Projeto e Implementação de um Descompressor PDC-ComPacket em um Processador SPARC

Eduardo Afonso Billo

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Projeto e Implementação de um Descompressor PDC-ComPacket em um Processador SPARC

Eduardo Afonso Billo

Abril de 2005

Banca Examinadora:

- Prof. Dr. Rodolfo Jardim de Azevedo (Orientador)
- Prof. Dr. Eduardo Braulio Wanderley Netto
  CEFET-RN
- Prof. Dr. Célio Cardoso Guimarães
  Instituto de Computação - Unicamp
- Prof. Dr. Guido Costa Souza de Araújo (Suplente)
  Instituto de Computação - Unicamp

Projeto e Implementação de um Descompressor PDC-ComPacket em um Processador SPARC

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Eduardo Afonso Billo e aprovada pela Banca Examinadora.

Campinas, 25 de Abril de 2005.

Prof. Dr. Rodolfo Jardim de Azevedo
(Orientador)

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

Agradecimentos

Agradeço a Deus, que certamente me acompanha e ilumina em todos os momentos.

Ao professor Rodolfo Azevedo, por ter acreditado em mim, sempre confiante e crucial para vencer vários obstáculos durante toda a trajetória.

Aos professores Guido Araújo e Paulo Centoducatte pelas dicas, incentivos, também importantes ao trabalho.

À minha família e namorada Gabriela, pessoas que amo, que sempre me incentivaram, mesmo sabendo que isso custaria a nossa distância.

Aos meus amigos, especialmente aqueles que convivi no dia-a-dia, dividindo frustrações e alegrias: Felipe, Marcos, Patrick, Cleyton, Márcio, Matias.

Valeu a todos - "Chega de bla bla bla"!

À minha mãe,
Ao meu pai,
À Gabriela.

Resumo

É cada vez mais comum encontrar implementações de complexos sistemas dedicados em um único *chip* (telefones celulares, PDA's, etc.). Quanto mais complexos, maiores as dificuldades para atingir requisitos como área de silício ocupada, desempenho e consumo de energia. A compressão de código, inicialmente concebida para diminuir a memória ocupada, através da compactação do software, atualmente traz vantagens também no desempenho e consumo de energia do sistema, através do aumento da taxa de acertos à *cache* do processador. Este trabalho propõe o projeto de um descompressor de código, baseado na técnica PDC-ComPacket, implementado de forma integrada ao *pipeline* do Leon2 (SPARC V8). Chegou-se a uma implementação prototipada em FPGA, com razões de compressão (tamanho final do programa comprimido e do descompressor em relação ao programa original) variando entre 72% e 88%, melhora no desempenho de até 45% e redução de energia de até 35%, validado através de dois *benchmarks*: *MediaBench* e *MiBench*. Além disso, são apresentados uma série de experimentos que exploram os *tradeoffs* envolvendo compressão, desempenho e consumo de energia.

Abstract

Implementations of Complex Dedicated Systems on a single chip has become very common (cell-phones, PDA's, etc.). As complexity grows, also grows the required effort to reach constraints such as the silicon area, performance and energy consumption. The code compression, initially conceived to decrease the memory size, today also brings advantages in the performance and energy consumption of the system, due to an increase in the processor's cache hit ratio. This document proposes the design of a code decompressor, based on the PDC-ComPacket technique, embedding it into the Leon2 (SPARC V8) pipeline. We have achieved a functional implementation on a FPGA, with compression ratios (compressed program plus decompressor size related to the original program) ranging from 72% to 88%, performance speed-up of up to 45% and a reduction on energy consumption of up to 35%, validated through two benchmarks: *MediaBench* e *MiBench*. In addiction, we present a bunch of experiments, exploiting the tradeoffs related to compression, performance and energy consumption.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A crescente demanda pelo uso de dispositivos móveis tem tornado cada vez mais comum a implementação de complexos sistemas em um único *chip*, os chamados *System-on-a-chip*. *PDAS*, telefones celulares e câmeras digitais são apenas alguns exemplos que fazem uso de SoCs. A complexidade desses sistemas faz com que os antigos processadores de 8/16 bits sejam cada vez menos usados, dando vez a processadores mais robustos de 32 bits, como os RISCs.

A implementação dos SoCs, que envolve, além do processador, um complexo subsistema de memória e E/S, antes tida como impossível, hoje tornou-se uma realidade graças às novas metodologias e técnicas que abordam os diversos níveis de abstração, desde o físico até o projeto de alto nível.

