimentos com um tamanho de linha de 32 bytes). Uma razão de compressão média de 72% é relatada, bem como reduções de até 30% no consumo de energia utilizando uma *cache* de instruções de 4Kbytes e memória de programa *on-chip*.

Dos trabalhos realizados no Instituto de Computação da Unicamp, podemos citar Wanderley [43–45], que propõe o método PDC-ComPacket para a arquitetura SPARCv8. Ele é baseado em um dicionário de 256 palavras de 32 bits. Existem quatro tipos de pacotes ComPacket, apresentados na figura 2.3. O formato 4 possui quatro índices de seis bits. Nenhum índice aponta para instruções do tipo *branch*. O formato 3B possui três índices de seis bits, onde um dos índices aponta para uma instrução de *branch*. O formato

3B também possui um *offset* de seis bits utilizado pela instrução de *branch*. O formato 3 possui três índices de oito bits (nenhum aponta para *branches*) e, finalmente, o formato 2B possui dois índices de oito bits, onde um índice aponta para um *branch*, além de um *offset*, também de 8 bits, utilizado pela instrução de *branch*. Ao contrário dos índices de oito bits, os de seis bits só servem para indexar as primeiras 64 posições do dicionário. Todos os formatos possuem uma sequência de escape de 8 bits composta por um *opcode* SPARCv8 inválido (utilizado para identificar o compacket), além dos bits TT, S e B. Os bits S e B são usados para determinar o formato utilizado pelo ComPacket. $S = 0$ indica índices de seis bits e $B = 0$ indica que nenhum dos índices aponta para uma instrução do tipo *branch* no dicionário. Os bits TT indicam a partir de qual índice a execução deve continuar caso a execução salte para dentro de um ComPacket. O algoritmo para compressão consiste em pegar um programa já compilado e tentar substituir sequências que aparecem no código e/ou sejam executadas com frequência e substituí-las por pacotes dos formatos 4, 3, 3B e 2B, caso seja possível. Caso não seja, então as instruções não comprimidas são mantidas. É necessário realizar *patching* de endereços após a compressão.

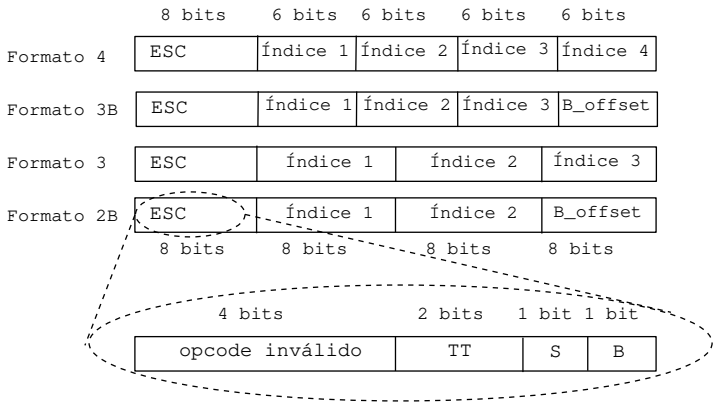


Figura 2.3: Formatos de ComPacket

Uma razão de compressão em torno de 75% é reportada. O descompressor foi implementado por Billo [10, 11] e foram observados aumento de desempenho de até 45% (redução de 22% no número de ciclos) e redução no consumo de energia de até 35%. As melhoras no desempenho e em consumo se devem à redução do número de *cache misses*, o que diminui o número de acessos à memória externa.

2.3 Codificações alternativas para o conjunto de instruções

Apesar do debate em torno de o que qualifica uma arquitetura como RISC, duas características são amplamente aceitas como requisitos básicos: máquina *load/store* e tamanho fixo de instruções. Como já foi mencionado anteriormente, o tamanho fixo de instruções, 32 bits na maior parte dos casos, é o grande responsável pela densidade de código baixa dos RISCs. Tendo isso em vista, Bunda et al. [12] publicaram, no início dos anos 90, um artigo investigando se código pouco denso é realmente o preço a ser pago pelo uso de uma arquitetura RISC.

Para tanto, é proposta uma nova codificação das instruções de 32 bits da arquitetura DLXe, uma variação do conjunto de instruções DLX de Hennessy e Patterson [23] que não permite operações de *load* e *store* nos registradores da unidade de ponto flutuante. A nova codificação, batizada de D16, consiste em instruções de 16 bits. A tabela 2.1 destaca as mudanças que tiveram que ser feitas para acomodar as instruções do DLXe em apenas 16 bits.

DLXe	D16
32 registradores visíveis	16 registradores visíveis
Instruções com 3 operandos	Instruções com 2 operandos
Imediatos maiores	Imediatos menores
Mais instruções com imediatos	Menos instruções com imediatos

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

Para determinar quais características do conjunto de instruções DLXe dão maior ganho no *tradeoff* desempenho x densidade de código, o compilador para DLXe é configurado seletivamente para não explorar uma determinada característica e então é utilizado para gerar o código, que sofre contagem estática e dinâmica de instruções. Os resultados são apresentados na tabela 2.2 e estão normalizados em relação aos parâmetros do código gerado especificamente para D16.

O uso de um conjunto de registradores menor fez com que o tráfego entre processador e memória aumentasse em 10% e que o número de instruções executadas ficasse ligeiramente maior. A redução nos campos de imediato gerou uma queda de desempenho de 9,5% pela necessidade de mais instruções para codificar os imediatos maiores. O uso de dois registradores por instrução no lugar de três causa um pequeno impacto negativo tanto no tamanho do programa quanto na quantidade de instruções executadas. Na configuração com 16 registradores e dois operandos por instrução, o código de 32 bits do DLXe ficou

1,62 vezes maior que o do D16 e foi executado em 95% do tempo (5% mais rápido que o D16).

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

Tabela 2.2: Resultados comparativos considerando D16=1.00 como referência

Seguindo essa linha de raciocínio e disposta a aumentar a densidade do código sem sacrificar o desempenho dos RISCs, a ARM e a MIPS lançaram recodificações dos seus conjuntos de instruções. A subseção 2.3.1 apresenta o Thumb da ARM, enquanto que o MIPS16 e o microMIPS são apresentados nas subseções 2.3.2 e 2.3.3, respectivamente.

2.3.1 ARM Thumb

A ARM [40] lançou, em 1995, uma extensão da arquitetura chamada Thumb [5], um conjunto de instruções de 16 bits. Os *cores* "*thumb aware*" são capazes de executar instruções tanto do conjunto original de 32 bits, quanto as novas instruções de tamanho reduzido. A distinção entre os modos de 16 e 32 bits é feita através de um bit de estado do processador que pode ser trocado através da instrução de salto `BX`.

Para que seja possível que o mesmo *core* execute instruções dos dois conjuntos, é necessário converter as instruções de 16 bits para suas correspondentes em 32 em tempo de execução. Isso é feito no estágio de decodificação do *pipeline*, quando as instruções Thumb passam por um descompressor, conforme pode ser visto na figura 2.4 (um exemplo de descompressão de uma instrução está disponível na figura 2.5). Não há impacto no *cycle time* do processador, uma vez que a conversão é feita numa fase do *clock* não utilizada pelo estágio em questão.

Assim como observado no caso do D16, a redução de tamanho das instruções diminui o poder de expressão das mesmas. O impacto mais óbvio ocorre na visibilidade do conjunto de registradores, reduzido a apenas 8 dos 15 registradores de propósito geral presentes na arquitetura. Apesar disso, algumas instruções do Thumb tornam possível acessar os registradores que ficam ocultos para as demais instruções. Os valores dos registradores não são alterados durante a transferência de modo entre 16 e 32 bits.

Os códigos convertidos para Thumb apresentam uma razão de compressão entre 55% e 70% e são executados em um tempo de 10% a 20% maior se colocados em um barramento de 32 bits. O desempenho do Thumb pode superar o do ARM em até 30% caso o

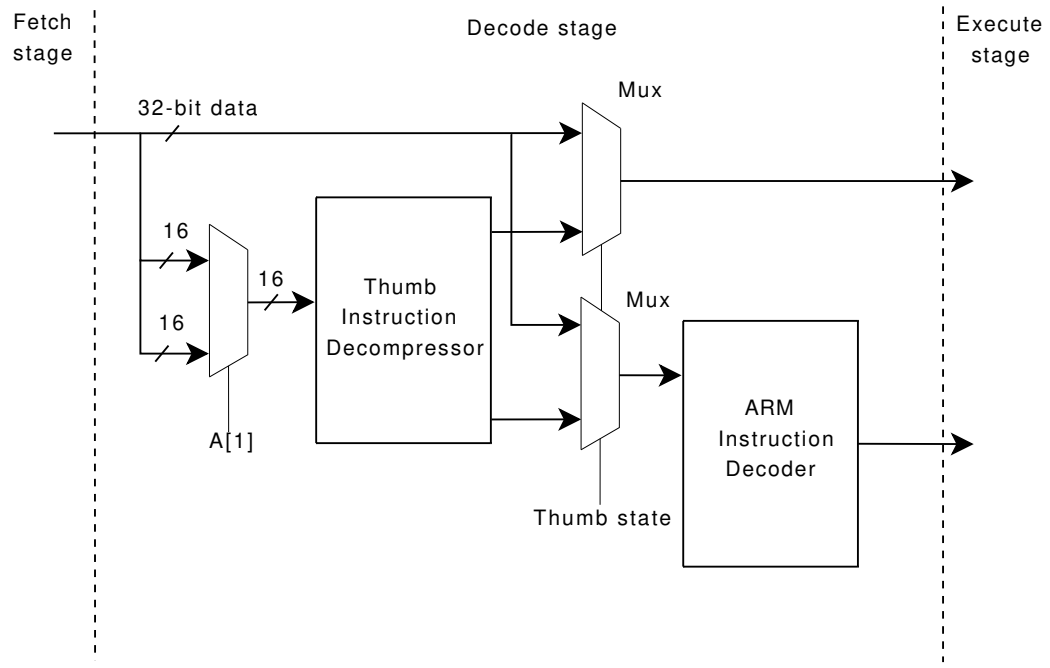


Figura 2.4: Decodificação e descompressão das instruções Thumb

barramento utilizado seja de 16 bits pois, nesse caso, são necessárias 2 leituras da memória para formar instruções de 32 bits enquanto as instruções de 16 bits são lidas de uma só vez.

Posteriormente a ARM criou o Thumb 2 [47], que extendeu o Thumb com novas instruções de 16 e de 32 bits (note que tratam-se de instruções de 32 bits do modo Thumb2, cujas codificações diferem das suas equivalentes ARM). Dentre as adições feitas ao Thumb estão instruções que acessam o coprocessador e instruções privilegiadas, tornando possível codificar o sistema operacional em modo 16 bits. A ARM reporta, para o Thumb 2, razões de compressão 5% menores do que os programas codificados em Thumb e uma perda de desempenho média de 2%.

2.3.2 MIPS16

A MIPS também desenvolveu um conjunto de instruções de 16 bits, o MIPS16 [29]. Assim como no modo Thumb, o mesmo *core* é usado para executar as instruções do MIPS16 e as do conjunto original. Conforme pode ser visto na figura 2.6, as instruções de 16 bits passam por um descompressor, posicionado entre a *cache* de instruções e o processador, que as converte em suas instruções correspondentes de 32 bits. É possível alternar entre os modos de execução através da instrução `JALX` (*Jump And Link with eXchange*). O estado do processador é mantido entre chamadas de procedimentos com troca do conjunto de

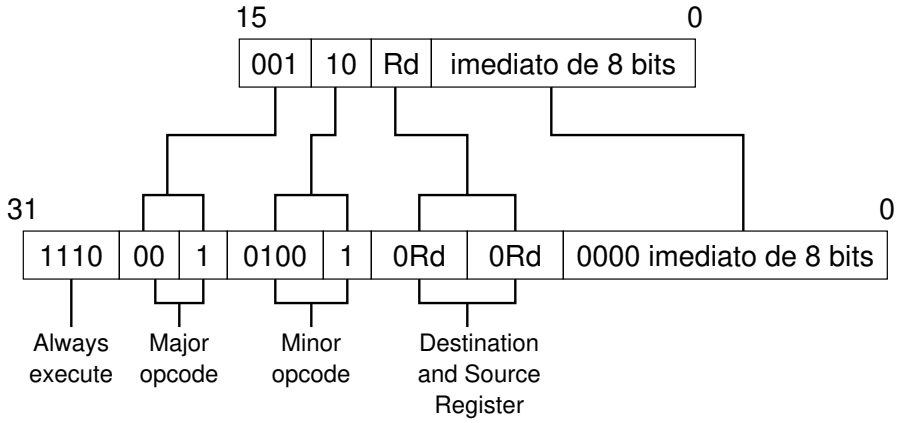


Figura 2.5: Instrução ARM `ADD Rd, #constante` representada nos dois formatos

instruções e também no processamento de interrupções.

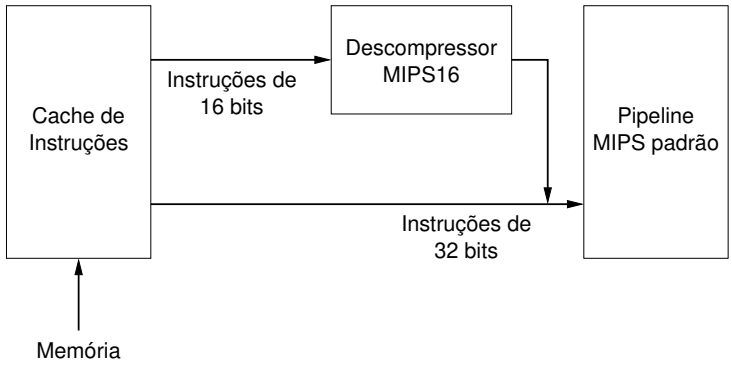


Figura 2.6: Diagrama de blocos de um processador MIPS com suporte ao conjunto de instruções MIPS16

Para conseguir reduzir o tamanho das instruções, algumas restrições sobre os tamanhos dos campos foram impostas. O conjunto de registradores de propósito geral visíveis foi reduzido para 8 (dos 32 presentes)[5] e, além disso, em muitos casos as instruções do MIPS16 só permitem dois registradores, obrigando um deles a servir como fonte e como destino do resultado da operação. Os campos de imediato também tiveram seu tamanho reduzido, no lugar dos 16 bits disponíveis nas instruções do conjunto original, as instruções do MIPS16 permitem apenas 5 bits. A figura 2.7 mostra um exemplo do mapeamento de uma instrução com campo de imediato em sua correspondente de 32 bits.

Alguns mecanismos foram implementados para contornar as limitações impostas pela falta de bits nas instruções:

[5]O *white paper* que apresenta o MIPS16 não fornece uma análise do *overhead* trazido por um conjunto reduzido de registradores visíveis.

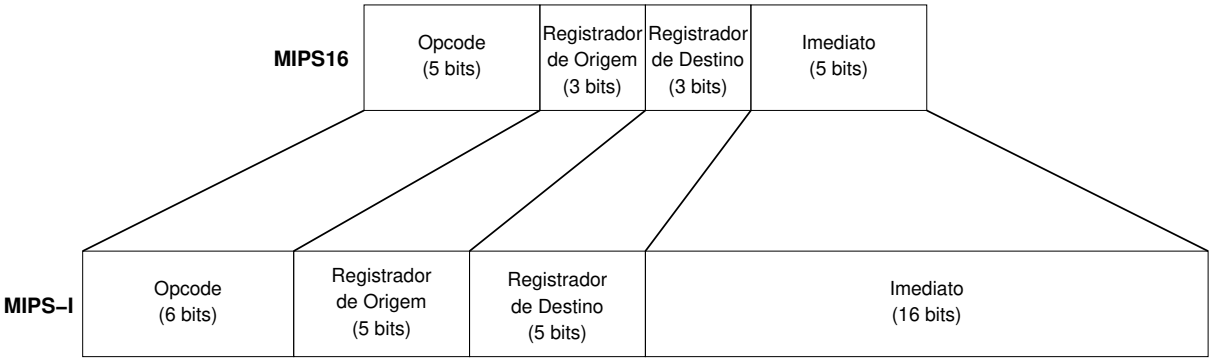


Figura 2.7: Conversão de uma instrução MIPS16 para o formato MIPS-I (32 bits)

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

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

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

Deslocamentos em *Loads/Stores*: De acordo com o tamanho do dado a ser buscado da memória, será automaticamente efetuado um deslocamento dos imediatos. Assim, se o processador efetuar um acesso a 32 bits da memória, o campo de imediato será deslocado 2 bits para a esquerda. No caso de acessos de 16 bits, ele será deslocado 1 bit apenas.

O artigo sobre o MIPS16 [29] cita uma razão de compressão média de 60% mas não informa o impacto no desempenho.

2.3.3 microMIPS

Recentemente, a MIPS apresentou o microMIPS [51]. Ao contrário do MIPS16, o microMIPS não é uma extensão, mas sim uma arquitetura completa. Ela consiste num *superset* do MIPS32, contendo versões de 16 bits das instruções mais comuns, além de todas as instruções do MIPS32 e do MIPS64. As instruções de 16 e de 32 bits podem ser intercaladas sem a necessidade de uma troca de modo. Para tornar isso possível, foi necessário um

remapeamento dos *opcodes* originais do MIPS32. A figura 2.8 exemplifica como isso foi feito para a instrução *load word*, que busca um valor na memória (no endereço dado pelo valor do registrador base somado ao *offset*) e o armazena no registrador rt. No MIPS32, o *opcode* dessa instrução é 100011_2. No microMIPS, essa mesma instrução possui o *opcode* 111111_2 para a versão de 32 bits e 011010_2 para a versão de 16.

Figura 2.8: Codificações da instrução *load word* de 16 e 32 bits no microMIPS

A MIPS também incluiu instruções novas visando comprimir o código e aumentar o desempenho. Uma delas é a instrução LWM (*load word multiple*), que tem sua codificação de 32 bits apresentada na figura 2.9 . Essa instrução especifica uma lista de 1 até 9 registradores[6] (no caso da codificação em 16 bits são até 5) que terão valores carregados a partir de posições sucessivas na memória. Outra instrução criada é a JRADDIUSP (*Jump Register, Adjust Stack Pointer*), cuja codificação de 16 bits é apresentada na figura 2.10. Ela transfere o controle da execução para o endereço armazenado no registrador de propósito geral 31 e adiciona um imediato de 7 bits ao *stack pointer*. Essa única instrução de 16 bits é capaz de realizar o trabalho de duas instruções MIPS32 comumente usadas ao entrar em uma subrotina. É interessante observar que tanto *lwm* quanto *jraddiusp* foram agrupadas com outras instruções (o *opcode* primário é um *pool* de instruções) e precisam de um *opcode* secundário para serem identificadas. Isso garante espaço para outras adições à arquitetura.

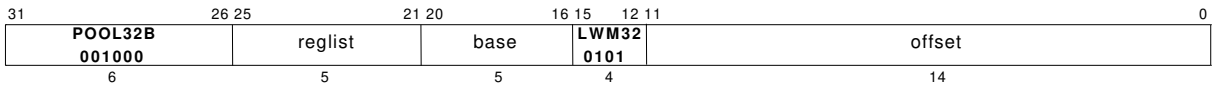


Figura 2.9: Codificação da instrução lwm32 no microMIPS

O microMIPS foi implementado nos *cores* M14K e M14Kc, através da adição de uma etapa de *recode*, que consiste em converter a instrução microMIPS em sua correspondente em MIPS32, dentro do estágio de *decode* do *pipeline*. Essa alternativa foi escolhida pelos projetistas devido ao já conhecido desempenho da microarquitetura MIPS32. Para rodar

[6]Apesar do campo reglist possuir 5 bits, não fica claro como ele é utilizado para determinar os 9 registradores a serem carregados. Apenas o whitepaper é aberto ao público, os manuais do microMIPS são restritos a quem adquire uma licença.

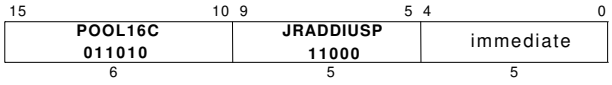


Figura 2.10: Codificação da instrução jraddiusp no microMIPS

código legado, a etapa de *recode* é ignorada e a instrução segue normalmente pelo *pipeline*. O uso de uma instrução para trocar entre o modo microMIPS e o modo MIPS32 se faz necessário apenas nessa situação. O artigo relata uma razão de compressão de 65% para o benchmark CSiBE [9] e um desempenho de 98% da arquitetura MIPS32 para o benchmark Dhrystone [52]. Não há dados sobre o impacto no consumo de energia.

2.4 Resumo das técnicas

As tabelas 2.3, 2.4, 2.5 e 2.6 trazem um resumo da revisão feita nesse capítulo. O campo tempo de execução é dado comparando-se a execução de código comprimido com a execução do código original (por exemplo, 75% de tempo de execução significa que o código comprimido levou apenas três quartos do tempo levado pelo código original para ser executado). Já o campo redução de energia apresenta a porcentagem que foi poupada (por exemplo, 28% de redução de energia significa que o programa comprimido consumiu 28% menos do que o original). A redução de consumo é decorrente da redução na atividade de transição no barramento entre o controlador de *cache* e a memória principal.

Note que na tabela que traz os resultados obtidos através da recodificação do conjunto de instruções, o campo redução de energia também foi omitido. A razão disso é que nenhum dos documentos fornecidos pela ARM e pela MIPS traz detalhes sobre esse aspecto, apesar de mencionarem possíveis reduções (devido ao menor número de acessos à memória externa).

Destaca-se que Azevedo, Pannain e Centoducatte levaram em consideração, ao apresentar a razão de compressão, uma memória que ocupasse a mesma área que o descompressor proposto.

Técnicas baseadas em software				
Autor	Arquitetura	*benchmarks*	Razão de compressão	Tempo de execução
Fraser e Proebsting [19]	SPARC	lcc e burg	50%	2000%
Kirovski [28]	SPARC	Mediabench	60%	110%
Liao [38, 39]	TMS320C25	Aipint2, bench, compress, dfx2hsh, gnucrypt, gzip, hill, jpeg, rsaref, rx, set	88%	—

Tabela 2.3: Resumo das técnicas baseadas em software

CDM					
Autor	Arquitetura	*benchmarks*	Razão de compressão	Tempo de execução	Redução de energia
Wolfe & Chanin [55]	MIPS	lex, pswarp, yacc, who, eightq, matrix25A, loop01, xlisp, espresso e spim	73%	—	—
IBM [26, 27]	PowerPC	—	60%	—	—
Pannain [4, 46]	MIPS	SPECint95	43%	—	—
Centoducatte [3, 13]	MIPS	SPECint95	60.7%	—	—
Azevedo [15]	MIPS	SPECint95	53,6%	—	—
Azevedo [15]	SPARC	SPECint95	61,4%	105,89%	—

Tabela 2.4: Resumo das técnicas CDM

PDC					
Autor	Arquitetura	*benchmarks*	Razão de compressão	Tempo de execução	Redução de energia
Lekatsas [34, 35]	SPARCv8	compress, diesel, i3d, key, mpeg, smo, trick	65%	75%	-28%
Lekatsas [36]	Xtensa 1040	compress, diesel, i3d, key, mpeg, smo, trick	65%	75%	—
Benini [8]	DLX	Ptolemy	72%	—	-30%
Lefurgy [31–33]	PowerPC	SPECint95	61%	—	—
Lefurgy [31–33]	ARM	SPECint95	66%	—	—
Lefurgy [31–33]	i386	SPECint95	75%	—	—
Billo [10, 11]	SPARCv8	susan, search, dijkstra, adpcm, pegwit, libmad	75%	78%	-45%

Tabela 2.5: Resumo das técnicas PDC

Recodificação da ISA				
Recodificação	Arquitetura	*benchmarks*	Razão de compressão	Tempo de execução
Thumb [5]	ARM	Eqntott, Xlisp, Espresso, Dhrystone	55%∼70%	110%∼120%
Thumb2 [47]	ARM	—	5% < Thumb	102%
MIPS16 [29]	MIPS	—	60%	—
microMIPS [51]	MIPS	CSiBE, Dhrystone	65%	102%

Tabela 2.6: Resumo das técnicas que envolvem recodificação da ISA

Capítulo 3

Avaliação de conjuntos de instruções

Para justificar a escolha da arquitetura SPARCv8 como alvo para os experimentos desse trabalho, foram avaliadas 15 variações de 7 arquiteturas diferentes levando-se em consideração o tamanho de código para um mesmo conjunto de programas, o benchmark SPEC 2006 [24]. O compilador GCC foi utilizado para compilar os programas para as 15 variações. As mesmas opções globais foram utilizadas em todos os casos — por opções globais entenda-se opções independentes de arquitetura. Das opções específicas para cada arquitetura, foram utilizadas as que resultam em menor tamanho de código.

A figura 3.1 mostra o tamanho total dos programas para cada variação de arquitetura suportada pelo GCC, ordenados de forma ascendente. Os tamanhos dos programas foram representados em 3 categorias diferentes e normalizados em relação aos menores em cada uma delas. Na figura, **AM** significa média aritmética de todos os programas, **GM** significa média geométrica e **TT** significa tamanho total. Tanto **AM** quanto **TT** foram normalizados em relação ao ARM (Thumb) e **GM** foi normalizado em relação ao M68K.

A melhor candidata a uma boa razão de compressão é uma arquitetura com baixa densidade de código. Além disso, também é importante que a arquitetura em questão ainda esteja em uso hoje em dia e que não possua um mecanismo oficial de compressão. Analisando da arquitetura com código menos denso para a de código mais denso, temos:

1. **Alpha:** As variações EV6 e EV7 possuem a pior densidade de código e, portanto, teriam maior potencial de compressão. Apesar disso, não foram escolhidas por terem sido descontinuadas.

2. **MIPS:** Tanto MIPS quanto MIPS32 são a próxima arquitetura com pior densidade de código. Mas a arquitetura MIPS já possui uma extensão de 16 bits chamada MIPS16. Infelizmente, o GCC fornecido para essa extensão não é capaz de compilar o benchmark SPEC2006 (problemas com a biblioteca C padrão newlib, focada em sistemas embarcados). De acordo com a documentação do MIPS16 [29], a razão de

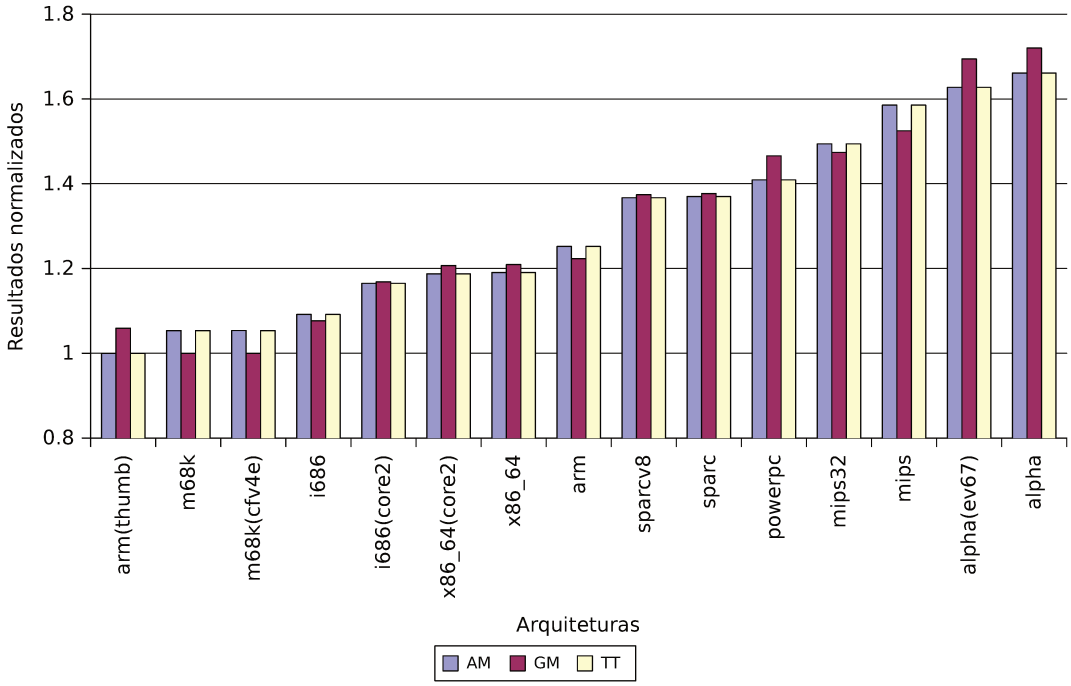


Figura 3.1: Tamanhos dos programas do benchmark SPEC 2006 compiladas com as mesmas opções globais do GCC

compressão é em torno de 60%, o que coloca a extensão MIPS16 no grupo de maior densidade de código.

3. **PowerPC:** Conforme mencionado no capítulo 2, já possui um mecanismo de compressão chamado CodePack [21]. Entra no mesmo conjunto que MIPS16 e Thumb.

4. **SPARC:** As arquiteturas SPARC e SPARCv8 [50] são as próximas no *rank* de baixa densidade de código, com tamanhos em torno de 40% maior do que o das arquiteturas de menor densidade. A arquitetura SPARC é o padrão IEEE 1754 e ganhou mercado com diferentes fabricantes ao redor do mundo (mais de 50 membros registrados na comunidade SPARC International, hoje em dia). Ela não possui um mecanismo oficial para execução de código comprimido e vem sendo usada em trabalhos acadêmicos como alvo de experimentos de compressão.

5. **ARM:** A arquitetura ARM possui as extensões Thumb e Thumb2, alcançando as maiores densidades de código dentre as arquiteturas avaliadas.

6. **i686 e x86_64:** As versões de 32 e 64 bits da arquitetura Intel. Uma observação

interessante é que o tamanho de código cresce com a evolução da arquitetura. Na média, tem-se um aumento de 15% no tamanho de código para os mesmo programas de i686 para x86_64.

7. **m68k:** A família Motorola 68000 possui código bastante denso, o menor se considerarmos apenas a média geométrica.

Baseado nas observações feitas acima, a arquitetura SPARCv8 aparece como uma excelente candidata, dado que possui uma densidade de código ruim e, portanto, tem potencial para reduções significativas no tamanho de código através do uso de uma codificação alternativa. Além disso, é uma arquitetura bastante utilizada atualmente.

Para comprimir a ISA SPARCv8, é preciso analisar como os campos das instruções são utilizados. Os campos de imediato merecem atenção especial uma vez que podem ocupar até metade dos bits de uma instrução. A figura 3.2 mostra a porcentagem acumulada de bits necessários para codificar um imediato em instruções lógicas, aritméticas ,de deslocamento e de load/store. Por exemplo, para operações aritméticas, um imediato de 3 bits é suficiente para codificar mais do que 60% das ocorrências dessa classe de instruções. Padrões similares também são observados para os programas do Mediabench.