À medida que os sistemas tornam-se mais complexos, tornam-se também os softwares que neles operam. Os requisitos de projeto levam a softwares com funcionalidades como *multi-thread*, tratamento de exceções, etc., o que, muitas vezes, é conseguido com o uso de sistemas operacionais inteiros. Junto com a complexidade, aumenta também o tamanho do código, em contraposição aos restritos requisitos de memória de um SoC. Neste sentido, surge uma técnica de alto nível que procura comprimir o código em tempo de compilação. A descompressão, por sua vez, é feita em tempo de execução.

Embora as técnicas de compressão tenham surgido com a finalidade de diminuir o tamanho do código, com o tempo alguns pesquisadores verificaram que elas também poderiam trazer vantagens em relação ao desempenho e ao consumo de energia do sistema, outros dois requisitos necessários no projeto de SoCs. A partir do momento que o código está comprimido, mais instruções são buscadas em cada acesso a memória externa, diminuindo a penalidade paga devido a esses acessos. Uma melhora no desempenho,

aliado à diminuição nas atividades de transição nos pinos de acesso à memória, leva, por sua vez, a uma redução no consumo de energia do circuito.

Este trabalho visa projetar e implementar um hardware descompressor, baseando-se na técnica de compressão desenvolvida como tese de doutorado de Wanderley [1]. O hardware é implementado em um processador SPARC, de modo que as instruções são descomprimidas *on-the-fly* e entregues ao *pipeline* do processador.

Além de chegar a uma implementação funcional do descompressor, devidamente integrado ao processador e prototipado em uma FPGA, este trabalho provê resultados em termos de compressão, desempenho e consumo de energia, previamente descritos como vantagens das técnicas de compressão (nem todas as técnicas focam nestes três requisitos).

Complementando os objetivos mencionados acima, foram definidos mais alguns requisitos desejáveis em relação ao projeto do descompressor, destacados a seguir e novamente explorados em capítulo posterior.

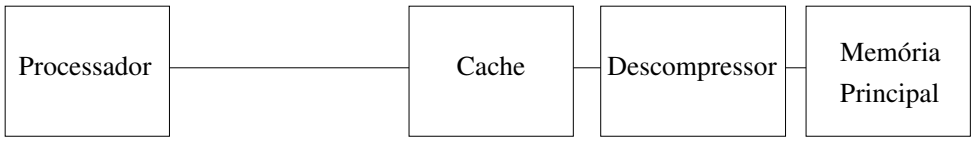
- Inserir o descompressor com o mínimo de alterações no processador original, usando o mínimo de hardware possível;
- O *cycle-time* do processador não deve ser degradado após a integração;
- Nenhuma bolha (ciclo extra) deve ser inserida devido ao descompressor.

1.1 Fundamentação Teórica

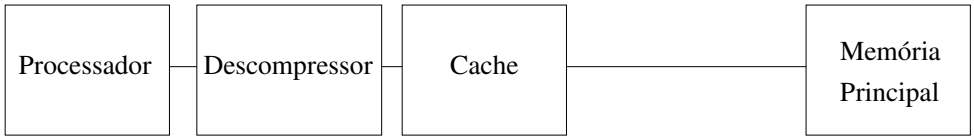
O principal assunto que deve ser fundamentado para permitir o posterior entendimento deste trabalho é o método de compressão usado, o PDC-ComPacket. *PDC* está relacionado ao tipo de arquitetura de descompressão associada, enquanto *ComPacket* é o nome dado à unidade comprimida, podendo ser traduzida como *pacote comprimido*.

1.1.1 Arquiteturas CDM X PDC

Na bibliografia relacionada, são encontradas dois tipos básicos de arquiteturas, *CDM* e *PDC*, identificando o posicionamento do descompressor em relação ao processador e subsistema de memória, como mostrado na Figura 1.1. A arquitetura *CDM* (*Cache-Descompressor-Memory*) indica que o descompressor está posicionado entre a *cache* e a memória principal, enquanto que a arquitetura *PDC* (*Processor-Decompressor-Cache*) posiciona o descompressor entre o processador e a *cache*.



(a) CDM



(b) PDC

Figura 1.1: Arquiteturas para descompressão de código

Existem algumas vantagens e desvantagens de cada uma das abordagens. As arquiteturas *PDC* geralmente oferecem melhores ganhos de desempenho, pois a taxa de acertos na *cache* é aumentada sensivelmente. Por outro lado, estas arquiteturas são mais difíceis de serem implementadas sem aumento no *cycle-time* do processador. A questão do desempenho foi um dos grandes motivadores para a escolha de uma arquitetura *PDC*.

1.1.2 Método de Compressão

Na computação em geral, faz-se uso de compressão de maneira indiscriminada, muitas vezes sem se perceber. Um arquivo *MP3* para compressão de aúdio, um *JPEG* para imagens, um *MPEG* para vídeos, ZIP para arquivos em geral, dentre outros, são exemplos de casos onde a compressão é bastante útil. Para se ter idéia, o formato MP3, que hoje é tão comumente usado, consegue tornar o arquivo comprimido dezenas de vezes menor do que o original, viabilizando novidades tecnológicas como tocadores de MP3 com formatos semelhantes a uma pequena caneta.

A maioria destes formatos segue uma mesma metodologia: eliminar as redundâncias ou repetições, procurando representá-las de uma maneira mais econômica. Imagine, por exemplo, um arquivo que representa a seguinte informação:

- AAABBBBCCCCCDDDDDD

Note que existe uma grande repetição de símbolos. Se cada letra fosse representada por 1 byte, seriam necessários 18 bytes. Agora, veja a seguinte representação da mesma

informação, formada por pares, onde o primeiro elemento identifica a letra, e o segundo identifica o número de repetições da mesma:

- A3B4C5D6

A mesma informação é representada, mas com 8 bytes. Embora o caso acima seja bastante didático, casos semelhantes se repetem na prática: uma imagem *BMP*, com uma grande área não pintada (em branco), ou um vídeo onde, de um quadro para outro, há pouquíssima movimentação do cenário, etc.

O método ComPacket segue exatamente a mesma idéia. Ele procura instruções que se repetem com bastante freqüência no código e as substitui por pequenos índices para um dicionário com estas instruções. Isto é feito posteriormente à compilação de um programa, seguindo os passos abaixo:

- Faz-se um ordenamento das instruções de acordo com o número de vezes que elas aparecem no código;
- Cria-se um dicionário com as instruções que mais aparecem no código (por exemplo, apenas as 256 instruções mais freqüentes);
- Substitui-se as instruções escolhidas por índices para o dicionário, o que comprime o código.

Um programa comprimido assumirá uma codificação completamente diferente da original, de modo que, se for submetido à execução, inevitavelmente, as instruções não serão reconhecidas, pois o processador não está preparado para isto. Neste ponto entra a arquitetura de descompressão, assunto principal deste trabalho, que visa estender e modificar o processador, para permitir que ele realize a descompressão em tempo de execução.

Para finalizar a fundamentação teórica, cabe definir o termo *razão de compressão*, que indica o quão comprimido um código está em relação a sua versão original. A seguinte equação define *razão de compressão*:

$$razão\ de\ compressão = \frac{\begin{array}{c}tamanho\ do\ código\ comprimido\ + \\ tamanho\ do\ dicionário\end{array}}{tamanho\ do\ código\ original}\ x100 \qquad (1.1)$$

Ao longo do trabalho, sempre que se falar em *razão de compressão*, estar-se-á levando em conta esta equação (exceto em alguns trabalhos correlatos, onde o dicionário não é levado em conta).

1.2 Contribuições

Neste trabalho, é proposta uma arquitetura de descompressão para o método PDC-ComPacket[1], chegando à implementação funcional em uma FPGA.

O projeto segue a arquitetura PDC, sendo que o descompressor é embutido ao sistema de *refill* da *cache* de instruções, de maneira que funcionalidades como *bursting* continuam possíveis.

A arquitetura foi implementada no LEON2, um processador bastante robusto, totalmente compatível com a versão 8 da especificação SPARC, usado intensamente tanto no meio acadêmico quanto comercial.

Em relação aos trabalhos correlatos, é a única arquitetura que permite descompressão e *bursting* ao mesmo tempo. Além disso, oferece um dos melhores ganhos de desempenho e consumo de energia em comparação aos demais trabalhos e, naturalmente, possibilitando a compressão do código.

1.3 Organização

Este trabalho está organizado da seguinte forma:

- No Capítulo 2 são apresentados os principais trabalhos correlatos encontrados na literatura.
- No Capítulo 3 é descrito o método PDC-ComPacket em detalhes.
- No Capítulo 4 é apresentado e detalhado o projeto da Arquitetura de Descompressão.
- No Capítulo 5 são avaliados os resultados sob três ângulos: compressão, desempenho e consumo de energia.
- No Capítulo 6 são apresentadas as conclusões e eventuais trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Neste capítulo, serão mostradas algumas arquiteturas para execução de códigos comprimidos, encontradas na literatura. Diferentemente do trabalho de Wanderley [1], em que a ênfase está no estudo de técnicas de compressão, este trabalho está focado em detalhes arquiteturais destas técnicas.