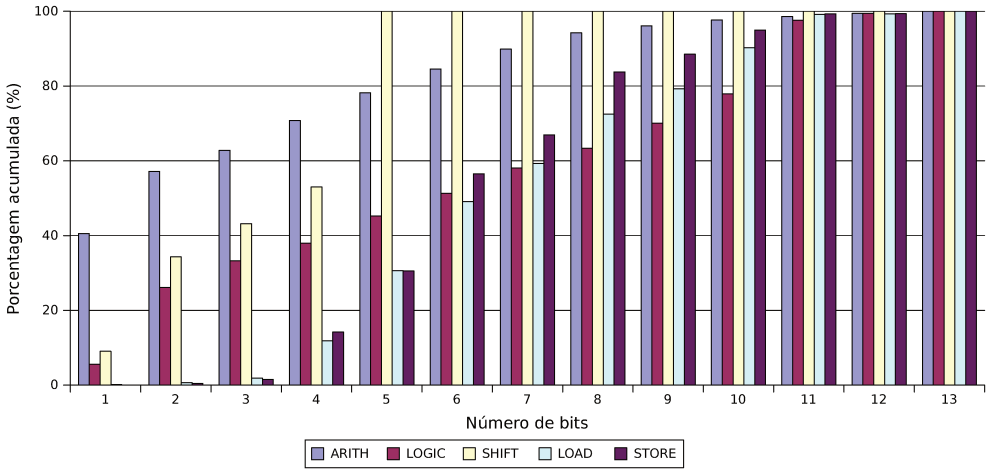


Figura 3.2: Tamanhos acumulados de imediatos utilizados em diferentes classes de instruções nos programas do MiBench

A seção 3.1 apresenta uma visão geral das características da arquitetura SPARCv8, o alvo dos experimentos de compressão desse trabalho. O capítulo 4 apresenta o mecanismo de compressão proposto para a arquitetura em questão, que consiste em uma recodificação das instruções.

3.1 A arquitetura SPARCv8

A arquitetura SPARC, cujo nome é um acrônimo para ***S**calable **P**rocessor **ARC**hitecture*, está publicada como o padrão IEEE 1754-1994. Ela define registradores de propósito geral, de ponto flutuante e de controle/status, além de 72 instruções codificadas em formatos de 32 bits.

Um processador SPARC é composto por uma unidade de inteiros e, opcionalmente, por uma unidade de ponto flutuante e um coprocessador. Todos os registradores, possivelmente com exceção dos que pertencem ao coprocessador, são de 32 bits. O processador possui dois modos de execução: **supervisor** e **usuário**. No modo supervisor, o processador pode executar qualquer instrução, incluindo as privilegiadas. No modo usuário, tentativas de executar instruções privilegiadas causam uma exceção e posterior transferência da execução para *software* supervisor.

Dentre as características principais dessa arquitetura, estão:

Formatos de instrução simples: Existem apenas 3 formatos de instrução, todos eles de 32 bits. Os *opcodes* e registradores são codificados de forma uniforme nos mesmos. Apenas instruções de carregamento ou armazenamento de dados são capazes de acessar a memória e interfaces de E/S.

Poucos modos de endereçamento: Um endereço de memória é dado pela soma de dois registradores ou de um registrador e de um imediato.

Instruções de 3 endereços: A maioria das instruções utiliza dois operandos (dois registradores ou um registrador e um imediato) e armazena o resultado em um terceiro registrador.

Janela de registradores: Em um determinado instante, um programa tem acesso a 8 registradores globais e a uma janela de 24 registradores dentro de um banco de registradores maior[1].

Um banco de registradores separado para a unidade de ponto flutuante: pode ser configurado através de *software* para funcionar como 32 registradores de 32 bits, 16 registradores de 64 bits ou 8 registradores de 128 bits.

A subseção 3.1.1 traz detalhes sobre os registradores da arquitetura. A subseção 3.1.2 apresenta de forma breve as instruções e seus formatos. Finalmente, a subseção 3.1.3 mostra como as chamadas de procedimento devem ser feitas.

[1] O número de janelas presente no banco de registradores é dependente da implementação, variando de 2 a 32.

3.1.1 Registradores

Um processador SPARC possui dois tipos de registradores: de propósito geral e de controle/status. Os registradores de propósito geral da unidade inteira são chamados registradores *r* e os da unidade de ponto flutuante são chamados *f*. Os registradores do coprocessador são dependentes da implementação. Dado que a extensão proposta nesse trabalho (SPARC16) dá suporte apenas a instruções da unidade inteira, essa seção foca apenas nos registradores da mesma.

Registradores de propósito geral

Uma implementação da unidade inteira pode conter de 40 a 520 registradores de propósito geral. Eles são particionados em 8 registradores globais (nomeados %g0 a %g7), além de um número dependente da implementação (que varia de 2 a 32) de conjuntos de 16 registradores. O conjunto de 16 registradores é dividido em 8 registradores de entrada (***ins***, nomeados %i0 a %i7) e 8 registradores locais (***locals***, nomeados %l0 a %l7).

Endereço do registrador na janela	Endereço do registrador *r*
in[0] — in[7]	r[24] — r[31]
local[0] — local[7]	r[16] — r[23]
out[0] — out[7]	r[8] — r[15]
global[0] — global[7]	r[0] — r[7]

Tabela 3.1: Endereçamento da janela

Em um dado momento, um programa pode acessar os 8 registradores globais e uma janela de 24 registradores dos registradores *r*. Uma janela é composta pelos 8 registradores de entrada e os 8 registradores locais de um determinado conjunto, juntamente com os 8 registradores de entrada do conjunto adjacente, endereçáveis a partir da janela corrente como registradores de saída (***outs***, nomeados %o0 a %o7). A tabela 3.1 mostra como os registradores globais, de entrada, de saída e locais são referenciados. A figura 3.3 apresenta 3 janelas sobrepostas, além dos 8 registradores globais.

A janela corrente é dada pelo ponteiro para a janela corrente (do inglês, *CWP, Current Window Pointer*), um campo de 5 bits dentro do registrador de estado do programa (do inglês, *PSR, Program State Register*). O *CWP* é incrementado através de uma instrução de *restore* (ou uma instrução *rett*) e decrementado por uma instrução de *save* ou uma exceção. O registrador de máscara de janela inválida (do inglês, *WIM, Window Invalid Mask*) é utilizado para detectar *overflow* ou *underflow* da janela. Ele é controlado por software executando no modo supervisor.

Cada janela compartilha os registradores de entrada e de saída com duas janelas adjacentes. Os registradores de saída da janela $CWP + 1$ são endereçáveis como os

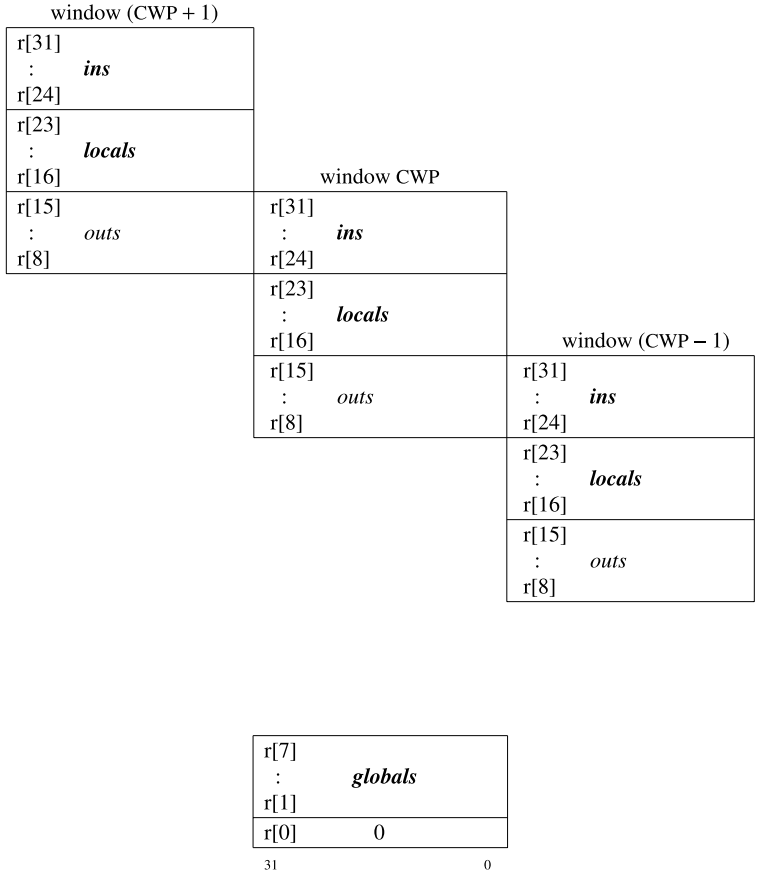


Figura 3.3: Três janelas sobrepostas e os 8 registradores globais (extraído de [50])

registradores de entrada da janela corrente. Os registradores de saída da janela corrente são endereçáveis como os registradores de entrada da janela $CWP - 1$. Os registradores locais são únicos a cada janela. A figura 3.4 ilustra essa sobreposição de registradores para um sistema com 8 janelas.

A utilização de 4 dos registradores r é fixada pela arquitetura. O registrador *%g0* sempre retorna o valor zero quando lido. O registrador *%o7* recebe o endereço de retorno quando uma instrução *call* (usada pra chamar uma rotina) é executada. Os registradores *%l1* e *%l2* recebem os valores de *pc* (o *program counter*) e *npc* (o valor que o *program counter* vai assumir no próximo ciclo), respectivamente, quando uma exceção ocorre.

Além disso, por convenção o registrador *%i6* é utilizado como *frame pointer* e o registrador *%o6* como *stack pointer*.

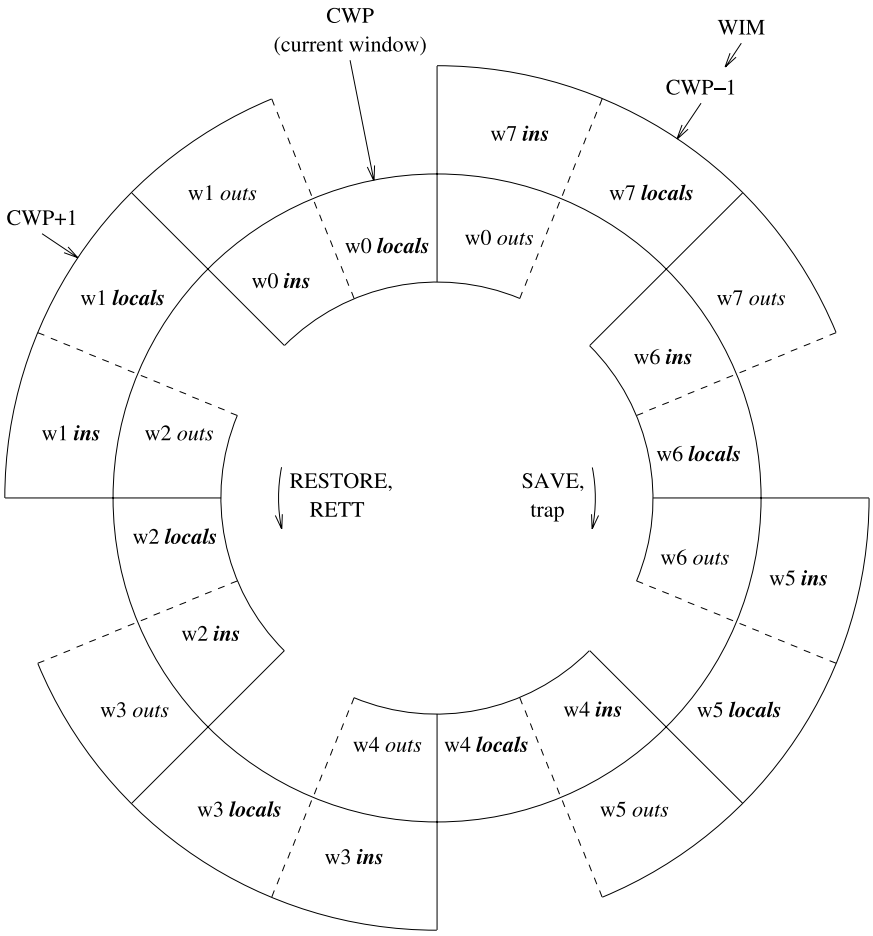


Figura 3.4: Janelas de registradores (extraído de [50])

Registradores de controle/*status*

Os registradores de controle/*status* da unidade inteira do SPARCv8 incluem o *PSR*, o *WIM*, o *TBR*, o registrador de multiplicação/divisão *Y*, além do *pc* e do *npc*. Eles são descritos a seguir:

***PSR*:** O registrador *PSR* é apresentado na figura 3.5.

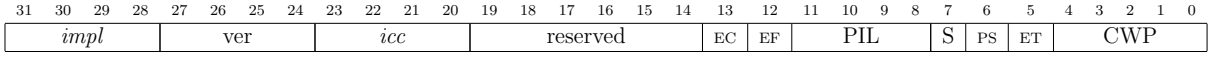


Figura 3.5: *Program State Register*

Os bits do campo *impl* identificam uma implementação (ou uma classe de implementações) da arquitetura. O campo *ver* pode ser apenas de leitura e, nesse caso, identifica um tipo particular de implementação, ou um campo de estado, cuja utilização depende da implementação. O campo *icc*, apresentado na figura 3.6, guarda os códigos de condição da unidade inteira. Esses bits são mudados pelas instruções aritméticas e lógicas cujo mnemônico termina em **cc**.

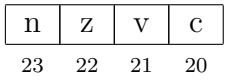


Figura 3.6: *Integer condition codes*

Os bits são afirmações feitas sobre o último resultado calculado pela ALU. O bit n indica se o valor calculado foi negativo. O bit z indica se o valor calculado foi zero. Os bits v e c indicam se ocorreu *overflow* ou *carry out*, respectivamente.

Os bits 19 a 14 são reservados para extensões arquiteturais. Os campos EC e EF indicam a presença de um coprocessador e de uma unidade de ponto-flutuante, respectivamente. O campo PIL indica a partir de qual nível de interrupções o processador vai aceitar uma interrupção. O campo S identifica o modo do processador (usuário ou supervisor). O campo PS guarda o valor que S possuía no momento da última interrupção. O bit ET determina se as interrupções estão ou não habilitadas. Finalmente, o campo CWP determina qual é a janela de registradores corrente, conforme explicado anteriormente.

***WIM*:** O registrador WIM (um acrônimo para *Window Invalid Mask*), apresentado na figura 3.7, é utilizado pelo *hardware* para determinar se uma interrupção de *overflow* ou *underflow* da janela foi gerada por uma instrução de *save*, *restore* ou *rett*.

Existe um bit de estado no *WIM* para cada janela de registradores da implementação. O bit $WIM[\mathrm{n}]$ corresponde à janela endereçada quando $CWP = n$.

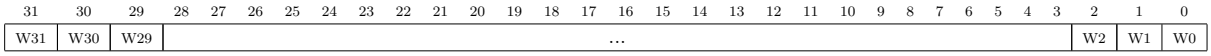


Figura 3.7: Registrador WIM (*Window Invalid Mask*)

Caso uma instrução *save*, *restore* ou *rett* faça com que o *CWP* aponte para uma janela inválida, ou seja, cujo bit correspondente em *WIM* seja 1 ($WIM[CWP] = 1$), uma exceção de *overflow* ou *underflow* da janela é causada.

***TBR*:** O registrador *TBR*, apresentado na figura 3.8, é utilizado para determinar o endereço para o qual a execução é transferida quando uma exceção ocorre. O campo *tba* contém os 20 bits mais significativos do endereço da tabela de interrupções. Esse campo é controlado por *software* supervisor, através da instrução *wrtbr*. Os bits 11 a 4 formam o campo *tt* (do inglês, *trap type*). Esse campo é escrito pelo *hardware* quando ocorre uma exceção e fornece um deslocamento dentro da tabela de interrupções. O campo *zero* contém apenas zeros e não é utilizado.

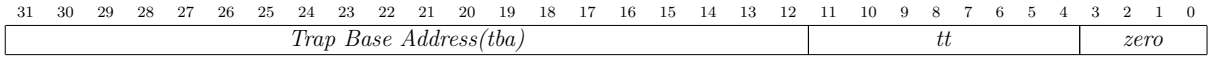


Figura 3.8: Campos do registrador *TBR* (*Trap Base Register*)

***Y*:** O registrador Y é utilizado pelas instruções de multiplicação e divisão. Nas instruções de multiplicação, os 32 bits mais significativos do resultado calculado são armazenados em Y. No caso de uma divisão, os 32 bits mais significativos do dividendo precisam ser colocados em Y. A leitura e escrita de Y podem ser feitas através das instruções *rdy* e *wry*, respectivamente.

***pc* e *npc*:** Os contadores de programa *pc* e *npc* contém o endereço da instrução sendo executada e da próxima instrução a ser executada, respectivamente.

3.1.2 Instruções

As instruções SPARCv8 são codificadas em 3 formatos de 32 bits. Os formatos 1, 2 e 3 são apresentados nas figuras 3.9, 3.10 e 3.11, respectivamente.

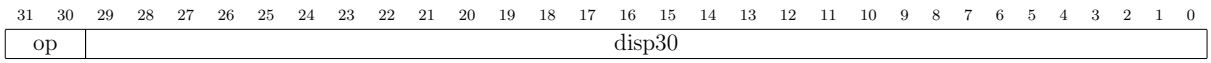


Figura 3.9: Formato 1

A tabela 3.2 mostra como a codificação do campo *op* determina o formato de uma instrução. O número 1 é utilizado para a instrução de salto incondicional *call* do formato

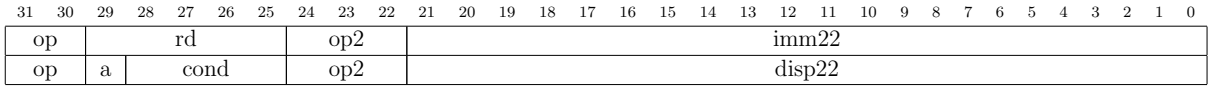


Figura 3.10: Formato 2

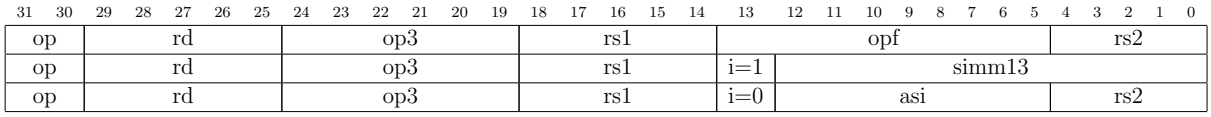


Figura 3.11: Formato 3

1. Ela possui apenas um campo de imediato de 30 bits. Esse campo é deslocado 2 bits à esquerda e somado ao *pc* quando a instrução *call* é executada. As instruções do formato 2 são codificadas com $op = 2$. Elas compreendem as instruções de *branch*, além da instrução *sethi* (utilizada para carregar um imediato de 22 bits nos bits mais significativos de um registrador). Finalmente, as instruções do formato 3 são codificadas usando $op = 2$ (para instruções lógicas, aritméticas e de deslocamento) e $op = 3$ (para instruções de acesso à memória).

Formato	*op*	Instruções
1	1	CALL
2	0	Bicc, FBfcc, CBccc, SETHI
3	3	Instruções de acesso à memória
3	2	Instruções lógicas, aritméticas, de deslocamento e restantes

Tabela 3.2: Codificação do campo *op*

As instruções pertencentes ao formato 2 possuem um *opcode* secundário (op2). A tabela 3.3 mostra como os valores de op2 são interpretados. Note que os valores 1, 3 e 5 não são utilizados para codificar nenhuma instrução e, portanto, podem ser utilizados por extensões da arquitetura. O valor 0 é utilizado para codificar a instrução *UNIMP*, que causa uma interrupção do tipo *illegal_instruction*.

A instrução *sethi* é codificada com $op2 = 4$. Para essa instrução, os bits 29 a 25 são interpretados como o endereço do registrador rd, cujos bits mais significativos receberão os 22 bits do campo imm22. Os valores 2, 6 e 7 são utilizados para codificar instruções de salto. Para essa classe de instruções, o bit *a* é interpretado como *annul bit*. Caso o valor desse bit seja *1* e a instrução seja um *branch* condicional não tomado ou um *branch* incondicional tomado, a instrução presente no *delay slot* não é executada. O campo *cond*, por sua vez, identifica qual condição deve ser verdadeira para que o salto seja tomado. A condição é verificada utilizando-se códigos de condição da unidade inteira, da unidade de ponto flutuante ou do coprocessador (de acordo com o valor de op2).

op2	Instruções
0	UNIMP
1	não implementado
2	Bicc (*branches* que utilizam códigos de condição da unidade inteira)
3	não implementado
4	SETHI
5	não implementado
6	FBfcc (*branches* que utilizam códigos de condição da unidade de ponto flutuante)
7	CBccc (*branches* que utilizam códigos de condição do coprocessador)

Tabela 3.3: Codificação do campo *op2*

Finalmente, as instruções do formato 3 são identificadas utilizando-se os bits de op e os 6 bits de um opcode secundário chamado op3. Esse formato é dividido em 3 subformatos. O primeiro deles é codificado usando $op3 = 110100$ (FPop1) ou $op3 = 110101$ (FPop2). Esses *opcodes* são utilizados para codificar instruções de ponto flutuante. A operação, em si, é determinada pelo valor do campo *opf*. As instruções FPop1 não alteram os códigos de condição da unidade de ponto flutuante, ao contrário das instruções FPop2. Vale ressaltar também que os campos rs1, rs2 e rd se referem a registradores da unidade de ponto flutuante.

Caso op3 não seja FPop1 ou FPop2, sabemos se tratar de um dos outros 2 subformatos. Eles são diferenciados pelo valor do campo i (o bit de número 13, que determina se a instrução faz uso ou não de um imediato). Caso $i = 1$, então os o campo formado pelos bits compreendidos entre a posição 12 e 0 são interpretados como um imediato sinalizado de 13 bits. As instruções desse subformato utilizam como operandos o imediato e o registrador rs1 e armazenam o resultado no registrador rd. Caso $i = 0$, sabemos que não há um imediato codificado na instrução. As instruções desse subformato realizam alguma operação com os registradores rs1 e rs2 e armazenam o resultado no registrador rd. Há também o campo *asi*, um identificador de espaços de endereçamento. Esse campo é utilizado pelas instruções *load/store alternate*, que permitem que o software acesse espaços de endereçamento diferentes do principal. Essas instruções só podem ser executadas quando o processador está em modo supervisor.

3.1.3 Chamadas de procedimentos

Como já foi explicado, os registradores de entrada (***ins***) e de saída (***outs***) tem como função principal passar parâmetros para procedimentos e receber os resultados calculados pelos mesmos. Quando uma rotina é chamada, os registradores de saída da rotina chamadora passam a ser vistos como os registradores de entrada da rotina chamada.

Outro aspecto que precisa ser observado para entender como a chamada de procedi-

mentos funciona é que convencionou-se utilizar o registrador %o6 como *stack pointer* e o registrador %i6 como *frame pointer*. Além disso, a instrução *call* coloca seu próprio endereço no registrador %o7 quando executada. Tendo isso em vista, vamos analisar o trecho de programa apresentado na figura 3.12 (note que o símbolo ';' é utilizado para comentários):

```
1    sum:
2      save  %sp, -96, %sp ; prólogo da função
3
4      add %i0, %i1, %l0   ; Soma os dois argumentos e coloca
5                          ; o resultado no registrador %l0
6
7      mov %l0, %i0   ; Coloca o valor calculado no registrador
8                     ; %i0 (o registrador de entrada também é utilizado para retornar
                         valores)
9
10     ; epílogo da função
11     ret         ; jmpl %i7+8, %g0
12     restore     ; restore %g0,%g0,%g0
13     nop         ; delay slot
14
15   main:
16     save %sp, -96, %sp ; prólogo
17
18     mov 10, %o0
19     mov 20, %o1
20     call sum ; poderia ser jmpl <endereço de sum>, %o7
21     nop        ; delay slot
22
23     ... ; Mais algum processamento
24
25     mov 0, %g1 ; Valor de retorno da função main
26     restore ; restore %g0, %g0, %g0
27     retl    ; jmpl %o7+8, %g0
28     nop     ; delay slot
```

Figura 3.12: Chamada de procedimento na arquitetura SPARCv8

O trecho de programa é composto por duas funções. A função *main* simplesmente coloca dois argumentos nos registradores de saída e chama a função *sum* (que retorna a soma desses dois argumentos). A passagem de argumentos e a chamada de *sum* são feitas pelas instruções das linhas 18, 19 e 20.

Agora vamos nos concentrar no código da função *sum*. Podemos dividí-lo em 3 partes: prólogo, corpo e epílogo. Essas partes são detalhadas abaixo:

Prólogo: É onde a pilha e os registradores são preparados para serem usados. No caso da arquitetura SPARCv8, duas ações precisam ser realizadas: decrementar o *CWP* (para que a função tenha acesso a uma nova janela de registradores) e decrementar o *stack pointer* (para garantir que a função tenha espaço na pilha para alocar variáveis locais). Isso é feito utilizando apenas a instrução *save*. Note que decrementar

o *CWP* faz com que o *stack pointer* da função chamadora (%o7) passe a ser o *frame pointer* da função chamada (%i7). Note também que a instrução *save* utiliza como operandos os registradores da janela antiga e armazena seu resultado em um registrador da janela nova. A instrução *save* da função *sum* está na linha 2.

Corpo: É onde o processamento é realizado. No caso da função *sum*, os dois argumentos são somados e o resultado é colocado em um registrador local (linha 4). Depois, o valor calculado é movido para o registrador de entrada %i0 para permitir que a função chamadora tenha acesso a ele (linha 7).

Epílogo: É o conjunto de instruções que reverte as ações feitas pelo prólogo e retorna o controle para a função chamadora. Na arquitetura SPARCv8, isso é feito através das pseudo-instruções[2] *ret* (*jmpl %i7+8, %g0*) e *restore* (*restore %g0, %g0, %g0*). A instrução *ret* (linha 11) retorna o controle para a função chamadora. Note que a instrução *call* colocou seu próprio endereço no registrador %o7 (disponível em %i7 na janela da função chamada) e que o imediato 8 é utilizado para que o controle seja passado para a instrução que sucede o *delay slot*. A instrução *restore* (linha 12) incrementa o *CWP* restaurando a janela da função chamadora.

O entendimento da convenção de chamadas utilizadas pela arquitetura SPARCv8 é importante para compreender como as instruções envolvidas foram recodificadas. Por exemplo, praticamente todas as ocorrências da instrução *save* tem como fonte e destino o registrador de *stack pointer*. Logo, foi criada uma instrução SPARC16 que utiliza esse registrador de forma implícita (fazendo com que mais bits estejam disponíveis para codificar o imediato).

3.2 Considerações finais e conclusões

Esse capítulo mostrou as razões que motivaram a escolha da arquitetura SPARCv8 como alvo dos experimentos de compressão desse trabalho. A avaliação de tamanhos de código de diferentes arquiteturas mostrou que Alpha e MIPS são as que possuem a densidade mais pobre. Entretanto, a primeira foi descontinuada e a segunda já possui dois mecanismos oficiais de compressão (MIPS16 e microMIPS), de modo que SPARCv8 passou a ser a escolha óbvia.

Além disso, uma visão geral das características da arquitetura SPARCv8 foram apresentadas, destacando-se a janela de registradores, os formatos de instruções, registradores

[2]Uma pseudo-instrução não é um *opcode* real da arquitetura e sim apenas uma facilidade fornecida pelo montador que torna o código mais legível.

de controle da unidade de inteiros e convenção de chamada de procedimentos. Esses aspectos foram abordados porque ajudam a entender decisões tomadas na hora de projetar a extensão SPARC16.

Capítulo 4

Uma Recodificação das instruções SPARCv8

Conforme mencionado anteriormente, este trabalho propõe uma recodificação das instruções SPARCv8 visando reduzir tamanho de código. Inicialmente, foram criadas versões de 16 bits das instruções SPARCv8. Para criá-las, foi desenvolvido um modelo de programação linear inteira (PLI) [42], apresentado na seção 4.1. Ao resolver o modelo, precauções foram tomadas para garantir que houvesse espaço para a inclusão manual de novas instruções, conforme fosse necessário.

A esse novo conjunto de instruções deu-se o nome de SPARC16. No decorrer do texto, SPARC16 será sempre tratado como uma extensão arquitetural. Isso reflete o fato de que SPARC16 precisa de um processador SPARCv8 para funcionar. Um descompressor traduz, em tempo de execução, as instruções SPARC16 para suas correspondentes em SPARCv8. Esse esquema de compressão é análogo aos modos MIPS16 e Thumb para as arquiteturas MIPS e ARM, respectivamente.

As instruções SPARC16 possuem imediatos menores do que suas correspondentes em SPARCv8. Além disso, o número de registradores visíveis é 8, ao contrário dos 32 na codificação original (devido ao número reduzido de bits para indexar o banco). Para contornar o problema imposto pela redução dos imediatos, criou-se a instrução EXTEND, que não faz nada a não ser carregar um imediato ou um registrador extra a ser usado pela instrução executada em sequência. Para lidar com o menor número de registradores visíveis, o registrador %g0 e os registradores associados à chamadas de procedimentos (*stack pointer, frame pointer* e *return address*) são referenciados unicamente de forma implícita por instruções especiais. Há, ainda, a possibilidade de fazer *register spilling* nos registradores invisíveis, também através de instruções especiais. Essas e outras características da extensão SPARC16 são detalhadas na seção 4.2. Um descompressor em *hardware* foi implementado no processador Leon 3, permitindo que instruções SPARC16

sejam executadas em um processador SPARCv8. Os detalhes sobre a implementação estão disponíveis no capítulo 5.

4.1 Um modelo de PLI utilizado para auxiliar a definição da codificação

Conforme mencionado no início deste capítulo, um modelo de programação linear inteira foi desenvolvido com o intuito de encontrar uma codificação de 16 bits para as instruções SPARCv8. A entrada para o modelo de PLI é uma tupla (S, F, I, c). Cada $s \in S$ é um conjunto de campos. A tabela 4.1 apresenta os campos que cada um dos conjuntos $s \in S$ possui. Os campos estão separados por categoria: *opcode* primário, *opcode* secundário, registrador(es) e imediato. O campo de *opcode* primário é obrigatório. Os campos de *opcode* secundário e imediato são opcionais e limitados a 1 por cada $s \in S$. Os campos de registradores também são opcionais, mas cada elemento $s \in S$ pode possuir até 3 deles.

O identificador de cada elemento $s \in S$ reflete os campos nele contidos: um R significa que o elemento possui um registrador, um I significa que o elemento possui um imediato e o número 2 significa que o elemento possui um campo de *opcode* secundário. Todos os elementos $s \in S$ possuem um *opcode* primário e, portanto, não há necessidade de incluir um carácter no seu identificador pra representar essa informação. Por exemplo, o elemento RI possui um *opcode* primário, um registrador e um imediato. O elemento RRI2 possui um *opcode* primário, dois registradores, um imediato e um *opcode* secundário.

Uma observação merece destaque: apesar de todos os conjuntos $s \in S$ compartilharem o mesmo campo de *opcode* primário (opc_1), essa afirmação não é válida para imediatos e *opcodes* secundários. A razão disso ficará clara depois de definirmos todos os elementos da tupla (S, F, I, c).

Uma atribuição de tamanhos a cada um dos campos de um conjunto $s \in S$ corresponde a um formato. Os formatos são armazenados no conjunto F e obedecem à duas regras: a soma dos tamanhos de seus campos é sempre 16 e o tamanho de um campo de registrador é sempre 3. Para cada $s \in S$, todos os formatos válidos são gerados. Por exemplo, a tabela 4.2 apresenta os formatos gerados para o conjunto RRI, que possui um *opcode* primário, dois registradores e um imediato. O tamanho máximo de um *opcode*, seja ele primário ou secundário, é de 7 bits. Isso porque um *opcode* primário de 7 bits implica a possibilidade de codificar até 128 instruções e utilizar mais bits pra esse campo reduziria os bits disponíveis para codificar registradores e imediatos.