2.1 Técnicas de Compressão

Existe uma série de técnicas de compressão propostas na bibliografia. A compressão pode ser alcançada no próprio projeto do processador, utilizando campos menores para codificar instruções mais freqüentes e campos maiores para as instruções pouco freqüentes, como foi o caso do projeto do Borroughs [2] e, posteriormente, do VAX [3].

Outras abordagens comprimem o código para uma forma executável ou interpretável, geralmente, a partir de uma representação do código (árvores de sufixos, por exemplo [4, 5, 6]). Partindo dessa idéia, Fraser e Hanson [7] adaptaram o compilador C (*lcc*) para produzir código compacto e um interpretador para este código, em uma arquitetura SPARC. Posteriomente, Ernst [8] repetiu a técnica para o RISC, chamando o novo processador de BRISC (*Byte-coded RISC*). Obviamente que, nestes casos, a partir do momento que o código passa a ser interpretado, uma alta penalidade no tempo de execução é imposta.

A compressão de código pode ainda ser atingida através de otimizações no compilador, evitando transformações como *procedure inlining* e *loop unrolling* [9].

Liao [10], por sua vez, propõe que sejam encontradas seqüências de instruções comuns

(chamadas de sub-rotinas), inserindo-nas em um dicionário. A seguir, estas sub-rotinas são substituídas no código por uma chamada CALL, de modo que a sub-rotina é executada e ao final, há uma instrução de RET para retornar ao ponto onde foi chamada.

Seguindo a mesma linha do Borroughs e VAX, alguns processadores atuais de 32 bits têm suas versões de 16 bits, como é o caso do Thumb [11] (versão de 16 bits do ARM) e do MIPS16 [12] (versão de 16 bits do MIPS). Para conseguir tal redução, ambas implementações fazem uso de alguns artifícios, como reduzir número de registradores especificados (4 bits para 3 bits), reduzir o número de registradores por instrução, etc. Em tempo de execução, existem hardwares descompressores que convertem as instruções comprimidas para o seu formato original, permitindo que o *core* original seja utilizado, ARM ou MIPS.

Exceto no caso do Thumb e MIPS16, todas as demais técnicas descritas rapidamente acima não necessitam de uma arquitetura implementada em hardware para execução do código comprimido, ou uma *arquitetura de descompressão*, como será usado daqui por diante. Nesta seção o objetivo foi apenas ter uma idéia bastante geral das técnicas existentes na literatura. A seguir, serão detalhadas aquelas técnicas semelhantes a este trabalho, que fazem uso de arquiteturas de descompressão CDM e PDC, definidas no capítulo anterior.

2.2 Arquiteturas CDM

Wolfe e Chanin [13] desenvolveram o CCRP (*Compressed Code RISC Processor*), aplicando um método de compressão bastante consagrado: código de Huffman [14]. Os códigos Huffman foram gerados através da análise de um histograma de ocorrências de bytes do programa. No que se refere à descompressão, o mecanismo proposto é inovador em alguns aspectos: foi o primeiro hardware descompressor implementado em um processador RISC (MIPS R2000) e foi a primeira técnica a usar as falhas de acesso à *cache* para acionar o mecanismo de descompressão.

A unidade de compressão usada é a linha da *cache* de instruções. A cada falha de acesso à *cache*, as instruções são buscadas na memória principal, descomprimidas e alimentam a linha da *cache* onde houve a falha. O fato de o CCRP já fazer a descompressão das instruções antes de armazená-las na *cache* é vantajoso, no sentido que os endereços de saltos contidos na *cache* são os mesmos do código original. Isto resolve a maioria dos problemas de endereçamento, não havendo necessidade de recorrer a artifícios como:

colocar hardware extra no processador para tratamento diferenciado dos saltos, fazer *patches* de endereços de salto, etc. Porém, ainda existe um problema de endereçamento ao usar este tipo de técnica: no caso de uma falha de acesso à *cache*, é necessário buscar na memória principal as instruções requisitadas, as quais têm endereços diferentes das instruções da *cache*, já que a memória contém código comprimido, enquanto a *cache* contém código não comprimido. Este problema é resolvido com uma tabela de tradução de endereços, chamada LAT (*Line Address Table*), que mapeia os endereços não comprimidos da *cache* em endereços na memória principal.