O conjunto I possui instruções candidatas a fazerem parte da extensão SPARC16. As instruções de I são instruções e pseudo-instruções da arquitetura SPARCv8. Pseudo-instruções foram incluídas uma vez que optamos por não deixarmos os registradores %g0,

Tabela 4.1: Campos contidos em $s \in S$

	Campos			
$s \in S$	Opcode primário	Opcode secundário	Registrador(es)	Imediato
I	opc_1			imm_1
RI	opc_1		reg_1	imm_2
RRI	opc_1		reg_1, reg_2	imm_3
RR	opc_1		reg_1, reg_2	
RRR	opc_1		reg_1, reg_2, reg_3	
I2	opc_1	opc_2		imm_4
RI2	opc_1	opc_3	reg_1	imm_5
RRI2	opc_1	opc_4	reg_1, reg_2	imm_6
RR2	opc_1	opc_5	reg_1, reg_2	
RRR2	opc_1	opc_6	reg_1, reg_2, reg_3	

Tabela 4.2: Formatos para o conjunto RRI

	Tamanho dos campos (bits)			
Formatos	opc_1	reg_1	reg_2	imm_3
F1	1	3	3	9
F2	2	3	3	8
F3	3	3	3	7
F4	4	3	3	6
F5	5	3	3	5
F6	6	3	3	4
F7	7	3	3	3

%sp, %fp e %ra visíveis na extensão SPARC16. O objetivo por trás dessa abordagem é evitar que a extensão SPARC16 tenha apenas 4 registradores utilizáveis, dado que os campos de registrador possuem 3 bits, possibilitando endereçar oito registradores diferentes (e quatro endereços já estariam comprometidos com os registradores previamente mencionados). Outro ganho importante decorrente de referenciar um registrador de forma implícita é que os bits não utilizados com um campo extra de registrador podem ser incorporados num campo de imediato, aumentando o poder de expressão da instrução.

A função de custo $c : I \times F \rightarrow \mathbb{I}$ especifica o custo de mapear $i \in I$ para o formato $f \in F$. Alguns mapeamentos são inválidos, como por exemplo tentar associar uma instrução que precisa de um imediato com um formato de um conjunto $s \in S$ que não possui um campo de imediato. Durante a geração de c, esses mapeamentos são detectados e desconsiderados. Os custos foram calculados usando programas dos benchmarks Mediabench [30] e Mibench [22]. Para cada instrução nesses programas, uma instrução equivalente em I era identificada e o custo de fazer tal associação era calculado levando em consideração fatores como o tamanho do campo do imediato ser grande o suficiente para acomodar o imediato da instrução e tentativas de representar uma instrução de 3 endereços como uma instrução de 2 endereços (forçando um registrador a trabalhar simultaneamente como fonte e destino de uma operação).

Para permitir que instruções $i \in I$ sejam descartadas, ou seja, não sejam escolhidas para compor a extensão SPARC16, o formato especial 0 foi criado. O custo de associar uma instrução $i \in I$ com 0 representa o custo de não suportar i na extensão SPARC16. Esse custo foi calculado estimando o número de instruções da extensão SPARC16 que devem ser executadas para se alcançar o mesmo efeito da instrução descartada. Esse valor é especulativo, ou seja, seria possível representar uma instrução *nor* através de uma instrução *or* seguida de uma instrução *not*, entretanto, como ainda não definimos quais instruções compõe a extensão SPARC16, não sabemos se as instruções *not* e *or* estarão disponíveis. Para tentar aumentar a probabilidade do valor especulado estar correto, certas instruções como *add, sub, and* e *or*, sabidamente imprescindíveis a qualquer arquitetura, tiveram seu custo de associação com o padrão 0 estabelecido como um valor infinito, garantindo sua inclusão na extensão SPARC16.

Dada a tupla (S, F, I, c), o problema é mapear toda candidata $i \in I$ para um único formato $f \in F$ minimizando o custo total incorrido no mapeamento (lembrando que é possível descartar uma instrução mapeando-a no formato especial 0). Entretanto, para que o resultado tenha utilidade prática, as seguintes condições precisam ser respeitadas:

1. Sejam $f1, f2 \in F$ formatos escolhidos, os campos presentes em $f1$ que também estão presentes em $f2$ precisam, necessariamente, ter o mesmo tamanho em bits. Por isso, todos os formatos possuem como *opcode* primário o campo opc_1. Dessa forma, garantimos que todos os formatos escolhidos vão possuir um *opcode* primário

de mesmo tamanho.

2. Se k é o tamanho do opcode primário, então no máximo 2^k *slots* podem ser alocados para instruções. As situações que implicam alocação de *slots* estão enumeradas abaixo:

   (a) Se a instrução está mapeada em um formato $f \in F$ sem *opcode* secundário, ela utiliza um *slot*.

   (b) Se a instrução está mapeada em um formato $f \in F$ que possui *opcode* secundário de p bits, o mapeamento de até 2^p instruções no formato f utiliza um *slot*, o mapeamento de $2^p + 1$ a $2 * 2^p$ instruções em f utiliza dois *slots* e assim sucessivamente.

   (c) Se a instrução está mapeada no formato especial 0, ela não utiliza nenhum *slot*.

Restrições foram impostas ao modelo para garantir essas condições. A descrição formal é dada a seguir:

Existem variáveis binárias x_{if} que indicam que uma instrução candidata $i \in I$ será mapeada em um formato $f \in F$ ou não. Também existem variáveis binárias y_f que especificam se um formato f foi escolhido — um formato é dito escolhido caso alguma candidata $i \in I$ seja mapeada nele. Um formato pode ter no máximo dois campos de opcode (um primário e outro secundário), sendo que o primário é obrigatório. Existem K campos de *opcode*, identificados por $opc_1, opc_2, \ldots, opc_K$. O *opcode* identificado por opc_1 é o *opcode* primário, enquanto que $opc_2, \ldots, opc_K$ são *opcodes* secundários. Cada *opcode* possui no máximo L bits. Existem variáveis binárias op_{kl} indicando que o k-ésimo *opcode* usa l bits. Existe uma variável especial inteira T que especifica o número total de *slots* do *opcode* primário. Para cada $k \in \{opc_2, \ldots, opc_K\}$ e $l \leq L$ são criadas variáveis inteiras G_{kl} que indicam o número de *slots* do *opcode* primário, dentre o total T, que serão utilizados por instruções que forem mapeadas em um formato que possui k com l bits como *opcode* secundário. Note que no formato em questão podem ser mapeadas no máximo $2^l G_{kl}$ instruções. A variável $G1$ indica o número de *slots* do *opcode* primário utilizados por instruções mapeadas em formatos que não possuem um *opcode* secundário. Os formatos $f \in F$ foram gerados a partir de um conjunto $s \in S$ específico e a notação $f \in s$ é utilizada pra expressar essa informação. A notação $i \in f$ é usada para indicar que a instrução $i \in I$ pode ser mapeada no formato $f \in F$, e F_i é o conjunto de formatos para os quais i pode ser mapeada. Abaixo, o modelo de PLI é descrito.

$$\text{Min} \sum_{i \in I} \sum_{f \in F_i} c(i, f) x_{if}$$

sujeito a

$$
\begin{array}{lll}
\sum_{f \in s} y_f & \leq 1, \text{ para } s \in S & (1) \\
x_{if} - y_f & \leq 0, \text{ para } i \in I \text{ e } f \in F_i & (2) \\
\sum_{f \in F_i} x_{if} & = 1, \text{ para } i \in I & (3) \\
y_f + y_{f'} & \leq 1, \text{ para } f, f' \in F \text{ e } f \text{ e } f' \text{ são inconsistentes} & (4) \\
\sum_{l \leq L} op_{kl} & = 1, \text{ para } k \leq K & (5) \\
y_f - op_{kl} & \leq 0, \text{ para cada } k \text{ e } l, \ f \text{ tem } op_{kl} & (6) \\
T - \sum_{l \leq L} 2^l op_{1l} & \leq 0 & (7) \\
G1 + \sum_{k=2}^{K} \sum_{l \leq L} G_{kl} - T & \leq 0 & (8) \\
G_{kl} - 2^L op_{kl} & \leq 0, \text{ para cada } k \text{ e } l & (9) \\
\sum_{(i,f) \in G1} x_{if} - G1 & \leq 0 & (10) \\
\sum_{(i,f) \in G_{kl}} x_{if} - 2^l G_{kl} & \leq 0, \text{ para cada } k \text{ e } l & (11)
\end{array}
$$

A restrição (1) garante que no máximo um formato $f \in F$ para cada conjunto $s \in S$ seja escolhido. A restrição (2) estabelece que uma instrução é atribuída a um formato apenas se esse formato foi escolhido. A restrição (3) garante que toda instrução $i \in I$ é mapeada em exatamente um formato (lembre-se de que as instruções podem ser mapeadas no formato 0). A restrição (4) garante que formatos inconsistentes, i.e formatos com alguns campos em comum mas tamanhos de campos diferentes, não possam ser escolhidos simultaneamente. A restrição (5) estabelece que os campos de *opcode* secundários terão no máximo l bits. A restrição (6) diz que um formato com o *opcode* k usando l bits pode ser usado apenas se a solução usa aquele *opcode* com l bits. A restrição (7) garante que se o *opcode* primário tiver k bits, então a quantidade total T de *slots* disponíveis será 2^k. A restrição (8) garante que o número de *slots* que são utilizados entre todos os grupos é no máximo T. A restrição (9) garante que o grupo G_{kl} será usado apenas se na solução o *opcode* k usa l bits. As restrições (10) e (11) limitam o número de instruções que podem ser atribuídas a cada grupo. No caso da restrição (10), $(i, f) \in G1$ se refere a uma instrução $i \in I$ mapeada em um formato $f \in F$ que não possui *opcode* secundário. No caso da restrição (11), $(i, f) \in G_{kl}$ se refere a uma instrução $i \in I$ mapeada em um formato $f \in F$ que possui k com l bits como *opcode* secundário.

4.2 A extensão SPARC16

A resolução do modelo de PLI nos dá um conjunto de formatos, com cada formato abrigando uma ou mais instruções. Entretanto, isso é apenas uma sugestão. Algumas modificações foram feitas de modo a tornar o conjunto de instruções mais regular e, além

disso, instruções especiais foram manualmente inseridas. Os formatos que compõe a extensão SPARC16 são apresentados na subseção 4.2.1. Os requisitos de alinhamento das instruções são introduzidos na subseção 4.2.2. Finalmente, os registradores acessíveis e as instruções de troca de modo da extensão SPARC16 são discutidos nas subseções 4.2.3 e 4.2.4, respectivamente. Uma descrição detalhada das instruções presentes no SPARC16 está disponível no apêndice A.

4.2.1 Formatos

A figura 4.1 apresenta os formatos das instruções de 16 bits do SPARC16. Note que um número grande de formatos foi criado. Apesar de fugir da característica de um conjunto de instruções RISC, essa diversidade garante que as operações que aparecem com mais frequência no código recebam campos maiores para seus imediatos ou que possam operar com três registradores (dois fontes e um destino). Outro aspecto que merece ser destacado é o uso de *opcodes* secundários (opc_2). Apesar de reduzir a quantidade de bits que pode ser usada para codificar um imediato ou forçar uma instrução a operar com apenas dois registradores (um registrador é simultaneamente fonte e destino da operação), isso permite que um conjunto de instruções seja agrupado e gaste apenas um *opcode* primário. Se esse recurso não fosse utilizado, SPARC16 ficaria restrito a apenas 32 instruções (dado os 5 bits utilizados para o *opcode* primário).

No formato I, foram alocadas as instruções de chamada de rotinas *call* e *call with exchange*, sendo que a última deve ser usada caso a rotina alvo esteja codificada com instruções SPARCv8. No formato IB, foram alocadas instruções de salto *branch always*, *branch if equal* e *branch if not equal*. Nesse formato, o bit de número 10 é interpretado como o *annul bit*. É importante destacar que, para economizar espaço, todas as instruções lógicas e aritméticas do SPARC16 alteram os códigos de condição (*condition codes*) da unidade inteira do processador. Esses códigos de condição são avaliados na hora de executar um salto condicional. O formato RI é utilizado por duas instruções: *cmp* e *mov*. A primeira compara o valor de um registrador com o de um imediato (e atualiza os códigos de condição da unidade inteira), enquanto que a segunda move um imediato para um registrador. O formato RRI é utilizado para codificar instruções que precisam de dois registradores e um imediato, como por exemplo instruções de soma, subtração e deslocamento.

Apesar das instruções *load word* e *store word* terem sido alocadas no formato RRI, instruções de carga e armazenamento menos convencionais (como *load unsigned half word*) foram alocadas no formato LW/ST. Esse formato possui um *opcode* secundário de 1 bit, de modo que o imediato só possui 4 bits. Um detalhe que deve ser destacado é que SPARC16 utiliza deslocamento de imediato em *loads/stores*, ou seja, ao carregar uma *double word*,

o imediato é deslocado 3 bits à esquerda (multiplicado por 8), dado que o padrão SPARC requer que *double words* estejam alinhadas em posições da memória múltiplas de 8.

O formato RRR agrupa instruções que utilizam como operandos apenas registradores. As operações realizadas com frequência, como soma e subtração, foram colocadas nesse formato. Destaca-se o fato dessas instruções utilizarem dois registradores como fonte e armazenarem o resultado da operação no terceiro registrador. Operações pouco frequentes foram acomodadas no formato RR, que força um dos registradores a ser fonte e destino simultaneamente.

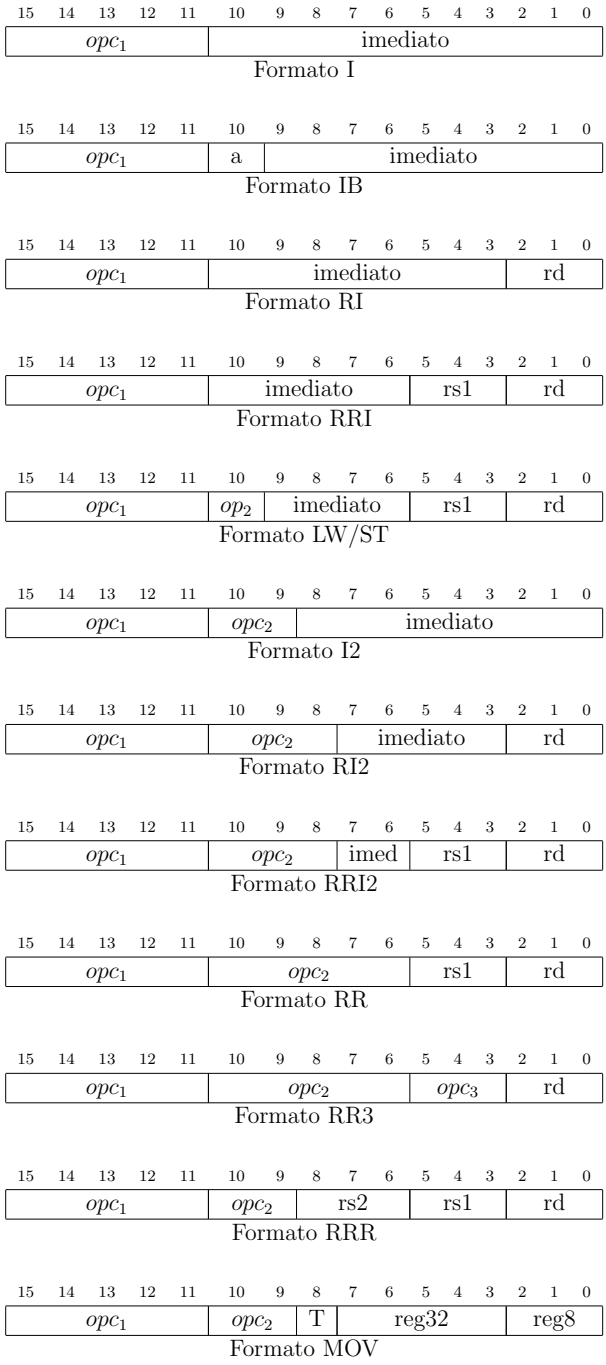


Figura 4.1: Formatos SPARC16

O formato I2, assim como o formato I, possui apenas um imediato. O que o difere é o uso de um *opcode* secundário de 2 bits, que permite que até 4 instruções sejam representadas sacrificando apenas um *opcode* primário. O preço a ser pago é um campo de imediato com 2 bits a menos. Nesse formato estão representadas as instruções *savesp* e a instrução *bx*. A primeira corresponde a uma instrução *save* que utiliza o *stack pointer* implicitamente. A segunda é uma instrução de salto incondicional que troca o modo de execução do processador (o endereço alvo contém código SPARCv8). O formato RI2 também faz uso de um *opcode* secundário e é utilizado para codificar instruções que utilizam o *stack pointer* ou o *frame pointer* de forma implícita — instruções para carregar ou armazenar palavras na memória onde o registrador de endereço base é implícito, além de instruções de soma onde um dos operandos fonte é um dos registradores mencionados. Nesse mesmo formato, codificamos as instruções *btst* e *clearword*. A primeira realiza uma operação E-lógico entre o imediato e o registrador codificados e atualiza os códigos de condição da unidade inteira. A última zera uma posição de memória dada pela soma do registrador com o imediato codificados. O formato RRI2 é similar ao formato RRI, a diferença consistindo num imediato de tamanho reduzido (apenas dois bits), devido ao uso de um *opcode* secundário. Nesse formato, foram alocadas instruções cuja frequência no código é menor, sendo exemplos as instruções de divisão e multiplicação. O formato RR, como já foi mencionado, é utilizado para codificar instruções que operam apenas em 2 registradores. Nesse caso, um dos registradores serve simultaneamente como fonte e destino de uma operação. As instruções de soma e subtração com carry (*addx* e *subx*) são exemplos de instruções codificadas nesse formato.

O formato RR3 é o único a utilizar um *opcode* terciário. Esse *opcode* adicional foi criado com o intuito de aproveitar ao máximo os bits utilizados para codificar as instruções SPARC16. Nesse formato, estão instruções que utilizam apenas um registrador. Como exemplos, podemos citar as instruções *call on register* e *jump on register*, que saltam para o endereço contido no registrador codificado. A diferença entre as duas é que a instrução de *call* escreve o valor do *program counter* no registrador %o7.

Finalmente, há o formato MOV, onde foram alocadas duas instruções capazes de mover os valores de registradores visíveis para a extensão SPARC16 para registradores invisíveis (leia-se acessíveis apenas em modo SPARCv8) e vice-versa. A subseção 4.2.3 traz mais detalhes sobre as restrições impostas ao acesso ao banco de registradores.

Para lidar com o problema de imediatos de tamanho reduzido, SPARC16 adotou uma estratégia similar ao mecanismo de EXTEND do MIPS16. A instrução de EXTEND não faz nada a não ser carregar um pedaço de um imediato ou um registrador extra a serem utilizados pela instrução a ser executada em seguida. Do ponto de vista prático, isso significa que SPARC16 possui instruções de 32 bits. Os formatos de 32 bits são apresentados nas figura 4.2.

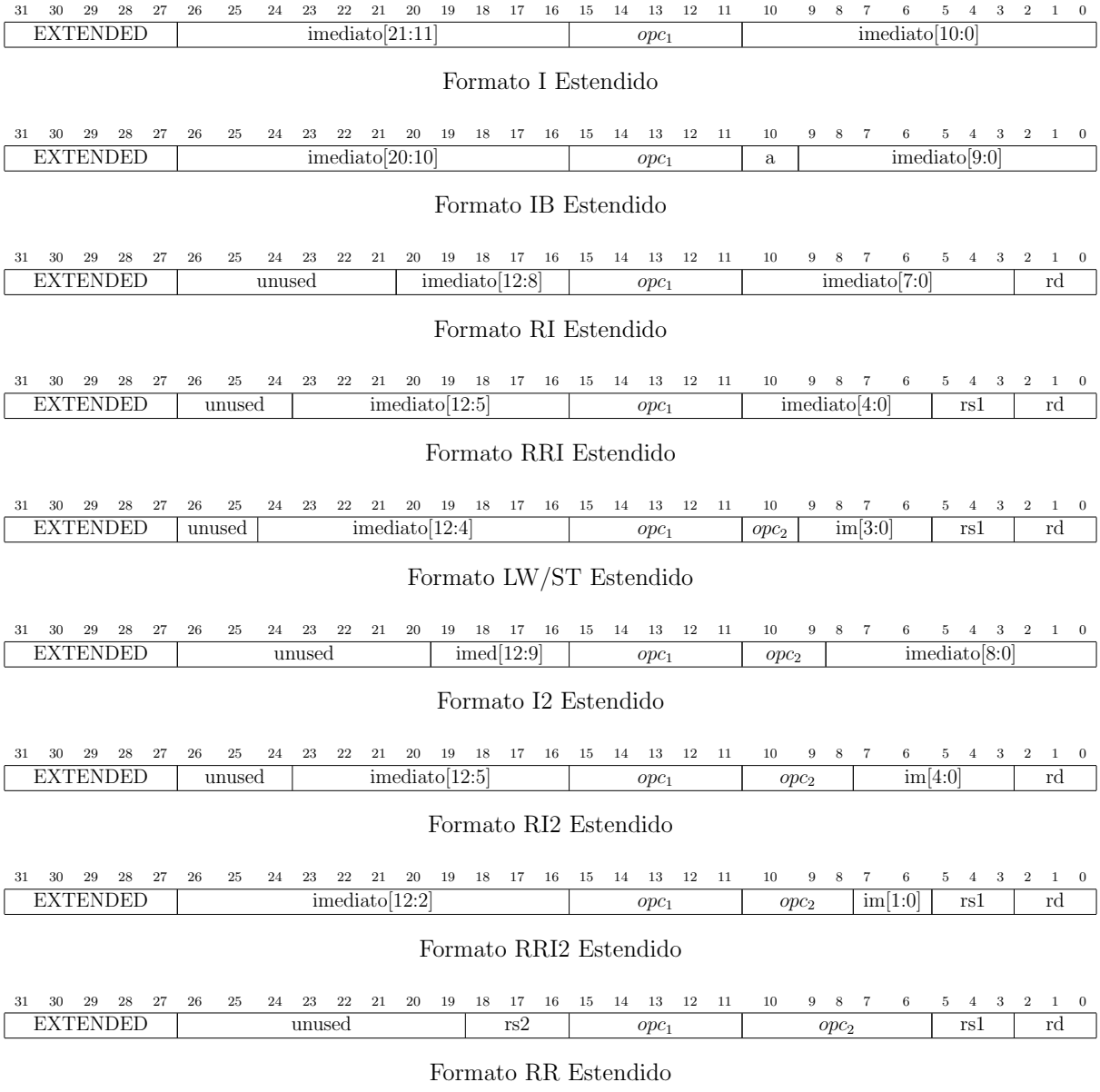


Figura 4.2: Formatos estendidos (32 bits)

Destaca-se que mesmo utilizando o mecanismo de EXTEND, a instrução de *call* do SPARC16 não atinge o mesmo número de bits de sua correspondente em SPARCv8. O *call* estendido em SPARC16 possui apenas 22 bits de imediato, enquanto que o *call* de SPARCv8 possui 30. A mesma afirmação pode ser feita sobre as instruções de *branch*, que quando estendidas em SPARC16 possuem 21 bits de imediato, contra os 22 bits da codificação SPARCv8. Apesar disso parecer uma desvantagem, as análises feitas indicam que os tamanhos de campo utilizados por SPARC16 são suficientes e permitem a geração de código com razões de compressão significativas. Outro ponto que merece ser ressaltado é o uso do mecanismo de EXTEND para especificar um registrador extra. Isso é útil para instruções do formato RR, nas quais um dos registradores precisa funcionar como fonte e destino da operação. O uso da instrução EXTEND permite que um registrador fonte extra seja especificado.

Finalmente, temos a instrução *sethi*, que carrega uma constante de 22 bits nos bits mais significativos de um registrador alvo. Optou-se por criar apenas um formato de 32 bits para essa instrução, tendo em vista que uma instrução de apenas 16 bits seria pouco eficaz, já que apenas 8 bits estariam disponíveis para codificar o imediato (5 seriam gastos com o *opcode* e 3 seriam utilizados para codificar o registrador). Mesmo que o mecanismo de EXTEND, fosse utilizado teríamos apenas 19 bits para compor o imediato. Pelo menos 20 seriam necessários para carregar uma constante em um registrador, dado que os 12 menos significativos seriam fornecidos por uma instrução OR (note que OR possui um imediato sinalizado de 13 bits, logo o bit mais significativo é estendido e precisaria ser 0_2). A figura 4.3 apresenta o formato SETHI. Destaca-se mais uma vez que *sethi* é uma instrução SPARC16 de 32 bits que não faz uso do mecanismo de EXTEND.

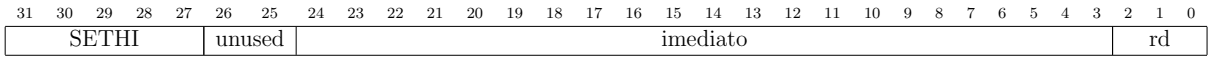


Figura 4.3: Formato SETHI (instrução de 32 bits)

4.2.2 Requisitos de alinhamento das instruções

As instruções SPARC16 precisam estar em endereços alinhados a 2 bytes. Essa regra também se aplica a instruções de 32 bits. Levando em consideração que os acessos à memória precisam ser alinhados a 4 bytes, um caso específico precisa ser observado: uma instrução SPARC16 de 32 bits pode estar em um endereço alinhado a 2 bytes que não seja múltiplo de 4. A figura 4.4 ilustra essa situação.

A instrução *sethi* não está em um endereço alinhado a 4 bytes. Essa situação é resolvida da seguinte maneira: Ao buscar os 2 bytes mais significativos da instrução, um *nop* é injetado no *pipeline*. No ciclo seguinte (ou muitos ciclos depois, no caso de um *cache miss*

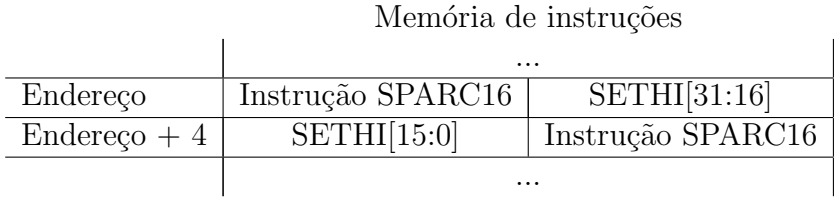


Figura 4.4: Instrução SPARC16 SETHI desalinhada

ter ocorrido), os últimos dois bytes da instrução são buscados na memória e a execução continua normalmente. Para evitar a inserção de *hardware* extra para restauração de estado no caso de ocorrer uma exceção entre o primeiro e o segundo *fetch*, optou-se por desabilitar as interrupções até que a metade menos significativa da instrução seja buscada.

4.2.3 Acesso ao banco de registradores

Como já foi mencionado na seção 3.1, a arquitetura SPARCv8 faz uso de um banco de registradores janelado. A extensão SPARC16 também utiliza esse mecanismo. Entretanto, dos 32 registradores da janela corrente acessíveis para a arquitetura SPARCv8, apenas 8 são visíveis e podem ser referenciados pelas instruções SPARC16. Para amenizar o impacto causado pelo uso de um número reduzido de registradores, algumas instruções SPARC16 referenciam registradores de forma implícita. A tabela 4.3 apresenta os registradores SPARCv8 que são acessíveis à extensão SPARC16.

Tabela 4.3: Registradores da extensão SPARC16

Registrador	**Acessível**	**Descrição**
%i0	Explicitamente	Registrador de entrada
%i1	Explicitamente	Registrador de entrada
%i2	Explicitamente	Registrador de entrada
%o0	Explicitamente	Registrador de saída
%o1	Explicitamente	Registrador de saída
%o2	Explicitamente	Registrador de saída
%l0	Explicitamente	Registrador local
%g1	Explicitamente	Registrador global
%g0	Implicitamente	Contém o valor zero
%fp	Implicitamente	*Frame pointer*
%sp	Implicitamente	*Stack pointer*
%ra	Implicitamente	Contém endereço de retorno

A decisão de quais registradores seriam acessíveis (explícita ou implicitamente) às ins-

truções SPARC16 foi tomada de forma cautelosa visando ser compatível com a convenção de chamadas da arquitetura SPARCv8. Por isso, existem 3 registradores de entrada e 3 registradores de saída que, como mencionado na seção 3.1 são usados para passagem de parâmetros. Obviamente, esses registradores só possuem essa função antes de uma função ser chamada e após ela retornar. Excetuando-se essas duas situações, o programador (ou compilador) é livre para utilizar esses registradores da forma que achar melhor. Os outros dois registradores explicitamente endereçáveis são o %l0 (um registrador local) e %g1 (um registrador global). O uso do registrador local fica a cargo do programador (ou do compilador). O registrador global, por outro lado, precisa ser usado de forma mais inteligente, dado que mesmo com o deslizamento da janela pelos vários registradores do banco, ele permanece sempre visível.

Além dos 8 registradores visíveis, 4 outros são utilizados implicitamente por instruções SPARCv8. A escolha dos registradores %g0, %sp (*stack pointer*), %fp (*frame pointer*) e %ra (*return address*) como alvos implícitos de instruções SPARC16 partiu da observação de padrões sistemáticos de utilização dos mesmos em código SPARCv8. A tabela 4.4 mostra como a observação dos padrões deu origem às instruções SPARC16. A primeira linha da tabela deve ser interpretada da seguinte forma: A pseudo-instrução *cmp %rs1, %rs2* é a instrução SPARCv8 *subcc %rs1, %rs2, %g0*, que subtrai o valor do registrador rs2 do registrador rs1 e descarta o resultado (escreve no registrador %g0). Essa pseudo-instrução deu origem à instrução *cmp* da extensão SPARC16.

Tabela 4.4: Instruções SPARC16 que utilizam registradores de forma implícita

Pseudo-Instrução SPARCv8	**Instrução SPARCv8**	**Instrução SPARC16**
cmp %rs1, %rs2	*subcc %rs1, %rs2, %g0*	*cmp %rs1, %rs2*
mov imm, %rd	*addcc %g0, imm, %rd*	*mov imm, %rd*
ret	*jmpl %i7+8, %g0*	*ret*
retl	*jmpl %o7+8, %g0*	*retl*
Não disponível	*save %sp, imm, %sp*	*savesp imm*
Não disponível	*ld [%sp+imm],%rd*	*ldsp imm, %rd*
Não disponível	*st %rd, [%sp+imm]*	*stsp imm, %rd*
Não disponível	*addcc %sp+imm, %rd*	*addsp imm, %rd*
Não disponível	*ld [%fp+imm],%rd*	*ldfp imm, %rd*
Não disponível	*st %rd, [%fp+imm]*	*stfp imm, %rd*
Não disponível	*addcc %fp+imm, %rd*	*addfp imm, %rd*

Note que os registradores de *frame pointer* e *stack pointer* não são utilizados por pseudo-instruções SPARCv8. Entretanto, instruções de soma e acessos à memória utilizando os mesmos são frequentes, fato que motivou a inclusão de instruções SPARC16 que

fazem uso desses registradores de forma implícita.

Dito isso, é importante destacar que além dos registradores mencionados, SPARC16 permite que registradores invisíveis sejam acessados. Isso é feito através das instruções *MOV8to32* e *MOV32to8*. A primeira move dados de um registrador SPARC16 para um registrador SPARCv8, enquanto que a segunda faz a operação inversa. O formato MOV é usado para representar essas instruções. O campo reg32 (5 bits) é utilizado para indexar o registrador SPARCv8, enquanto que o campo reg8 (3 bits) referencia um registrador SPARC16. O campo T é usado para diferenciar as duas instruções ($T = 0$ representa uma instrução MOV32to8, $T = 1$ representa MOV8to32).

4.2.4 Trocas de modo

Como já foi mencionado, SPARC16 é uma recodificação das instruções SPARCv8. As instruções SPARC16 são convertidas em tempo de execução para suas correspondentes em SPARCv8 e só então prosseguem pelo restante do *pipeline*. A tradução precisa ser feita antes do estágio de *decode* do *pipeline* (ou exatamente no início do mesmo). A figura 4.5 apresenta uma implementação possível.

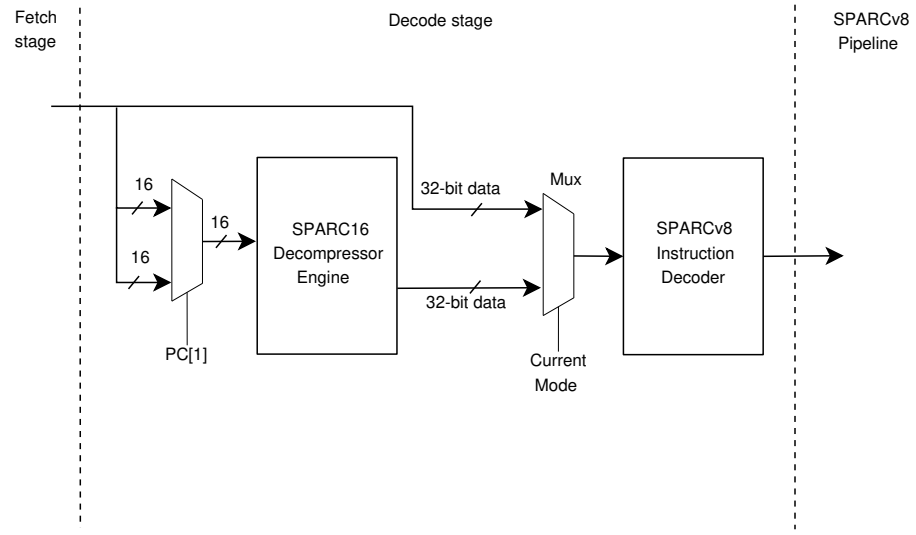


Figura 4.5: Uma possível implementação de SPARC16

Nesse esquema, o descompressor foi posicionado no início do estágio de decodificação do *pipeline*. Note que a instrução SPARCv8 que é efetivamente decodificada é a presente na saída do multiplexador — controlado por um bit chamado de modo corrente. Esse modo não se refere ao modo usuário ou supervisor, mas sim ao modo SPARC16 ou SPARCv8. Ou seja, o *core* que implementa a extensão SPARC16 é capaz de rodar tanto código SPARCv8 como código SPARC16. É importante destacar que as instruções

desses dois modos não podem estar simplesmente intercaladas. As instruções SPARC16 só são corretamente executadas caso o modo corrente seja SPARC16. O mesmo vale para o modo SPARCv8. Para a troca do modo de execução, instruções específicas precisam ser executadas. Essas instruções são apresentadas a seguir:

Saltos de SPARCv8 para SPARC16

Duas instruções SPARCv8 foram criadas para saltar para código SPARC16: *branch with exchange* (figura 4.6) e *jump and link with exchange* (figura 4.7). Repare que as duas estão alocadas no formato 3 da arquitetura SPARC (um *opcode* primário de 2 bits e um *opcode* secundário de 6 bits).

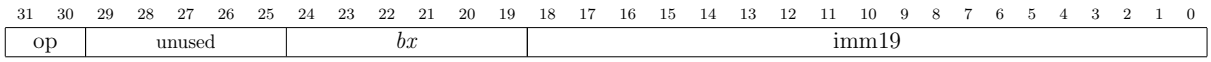


Figura 4.6: Instrução SPARCv8 Branch with Exchange (bx).

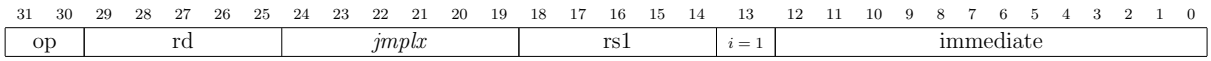


Figura 4.7: Instrução SPARCv8 Jump and Link with Exchange (jmplx)

A instrução *bx* salta para o endereço dado pela soma do *pc* com o imediato deslocado 1 bit à esquerda (uma vez que estamos saltando para código de 16 bits, multiplicamos o imediato por 2) e ativa o modo SPARC16. A instrução *jmplx* também é usada para ativar o modo SPARC16, mas ao contrário de *bx*, o endereço alvo do salto é dado pela soma do registrador rs1 com o imediato. Além disso, *jmplx* armazena o seu próprio endereço no registrador rd. Essa instrução deve ser utilizada para efetuar chamada de procedimentos. Nos dois casos, o *delay slot* precisa ser preenchido com uma instrução SPARCv8.

Como tanto *bx* como *jmplx* são instruções SPARCv8, foi preciso utilizar *opcodes* livres para codificá-las (disponíveis no formato 3 da arquitetura SPARCv8). O opcode alocado para a instrução *bx* é 011001_2, enquanto que a instrução *jmplx* utiliza 001101_2. As duas utilizam $op = 10_2$.

A figura 4.8 traz um exemplo em linguagem de montagem do uso das duas instruções (o símbolo ';' é usado para comentários). Nas linhas 5 e 6 o endereço de uma função codificada em sparc16 (sparc16_code) é carregado no registrador %l0 através das instruções *sethi* e *or*. Na linha 7, a instrução *jmplx* salta para a função sparc16_function (o endereço alvo é dado por $\%l0 + \%g0$) e armazena o endereço de retorno no registrador %o7, de modo que a função sparc16 possa retornar de forma apropriada.

A linha 13 traz um exemplo do uso da instrução sparcv8bx. Basta chamá-la com o *label* do endereço alvo. Obviamente, o alvo precisa estar perto o suficiente (o imediato

da instrução possui 19 bits). Outro detalhe importante é que essa instrução não guarda o endereço de retorno em um registrador.

```
1   sparcv8_code:
2       ...
3
4     ; Chamada de código sparc16 usando jmplx.
5       sethi %hi(sparc16_function), %l0
6       or  %l0, %lo(sparc16_function), %l0
7       jmplx %l0, 0, %o7
8       nop
9
10      ...
11
12    ; Chamada de código sparc16 usando branch with exchange.
13      sparcv8bx sparc16_code
14      nop
15
16      ...
```

Figura 4.8: Exemplo em assembly de chamada de código SPARC16 a partir de código SPARCv8

Saltos de SPARC16 para SPARCv8

Existem 4 instruções SPARC16 de salto que ativam o modo SPARCv8. Elas são apresentadas abaixo.

***callx (call with exchange)*:** Apresentada na figura 4.9. Essa instrução foi alocada no formato I. O endereço do salto é obtido da mesma forma que para *bx*. Entretanto, ao ser executada, essa instrução coloca seu próprio endereço no registrador %o7, devendo, por esse motivo, ser utilizada na chamada de procedimentos.

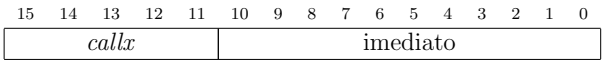


Figura 4.9: Instrução Call with Exchange (callx).

call on register with exchange : Apresentada na figura 4.10. Essa instrução foi alocada no formato RR3. O endereço do salto é dado pelo registrador rd. O endereço de *callregx* é armazenado no registrador %o7, permitindo que essa instrução seja utilizada para chamar procedimentos.

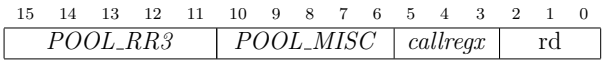


Figura 4.10: Instrução Call on Register with Exchange (callregx)

jump on register with exchange : Apresentada na figura 4.11. Essa instrução também foi alocada no formato RR3. O endereço do salto é dado pelo registrador rd. Ao contrário da instrução *callregx*, não armazena o endereço o de retorno.

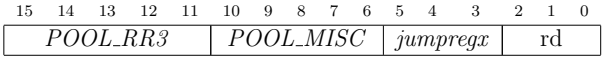


Figura 4.11: Instrução Jump on Register with Exchange (jumpregx)

***bx (branch with exchange)*:** Apresentada na figura 4.12. Essa instrução foi alocada no formato I2. O endereço do salto é obtido somando-se o *pc* ao imediato deslocado 2 bits à esquerda (já que o código SPARCv8 é alinhado a 4 bytes). Os 2 bits menos significativos do *pc* são ignorados.

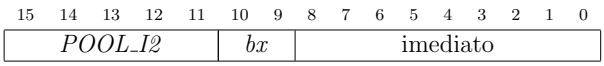


Figura 4.12: Instrução Branch with Exchange (branchx)

Todas essas instruções requerem que o *delay slot* seja preenchido com uma instrução SPARC16. A figura 4.13 mostra um exemplo em assembly do uso dessas instruções. Na linha 3 há um exemplo de uso da instrução *callx*. Basta chamá-la com o *label* do endereço alvo, lembrando que o imediato da instrução possui 10 bits (20 bits caso o mecanismo de EXTEND seja usado). A linha 11 traz um exemplo de uso da instrução *callregx*, que requer que um endereço seja previamente carregado em um registrador (a carga do endereço é feita nas linhas 9 e 10 através das instruções *sethi* e *or* estendido). Finalmente, na linha 17 há uma instrução *bx*.

4.3 Considerações finais e Conclusões

Esse capítulo detalhou a extensão SPARC16. Inicialmente, o modelo de programação linear inteira utilizado para auxiliar na criação dos formatos e instruções SPARC16 foi

```
1     sparc16_code:
2       ; Chamada de código sparcv8 usando callx
3         callx sparcv8_function
4         nop
5
6         ...
7
8       ; Chamada de código sparcv8 usando callregx .
9         sethi %hi(sparcv8_function), %l0
10        eor  %l0, %lo(sparcv8_function), %l0
11        callregx %l0 ; poderia ser jumpregx %l0
12        nop
13
14        ...
15
16      ; Chamada de código sparcv8 usando branch with exchange.
17        bx sparcv8_code
18        nop
19
20        ...
```

Figura 4.13: Exemplo em assembly de chamada de código SPARCv8 a partir de código SPARC16

descrito. Em seguida, os formatos escolhidos foram expostos. Finalmente, as instruções de troca de modo de execução e os registradores visíveis (quando o processador está em modo SPARC16) foram apresentados.

É possível notar que, ao contrário do conjunto de instruções original, a extensão SPARC16 possui um grande número de formatos. Isso é necessário para garantir que as operações mais frequentes, como uma instrução de soma, recebam campos de imediato com um número maior de bits. Para as operações menos frequentes, foram utilizados *opcodes* secundários (que acabam reduzindo o número de *bits* disponíveis para codificar o imediato), permitindo alocar um grande número de instruções em um único *opcode* primário.

Capítulo 5

Implementação e integração do descompressor no Leon 3

Neste capítulo, será apresentado o descompressor projetado para suportar a extensão SPARC16, bem como os detalhes da integração do mesmo no processador Leon 3.

Antes de iniciar o desenvolvimento, os seguintes requisitos foram fixados:

- Inserir o descompressor com o mínimo de alterações no processador original usando o mínimo de *hardware* possível;
- Não degradar o *cycle-time* do processador após a integração;
- Garantir que as instruções SPARC16 sejam alcançáveis através de saltos, mesmo não estando alinhadas em posições de memória múltiplas de 32 bits.

5.1 Processador Leon 3

Esta seção apresenta o Leon 3 [20], o processador utilizado como alvo dos experimentos para validar a extensão SPARC16. O Leon 3 é um modelo VHDL de um processador totalmente compatível com a arquitetura SPARCv8, ideal para uso em *SoCs* (*System-on-a-chip*). A figura 5.1 mostra um exemplo de um *SoC* projetado utilizando o Leon 3.

Dentre as características desse processador estão:

- Unidade inteira com 7 estágios de *pipeline*;
- *Cache* de instruções e dados separadas (Arquitetura Harvard). As caches são configuráveis e podem ter até 4 vias (com tamanho de 1kb a 256kb) e utilizar diferentes

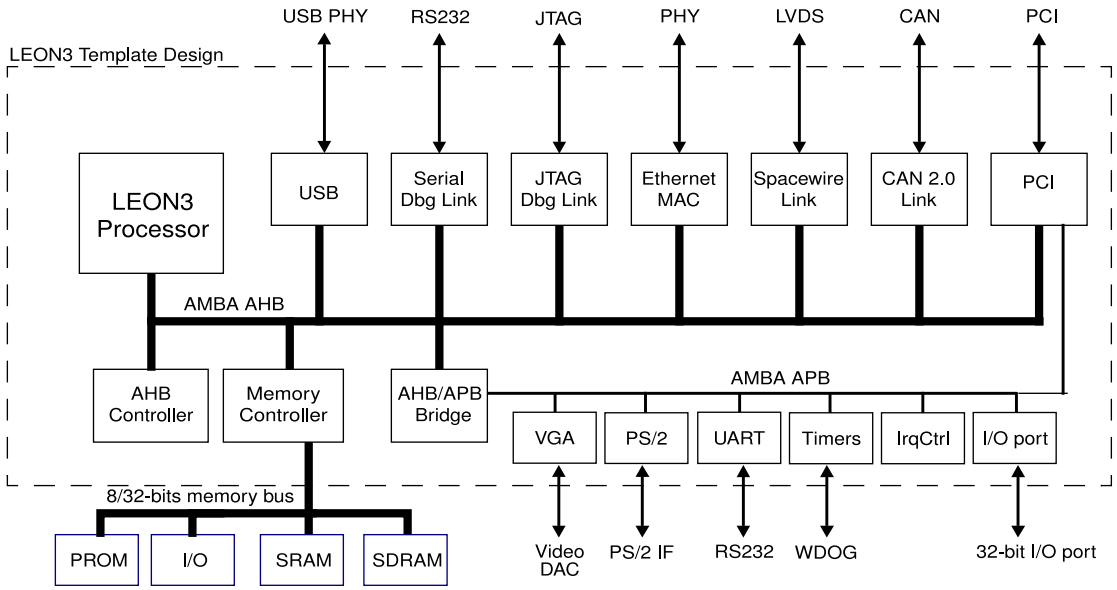


Figura 5.1: Diagrama do processador Leon 3. Fonte: [48]

algoritmos de substituição (LRU[1], LRR[2] ou aleatório);

- Suporta de 2 a 32 janelas de registradores;
- Suporte a memória virtual (*Memory Management Unit* - MMU);
- Multiplicação e divisão implementadas em *hardware* através de instruções *umul*, *smul*, *udiv* e *sdiv*.
- Diversos periféricos implementados on-chip: *UARTs*, controlador ethernet, controlador vga, interface PCI etc;
- Controlador de interrupções;
- Controlador de memória para SRAM, SDRAM, PROM e I/O;
- Unidade de depuração implementada em *hardware*: *DSU - Debug Support Unit*;
- Utiliza o barramento AMBA. Destaca-se que AMBA [41] é dominante no mercado (processadores ARM), bem documentado e não possui restrições de licenças.

As principais razões que motivaram a escolha desse processador foram:

[1]Do inglês, *Least Recently Used.*
[2]Do inglês, *Least Recently Replaced.*

RISC: Como já dito, processadores RISC atendem as necessidades de complexidade dos sistemas embarcados atuais. Além disso, os processadores RISC são conhecidos por oferecerem um conjunto de instruções regular (as instruções têm 32 bits, independente da complexidade da instrução). Com isso, a densidade de código se torna bastante baixa se comparada com os processadores CISC e DSP. Desta forma, o impacto no uso de técnicas de compressão será ainda maior.

SPARC Versão 8: O Leon é certificado pela organização SPARC, como sendo um processador totalmente compatível com a arquitetura SPARC versão 8. Para obter a certificação, o processador passou por uma bateria de testes de compatibilidade e tolerância a erros. Sendo assim, a implementação final deste trabalho vai oferecer um processador bastante confiável, com pequena possibilidade de ter erros.

Código aberto: O código completo do Leon 3 é aberto, podendo ser obtido gratuitamente tanto para fins acadêmicos quanto comerciais, segundo os termos da licença GNU LGPL. O código está escrito em VHDL, uma linguagem de alto nível para descrição de *hardware*.

Potencial para o mercado de sistemas dedicados: A ausência do pagamento de *royalties* fez com que crescesse o uso do Leon em sistemas dedicados. O financiamento pela Agência Espacial Europeia serve como forte fomento ao desenvolvimento do processador. Um modelo de compressão de código que aumente o desempenho e reduza o consumo de energia pode melhorar mais ainda a aceitação no mercado.

5.1.1 Pipeline da unidade de inteiros do Leon 3

Essa subseção apresenta o *pipeline* da unidade inteira do Leon 3. Entender em detalhes a função executada por cada um dos estágios é importante para compreender as modificações realizadas para dar suporte à extensão SPARC16. Um diagrama do *pipeline* é mostrado na figura 5.2.

A descrição dos estágios é dada a seguir:

Fetch (FE): Se a *cache* de instruções está habilitada, a instrução é buscada nela. Caso contrário (ou caso ocorra um *cache miss*), a busca é redirecionada para o controlador de memória. A instrução é válida ao final do ciclo e é armazenada dentro da unidade de inteiros.

Decode (DE): A instrução é decodificada e o endereço alvo de instruções *call* e de *branches* é calculado. Nesse estágio também é verificado se é necessário travar o *pipeline* devido a algum *hazard* de dados.

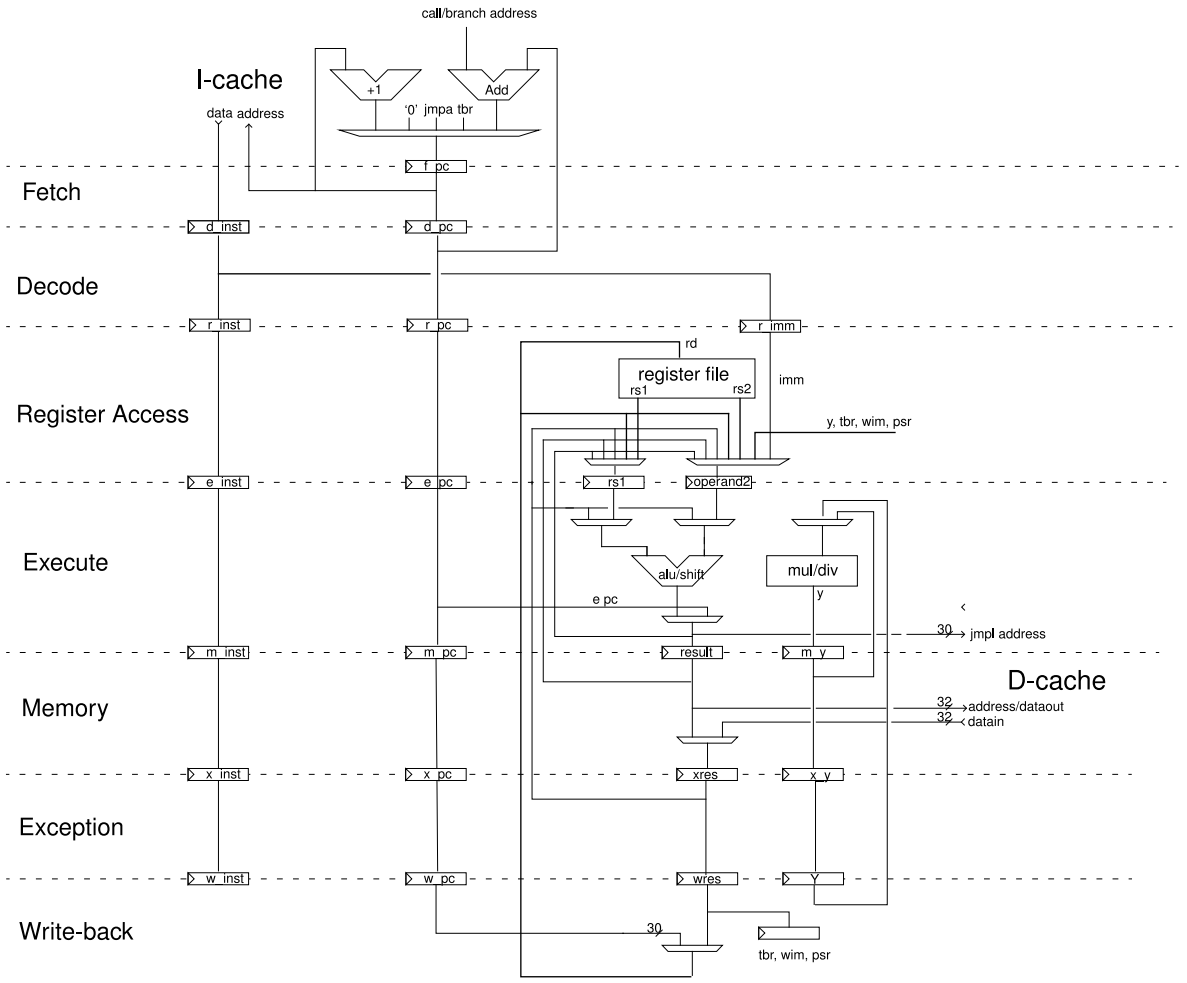


Figura 5.2: Diagrama do pipeline do processador Leon 3. Fonte: [48]

Register Access (RA): Os operandos da instrução são lidos do banco de registradores ou através de *bypasses*.

Execute (EX): Operações lógicas, aritméticas ou de deslocamento são executadas. Para operações de memória ou de transferência indireta de controle (*jmpl* ou *rett*), o endereço alvo é calculado.

Memory Access (ME): A *cache* de dados é acessada, no caso de instruções que carregam ou armazenam dados.

Exception Detect (XC): *Traps* e interrupções são tratados nesse estágio.

Write-back (WB): O resultado de qualquer operação aritmética, lógica, de deslocamento ou o dado lido da memória é escrito no banco de registradores.

5.1.2 Mecanismo de *bursting* da *cache*

Essa seção apresenta o mecanismo de *bursting* presente no sistema de memória do Leon 3. Um entendimento do mesmo é importante para compreender as alterações feitas durante a integração do descompressor projetado.

Quando ocorre uma falha de acesso na *cache* de instruções, o estado do controlador muda de *hit* para *miss*. Nesse estado, o controlador busca sequencialmente as instruções na memória externa até que uma linha da *cache* seja preenchida. Ao mesmo tempo que essas instruções são devolvidas pela memória externa, elas são encaminhadas ao processador. Esse mecanismo — que envia as instruções ao processador diretamente da memória externa — é chamado de *bursting*. A figura 5.3 ilustra o funcionamento do mesmo. Na situação apresentada, ocorre uma falha quando o processador tenta acessar uma instrução no endereço 0x40000014 e, posteriormente, são buscadas 3 instruções SPARCv8 na memória externa — armazenadas nas posições 5, 6 e 7 da linha da *cache*. Ao mesmo tempo, as instruções são enviadas ao processador de modo que a execução prossegue sem que novas bolhas sejam inseridas no *pipeline*.

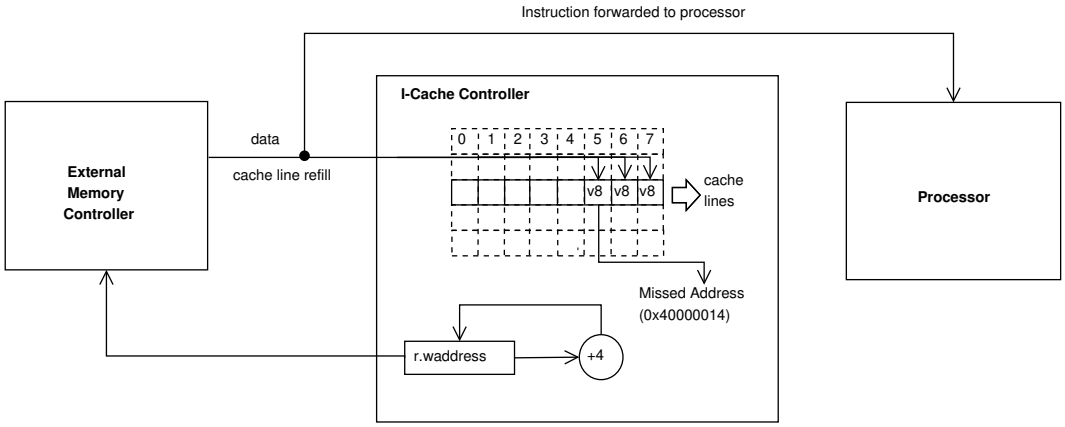


Figura 5.3: Leon 3 em em modo *burst*.

Um detalhe importante sobre o Leon 3 diz respeito a como instruções de salto que ocorrem enquanto o processador está em modo *burst* são tratadas. Nessas situações, o controlador continua solicitando instruções à memória externa (até encher uma linha da *cache*), mas para de encaminhá-las ao processador depois que a instrução de salto e a instrução que ocupa o *delay slot* são executadas.

5.2 Projeto do descompressor

Conforme explicado no capítulo anterior, cada instrução SPARC16 é traduzida em uma única instrução SPARCv8. Portanto, o processo de tradução é, basicamente, combinacional. Apesar disso, devido ao uso do mecanismo de EXTEND e da instrução *sethi* de 32 bits, foi necessário incluir um elemento sequencial (estado) no projeto do descompressor. A figura 5.4 apresenta um diagrama do mesmo.

Observe que o descompressor possui 4 entradas (bit_stream, current_pc[1], current_mode e write_enable) e 3 saídas (v8_insn, *trigger_switch_to_v8* e *trigger_switch_to_16*). As entradas de *clock* e *reset* foram omitidas para não poluir o diagrama, mas são utilizadas pela parte sequencial do circuito. As saídas *trigger_switch_to_v8* e *trigger_switch_to_16* são utilizadas para que seja possível alternar o modo do processador (de SPARCv8 para SPARC16 e vice-versa) e são explicadas em detalhe na seção 5.3.2. As entradas estão localizadas no lado esquerdo da figura e as saídas do lado direito.

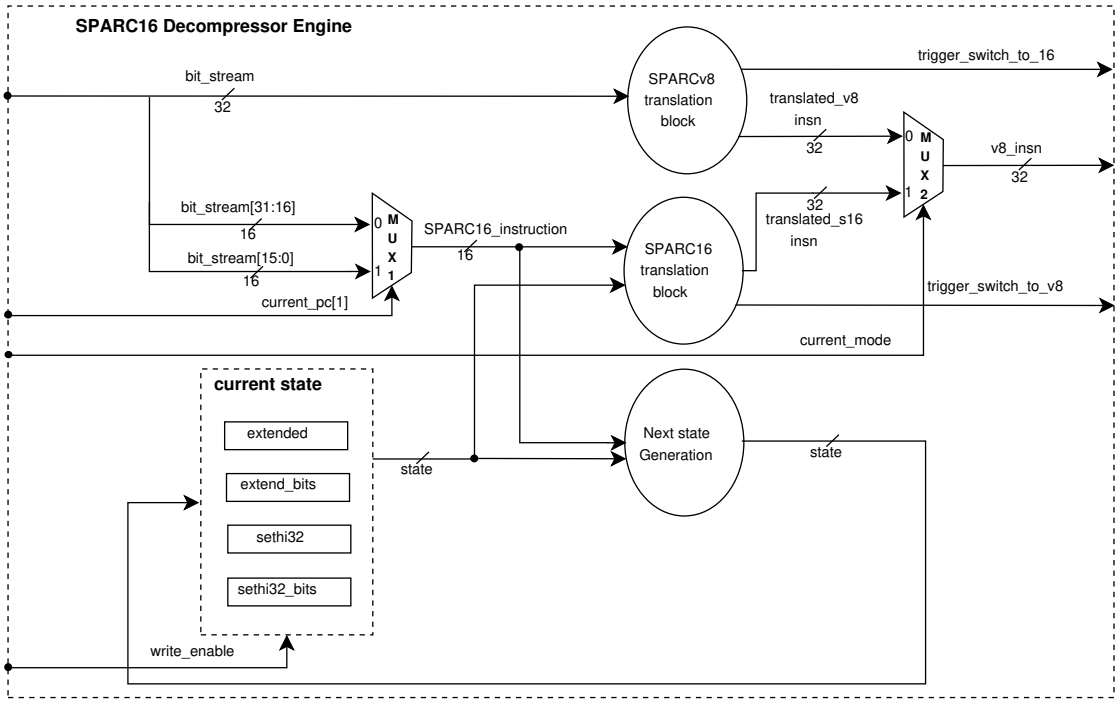


Figura 5.4: Diagrama do descompressor que dá suporte à extensão SPARC16

As 3 elipses representam os 3 blocos puramente combinacionais do descompressor: *SPARCv8 translation block, SPARC16 translation block* e *next state generation.*

O bloco *SPARCv8 translation block* gera como saídas o sinal *trigger_switch_to_16* e o sinal *translated_v8_insn* (a entrada 0 do MUX2). Nesse bloco, as instruções de salto de código SPARCv8 para código SPARC16 *bx* e *jmplx* são convertidas, respectivamente,

para um *branch always* e um *jmpl* com o bit menos significativo do imediato no nível lógico '1'. As instruções SPARCv8 originais simplesmente passam pelo bloco sem sofrer qualquer tipo de processamento.

O bloco *SPARC16 translation block* gera como saídas o sinal *trigger_switch_to_v8* e o sinal *translated_s16_insn* (a entrada 1 do MUX2). Esse bloco traduz a instrução SPARC16 (sinal *SPARC16_instruction*). A tradução é simples e consiste em extender campos de imediato, traduzir os registradores e converter *opcodes*. O estado corrente é utilizado para implementar as instruções que fazem uso do mecanismo de EXTEND e também a instrução *sethi*. Ele é composto por quatro sinais, descritos na tabela 5.1.

Sinal	**Descrição**
extended	Indica que os 16 bits processados no ciclo anterior eram uma instrução EXTEND.
extend_bits	Os bits mais significativos do imediato da instrução estendida. Só são relevantes caso $extended = 1$.
*sethi*32	Indica que os 16 bits processados no ciclo anterior eram os primeiros dois *bytes* de uma instrução *sethi*.
*sethi*32*_bits*	Os 9 bits mais significativos do imediato da instrução *sethi*. Só é relevante caso $sethi32 = 1$.

Tabela 5.1: Sinais que compõe o estado corrente do descompressor

Por último, o bloco *next state generation* é responsável pela geração do próximo estado. Ele recebe como entrada a instrução SPARC16 e o estado corrente. A saída é gerada de acordo com o fluxograma da figura 5.5.

Como pode ser observado na figura, caso o descompressor esteja processando a segunda parte de uma instrução SPARC16 de 32 bits (ou uma instrução SPARC16 de 16 bits), então o estado é resetado. Se, por outro lado, o descompressor estiver lidando com uma instrução estendida ou com os primeiros 16 bits de uma instrução *sethi*, os bits do imediato são devidamente carregados e os sinais *extended* e *sethi32* são atribuídos de forma apropriada.