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

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

A Figura 2.1 mostra a arquitetura de descompressão do CCRP. Note que o descompressor está posicionado entre a I-cache e a memória principal, de maneira que as instruções armazenadas na *cache* já estão descomprimidas e prontas para serem usadas pelo processador. Desta forma, quando o processador solicita a próxima instrução e há um acerto na I-cache, a instrução é imediatamente repassada, sem necessidade de acionamento do mecanismo de descompressão. Entretanto, no caso de uma falha, o mecanismo de descompressão (*cache refill engine*) procura na CLB pelo endereço solicitado. Se houver um acerto na CLB, o endereço comprimido correspondente é usado para buscar a instrução na memória, que é então descomprimida e repassada à I-cache e processador. No caso de falha na consulta à CLB, é necessário buscar o endereço na LAT. Após isso, o endereço comprimido obtido da LAT é atualizado na CLB e usado para recuperar a instrução na memória principal, que é então descomprimida e repassada à I-cache e processador.

O descompressor consegue descomprimir 2 bytes por ciclo. Portanto, para descomprimir os 32 bytes de uma linha da *cache*, são usados no mínimo 16 ciclos de

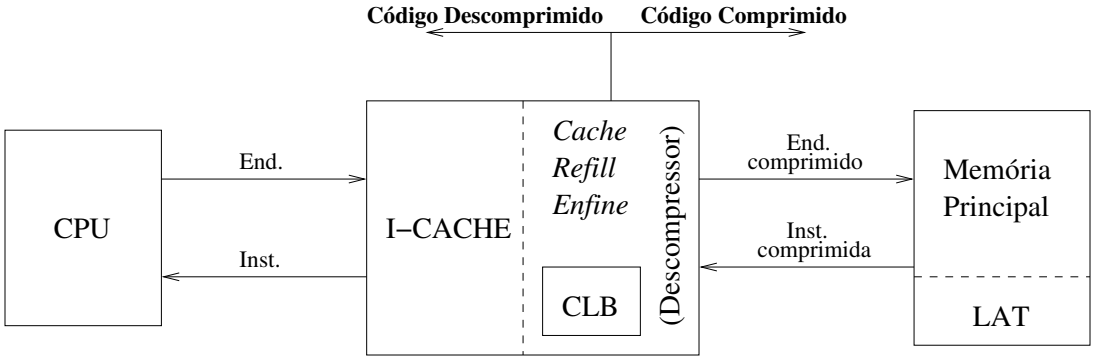


Figura 2.1: Descompressor do CCRP

relógio, podendo este número aumentar no caso de memórias lentas, onde o mecanismo de descompressão não é o gargalo. Para conseguir esta velocidade de descompressão, o autor optou por usar uma lógica *hardwired*, o que traz resultados bastantes satisfatórios quando são usadas boas ferramentas de síntese. Outras alternativas poderiam ser usadas, como ROM's ou PLA's, mas como o fator limitante é, de fato, o tempo de descompressão, esta foi a implementação escolhida.

A técnica mostrou razões de compressão de 73%, em média. Para modelos de memórias mais lentos, o desempenho do processador foi na maioria das vezes suavemente melhorado. Para modelos mais rápidos, o desempenho sofreu uma leve degradação.

Uma implementação do CCRP foi feita por Benes, Wolfe e Nowick [15] em 1997, com o objetivo de investigar o custo de um descompressor Huffman. Os melhores resultados apontam uma taxa de 32bits/39ns em um processo MOSIS CMOSX 0,8mm com 3 camadas de metal. A área total do descompressor ocupa 0,75mm^2, o equivalente a aproximadamente 3Kbytes de ROM. Mais tarde, em 1998, os mesmos autores conseguiram melhorar o desempenho do descompressor para 32bits/25ns [16].

Um outro trabalho interessante sugerindo uma arquitetura CDM é o CodePack [17][18], aplicado em processadores PowerPC. O método de compressão tira proveito do formato de instrução do PowerPC. O autor verificou que a parte alta da instrução (16 bits mais significativos) possui muitas diferenças de codificação em relação a parte baixa (16 bits menos significativos) e resolveu construir dois dicionários distintos, um para cada parte da instrução, trazendo melhores resultados de compressão.

A codificação usada é mostrada na Figura 2.2. Cada símbolo consiste de um ou dois campos (cada bit é representado por n). Os oito símbolos mais freqüentes recebem a codificação mais densa, 00_2. Os próximos 32 recebem a codificação 01_2, e assim por diante,

até os menos freqüentes, que recebem sua codificação original (16 bits não comprimidos).