É importante observar que o próximo estado só é armazenado no estado corrente se o sinal *write_enable* estiver no nível lógico alto. Isso é necessário porque instruções SPARC16 de 32 bits podem não estar alinhadas em endereços múltiplos de 4 na memória, de modo que é possível que ocorra um *cache miss* entre o *fetch* da primeira e da segunda parte da instrução, o que causaria uma perda do estado caso o estado atual fosse escrito a cada subida do *clock*. Mais detalhes são dados na seção 5.3, que trata sobre a integração do descompressor no Leon 3.

Finalmente, dois multiplexadores fazem parte do descompressor:

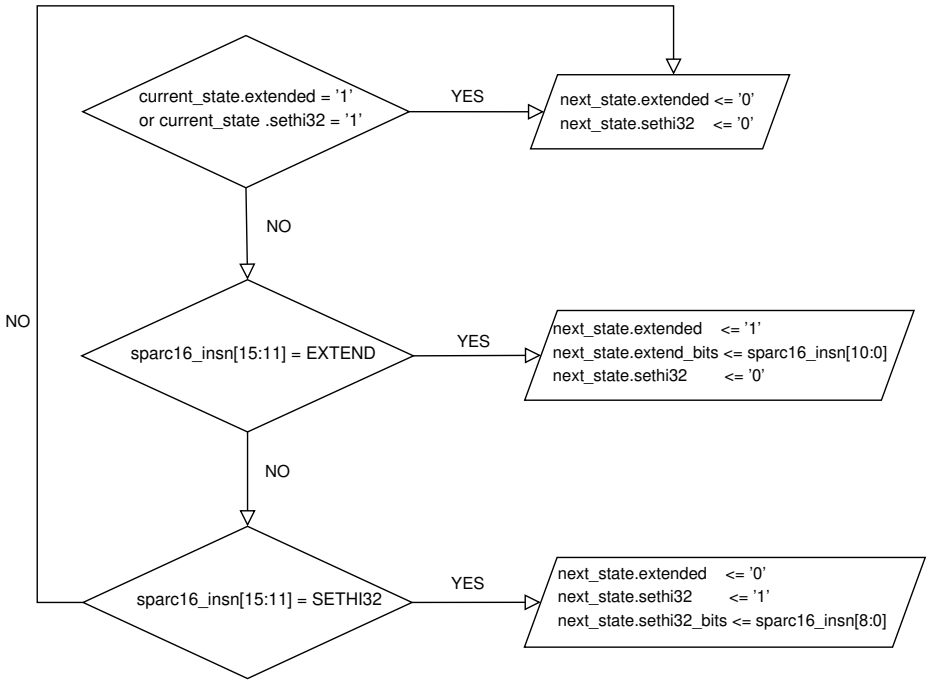


Figura 5.5: Fluxograma do bloco responsável pela geração do próximo estado do descompressor

MUX1: Pretendemos integrar o descompressor no processador Leon 3, que se comunica com a memória através de um barramento de 32 bits. Logo, quando código SPARC16 estiver sendo executado, precisamos de um multiplexador para selecionar instruções de 2 bytes dentro de uma palavra de 4 bytes. O multiplexador é controlado pelo bit de número 1 do *pc* (zero seleciona os dois bytes mais significativos e um os dois menos significativos). A saída desse multiplexador alimenta o bloco *SPARC16 translation block*.

MUX2: Controlado pelo modo corrente. A saída desse multiplexador é o sinal v8_insn (a instrução SPARCv8 que seguirá adiante no *pipeline*). Se o modo corrente for SPARCv8, então a saída do bloco *SPARCv8 translation block* é selecionada. Caso contrário, estamos executando código SPARC16 e a saída do bloco *SPARC16 translation block* é selecionada.

5.2.1 Instruções SPARC16 de 32 bits

Existem duas classes de instruções SPARC16 que possuem 32 bits. São elas a instrução SETHI e as instruções estendidas. Uma instrução estendida é, na verdade, uma instrução EXTEND seguida por uma instrução SPARC16, ou seja, ela possui dois *opcodes* (vide

figura 4.2). Entretanto, as duas classes são processadas de forma similar. O descompressor leva 2 ciclos para descomprimí-las:

Primeiro ciclo: A parte mais significativa do imediato é carregada para ser usada no próximo ciclo. Um NOP é injetado no *pipeline*.

Segundo ciclo: O descompressor sabe, através do estado, que está processando a segunda parte de uma instrução de 32 bits. A instrução é decodificada e injetada no *pipeline*.

Dois exemplos de processamento de instruções de 32 bits são mostrados a seguir. A figura 5.6 traz um trecho de código SPARC16 (posicionado a partir do endereço 0x40000000) que desliza a janela de registradores e decrementa o *stack pointer* (linha 2), além de carregar uma constante no registrador %*l*0 através das instruções *sethi32* (linha 4) e EXTENDED *or* (linha 5).

A instrução *savesp* possui 16 bits e é processada em um ciclo. Em seguida, a instrução *sethi* (no endereço 0x40000002) é descomprimida em 2 ciclos.

```
1    sparc16_function: ; @ 0x40000000
2      savesp  -96
3
4      sethi32 %hi(.LLC0), %l0
5      eor16   %l0, %lo(.LLC0), %l0
6
7      mov 25, %o0
8
9      ...
```

Figura 5.6: Carga de constante utilizando instruções SPARC16

A figura 5.7 mostra uma instrução *sethi* com o imediato dividido em duas partes (imediato[22:13] e imediato[12:0]). A primeira parte (que contém os *bits* mais significativos) é carregada durante o primeiro ciclo de processamento. A segunda parte é concatenada com a primeira durante o segundo ciclo, formando a instrução SETHI codificada em SPARCv8.

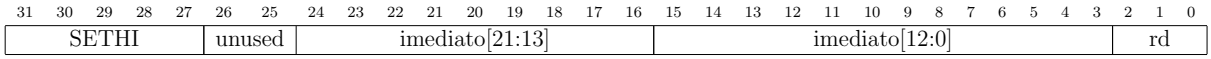
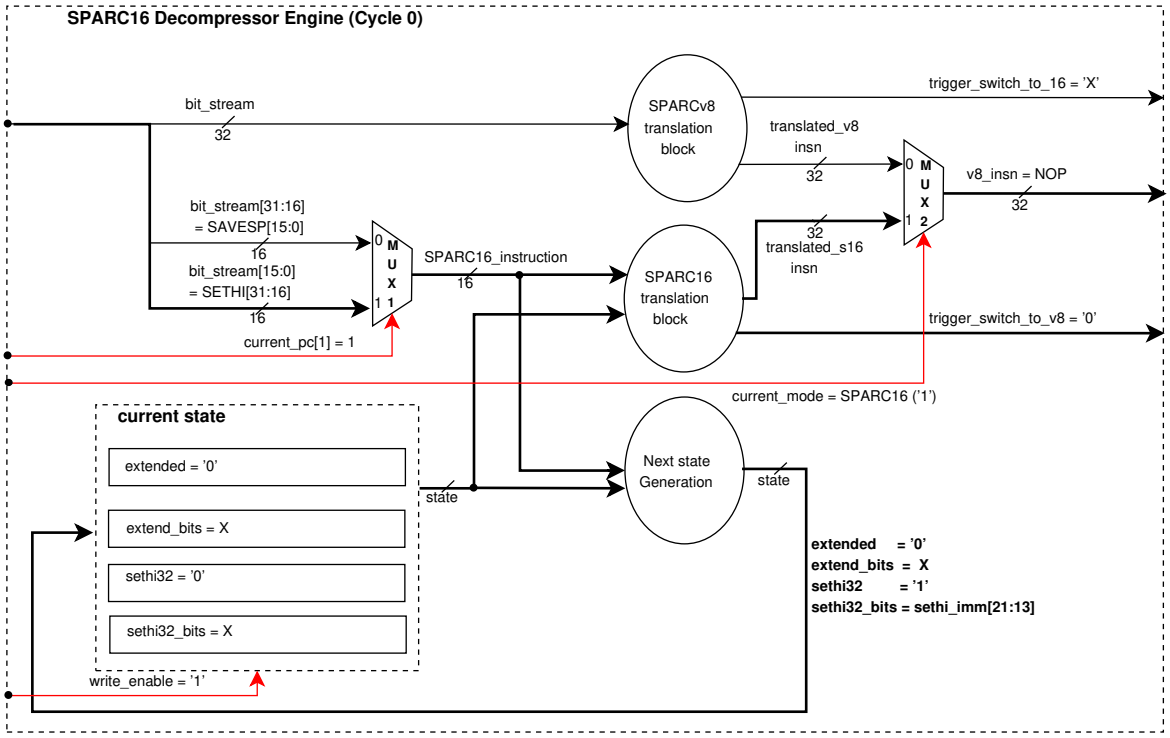


Figura 5.7: Instrução SETHI (32 bits)

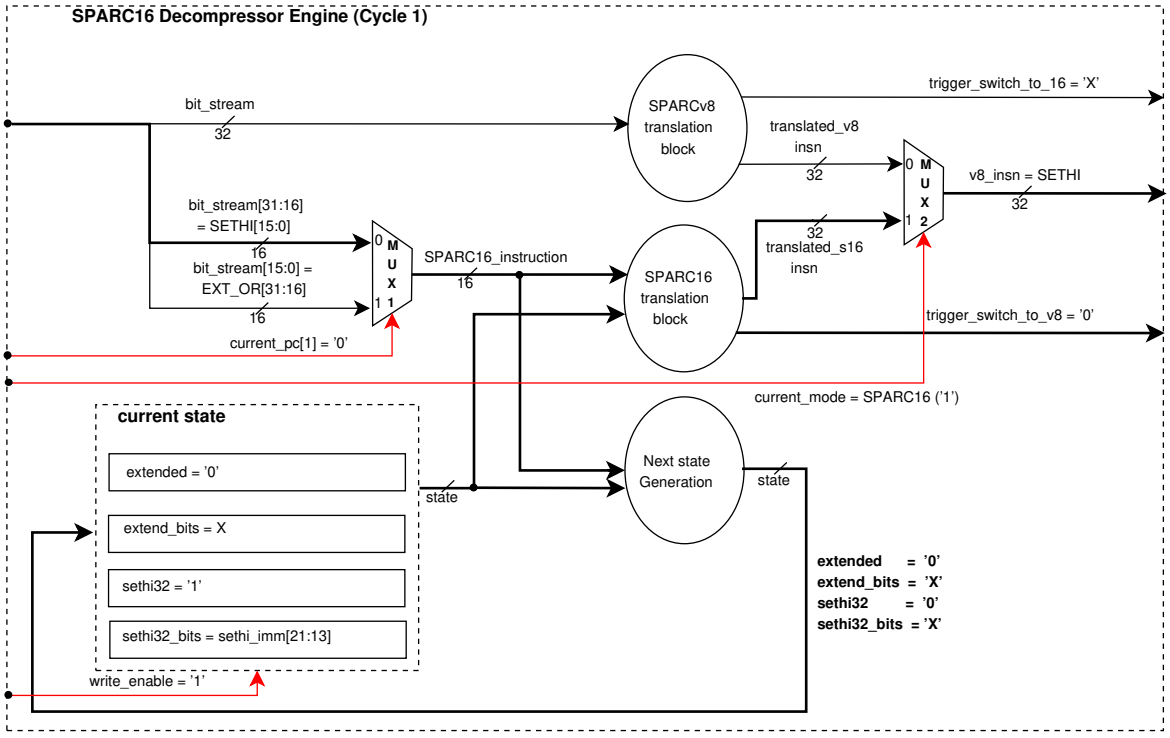
As figuras 5.8(a) e 5.8(b) ilustram o processamento da instrução no primeiro e no segundo ciclo, respectivamente. Os sinais *current_pc*, *current_mode* e *write_enable* são sinais de controle. As linhas em negrito representam o caminho percorrido pela instrução SPARC16 (e também pelos bits que representam o estado corrente) enquanto a mesma se propaga pelo descompressor.

No primeiro ciclo, o descompressor opera com os 16 bits mais significativos da instrução SETHI (endereço 0x40000002). O bloco *SPARC16 translation block* coloca um NOP em sua saída (o sinal *trigger_switch_to_v8* também é zero). O bloco *next state generation* detecta que trata-se de uma instrução SETHI de 32 bits e gera o próximo estado de acordo (*sethi32='1', sethi32_bits=sethi_imm[22:13], extended='0' e extend_bits='X'*). O 'X' significa *don't care*. Como o sinal *state_write_enable* é '1', o estado corrente será atualizado na próxima borda de subida do *clock*.

No segundo ciclo, o descompressor processa os 16 bits menos significativos da instrução SETHI (endereço 0x40000004). O estado corrente reflete a execução da primeira parte do SETHI no ciclo anterior. O bloco *SPARC16 translation block* sabe (devido ao estado) que está processando a segunda parte de uma instrução SETHI. A saída é uma instrução SETHI codificada em SPARCv8. O bloco *next state generation*, por sua vez, "zera" os sinais *sethi32* e *extended*.



(a) Primeiro ciclo



(b) Segundo ciclo

Figura 5.8: Processamento de uma instrução SETHI de 32 bits no descompressor SPARC16

Continuando o exemplo, uma instrução *or* estendida é executada depois do *sethi*. A figura 5.9 mostra a instrução com o imediato dividido em duas partes (imediato[12:2] e imediato[1:0]), determinando a forma como o mesmo é processado pelo descompressor.

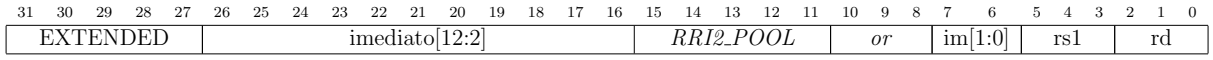
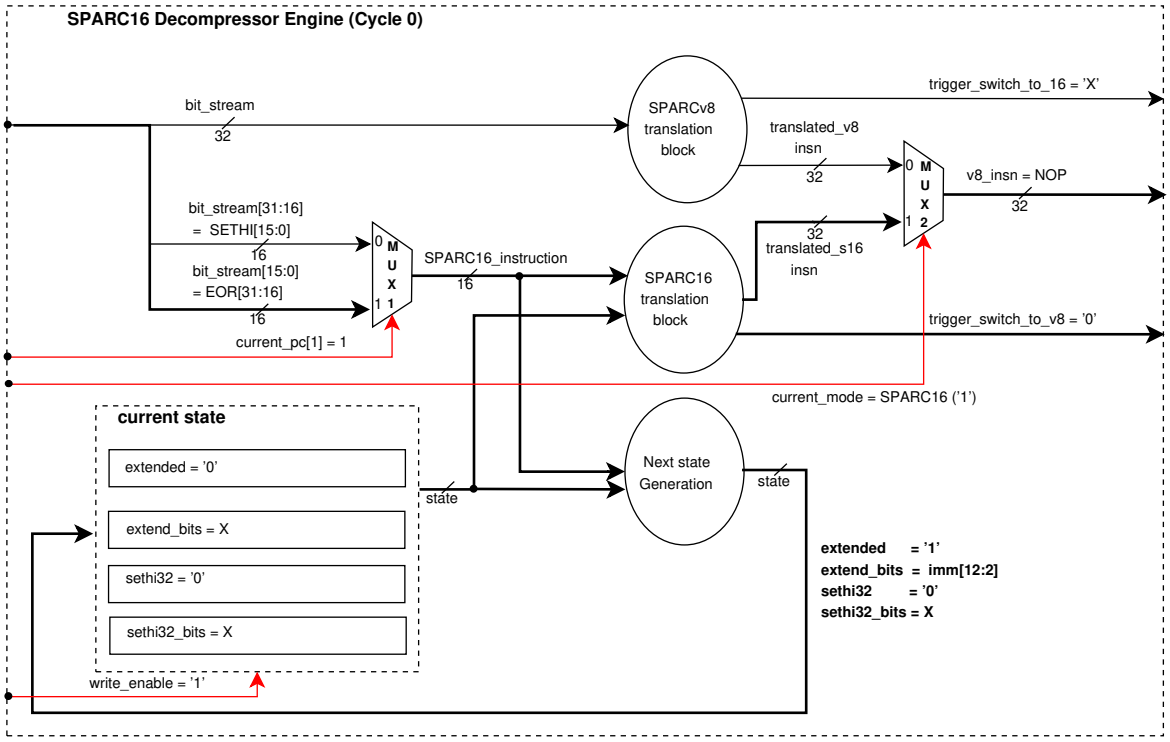


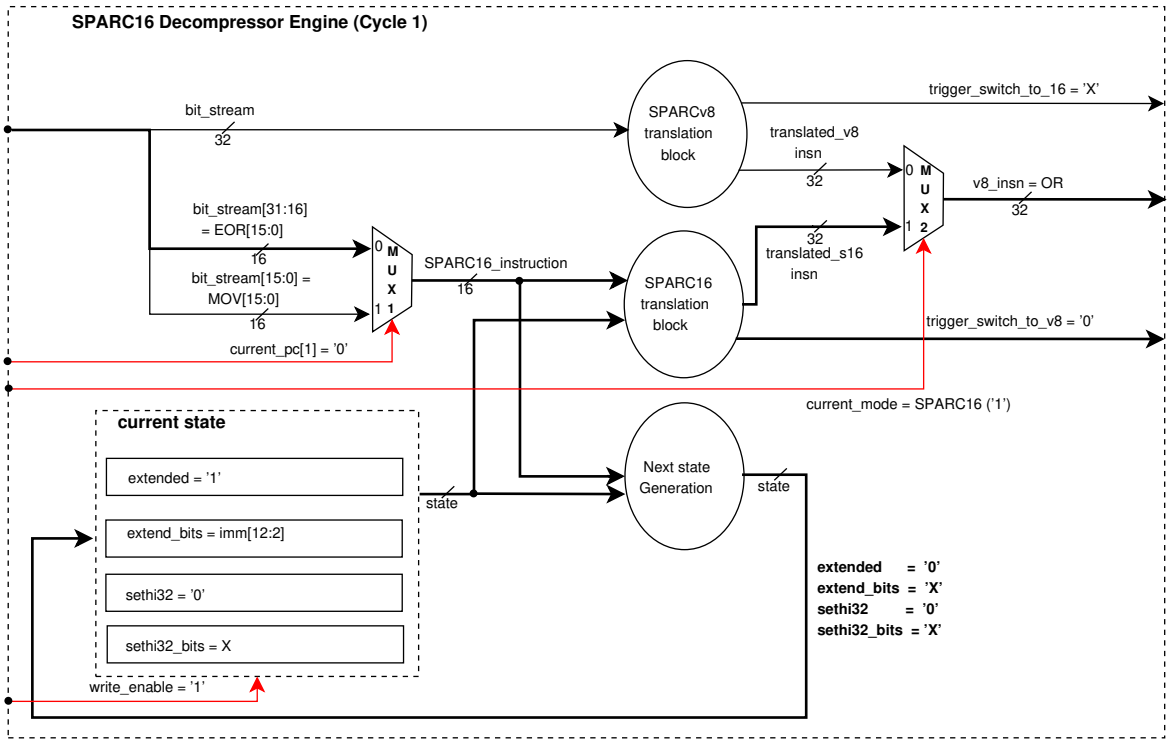
Figura 5.9: Instrução *or* Estendida

As figuras 5.10(a) e 5.10(b) ilustram o processamento da instrução no primeiro e no segundo ciclo, respectivamente. No primeiro ciclo, o descompressor processa a instrução EXTEND (endereço 0x40000006). O bloco *SPARC16 translation block* coloca um NOP em sua saída (o sinal *trigger_switch_to_v8* também é zero). O bloco *next state generation* detecta que trata-se de uma instrução estendida e gera o próximo estado de acordo (*sethi32='0', sethi32_bits='X', extended='1' e extend_bits=imm[12:2]*).

No segundo ciclo, o descompressor processa a a instrução *or* (endereço 0x40000008). Através do estado, o descompressor sabe que está processando uma instrução estendida. Os bits de imediato carregados no ciclo anterior são concatenados aos bits presentes na *half word* sendo processada para formar a instrução *or*. O bloco *next state generation*, por sua vez, "zera" os sinais *sethi32* e *extended*.



(a) Primeiro ciclo



(b) Segundo ciclo

Figura 5.10: Processamento de uma instrução estendida OR no descompressor SPARC16

5.3 Integração do descompressor

Para integrar o descompressor no processador Leon 3, três abordagens foram cogitadas. Conforme pode ser observado na figura 5.11, na abordagem 1, o descompressor é posicionado no final do estágio de *fetch* do pipeline, enquanto que na abordagem 3 o descompressor é posicionado no início do estágio de *decode*. Uma abordagem híbrida (2), com a lógica de descompressão espalhada entre os estágio de *fetch* e *decode* também seria possível.

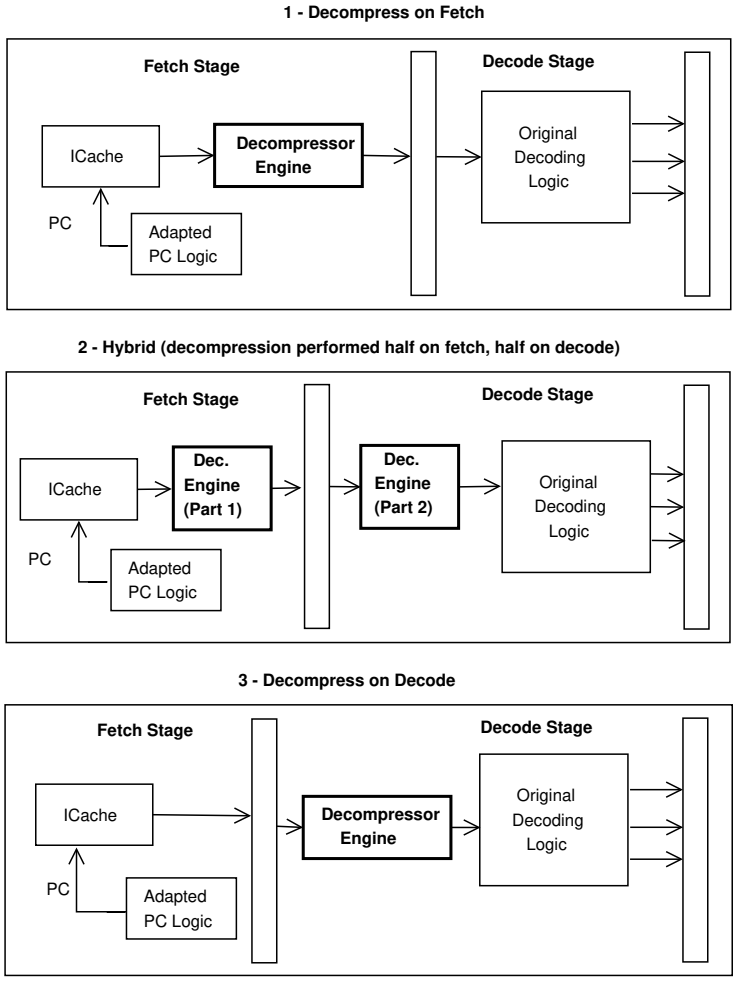


Figura 5.11: Opções de posicionamento do descompressor no *pipeline*.

Entretanto, verificou-se que o caminho crítico do processador estava no estágio de *decode* (mais precisamente, no cálculo do endereço alvo de instruções de salto). Portanto, a primeira alternativa foi a escolhida e, a partir de agora, só serão abordados, no texto, aspectos relacionados a esse estilo de implementação (descompressor no estágio de *fetch*).

Além disso, é preciso destacar que devido às diferenças entre a extensão SPARC16 e o conjunto de instruções original, a simples integração do descompressor em um estágio do *pipeline* não foi suficiente. Outras mudanças precisaram ser feitas na unidade de inteiros e no controlador de *cache*. São elas:

- Adaptação na lógica que calcula o *program counter*.
- Inclusão de suporte à troca de modos.
- Adaptação na lógica que calcula o endereço alvo de instruções *branch* e *call*.
- Correção do cálculo feito para obter o endereço de retorno de uma função para sua chamadora (codificada em modo diferente).
- Alteração no mecanismo de *bursting* da *cache* para suportar a execução de código SPARC16.
- Alteração no mecanismo de tratamento de interrupções para lidar com instruções SPARC16 de 4 bytes.

Essas alterações são discutidas em detalhe nas próximas seções.

5.3.1 Alteração na lógica do *program counter*

Para suportar a extensão SPARC16, a implementação do Leon 3 foi modificada para utilizar os 2 bits menos significativos do *pc*. O bit de número 1 é necessário porque as instruções SPARC16 podem estar localizadas em endereços pares que não sejam múltiplos de quatro.

O bit 0, por sua vez, é utilizado para armazenar o modo corrente de execução. Se ele possuir como valor o nível lógico 0, o descompressor interpreta as instruções como código SPARCv8 (e, portanto, o *pc* é incrementado de 4). Caso contrário, significa que código SPARC16 está sendo executado (e o *pc* é incrementado de 2). A figura 5.12 apresenta a lógica para cálculo do *pc* modificada (o MUX 2 foi adicionado). Note que além do *pc* incrementado, outros 4 endereços podem ser selecionados como próximo *program counter*: o alvo de um *branch*/*call*, o endereço contido no registrador *TBR*, o endereço de uma instrução de *jmpl* e o endereço de *reset*.

Armazenar o modo corrente de execução no bit menos significativo do *pc* foi extremamente útil para lidar com o caso de uma função chamando outra codificada usando instruções de um modo diferente. Por exemplo, suponha a seguinte situação: A função A16 (codificada em SPARC16) chama a função BV8 (codificada em SPARCv8). A função BV8 realiza o seu processamento (que inclui chamadas a outras funções) e, ao retornar,

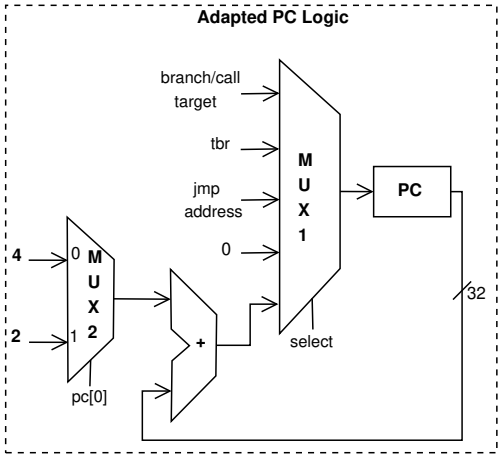


Figura 5.12: Adaptação da lógica que calcula o *program counter*

o endereço de retorno contém o modo de execução da função chamadora no seu bit menos significativo. Isso permite que a função A16 possa ser chamada tanto por código SPARCv8 quanto por código SPARC16. Da mesma forma, uma função SPARCv8 pode ser chamada por uma função SPARC16.

5.3.2 Inclusão de suporte à troca de modos de execução

Conforme explicado na seção 4.2.4, a troca do modo de execução do processador é feita através de instruções de salto. Para facilitar o entendimento de como elas foram implementadas no Leon 3, é útil separá-las em dois grupos: salto indireto, no qual o destino do salto é dado por um registrador ou pela soma de um registrador com um imediato, e salto direto, no qual o endereço alvo é dado pela soma do *program counter* com um imediato codificado na instrução.

Instruções de salto indireto com troca de modo

Como o bit menos significativo do *program counter* é utilizado para armazenar o modo corrente, essas instruções não exigiram mudança na unidade de inteiros. Isso porque o descompressor, ao traduzi-las, garante que o endereço alvo do salto seja ímpar caso estejamos saltando de código SPARCv8 para código SPARC16 e par caso contrário.

Por exemplo, na figura 5.13 temos um trecho de código SPARCv8 que faz uso da instrução *jmplx*. Primeiro, o endereço de uma função SPARC16 (0x40001200) é carregado no registrador $\%l0$ e depois a instrução *jmplx* é executada. O que o descompressor faz é mudar o bit menos significativo do imediato para 1, fazendo com que o processador salte para o endereço 0x40001201 ao invés de 0x40001200.

```
1   sparcv8_code:
2       ...
3
4     ; Chamada de código sparc16 usando jmplx.
5       sethi %hi(sparc16_function), %l0
6       or  %l0, %lo(sparc16_function), %l0
7       jmplx %l0, 0, %o7
8       nop
9
10    ; Alocada no endereço 0x40001200
11  sparc16_function:
12      ...
```

Figura 5.13: Instrução SPARCv8 de salto indireto com troca de modo

É importante notar que o mesmo (mudar o bit menos significativo do imediato para 1) precisa ser feito nas instruções de salto indireto (sem troca de modo) de código SPARC16 para código SPARC16 . Não mudar o valor desse bit faz com que ocorra uma troca para modo SPARCv8. Um exemplo é apresentado na figura 5.14. Na linha 6 a instrução *call on register* é utilizada para saltar de código SPARC16 para código SPARC16. Na linha 16 a instrução *call on register with exchange* é utilizada para saltar de código SPARC16 para código SPARCv8.

```
1   sparc16_function:
2       savesp -96
3       ; esse trecho chama outra função sparc16
4       sethi %hi(sparc16_function_2), %l0
5       or  %l0, %lo(sparc16_function_2), %l0
6       callreg %l0 ; traduzido para SPARCv8 como jmpl %l0, 1, %o7
7       nop
8       ...
9
10  ; alocada no endereço 0x40001100
11  sparc16_function_2:
12      savesp -96
13      ; esse trecho chama uma função codificada em SPARCv8
14      sethi %hi(sparcv8_function), %l0
15      or  %l0, %lo(sparcv8_function), %l0
16      callregx %l0 ; traduzido para SPARCv8 como jmpl %l0, 0, %o7
17      nop
18      ...
19
20  ; Alocada no endereço 0x40001200
21  sparcv8_function:
22      ...
```

Figura 5.14: Instrução SPARC16 de salto indireto com troca de modo

Instruções de salto direto com troca de modo

Não é possível utilizar a abordagem anterior para as instruções de salto direto porque, para esse grupo (que contém instruções de *call* e *branch* com troca de modo), o imediato é deslocado a esquerda. Para isso, o descompressor gera os sinais *trigger_switch_to_v8* e *trigger_switch_to_16*. Quando o processador está em modo SPARCv8 e a instrução *bx* é executada, o sinal *trigger_switch_to_16* assume o nível lógico 1. Analogamente, quando o processador está em modo SPARC16 e as instruções *bx* ou *callx* são executadas, o sinal *trigger_switch_to_v8* assume o nível lógico 1.

Note que o descompressor está posicionado no primeiro estágio do *pipeline* (*fetch*), enquanto que o endereço alvo de um salto direto é calculado no segundo estágio (*decode*). Dessa forma, os sinais que disparam uma troca de modo são armazenados no registrador FE/DE do *pipeline* (conforme pode ser visto na figura 5.15), para serem utilizados quando a instrução estiver no estágio de *decode*. Perceba que o cálculo do alvo de uma instrução de salto direto (sinal *de_branch_address*) só usa os 31 bits mais significativos do *program counter* e do imediato estendido. O bit menos significativo (que determina em qual modo o processador está operando) é calculado utilizando o modo corrente (seletor do MUX 1) e os sinais *trigger_switch_to_v8* e *trigger_switch_to_16*.