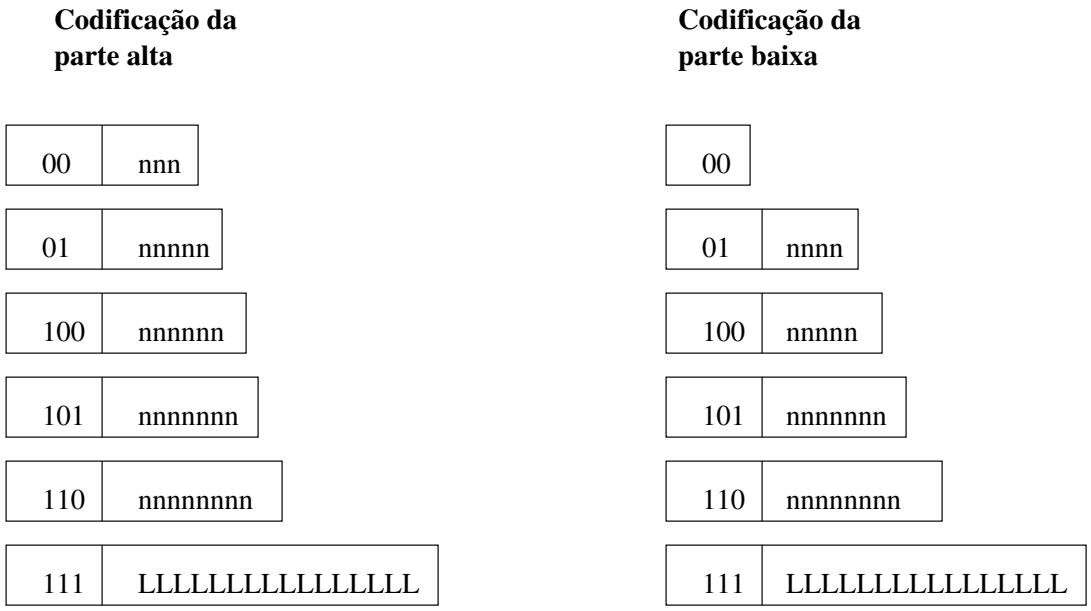


Figura 2.2: Codificação de símbolos do CodePack

A arquitetura de descompressão é muito semelhante ao CCRP, fazendo parte do sistema de memória. Uma tabela de mapeamento é responsável por fazer o mapeamento de endereços não comprimidos (I-cache) em endereços comprimidos (memória principal). Uma implementação desta arquitetura, sem otimizações, consegue fornecer uma instrução ao processador a cada três ciclos de relógio, o que é uma taxa bastante lenta. Para melhorar o desempenho, o autor sugere que as tabelas de mapeamento apontem para blocos de instruções (16 instruções por bloco) em vez de manter uma entrada na tabela para cada instrução. Com isso, o descompressor pode criar estágios de descompressão na forma de um *pipeline*, o que melhora o *throughput* para quase uma instrução por ciclo (desconsiderando o tempo para encher o *pipeline*). A segunda otimização parte do princípio que o processador fica grande parte do tempo executando trechos de códigos seqüenciais (sem saltos). Baseado nisto, o descompressor já conhece de antemão a próxima linha da *cache* a ser buscada na memória, sem precisar consultar a tabela de mapeamento, o que leva a economia de um ciclo de relógio em cada *cache refill*. No caso de haver saltos, esta segunda abordagem naturalmente não pode ser usada.

O CodePack foi implementado no PowerPC usando a tecnologia IBM CMOS 5S ASIC, ocupando 25.484 células lógicas, mais 10.814 células lógicas referentes à tabela de símbolos (512 entradas). A razão de compressão alcançada ficou em torno de 60%.

Não é fornecido um número exato sobre desempenho, sendo apenas citado que, para taxas altas de acertos na I-cache, a queda de desempenho é muito pequena, mas que esse valor mostra-se considerável à medida que a taxa de acertos cai, devido ao atraso oferecido pelo hardware de descompressão.

Azevedo [19] sugere o IBC (*Instruction Based Compression*), onde é feita a divisão do conjunto de instruções do processador em classes, de acordo com o número de ocorrências e a quantidade de elementos de cada classe. O estudo feito mostra que, geralmente, os resultados de compressão são melhores para 4 classes de instruções. A compressão consiste em substituir o código original por pares no formato [prefixo, *codeword*], onde o prefixo indica a classe da instrução e o *codeword* é um índice para uma tabela de instruções. Além disso, assim como nos métodos descritos anteriormente, este método necessita de uma tabela de conversão de endereços descomprimidos (instruções da I-cache) para endereços comprimidos (memória principal).