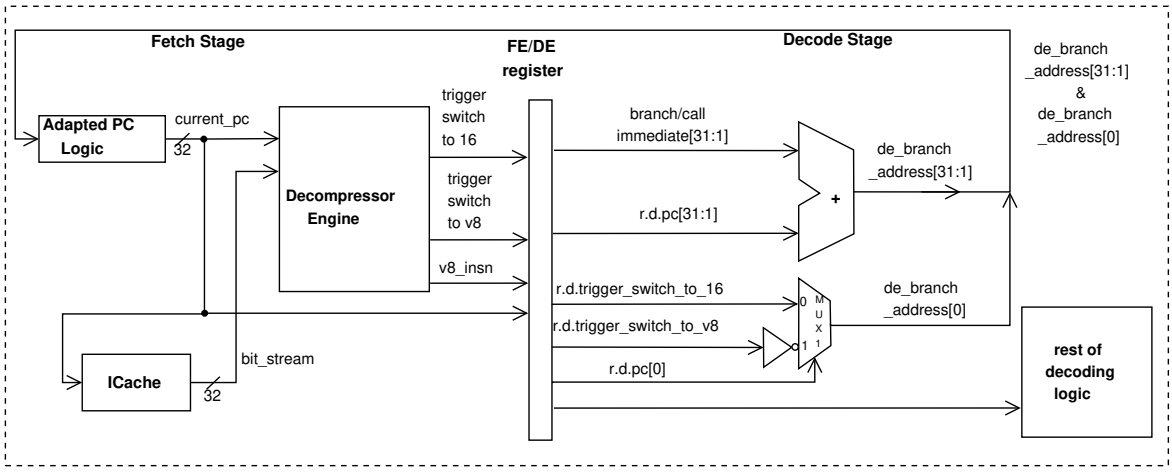


Figura 5.15: Uso dos sinais *trigger_switch_to_v8* e *trigger_switch_to_16*

Um exemplo de uma troca de modo (de SPARC16 para SPARCv8) é dado a seguir. Suponha a situação ilustrada na figura 5.16. A figura mostra o *program counter*, o endereço da memória que é efetivamente enviado à *cache* de instruções e, finalmente, na última coluna, a instrução SPARC16 devolvida pela *cache*. Note que, como o processador está em modo SPARC16, o bit menos significativo do *program counter* é 1. No endereço 0x40000000 está a instrução de *branch with exchange* enquanto que o *delay slot* (no endereço 0x40000002), conforme pode ser visto no código do programa, é uma instrução

NOP.

program counter	Endereço da memória	Memória de instruções
		...
0x40000001	0x40000000	*bx v8_function*
0x40000003	0x40000002	*nop*
		...

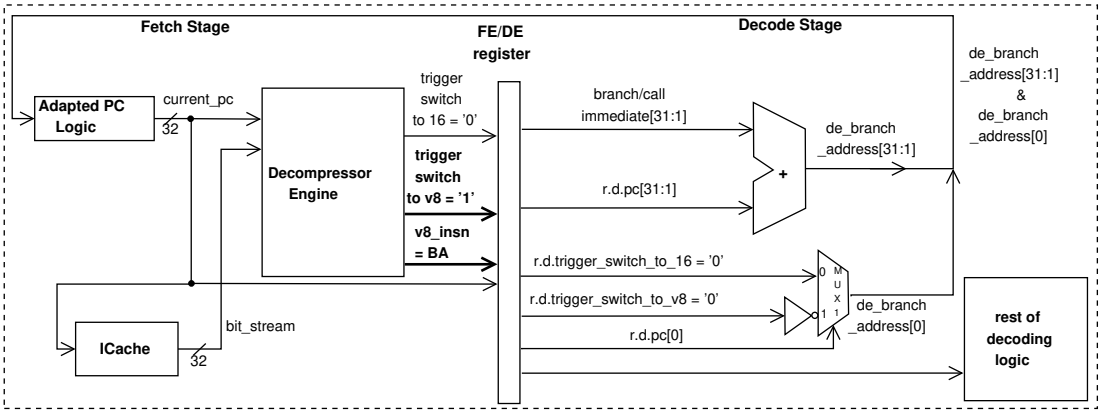
Figura 5.16: *Snapshot* da memória contendo uma instrução SPARC16 *branch with exchange*

A figura 5.17(a) mostra a instrução *branch with exchange* sendo processada no estágio de *fetch* do *pipeline* (os sinais relevantes estão em negrito). Note que a instrução é convertida em um *branch always* e que o sinal *trigger_switch_to_v8* assume o nível lógico alto (e é armazenado no registrador FE/DE). A figura 5.17(b) mostra a instrução de *branch* sendo processada no estágio de *pipeline* (o NOP do *delay slot* está no estágio de *fetch*). O sinal *r.d.trigger_switch_to_v8*[3] possui o nível lógico 1, passa por um inversor (e, por isso, se torna 0) e alimenta a entrada 1 do multiplexador MUX 1. O multiplexador é controlado pelo modo corrente (SPARC16) e, portanto, seleciona a entrada 1 em sua saída, fazendo com que o modo do processador seja trocado.

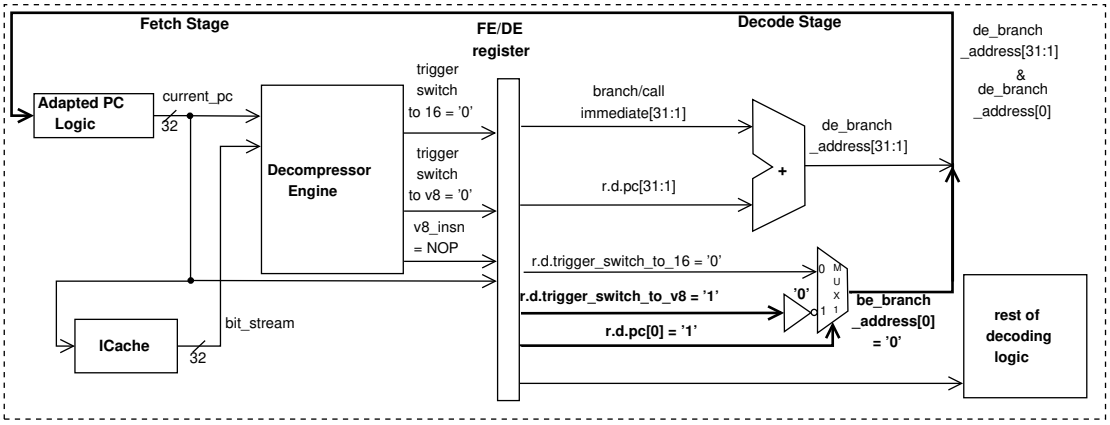
O *branch with exchange* de SPARCv8 para SPARC16 funciona de forma análoga. Suponha a situação ilustrada na figura 5.18. Perceba que, nesse caso, o bit menos significativo do *program counter* é zero. A instrução *bx* está alocada no endereço 0x40000000 e o *delay slot* (preenchido com um NOP) no endereço 0x40000004.

A figura 5.19(a) mostra a instrução *branch with exchange* sendo processada no estágio de *fetch*. Ela é traduzida para um instrução *branch always* e, além disso, o sinal *trigger_switch_to_16* assume o nível lógico 1. Quando a instrução chega no estágio de *decode* (figura 5.19(b)), o sinal *r.d.trigger_switch_to_v8*, que alimenta a entrada 0 do MUX1, é 1. Como o modo corrente é SPARCv8 (portanto, *pc*[0] ='0'), é esse sinal que se torna o bit menos significativo do alvo da instrução de *branch*, fazendo com que ocorra uma troca de modo.

[3]Nome do sinal na implementação em VHDL



(a) SPARC16 *branch with exchange* no estágio de *fetch* do *pipeline*

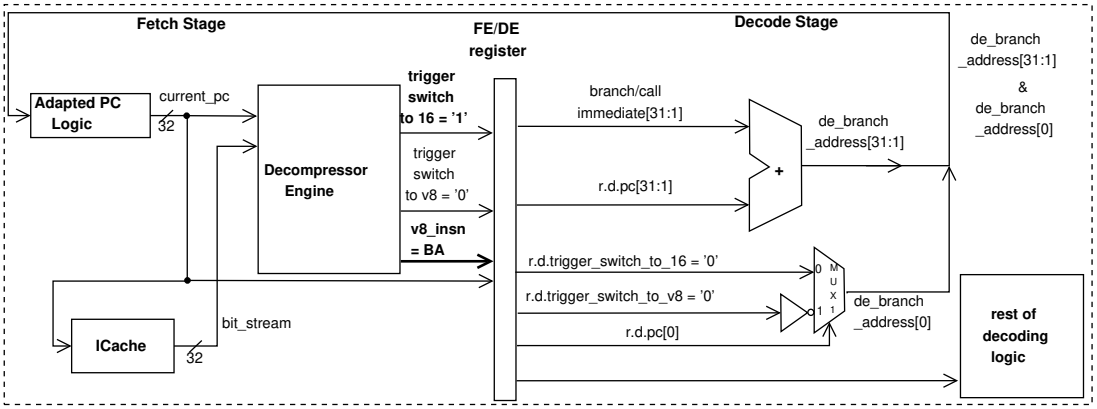


(b) SPARC16 *branch with exchange* no estágio de *decode* do *pipeline*

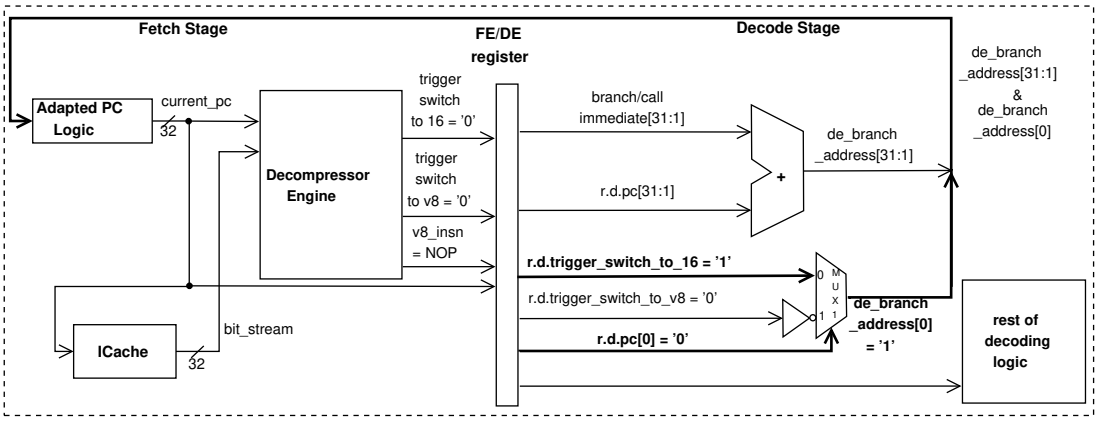
Figura 5.17: Processamento de uma instrução SPARC16 *branch with exchange* no *pipeline* do Leon 3

program counter	Endereço da memória	Memória de instruções
		...
0x40000000	0x40000000	*bx s16_function*
0x40000004	0x40000004	*nop*
		...

Figura 5.18: *Snapshot* da memória contendo uma instrução SPARCv8 *branch with exchange*



(a) SPARCv8 *branch with exchange* no estágio de *fetch* do *pipeline*



(b) SPARCv8 *branch with exchange* no estágio de *decode* do *pipeline*

Figura 5.19: Processamento de uma instrução SPARCv8 *branch with exchange* no *pipeline* do Leon 3

5.3.3 Mudanças no cálculo do endereço-alvo de *branches* e *calls*

Os requisitos de alinhamento de código na memória são diferentes entre SPARC16 e SPARCv8. Código SPARC16 precisa estar alinhado em endereços múltiplos de 2, enquanto que código SPARCv8 precisa estar alinhado em endereços múltiplos de 4. Levando em consideração que o imediato de instruções de *branch* e *call* são deslocados (antes de serem somados ao *pc* para se obter o endereço alvo), uma pequena modificação precisou ser feita.

Originalmente, o imediato era deslocado 2 bits à esquerda (ou seja, era multiplicado por 4). Para suportar a extensão SPARC16, o deslocamento passou a ser variável: caso a rotina alvo esteja codificada em SPARCv8, continuamos deslocando 2 bits à esquerda. Caso contrário, deslocamos apenas 1 bit à esquerda (multiplicamos por 2).

O modo no qual a rotina alvo está codificada é identificado através dos sinais *trigger_switch_to_v8* e *trigger_switch_to_16*, de acordo com a tabela 5.2. Repare que a tabela não lista uma situação onde os dois sinais estejam no nível lógico 1 porque ela nunca acontece.

trigger switch to v8	**trigger switch to 16**	**Descrição**
0	1	Código SPARCv8 está sendo executado, mas o alvo da instrução de salto é código SPARC16.
1	0	Código SPARC16 está sendo executado, mas o alvo da instrução de salto é código SPARCv8.
0	0	O alvo está codificado usando o mesmo modo que a instrução de salto.

Tabela 5.2: Identificação do modo em que a rotina alvo foi escrita

5.3.4 Cálculo do endereço de retorno

Quando uma instrução *call* é executada, seu próprio endereço é armazenado no registrador %o7. Portanto, depois que o procedimento chamado termina seu processamento, ele executa a instrução *jmpl %o7+8, %g0* para retornar o controle à função chamadora. A constante 8 é necessária porque %o7 contém o endereço do *call*, e $\%o7 + 4$ é o endereço do *delay slot*. Logo, o endereço da próxima instrução a ser executada é dado por %o7+8. Isso funciona perfeitamente para procedimentos SPARCv8 chamando outros procedimentos também codificados em SPARCv8.

Entretanto, surge um problema quando uma função SPARC16 chama código SPARCv8 e vice-versa. A figura 5.20 ilustra essa situação. Nela, é possível observar um trecho de

uma função codificada usando instruções SPARC16 posicionada a partir do endereço 0x40000000.

	Memória de instruções
	...
0x40000000	*callx v8_function*
0x40000002	*nop*
0x40000004	*mov 0, %i0*
0x40000006	*restore*
0x40000008	*retl*
	...

Figura 5.20: Trecho de código SPARC16 chamando código SPARCv8

A instrução *call* (endereço 0x40000000) chama uma função codificada em SPARCv8 (v8_function). O *delay slot* (posicionado em 0x40000002) foi preenchido com um NOP. Portanto, a próxima instrução a ser executada (depois do retorno da função v8_function) é *mov 0, %i0* (endereço 0x40000004). Porém, para retornar o controle à função chamadora, v8_function executa *jmpl %o7+8, %g0*, o que faz com que o *pc* se torne 0x4000008.

Para corrigir isso, o estágio de EXECUTE foi modificado. Caso uma instrução *jmpl* esteja sendo executada para retornar o controle de um procedimento SPARCv8 para outro em SPARC16, o imediato é deslocado 1 bit a direita (dividido por 2). Da mesma forma, caso uma instrução *jmpl* esteja sendo executada para retornar o controle de um procedimento SPARC16 para outro escrito em SPARCv8, o imediato é deslocado 1 bit a esquerda (multiplicado por 2).

5.3.5 Mudanças no mecanismo de *bursting* da *cache*

Conforme explicado na seção 5.1.2, o processador Leon 3 faz uso do mecanismo de *bursting*. Quando em modo *burst*, o controlador de *cache* encaminha as instruções SPARCv8 (que possuem 4 bytes) diretamente da memória externa para o *pipeline*. Isso se torna um problema quando estamos executando código SPARC16, dado que a maioria das instruções dessa extensão possuem apenas 2 bytes. O controlador de *cache* original ignora esse fato e continua entregando 4 bytes para o processador a cada ciclo, fazendo com que o Leon 3 pule algumas instruções SPARC16. A figura 5.21 ilustra essa situação. Nela, observa-se que ocorreu uma falha de acesso à *cache* e que o controlador, depois de buscar as instruções (palavras de 4 bytes na memória) as repassa ao processador (uma palavra nova a cada ciclo). Como cada instrução SPARC16 possui 2 bytes, o processador deixa de executar as instruções $s16_2$ e $s16_4$.

Levando em consideração que o mecanismo de *bursting* impacta de forma positiva o

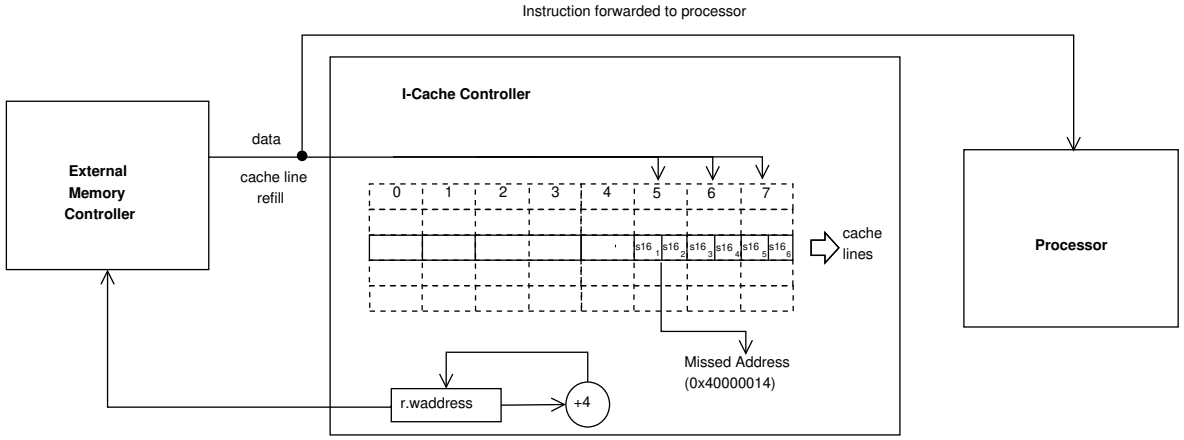


Figura 5.21: Modo *burst* com código SPARC16

desempenho [10, 11], desabilitá-lo para resolver esse problema não é aceitável. A solução encontrada foi fazer com que o controlador de *cache* continue solicitando palavras da memória até que a linha da *cache* seja preenchida, porém, caso código SPARC16 esteja sendo executado, as instruções param de ser encaminhadas ao processador. A mesma abordagem é adotada pela controlador de *cache* para lidar com saltos que ocorram durante o *bursting*.

A tabela 5.3 apresenta a ordem incorreta (mecanismo original) e a ordem correta (obtida através da modificação discutida acima) de execução.

Ciclo	**Ordem Incorreta** (Mecanismo original)	**Ordem Correta** (Mecanismo de *bursting* modificado)
0	$s16_1$	$s16_1$
1	$s16_3$	$s16_2$
2	$s16_5$	$s16_3$
...	...	...

Tabela 5.3: Ordem de entrega de instruções SPARC16 ao processador

5.3.6 Alteração no tratador de interrupções

Conforme já foi mencionado, quando estamos executando código SPARC16, o descompressor projetado processa 2 bytes por ciclo. No caso de instruções SPARC16 de 32 bits, em um ciclo os primeiros 2 bytes da instrução são processados — um NOP é injetado no *pipeline* e o estado do descompressor é alterado — e no ciclo seguinte, os últimos 2 bytes da instrução e o estado do descompressor são utilizados para traduzir a instrução SPARC16.

Um problema trazido por essa abordagem é como lidar com interrupções que acontecem exatamente entre os dois ciclos necessários para processar uma instrução SPARC16 de 32 bits. A solução que foi adotada é simples: o tratamento das interrupções é adiado até que a segunda parte de uma instrução de 32 bits seja processada. Isso faz com que não seja necessário incluir um circuito que salve o estado do descompressor quando o processador é interrompido.

5.4 Um exemplo prático

Essa seção traz um exemplo da execução de um programa que utiliza instruções de troca de modo e também instruções SPARC16 de 32 bits. O programa em *assembly* é apresentado na figura 5.22. O programa simplesmente imprime uma mensagem através de uma chamada à função *printf* da *libc*. É importante destacar que a função *printf* utilizada está codificada com instruções SPARCv8. Por enquanto, devido a falta de um compilador para a extensão SPARC16, não foi possível gerar a *libc* em SPARC16.

```
1  ; Constante
2  .LLC0:
3          .asciz  "Teste numero %d!\n"
4          .section        ".text"
5          .align 4
6          .global main
7
8  ; trecho  de código SPARCv8
9  main:
10         sparcv8_bx main_16
11         sparcv8_nop
12
13 ; trecho de código SPARCv8
14 main_16:
15         savesp  -96
16         sethi32 %hi(.LLC0), %g1
17         extended_or %g1, %lo(.LLC0), %o0
18         mov     1, %o1
19         callx   printf
20         nop
21         mov     0, %i0
22         trestore
23         retl
24         nop
```

Figura 5.22: Exemplo de programa SPARC16

Outro detalhe importante é que o processador sempre inicia sua execução em modo SPARCv8 e, para podermos executar código SPARC16, é necessária uma troca de modos (perceba que o montador SPARC16 requer que as instruções SPARCv8 possuam o prefixo "*sparcv8_*"). Isso é feito através da instrução *bx* na linha 10 (o delay slot foi preenchido com um NOP).

Depois do *branch*, o processador passa a executar a função main_16, codificada com instruções SPARC16. A primeira coisa feita por essa função é chamar a instrução *savesp* (linha 15), que desliza a janela de registradores e decrementa o *stack pointer*. Depois, o endereço da *format string* que será passada como argumento para o *printf* é carregado no registrador %*o*0. Isso é feito através das instruções *sethi32* e *or* estendida (linhas 16 e 17). Depois, na linha 18, a constante 1 é carregada no registrador %*o*1 (ele é o segundo argumento do *printf*). Finalmente, na linha 19 um *callx* é utilizado para chamar a função *printf* (note que, como a função alvo está codificada em SPARCv8 foi necessário utilizar a instrução *call with exchange*). O *delay slot* foi preenchido com um NOP. As linhas 21, 22, 23 e 24 são o epílogo da função (armazena o valor de retorno no registrador %i0, restaura a janela de registradores e retorna para a função chamadora).

As figuras 5.23 e 5.24 mostram como as instruções SPARCv8 e SPARC16 foram alocadas na memória, respectivamente. Repare que na figura 5.24 cada *word* da memória possui duas instruções (dado o tamanho reduzido das instruções SPARC16). Perceba também que a instrução *sethi32* ficou desalinhada.

Endereço da memória	Instrução
	...
0x400011C8	*sparcv8_bx*
0x400011CC	*sparcv8_nop*
	...

Figura 5.23: Instruções SPARCv8 na memória

Endereço da memória	Instruções	
	...	
0x400011D0	savesp	sethi[31:16]
0x400011D4	sethi[15:0]	extend
0x400011D8	or	mov
0x400011DC	callx	nop
0x400011E0	mov	trestore
0x400011E4	retl	nop
	...	

Figura 5.24: Instruções SPARC16 na memória

A figura 5.25 apresenta uma forma de onda que mostra como a instrução SPARCv8 *bx* é executada. Repare que no ciclo 1, quando a instrução *bx* está no estágio de *fetch* do *pipeline*, o sinal *trigger_switch_to_16* assume o nível lógico 1. Esse mesmo valor é armazenado no registrador *r.d.trigger_switch_to_16* ao final do primeiro ciclo, de modo

que no ciclo 2, quando a instrução *bx* está no estágio de *decode*, o processador sabe que trata-se de uma instrução de troca para modo SPARC16 e, ao calcular o endereço alvo do salto, garante que o *bit* menos significativo do mesmo seja 1. No ciclo 3, o processador passa a executar as instruções do endereço 0x400011D0 (função main_16). Perceba que o bit menos significativo do *program counter* corrente (sinal *current_pc*) é 1.

Note também que o sinal *holdn* assume o nível lógico 0. Esse sinal é responsável por travar o *pipeline*. Nesse caso específico, o travamento ocorreu porque o controlador de *cache*, funcionando em modo *burst*, parou de encaminhar as instruções ao processador, dado que o mesmo passou a executar código SPARC16 (lembre-se da seção 5.3.5).

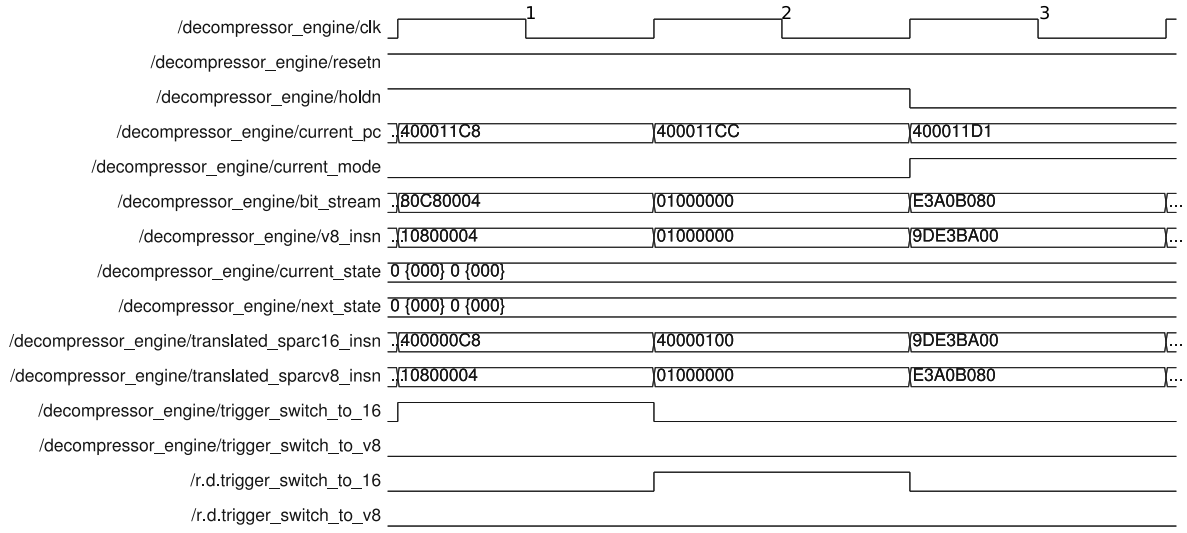


Figura 5.25: Forma de onda mostrando troca de modo de SPARCv8 para SPARC16

Depois que a linha de *cache* termina de ser preenchida, o sinal *holdn* passa a ser 1, destravando o *pipeline*. A figura 5.26 apresenta uma forma de onda que mostra as instruções SPARC16 sendo descomprimidas. No ciclo 1, a instrução do endereço 0x400011D0 (um *savesp*) é traduzido para o seu equivalente em SPARC16.

No ciclo 2, o descompressor processa a primeira parte de uma instrução *sethi*. Note que o sinal *v8_insn*, que fornece a instrução que seguirá pelo *pipeline*, contém o valor 0x01000000 (a codificação de um NOP em SPARCv8). Perceba também que os sinais *next_state.sethi32* e *next_state.sethi32_bits* assumem o valor 1 e 080, respectivamente. Esses sinais são armazenados nos registradores *current_state.sethi32* e *current_state.sethi32_bits* ao final do ciclo. Assim, no ciclo seguinte, quando os 2 bytes do endereço 0x400011D4 são processados, o descompressor sabe que está processando a segunda parte de uma instrução *sethi*. Ele utiliza os bits do imediato carregados no ciclo anterior (juntamente com os presentes no segundo pedaço da instrução) para formar a instrução *sethi* codificada em SPARCv8 (sinal *v8_insn*).

No ciclo 4, o descompressor processa a instrução EXTEND (endereço 0x400011D6). De forma análoga ao que aconteceu quando a instrução *sethi* foi processada, um NOP é injetado no *pipeline* e os sinais *next_state.extended* e *next_state.extend_bits* assumem o valor 1 e 068, respectivamente. No ciclo seguinte, esses sinais estão disponíveis nos registradores *current_state.extended* e *current_state.extend_bits* e, juntamente com as informações contidas na instrução SPARC16 *or*, são usados para gerar uma instrução *or* codificada em SPARCv8. Isso conclui a carga de uma constante no registrador %o0.

No ciclo 6, o valor 1 é colocado no registrador %o1 através da instrução SPARC16 *mov*. Essa instrução é traduzida para um *or* que utiliza o registrador %g0 de forma implícita. Isso conclui a passagem de argumentos para a função *printf* (o endereço da *format string* está no registrador %o0, e o inteiro a ser impresso na tela está no registrador %o1).

No ciclo 7, a instrução *callx* (endereço 0x400011DC) passa pelo descompressor. A troca de modos é necessária porque, como já foi mencionado anteriormente, estamos chamando a função *printf* codificada com instruções SPARCv8. A instrução *callx* é traduzida para um *call* além de fazer com que o sinal *trigger_switch_to_v8* assuma o nível lógico 1. Esse mesmo sinal é armazenado, ao final do ciclo, no registrador *r.d.trigger_switch_to_v8*. Isso faz com que, no ciclo 8, quando a instrução está no estágio de *decode*, o bit menos significativo do endereço alvo seja 0.

Portanto, como pode ser visto na figura, no ciclo 9 o processador passa a executar instruções do endereço 0x40001210, o endereço da função *printf*. Note que o bit menos significativo do sinal *current_pc* é 0, ou seja, o processador está executando instruções SPARCv8. A função *printf* é executada e causa a impressão na tela da seguinte mensagem: *Teste número 1*. O retorno da função *printf* para o código chamador é trivial (dado que o bit menos significativo do endereço de retorno contém o modo de execução da função chamadora) e, por isso, foi omitido.

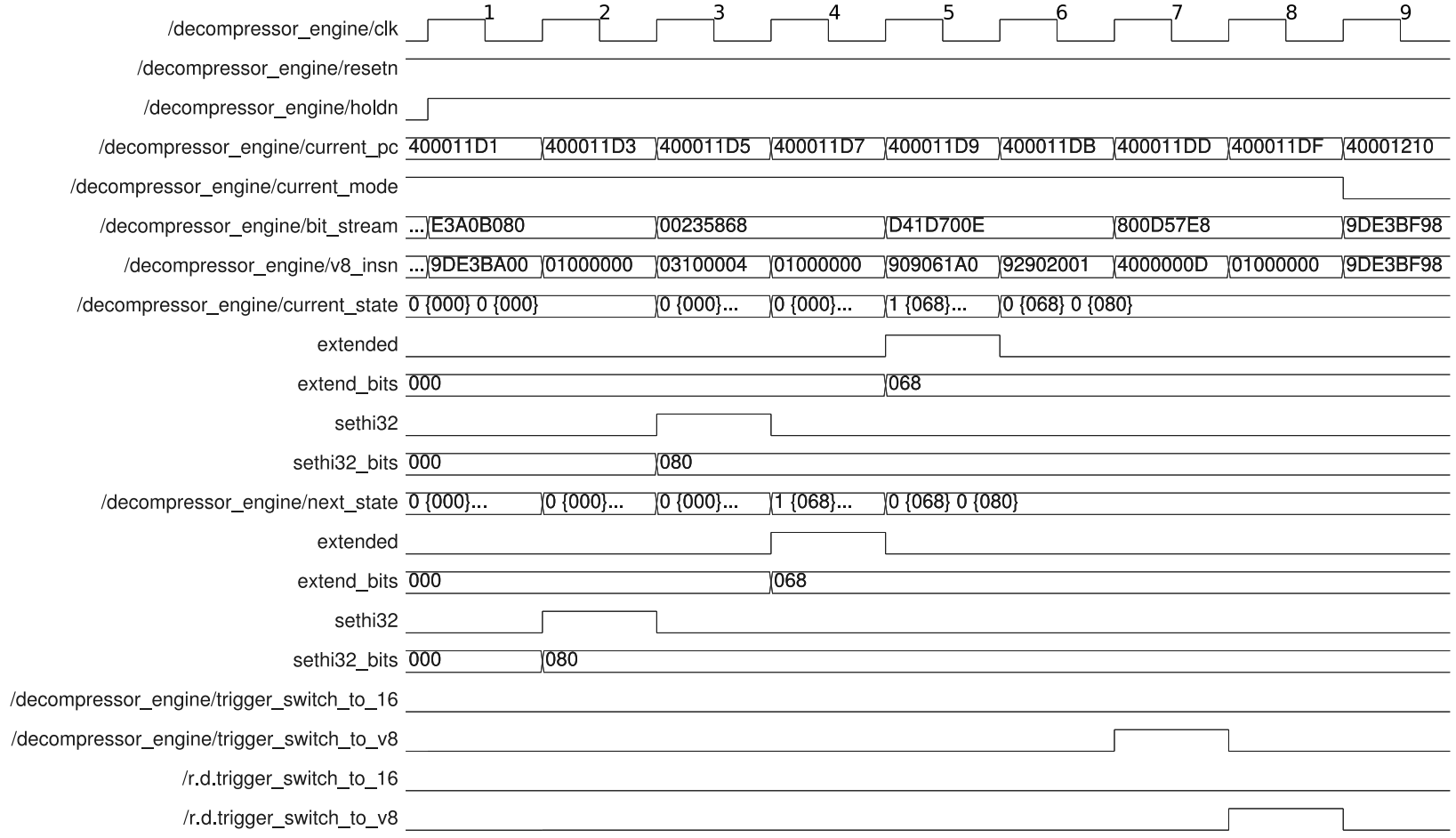


Figura 5.26: Forma de onda mostrando código SPARC16 sendo executado

5.5 Considerações finais e Conclusões

O grande desafio do projeto do descompressor era tratar instruções de 32 bits. Isso foi resolvido incluindo um elemento sequencial (estado) no descompressor. Além disso, fazer com que instruções SPARC16 pudessem ser alcançadas através de saltos, mesmo quando não alocadas em endereços múltiplos de 32 *bits*, foi trivial (só foi necessário impedir que uma exceção fosse gerada caso código SPARC16 estivesse sendo executado e um salto para um endereço desalinhado ocorresse).

O tamanho do descompressor ficou maior do que o previsto (os resultados são detalhados no próximo capítulo). Por outro lado, a latência trazida pela descompressão é mascarada pelo multiplicador/divisor implementado em *hardware*.

A validação da extensão SPARC16 foi feita através de programas escritos em linguagem de montagem. A maior parte das instruções requer apenas lógica combinacional, de modo que um programa contendo todas as instruções SPARC16 foi escrito e executado. Além disso, programas foram escritos para garantir que o descompressor consegue lidar com instruções de 32 bits desalinhadas (mesmo caso ocorra um *cache miss* ou uma interrupção entre o processamento da primeira e da segunda parte dessas instruções).

Finalmente, a implementação de um descompressor em um processador consagrado no meio comercial, juntamente com um conjunto de ferramentas para geração de código comprimido (que estão dentro do escopo do trabalho de um aluno de doutorado do LSC) tornam essa abordagem atrativa para usos no futuro.

Capítulo 6

Resultados experimentais

Este capítulo apresenta os resultados obtidos com esse trabalho e está dividida em duas partes:

- As seções 6.1 e 6.2 descrevem a infraestrutura utilizada para gerar programas SPARC16 e prototipar o Leon 3, além dos resultados da síntese para FPGA.
- A seção 6.3 apresenta estimativas[1] de razões de compressão para os programas do Mediabench e do Mibench, além de uma avaliação das mudanças no padrão de acesso à *cache* de instruções que deve ser observada com o uso de SPARC16.

Foi necessário estimar as razões de compressão para os programas do Mediabench e do Mibench porque um compilador para a extensão SPARC16 ainda não está disponível. Apenas programas simples escritos em linguagem de montagem foram executados no sistema prototipado em FPGA.

6.1 Infraestrutura Utilizada

A infraestrutura utilizada para executar programas SPARC16 envolve os utilitários (montador SPARC16 e um *linker* híbrido) responsáveis por gerar binários com instruções SPARC16, além das ferramentas da Altera/Modelsim que permitiram o projeto e prototipação do Leon 3 com o descompressor SPARC16 em FPGA.

[1]Conforme mencionado na introdução, o compilador SPARC16 está dentro do escopo do trabalho de doutorado de Bruno Cardoso Lopes. Para contornar esse problema, um método para inferir como código SPARCv8 seria codificado com instruções SPARC16 foi desenvolvido para estimar razões de compressão.

6.1.1 Geração de código SPARC16

Como já mencionado anteriormente, um compilador para a extensão SPARC16 ainda está sendo desenvolvido. Atualmente, estão disponíveis apenas um montador SPARC16 e um *linker* híbrido (capaz de gerar binários contendo código SPARCv8 e SPARC16).

A figura 6.1 mostra o fluxo utilizado para gerar programas SPARC16. Primeiramente, o programa é escrito utilizando a linguagem de montagem da extensão SPARC16. O montador é, então, utilizado para gerar um arquivo no formato ELF. Dado que o compilador SPARC16 ainda está sendo desenvolvido, a *GNU C Library* foi compilada para SPARCv8 e suas funções são chamadas pelos programas SPARC16 através das instruções de troca de modo apresentadas na seção 4.2.4.

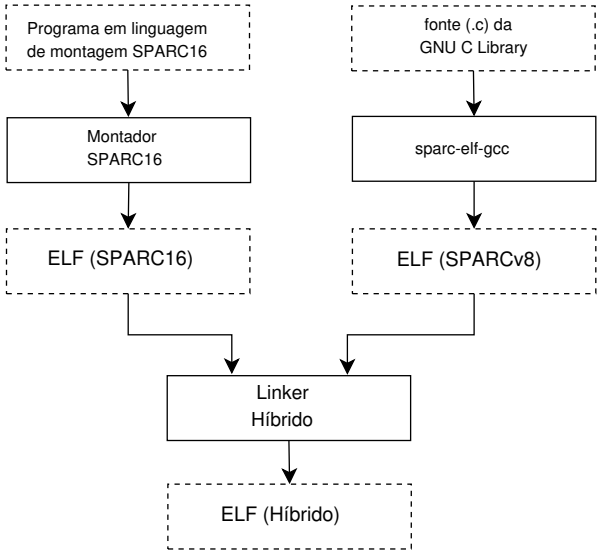


Figura 6.1: Fluxo para geração de programas SPARC16

Finalmente, o *linker* é chamado e gera um arquivo binário no formato ELF contendo tanto código SPARCv8 quanto código SPARC16.

6.1.2 Desenvolvimento e síntese do descompressor

A figura 6.2 ilustra a infraestrutura utilizada durante as etapas de desenvolvimento e síntese do descompressor. Foram utilizadas ferramentas da Altera para síntese e da Mentor Graphics para simulação do projeto.

Primeiramente, simulações RTL são executadas para garantir que o descompressor funciona de acordo com o que foi especificado. Depois, a ferramenta *quartus_map* é utilizada para fazer a síntese lógica, que consiste em traduzir o modelo RTL para um conjunto de elementos lógicos interconectados (*netlist*). Para garantir que a *netlist* é

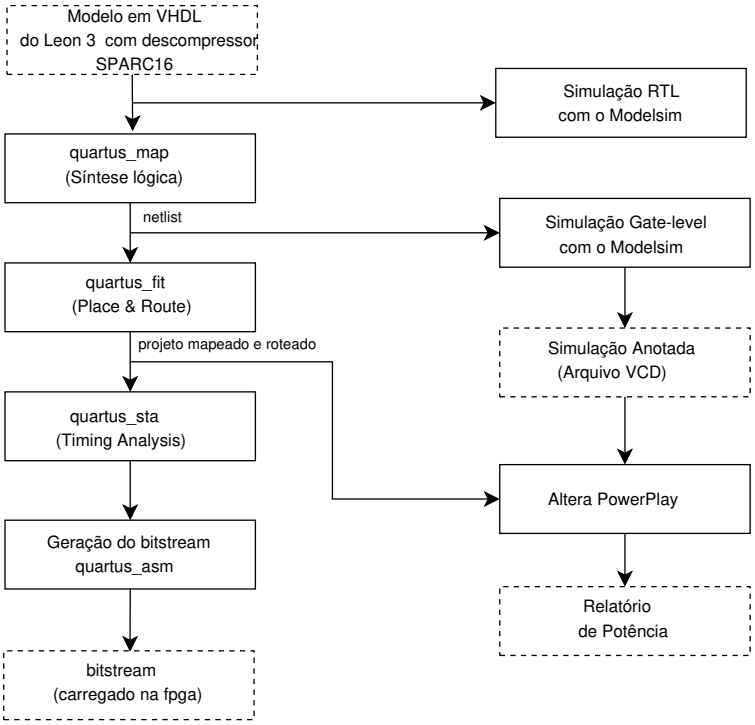


Figura 6.2: Infraestrutura de desenvolvimento

equivalente ao modelo RTL, simulações *gate-level* são realizadas. Além disso, a simulação *gate-level* pode, opcionalmente, gerar um arquivo com a anotação de todas as transições de sinais. Esse arquivo é usado para alimentar a ferramenta que gera estimativas de consumo de energia (essa etapa não foi realizada nesse trabalho).

Em seguida, a ferramenta *quartus_fit* é utilizada para fazer o *place and route*, que consiste em posicionar os elementos lógicos da *netlist* na FPGA (*place*) e interconectá-los (*route*), visando atender os requisitos de *timing* do projeto (que são conferidos, numa etapa posterior, pela ferramenta *quartus_sta*). Finalmente, o *bitstream* é gerado e pode ser enviado para a FPGA.

Todas as ferramentas de síntese citadas fazem parte do Quartus II 9.1.

6.1.3 Ambiente de prototipagem

A placa *Nios II Development Kit, Stratix II Edition* [14] foi utilizada para prototipar o Leon 3 com o descompressor SPARC16. Dentre as características da mesma, estão:

- 2MB de memória SRAM;
- 16MB de memória DDR SDRAM;

- Interface JTAG;
- FPGA Stratix II EP2S60;
- Interface Ethernet 10/100;

Conforme pode ser visto na figura 6.3, a interface JTAG é utilizada para enviar o *bitstream* para a FPGA através do programa *quartus_pgm*, que roda em uma estação de trabalho.

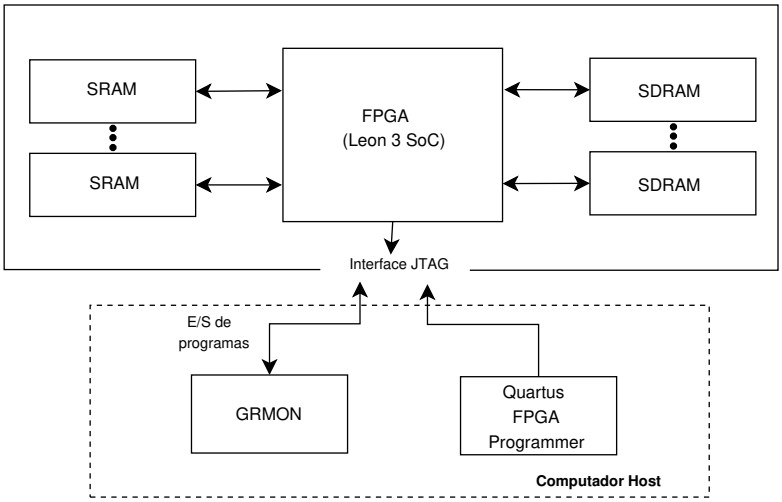


Figura 6.3: Placa de prototipação

A mesma interface é, posteriormente, utilizada para que o programa GRMON [2], que também roda em uma estação de trabalho, se conecte ao sistema com o Leon 3 rodando na FPGA. O GRMON é um monitor de *debug* fornecido pela própria Gaisler que possui as seguintes funcionalidades:

- Acesso de leitura/escrita a todos os registradores do sistema e da memória;
- Gerenciamento de *trace buffer*;
- Carregamento e execução de aplicações no Leon 3;
- Gerenciamento de *breakpoints*;
- Conexão remota ao *GNU Debugger* (gdb);
- Suporte à USB, JTAG, RS232, Ethernet e SpaceWire;

Destaca-se, dentre as características mencionadas, o *trace buffer*, que é um mecanismo capaz de armazenar as últimas instruções executadas e as últimas transferências que ocorreram no barramento AHB. Através dele, foi possível verificar que as instruções SPARC16 estavam sendo decodificadas de forma correta.

6.1.4 Validação e Programas Avaliados

Devido a falta de um compilador, apenas programas simples escritos em linguagem de montagem foram executados na placa de prototipação. Dentre os programas executados estavam, por exemplo, implementações recursivas e iterativas de fatorial e fibonacci. O uso do *trace buffer* e de *breakpoints* permitiu observar que a decodificação das instruções estava sendo feita de forma correta.

Dado que o processo de decodificação de instruções SPARC16 é majoritariamente combinacional, um programa contendo todas as instruções SPARC16 foi escrito e executado. Também tomou-se o cuidado de verificar que o descompressor consegue lidar com instruções SPARC16 de 4 bytes desalinhadas, mesmo nas situações em que um *cache miss* ocorra entre o processamento da primeira e da segunda parte dessas instruções. Além disso, os programas executados utilizam funções da *GNU C Library* codificadas em SPARCv8, validando as instruções de troca de modo implementadas.

Finalmente, destaca-se que um esforço maior na validação do descompressor ficou comprometido devido a falta de um compilador.

6.2 Resultados da síntese em FPGA

A tabela 6.1 mostra o aumento que a inclusão do descompressor causou na área ocupada pelo Leon 3, em torno de 734 ALUTs[2] (24%). Entretanto, é preciso lembrar que o Leon 3 é uma implementação de um RISC de 32 bits, ou seja, pequeno. Além disso, o descompressor adiciona suporte a um novo conjunto de instruções.

Tabela 6.1: Área ocupada pelo núcleo do Leon 3

Número original de ALUTs	3051
Número de ALUTs com o descompressor	3785
Aumento da área (%)	24%

[2]Do inglês, *Adaptative Look-up Table*. Bloco básico da arquitetura da FPGA Stratix II.

Tendo isso em mente, é plausível analisar o aumento de área levando em consideração a síntese do sistema inteiro, com *cache*, suporte a Ethernet e UART, gerenciador de memória e multiplicador/divisor em *hardware*. O resultado da síntese completa está disponível na tabela 6.2. Destaca-se que a latência imposta pelo descompressor é mascarada pelo multiplicador/divisor em *hardware*.

Tabela 6.2: Utilização da área da FPGA

Recurso	**Configuração**
Número de janelas de registradores	8
I-*cache*	2 sets de 8Kbytes
D-*cache*	2 sets de 4 Kbytes
Ethernet	SIM
PCI	SIM
Unidade de Gerenciamento de Memória	SIM
Mul/Div em *hardware*	SIM
Trace buffer	128 linhas
Unidade de ponto flutuante	NÃO
Número original de ALUTs	10.894
Número de ALUTs com o descompressor	11.924
Aumento da área (%)	9%

6.3 Experimentos com benchmarks: Mibench e Mediabench

Os programas executados no sistema prototipado em FPGA não são representativos o suficiente para avaliar as razões de compressão que podem ser obtidas com o uso da extensão SPARC16. Para contornar essa limitação, os programas dos *benchmarks* Mediabench e Mibench foram compilados para a arquitetura SPARCv8. Dado que a extensão SPARC16 é apenas uma recodificação das instruções SPARCv8, é trivial inferir como um programa codificado com instruções SPARCv8 seria representado usando instruções SPARC16.

Para ilustrar como o processo de inferência funciona, vamos utilizar um programa SPARCv8 como exemplo (mostrado na figura 6.4). As instruções desse programa foram escolhidas de modo a mostrar todos os casos de mapeamento.

Começando pela instrução de soma da linha 3, é possível notar que o imediato é pequeno o suficiente para ser codificado em um campo de 5 bits. Por isso, é possível

representá-la com uma instrução SPARC16 do formato RRI (figura 6.5). O mesmo não é verdade para a instrução de soma da linha 4 (o imediato não cabe em 5 bits). Portanto, ela é representada em SPARC16 com uma instrução estendida do formato RRI (figura 6.6). Esse mesmo procedimento, que consiste em verificar se o imediato é suficientemente pequeno para caber no campo da instrução SPARC16 correspondente, é aplicado para todas as instruções lógicas e aritméticas que fazem uso de imediato.

Na sequência, temos duas instruções que operam apenas em registradores: um ORN (linha 6) e um XNOR (linha 7). Em SPARC16, essas duas operações são representadas por instruções que operam em 2 endereços (ou seja, um dos registradores funciona como fonte e destino). Tendo isso em mente, observamos que a instrução ORN possui um registrador funcionando como fonte e destino (%i0) e, por conseguinte, é representada por uma instrução SPARC16 do formato RR (figura 6.7). O mesmo não se pode afirmar da instrução XNOR. Portanto, a mesma precisa ser representada por uma instrução estendida do formato RR (figura 6.8). Note que nesse caso o mecanismo de EXTEND é usado para fornecer um registrador extra.

Prosseguindo, temos uma instrução SETHI na linha 9. Esse caso é trivial, dado que SPARC16 possui uma instrução correspondente (que também ocupa 32 bits). A codificação SPARC16 do SETHI é mostrada na figura 6.9.

Finalmente, nas linhas 11 e 12 estão uma instrução CALL seguida do *delay slot* (preenchido com um NOP). Existe uma instrução CALL em SPARC16 (formato I) que possui um campo de imediato com 11 bits. A mesma instrução pode ser estendida e, assim, ter um campo de imediato de 22 bits.

Entretanto, a codificação da mesma instrução em SPARCv8 possui 30 bits. Ou seja, é possível que mesmo uma instrução CALL estendida em SPARC16 não seja suficiente para representar o CALL em SPARCv8. Nesses casos, é necessário carregar o endereço alvo em um registrador através de uma instrução SETHI e de um *or* estendido e, posteriormente, utilizar a instrução *call on register* (formato RR3). As instruções mencionadas são ilustradas na figura 6.10 (inclusive o *delay slot*). Note que há uma perda significativa de compressão nessa situação (32 bits em SPARCv8 contra 80 em SPARC16), contudo esse caso é bastante raro. O mapeamento do NOP é trivial (possui um correspondente em SPARC16 no formato RR3)

Esse método para estimar razões de compressão não leva em consideração um possível aumento no tamanho de código causado por um número reduzido de registradores visíveis. Entretanto, devido à existência de instruções SPARC16 que utilizam registradores de forma implícita, além das instruções especiais capazes de mover valores de registradores visíveis para invisíveis (e vice-versa), é provável que esse aumento seja pequeno.

As figuras 6.11 e 6.12 apresentam as estimativas de razão de compressão para os programas do Mediabench e do Mibench, respectivamente. É possível perceber que a

melhor razão de compressão foi obtida para o programa *cjpeg* (em torno de 58%), enquanto que a pior razão foi obtida para o programa *lame* (em torno de 64%), cujas instruções possuem imediatos maiores, de modo que, se codificadas em SPARC16, usariam com mais frequência o mecanismo de EXTEND.

A razão de compressão média (ponderada) foi de 61.27% para os programas do Mediabench e de 60.77% para os programas do Mibench. A imagem do Linux compilado para SPARCv8 também foi avaliada utilizando a mesma metodologia e obteve-se 65.42% de razão de compressão. Um dos fatores que contribuíram para a obtenção de uma compressão baixa foi o fato de que algumas instruções privilegiadas ainda não são suportadas por SPARC16 e, portanto, precisam ser tratadas da seguinte forma:

1. Troca de modo para SPARCv8.
2. Execução da instrução privilegiada.
3. Troca de modo para SPARC16.

```
1   sparcv8_code:
2
3       add %i0, 5, %i1
4       add $i1, 480, %i2
5
6       orn  %i0, %i1, %i0
7       xnor %i0, %i1, %i2
8
9       sethi %hi(label), %o0
10
11      call other_function
12      nop
```

Figura 6.4: Exemplo em assembly de código SPARCv8

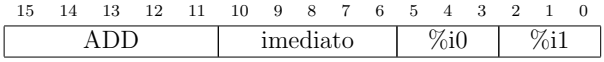


Figura 6.5: ADD da linha 3 representado em SPARC16

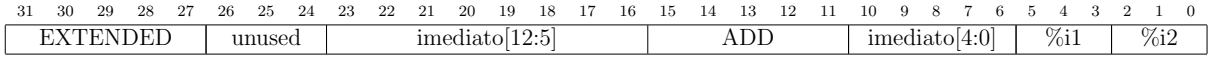


Figura 6.6: ADD da linha 4 representado em SPARC16

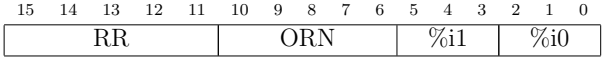


Figura 6.7: ORN da linha 6 representado em SPARC16

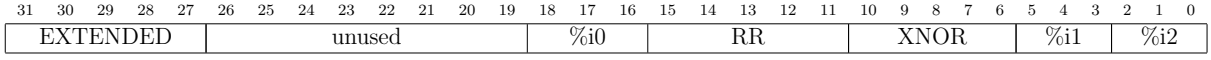


Figura 6.8: XNOR da linha 7 representado em SPARC16

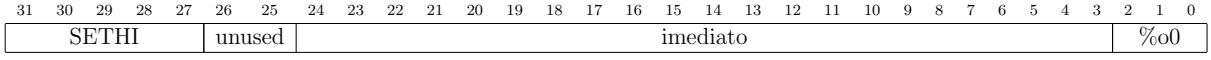


Figura 6.9: Instrução SETHI (codificada em SPARC16)

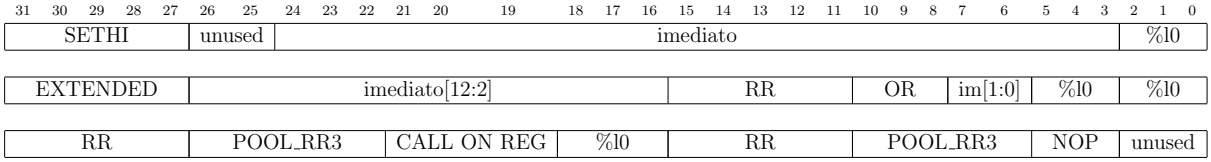


Figura 6.10: CALL da linha 11 representado em SPARC16

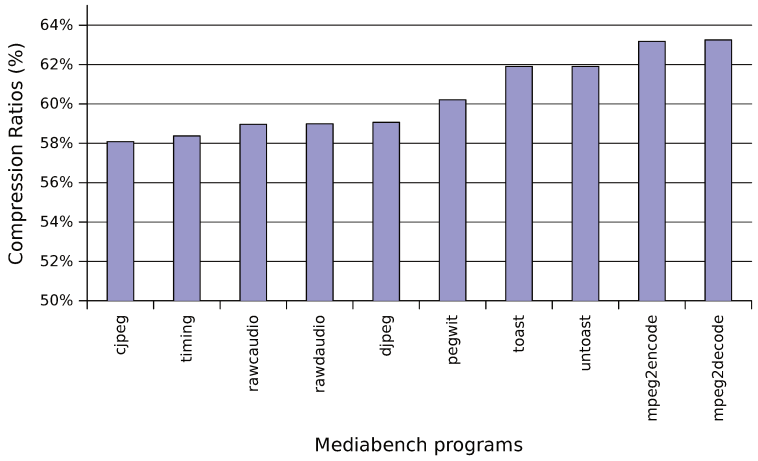


Figura 6.11: Razões de compressão para os programas do Mediabench

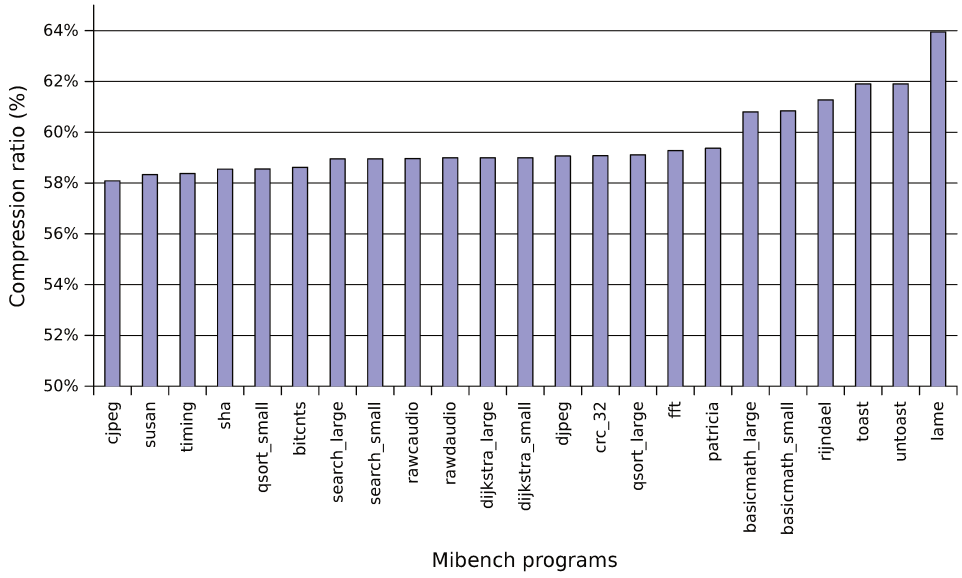


Figura 6.12: Razões de compressão para os programas do Mibench

6.3.1 Avaliação da mudança nos padrões de acesso à *cache* de instruções

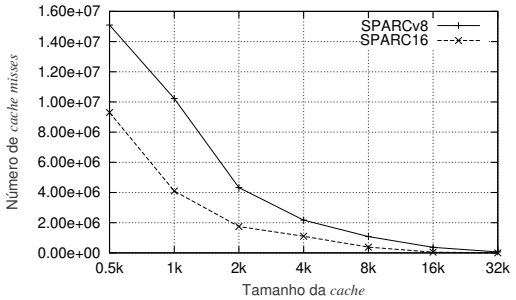
Essa seção apresenta a análise da mudança que o uso da extensão SPARC16 trará nos padrões de acesso à *cache* de instruções. Para tanto, foi utilizado um modelo de um processador SPARCv8 descrito em ArchC [49], uma linguagem de descrição de arquiteturas.

Foram coletados os *traces* da execução dos programas dos *benchmarks* Mediabench e Mibench no modelo ArchC do SPARCv8 e, através do método para inferir como código SPARCv8 seria representado em SPARC16 mostrado na seção anterior, os *traces* SPARCv8 convertidos para *traces* SPARC16.

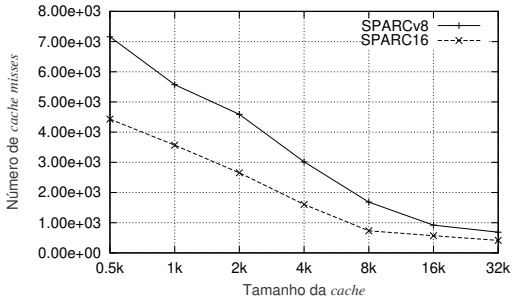
Posteriormente, os *traces* dos dois conjuntos de instruções foram analisados utilizando a ferramenta de análise de *cache* DineroIV [16]. Foram testados diferentes tamanhos para uma *cache* de associatividade 2 com 16 *bytes* por linha. Os resultados estão disponíveis na figura 6.13.

Por exemplo, para o programa basicmath (figura 6.13(a)), o uso de instruções SPARCv8 ocasiona em torno de $1.5 * 10^7$ falhas de acesso à *cache* de instruções, quando usamos uma *cache* de 0.5 *kilobytes*. No caso de SPARC16, o número cai para algo próximo de $1.0 * 10^7$.

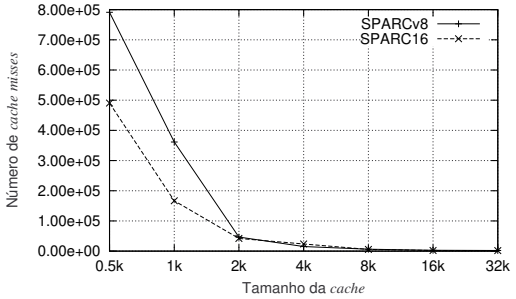
Note que para alguns programas, como é o caso do *sha* (figura 6.13(r)), o número de *misses* da extensão SPARC16 é menos do que 50% do que o observado para SPARCv8. Apesar de parecer contra intuitivo, isso é explicado pelo fato de que, para *caches* muito pequenas, os trechos do programa que são executados com frequência são suficientemente pequenos para caber na mesma quando codificados em SPARC16, o que não acontece com código SPARCv8 (causando uma substituição frequente de linhas).