O autor faz duas implementações do método, para o processador MIPS e para o processador SPARC, com arquiteturas de descompressão semelhantes. A Figura 2.3 mostra a implementação do descompressor para o SPARC V8 LEON [20]. A descompressão é realizada em 4 estágios de *pipeline*. O primeiro estágio, chamado de *Input*, é responsável por converter o endereço do processador (código não comprimido) para o endereço da memória principal. O segundo estágio, *Fetch*, é responsável por fazer a busca da palavra comprimida na memória principal. O terceiro estágio, *Decode*, é o que efetivamente decodifica os *codewords*. O quarto estágio, *Output*, consulta a tabela de instruções e fornece a instrução descomprimida ao processador.

A razão de compressão deste método ficou em 53,6% para a arquitetura MIPS e 61,4% para arquitetura SPARC. Houve uma perda de desempenho de 5,89%, em média.

2.3 Arquiteturas PDC

Lekatsas [21] propõe um método de compressão onde as instruções são agrupadas em quatro grupos distintos para a arquitetura SPARC:

- Grupo 1 - Instruções com imediatos
- Grupo 2 - Instruções de salto
- Grupo 3 - Instruções de acesso rápido

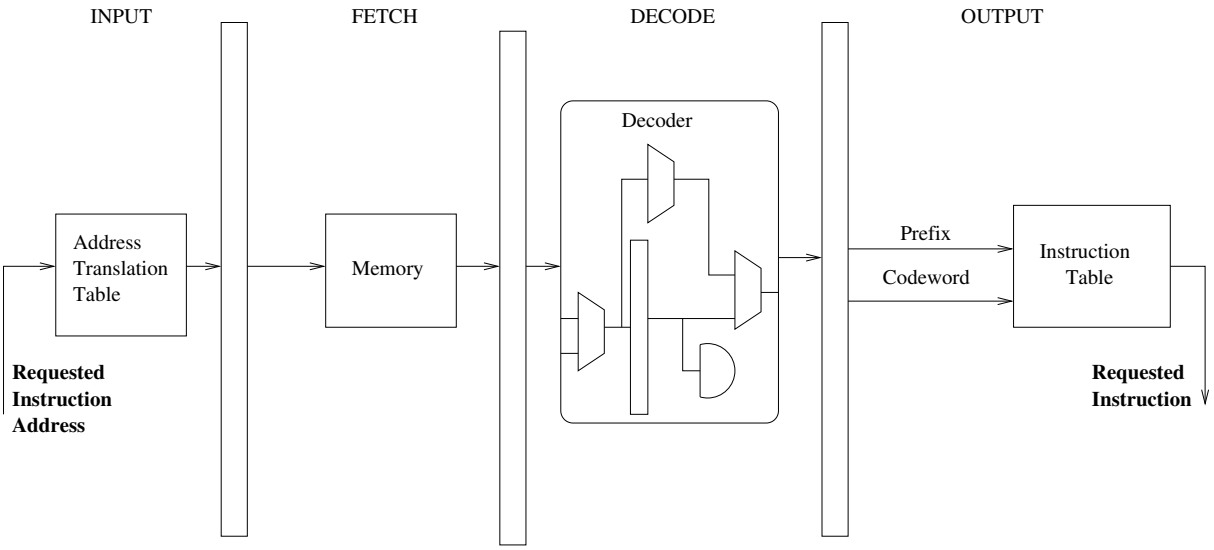


Figura 2.3: Descompressor IBC para o LEON

- Grupo 4 - Instruções não comprimidas

Cada grupo é identificado por uma seqüência de bits: grupo 1 = 0_2, grupo 2 = 11_2, grupo 3 = 100_2 e grupo 4 = 101_2, de modo que o hardware possa fazer a descompressão adequada de acordo com o grupo da instrução, já que cada grupo tem sua codificação própria: no grupo 1, usa-se codificação aritmética (este tipo de codificação tem vantagens significativas sobre a conhecida codificação de Huffman, especialmente adequada quando as ocorrências são bastante espaçadas. Geralmente, a tabela de codificação é mais genérica e produz uma razão de compressão melhor do que a codificação de Huffman [22]). Instruções do grupo 2 são comprimidas através da compactação do campo de deslocamento. Para obter a compactação, codifica-se apenas a mínima quantidade de bits necessária para representar o deslocamento em questão. Instruções do grupo 3 são índices para um dicionário de instruções e, como podem ser descomprimidas com um simples acesso a um dicionário, em apenas um ciclo do processador, são chamadas de *instruções de acesso rápido*.