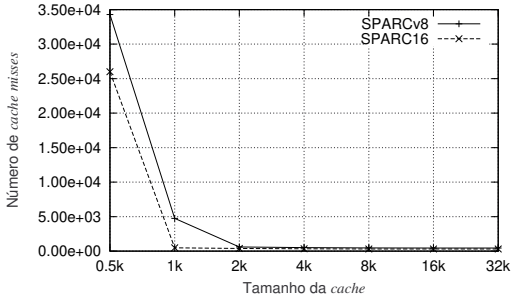
(a) basicmath



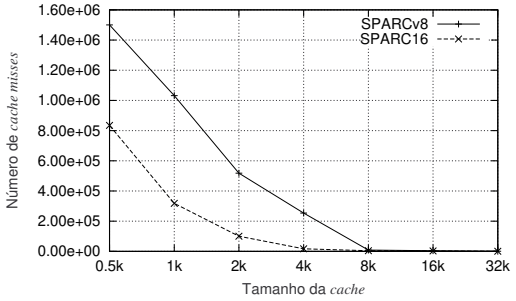
(b) bitcnts



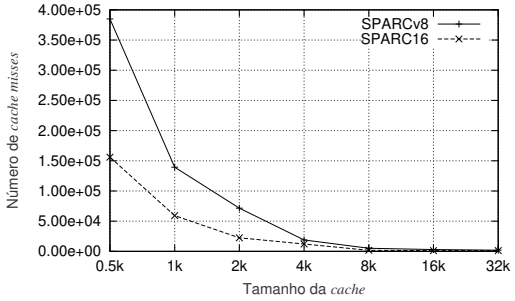
(c) cjpeg



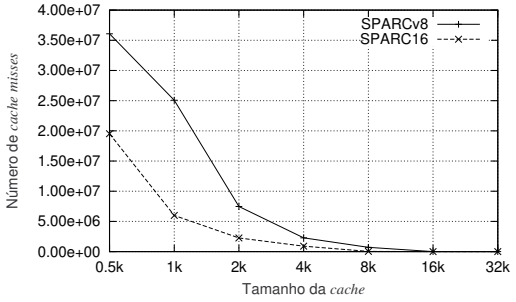
(d) crc_32



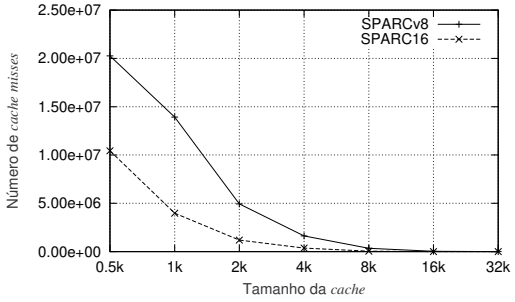
(e) dijkstra



(f) djpeg

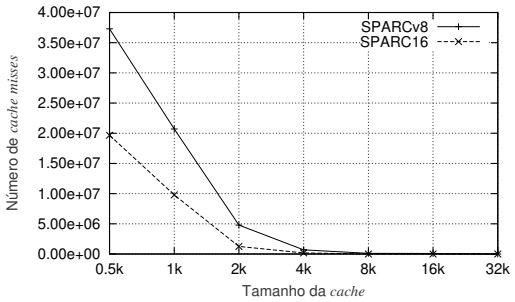


(g) fft

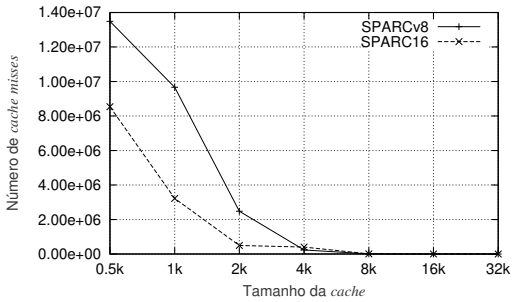


(h) lame

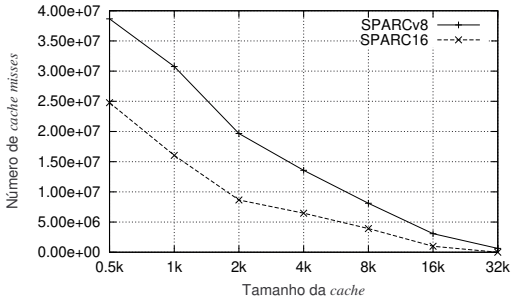
Figura 6.13: Análise de Cache: SPARC16 x SPARCv8



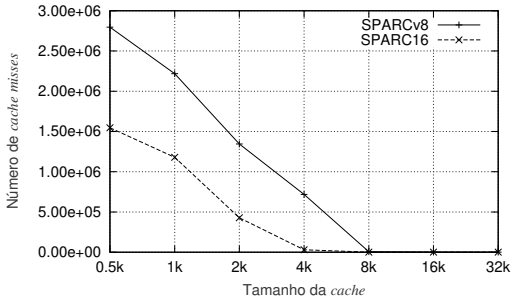
(i) mpeg2decode



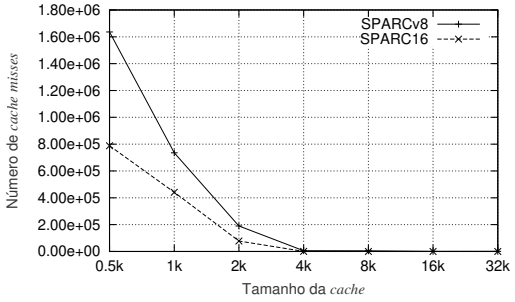
(j) mpeg2encode



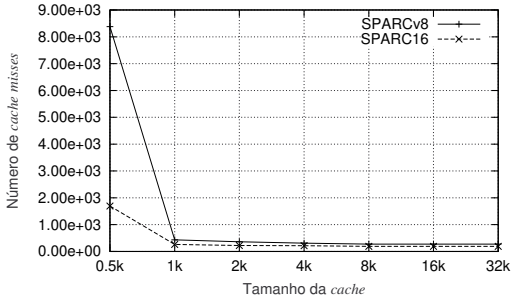
(k) patricia



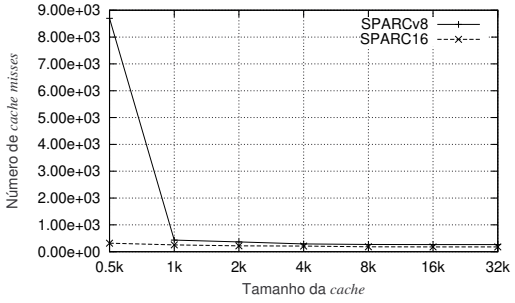
(l) pegwit



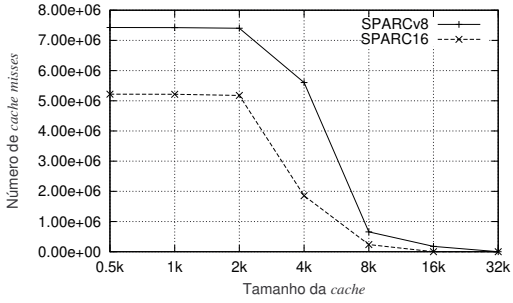
(m) qsort



(n) rawcaudio

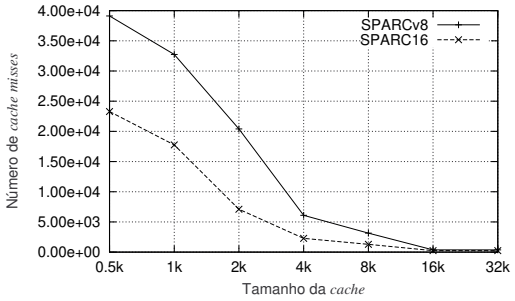


(o) rawdaudio

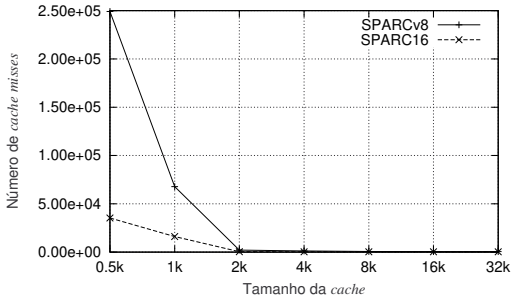


(p) rijndael

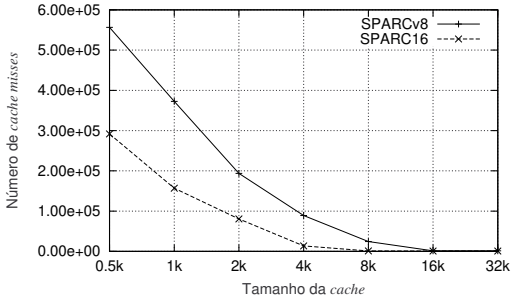
Figura 6.13: Análise de Cache: SPARC16 x SPARCv8 (Cont.)



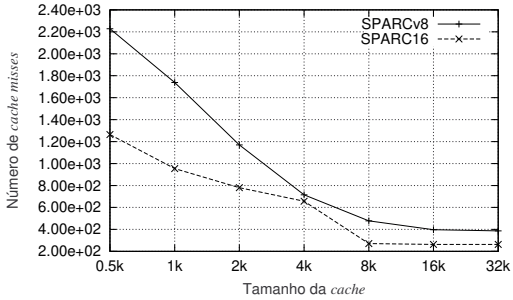
(q) string search



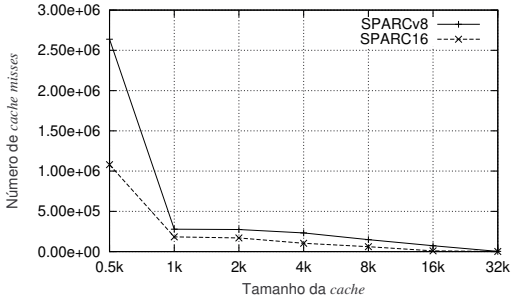
(r) sha



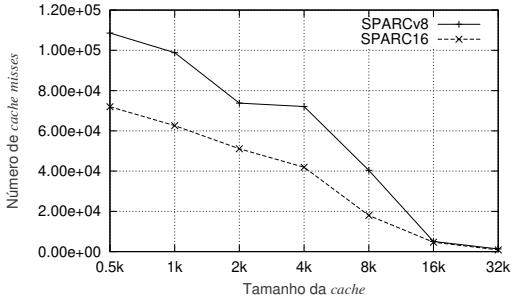
(s) susan



(t) timing



(u) toast



(v) untoast

Figura 6.13: Análise de Cache: SPARC16 x SPARCv8 (Cont.)

6.3.2 Avaliação de impacto no desempenho

Para mostrar como o uso de SPARC16 pode ocasionar ganho de desempenho, suponha o programa *rijndael* sendo executado em um processador SPARCv8 que executa exatamente uma instrução a cada ciclo do relógio. Suponha também que uma *cache* de instruções de associatividade 2 e com 16 *bytes* por linha com 2KB de tamanho é usada pelo sistema.

Para simplificar a análise, apenas falhas de acesso à *cache* de instruções serão consideradas. Portanto, o número total de ciclos gastos na execução de um programa é dado por:

$$\text{número de ciclos} = \text{IE} + \text{ICF} * \text{MP}$$

onde IE é o número total de instruções executadas, ICF é o número de falhas de acesso à *cache* de instruções e MP é penalidade (em ciclos) causada por uma falha de acesso. A tabela 6.3 apresenta os valores de IE e ICF para a execução do programa *rijndael* em SPARC16 e SPARCv8 em um sistema com a configuração de *cache* apresentada anteriormente. Os valores para SPARCv8 foram obtidos através da execução do programa em um modelo ArchC. Devido à falta de um compilador SPARC16, os valores para SPARC16 foram inferidos utilizando o mesmo método aplicado para gerar as análises de *cache*.

Tabela 6.3: Valores de IE e ECF para o programa *rijndael*

	SPARCv8	**SPARC16**
IE	32559781	43253303
ICF	7404315	5177076

Utilizando a fórmula apresentada anteriormente, podemos estimar o *speedup* que será conseguido com a extensão SPARC16 para diferentes valores de MP. A tabela 6.4 apresenta as estimativas de *speedups* obtidas. Note que, quanto maior for a penalidade por um *cache miss*, maior é o *speedup*.

Por exemplo, caso a penalidade para uma falha de acesso à *cache* de instruções for 10 ciclos, então o programa em SPARC16 executará 1.12 vezes mais rápido (89% do tempo gasto por SPARCv8). Caso a penalidade seja 50 ciclos, então o programa em SPARC16 executará 1.33 vezes mais rápido (75% do tempo gasto por SPARCv8).

6.4 Considerações finais e conclusões

Chegou-se a um protótipo funcional do processador Leon 3 com o descompressor SPARC16 integrado. O acréscimo de área devido ao *hardware* extra foi maior do que originalmente

Tabela 6.4: Estimativa do *speedup* que pode ser alcançado através do uso de SPARC16

MP	**Ciclos(SPARCv8/SPARC16)**
10	1.12
20	1.23
30	1.28
40	1.31
50	1.33

esperado (se consideramos apenas a área do *core* SPARCv8, o acréscimo chega próximo de um quarto), porém, o fato de esse tipo de informação não ser disponibilizado pela ARM e pela MIPS (empresas que desenvolveram extensões similares para suas arquiteturas) inviabilizou uma comparação.

A análise das razões de compressão alcançadas com SPARC16 ficou comprometida devido à falta de um compilador, todavia esse problema foi contornado estimando como código SPARCv8 seria representado em SPARC16 (o método para realizar as estimativas foi apresentado na seção 6.3). As estimativas de razões de compressão obtidas mostram que SPARC16 é competitivo quando comparado com abordagens similares aplicadas à ARM e MIPS, conforme pode ser observado na tabela 6.5.

Um ponto que precisa ser destacado é que, ao compilar o *benchmark* SPEC2006 para diversas arquiteturas[3], as razões de compressão reportadas pela ARM para a extensão Thumb não são alcançadas, conforme foi apresentado na figura 3.1. Se na mesma figura incluíssemos SPARC16 com o valor alcançado através da multiplicação do tamanho de código SPARCv8 por 61% (razão de compressão média de SPARC16), chegaríamos a figura 6.14 (com SPARC16 possuindo o código mais denso).

No que diz respeito a outras técnicas apresentadas na seção de trabalhos relacionados, como a abordagem proposta por Billo [10,11], SPARC16 perde em área do descompressor mas ganha facilmente em generalidade. O esquema proposto por Billo requer que um dicionário seja gerado para cada programa e, além de precisar de *patching* de endereços nos programas comprimidos (processo que o autor realizou manualmente). A extensão SPARC16 utiliza um mesmo conjunto de instruções para executar diferentes programas — apenas um compilador é necessário.

A análise das mudanças nos padrões de acesso à *cache* que serão ocasionadas pelo uso da extensão SPARC16 mostram uma redução significativa no número de *cache misses*. Por exemplo, para o programa *bitcnts* (figura 6.13(b)), o programa codificado em SPARC16

[3]Problemas com a biblioteca *GNU C Library* inviabilizaram a geração de código para MIPS16 e microMIPS.

Recodificação da ISA				
Recodificação	Arquitetura	*benchmarks*	Razão de compressão	Tempo de execução
Thumb [5]	ARM	Eqntott, Xlisp, Espresso, Dhrystone	55%∼70%	110%∼120%
Thumb2 [47]	ARM	—	5% < Thumb	102%
MIPS16 [29]	MIPS	—	60%	—
microMIPS [51]	MIPS	CSiBE e Dhrystone	65%	102%
SPARC16	SPARCv8	Mediabench e Mibench	60%	89%

Tabela 6.5: Comparação de tamanho de código entre recodificações da ISA

utilizando uma *cache* de 2kb garantiria um número de *misses* menor do que o mesmo programa codificado em SPARCv8 com uma *cache* de 4kb (o dobro do tamanho).

Se levarmos em consideração um sistema multitarefa e com o próprio sistema operacional comprimido e estabelecêssemos um desempenho mínimo, mesmo com o aumento na área causado pelo descompressor, o uso da abordagem proposta nesse trabalho contribuiria para a redução da área do circuito. Destaca-se, ainda, que de acordo com trabalhos anteriores [10, 11] o menor número de falhas no acesso à *cache* ocasionará uma redução no consumo de energia.

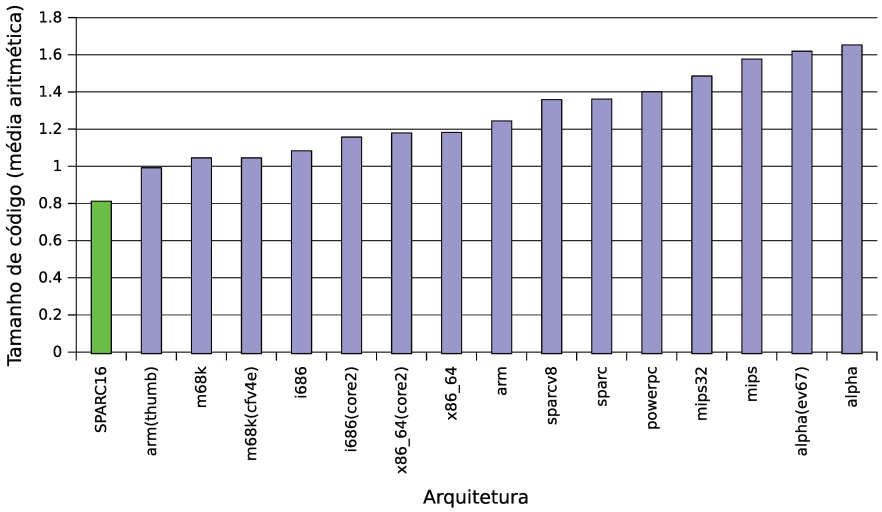


Figura 6.14: Tamanhos de código para programas do SPEC2006 para diferentes arquiteturas

Capítulo 7

Conclusão

Esse trabalho atingiu os objetivos traçados inicialmente, chegando a um protótipo funcional de um descompressor SPARC16 integrado ao processador Leon 3. O aumento de área causado pela incorporação do descompressor foi maior do que o esperado (em torno de um quarto da área original), porém, o requisito de integrar a extensão SPARC16 "minimizando o aumento de área" é subjetivo, dado que as empresas que implementaram extensões semelhantes para outras arquiteturas (ARM e MIPS) não divulgam o impacto em área causado pelo suporte às extensões de 16 *bits*.

Devido a falta de um compilador, apenas programas pequenos, codificados em linguagem de montagem, foram de fato executados em FPGA. Todavia, é possível estimar como código SPARCv8 seria representado com instruções SPARC16, de modo que uma análise das razões de compressão que podem ser atingidas através do uso dessa extensão foi feito. Também foi feita uma avaliação de como SPARC16 deve alterar os padrões de acesso à *cache*.

As análises mostraram uma razão média de compressão de 61.27% e 60.77% para os programas do Mediabench e do Mibench, respectivamente. A avaliação das mudanças nos padrões de acesso à *cache* de instruções mostra que, para *caches* muito pequenas, o número de falhas de acesso pode ser até menor do que 50% para os programas codificados em SPARC16 (quando em comparação com SPARCv8).

Isso indica que SPARC16 é competitivo e tem potencial para atingir a maior densidade de código dentre as arquiteturas difundidas na indústria. A implementação do descompressor no Leon 3 juntamente com um conjunto de utilitários para a geração de código pode aumentar o alcance desse trabalho no meio comercial.

7.1 Trabalhos Futuros

Além de um compilador (que está em processo de desenvolvimento), destacam-se as seguintes possíveis continuações desse trabalho:

- Aplicar o modelo de programação linear inteira proposto a outras arquiteturas.
- Reduzir a área do descompressor.
- Incluir suporte a ponto flutuante na extensão SPARC16. Até o presente momento, apenas instruções da unidade de inteiros são suportadas.
- Portar o sistema operacional Linux para SPARC16. Apesar de ser possível executar o Linux em 32 bits e rodar programas codificados em SPARC16, seria interessante rodar o próprio sistema operacional comprimido.
- Chegar a uma implementação em ASIC do Leon 3 com suporte à SPARC16.

Referências Bibliográficas

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

[2] Aeroflex Gaisler AB. *GRMON User's manual*. Aeroflex Gaisler AB, 2009.

[3] G. Araujo, P. Centoducatte, R. Azevedo, and R. Pannain. Expression-tree-based algorithms for code compression on embedded risc architectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 8(5):530–533, Oct 2000.

[4] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain. Code compression based on operand factorization. *Microarchitecture, 1998. MICRO-31. Proceedings. 31st Annual ACM/IEEE International Symposium on*, pages 194–201, Nov-2 Dec 1998.

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

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

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

[8] L. Benini, A. Macci, and A. Nannarelli. Cached-code compression for energy minimization in embedded processor. In *International Symposium on Low Power Electronic Design*, pages 322–327. IEEE, 2001.

[9] Árpád Beszédes. CSiBE Benchmark: One year perspective and plans. pages 7–15, 2004.

[10] E. Billo, R. Azevedo, G. Araujo, P. Centoducatte, and E. Wanderley Netto. Design of a decompressor engine on a sparc processor. In *SBCCI '05: Proceedings of the 18th annual symposium on Integrated circuits and system design*, pages 110–114, New York, NY, USA, 2005. ACM.

[11] Eduardo Afonso Billo. Projeto e implementação de um descompressor pdc-compacket em um processador sparc. Master's thesis, Instituto de Computação - UNICAMP, 2005.

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

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

[14] Altera Corporation. *Nios II Development Board Stratix II Edition, Reference Manual*. Altera Corporation, 2007.

[15] Rodolfo Jardim de Azevedo. *Uma Arquitetura para Execução de Código Comprimido em Sistemas Dedicados*. PhD thesis, Instituto de Computação - UNICAMP, 2002.

[16] J. Edler and M.D. Hill. Dinero iv trace-driven uniprocessor cache simulator, online at http://www.cs.wisc.edu/ markhill/dineroiv/, 2003.

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

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

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

[20] Jiri Gaisler. The leon3 processor. online, 2008. `http://www.gaisler.com`.

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

[22] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, Dec. 2001.

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

[24] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

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

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

[27] T.M. Kemp, R.M. Montoye, J.D. Harper, J.D. Palmer, and D.J. Auerbach. A decompression core for PowerPC. *IBM Journal of Research and Development*, 42(6), Nov 1998.

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

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

[30] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

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

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

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

[34] H. Lekatsas, J. Henkel, and W. Wolf. Design and simulation of a pipelined decompression architecture for embedded systems. In *Proc. ACM/IEEE Int'l Symp. on System Synthesis*, October 2001.

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

[36] Haris Lekatsas, Jörg Henkel, and Venkata Jakkula. Design of an one-cycle decompression hardware for performance increase in embedded systems. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 34–39, New York, NY, USA, 2002. ACM.

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

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

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

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

[41] Advanced RISC Machines Ltd. AMBA Specification (Rev. 2.0). *http://www.arm.com*, 1999.

[42] George L. Nemhauser and Laurence A. Wolsey. *Integer and combinatorial optimization*. Wiley-Interscience, New York, NY, USA, 1988.

[43] E. Wanderley Netto, R. Azevedo, P. Centoducatte, and G. Araujo. Multi-profile based code compression. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 244–249, New York, NY, USA, 2004. ACM.

[44] Eduardo Bráulio Wanderley Netto. *Compressão de Código Baseada em Multi-Profiles.* PhD thesis, Instituto de Computação - UNICAMP, 2004.

[45] E.W. Netto, R. Azevedo, P. Centoducatte, and G. Araujo. Multi-profile instruction based compression. *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, pages 23–29, 27-29 Oct. 2004.

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

[47] Richard Phelan. *Improving ARM Code Density and Performance: New thumb extensions to the ARM architecture*. Advanced RISC Machines Ltd., June 2003.

[48] Gaisler Research. Grlib ip user's manual. online, 2009. `http://www.gaisler.com`.

[49] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. Archc: a systemc-based architecture description language. pages 66–73, Oct. 2004.

[50] SPARC International Inc. *SPARC V8 Architecture Manual*, revision sav080si9308 edition, 1992.

[51] MIPS Technologies. *microMIPS Instruction Set Architecture: Uncompromised performance, mininum system cost*. Silicon Graphics MIPS Group, October 2009.

[52] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, 1984.

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

[54] Waine Wolf. *Computer as Components: Principles of embedded system design*. Morgan Kaufmann.

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

Apêndice A

Lista de instruções SPARC16

As tabelas a seguir apresentam as codificações dos *opcodes* das instruções SPARC16. As instruções das tabelas A.2, A.6 e A.10 utilizam os códigos de condição da unidade de inteiros (abreviados de *icc*[1]).

Tabela A.1: Instruções do Formato I

Mnemônico	**Opcode$_1$**	**Descrição**
call	00000	Chamada de procedimento.
call with exchange	10000	Chamada de procedimento com troca de modo.

Tabela A.2: Instruções do Formato IB

Mnemônico	**Opcode$_1$**	**Descrição**
b	01000	Salta sempre.
be	00001	Salta se for igual.
bne	01001	Salta se não for igual.

[1]Do inglês, *Integer Condition Codes.*

Tabela A.3: Instruções do Formato RI

Mnemônico	**Opcode**$_1$	**Descrição**
cmp	01101	Compara e altera os *condition codes* da unidade inteira.
mov	01110	Move constante para registrador.

Tabela A.4: Instruções do Formato RRI

Mnemônico	**Opcode**$_1$	**Descrição**
add	00010	Soma.
and	00011	Operação E lógico.
ld	11000	Carga de *word* da memória.
st	01100	Armazenamento de *word* na memória.
sra	00111	Deslocamento à direita aritmético.
srl	00110	Deslocamento à direita lógico.
sll	00101	Deslocamento à esquerda lógico.

Tabela A.5: Instruções do Formato LWST

Mnemônico	**Opcode**$_1$	**Opcode**$_2$	**Descrição**
ldd	11011	0	Carga de *double word*.
std	11011	1	Armazenamento de *double word*.
stb	01111	0	Armazenamento de *byte*.
sth	01111	1	Armazenamento de *half word*.
ldsb	10100	0	Carga de *byte*. Bit de sinal é estendido.
ldub	10100	1	Carga de *byte*.
ldsh	00100	0	Carga de *half word*. Bit de sinal é estendido.
lduh	00100	1	Carga de *half word*.

Tabela A.6: Instruções do Formato I2

Mnemônico	**Opcode**$_1$	**Opcode**$_2$	**Descrição**
bn	11100	00	Salta (nunca).
ble	11100	10	Salta se for menor ou igual.
bl	11100	11	Salta se for menor.
bleu	11101	00	Salta se for menor ou igual (comparação não sinalizada).
bcs	11101	01	Salta se o *bit* de *carry* do *icc* possuir o nível lógico 1.
bneg	11101	10	Salta se o último resultado calculado for negativo.
bvs	11101	11	Salta se o *bit* de *overflow* possui o nível lógico 1.
bg	11110	10	Salta se for maior.
bge	11110	11	Salta se for maior ou igual.
bgu	11111	00	Salta se for maior (comparação não sinalizada).
bcc	11111	01	Salta se o *bit* de *carry* do *icc* estiver no nível lógico 0.
bpos	11111	10	Salta se o último resultado calculado for positivo.
bvc	11111	11	Salta se o *bit* de *overflow* do *icc* estiver no nível lógico 1.
savesp	11100	01	SAVE (desliza a janela de registradores) que utiliza o *stack pointer* de forma implícita. O imediato é deslocado 4 bits à esquerda (multiplicado por 16).
bx	11110	01	Salta para o endereço alvo e troca o modo de execução para SPARCv8.

Tabela A.7: Instruções do Formato RI2

Mnemônico	**Opcode**$_1$	**Opcode**$_2$	**Descrição**
addfp	10011	000	Soma.
stfp	10011	001	Armazenamento de *word*.
ldfp	10011	010	Carga de *word*;
addsp	10011	011	Soma.
stsp	10011	100	Armazenamento de *word*.
ldsp	10011	101	Carga de *word*.
btst	10011	110	Testa os bits do registrador e altera os *condition codes* da unidade inteira.
clearword	10011	111	Palavra da memória cujo endereço é dado pela soma do registrador com o imediato é zerada.

Tabela A.8: Instruções do Formato RRI2

Mnemônico	**Opcode**$_1$	**Opcode**$_2$	**Descrição**
addx	11001	000	Soma. (Utiliza o *bit* de *overflow*dos *integer condition codes*).
subx	11001	001	Subtração. (Utiliza o *bit* de *overflow*dos *integer condition codes*).
smul	11010	000	Multiplicação com sinal.
sdiv	11010	001	Divisão com sinal.
umul	11010	010	Multiplicação sem sinal.
udiv	11010	011	Divisão sem sinal.
or	11010	100	OU lógico.
xor	11010	101	OU exclusivo.
orn	11010	110	ORN lógico.
xnor	11010	111	XNOR lógico.

Tabela A.9: Instruções do Formato RR

Mnemônico	**Opcode**$_1$	**Opcode**$_2$	**Descrição**
addx	01010	00100	Soma. (Utiliza o *bit* de *overflow*dos *integer condition codes*).
subx	01010	01100	Subtração. (Utiliza o *bit* de *overflow*dos *integer condition codes*).
btst	01010	11000	Testa os bits do registrador e altera os *condition codes* da unidade inteira.
clearword	01010	11010	Palavra da memória cujo endereço é dado pela soma do registrador com o imediato é zerada.
clearbyte	01010	01000	*Byte* da memória cujo endereço é dado pela soma do registrador com o imediato é zerada.
clearhalf	01010	11110	*Half word* da memória cujo endereço é dado pela soma do registrador com o imediato é zerada.
cmp	01010	10100	Compara e altera os *condition codes* da unidade inteira.
stb	01010	00101	Armazenamento de *byte*.
sth	01010	00110	Armazenamento de *half word*.
lduh	01010	00010	Carga de *half word*.
ldsh	01010	01010	Carga de *half word*, *bit* de sinal é estendido.
ldub	01010	00001	Carga de *byte*.
ldsb	01010	01010	Carga de *byte*, *bit* de sinal é estendido.
sra	01010	10111	Deslocamento à direita aritmético.
srl	01010	10110	Deslocamento à direita lógico.
sll	01010	10101	Deslocamento à esquerda lógico.
orn	01010	11100	ORN lógico.
xnor	01010	11011	XNOR lógico.
andn	01010	01101	ANDN lógico.
restore	01010	00000	Restaura a janela de registradores.
smul	01010	00011	Multiplicação sinalizada.
sdiv	01010	00111	Divisão sinalizada.
umul	01010	10001	Multiplicação não sinalizada.
udiv	01010	10010	Divisão não sinalizada.
xor	01010	10011	XOR lógico.

Tabela A.10: Instruções do Formato RR3 (*traps*)

Mnemônico	**Opcode$_1$**	**Opcode$_2$**	**Opcode$_3$**	**Descrição**
ta	01010	01011	000	Causa uma *trap* (sempre).
tn	01010	01011	001	Causa uma *trap* (nunca).
tne	01010	01011	010	Causa uma *trap* caso não seja igual.
te	01010	01011	011	Causa uma *trap* caso seja igual.
tg	01010	01011	100	Causa uma *trap* caso seja maior.
tle	01010	01011	101	Causa uma *trap* caso menor ou igual.
tge	01010	01011	110	Causa uma *trap* caso seja maior ou igual
tl	01010	01011	111	Causa uma *trap* caso seja menor.
tgu	01010	10000	000	Causa uma *trap* caso seja maior (não sinalizado)
tleu	01010	10000	001	Causa uma *trap* caso seja menor ou igual (não sinalizado)
tcc	01010	10000	010	Causa uma *trap* caso o *bit* de *carry* do *icc* esteja no nivel lógico 0.
tcs	01010	10000	011	Causa uma *trap* caso o *bit* de *carry* do *icc* esteja no nivel lógico 1.
tpos	01010	10000	100	Causa uma *trap* caso o últimno resultado computado seja positivo.
tneg	01010	10000	101	Causa uma *trap* caso o último resultado computado seja negativo.
tvc	01010	10000	110	Causa uma *trap* caso o *bit* de *overflow* esteja no nível lógico 0.
tvs	01010	10000	111	Causa uma *trap* caso o *bit* de *overflow* esteja no nível lógico 1.

Tabela A.11: Instruções do Formato RR3

Mnemônico	**Opcode$_1$**	**Opcode$_2$**	**Opcode$_3$**	**Descrição**
rdy	01010	01111	000	Lê o registrador Y
rdpsr	01010	01111	001	Lê o registrador PSR
rdwim	01010	01111	010	Lê o registrador WIM
rdtbr	01010	01111	011	Lê o registrador TBR
wry	01010	01110	000	Escreve no registrador Y
wrdpsr	01010	01110	001	Escreve no registrador PSR
wrwim	01010	01110	010	Escreve no registrador WIM
wrtbr	01010	01110	011	Escreve no registrador TBR
callreg	01010	11111	000	Salta para o endereço do registrador.
jumpreg	01010	11111	001	Salta para o endereço do registrador.
ret	01010	11111	010	Retorna de procedimento.
retl	01010	11111	011	Retorna de procedimento folha.
trivialrestore	01010	11111	100	*Restore* trivial.
jumpregx	01010	11111	101	Salta para o endereço do registrador e troca o modo para SPARCv8.
callregx	01010	11111	111	Salta para o endereço do registrador e troca o modo para SPARCv8.

Tabela A.12: Instruções do Formato RRR

Mnemônico	**Opcode$_1$**	**Opcode$_2$**	**Descrição**
add	10101	00	Soma
sub	10101	01	Subtração.
and	10101	10	E lógico
or	10101	11	OU lógico
st	10111	00	Armazenamento de *word.*.
ld	10111	01	Carga de *word.*
std	10111	10	Armazenamento de *double word.*
ldd	10111	11	Carga de *double word.*

Tabela A.13: Instruções do Formato MOV

Mnemônico	**Opcode$_1$**	**Opcode$_2$**	**T**	**Descrição**
mov32to8	11110	00	0	Move um registrador invisível para um registrador visível.
mov8to32	11110	00	1	Move um registrador visível para um registrador invisível.

Tabela A.14: Instrução SETHI

Mnemônico	**Opcode$_1$**	**Descrição**
SETHI	10110	Carrega os 22 bits mais significativos do registrador.

Tabela A.15: Instrução EXTEND

Mnemônico	**Opcode$_1$**	**Descrição**
—	01011	Carrega os bits mais significativos do imediato da instrução executada na sequência.