Como continuação deste trabalho [23], Lekatsas sugere uma Arquitetura de descompressão para este método. O descompressor é localizado entre a *cache* e a CPU, com a intenção de aumentar a largura de banda entre processador/memória e, desta forma, alcançar três grandes objetivos: ganho de desempenho, diminuição de consumo de energia e diminuição da área do sistema como um todo. O descompressor é combinado com o *pipeline* do processador como mostra a Figura 2.4. São adicionados quatro *pipelines*, um

para cada grupo de instruções descritos acima. Antes da unidade de descompressão existe um buffer de entrada (I-Buf), que tem como objetivo guardar as instruções comprimidas vindas da *cache* de instruções. Após a unidade de descompressão existe um buffer de saída (O-Buf), com o objetivo de guardar as instruções descomprimidas e fornecê-las ao *pipeline* do processador a medida que são requisitadas. Um ponto importante nesta arquitetura é que ela permite que os quatro *pipelines* executem ao mesmo tempo, o que traz uma dificuldade extra na modelagem do hardware descompressor. Pode ocorrer o caso de uma instrução B (que vem a seguir de uma instrução A no programa) ser descomprimida mais rapidamente que a instrução A, pois é do grupo 3 (descompressão em apenas um ciclo). Isto faz com que as instruções chegem ao O-Buf fora de ordem, implicando em hardware extra para controle e entrega ordenada das instruções ao processador.

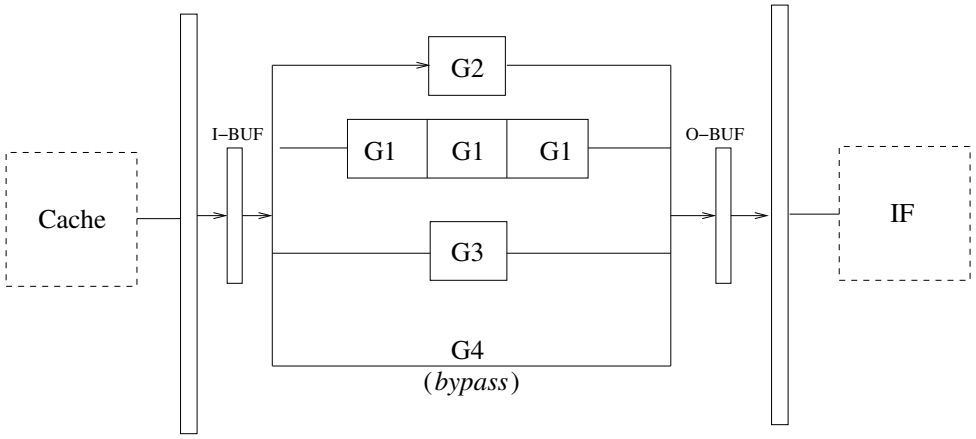


Figura 2.4: Decompressor *pipelined* sugerido por Lekatsas

Os resultados encontrados após a conclusão destes dois trabalhos mostram que, em média, 53% das instruções pertencem ao grupo 1, 26,7% ao grupo 2, 19,7% ao grupo 3 e apenas 0,6% ao grupo 4 e, portanto, não são comprimidas. A razão de compressão ficou em torno de 65%. Cabe lembrar que esses valores não levam em conta o *overhead* do hardware descompressor, que é bastante complexo, o que impossibilita usar esta métrica para comparação com outros métodos. Além disso, houve uma estimativa de redução no consumo de energia de 28% e se alcançou em média 25% de ganho de desempenho.

Em um outro trabalho [24], Lekatsas propõe uma outra arquitetura de descompressão. Neste trabalho, o objetivo é projetar um descompressor de apenas um ciclo, sem alterar o tempo de ciclo do processador alvo, o Xtensa-1040[25]. Para isso, é utilizado um método de compressão mais simples que o anterior, completamente baseado em dicionários. A partir

de estatísticas do programa, as instruções que mais aparecem no código são substituídas por índices para uma tabela de códigos (dicionário de 256 entradas). As instruções de 24 bits do Xtensa-1040 podem receber dois tipos de codificação, de 8 bits e 16 bits. No caso da codificação de 8 bits, o código inteiro é usado para endereçar o dicionário. Quando a codificação de 16 bits é usada, apenas os 8 bits mais significativos são usados para endereçamento do dicionário, enquanto os 8 bits menos signitivativos são usados para identificar o tipo de instrução (se é comprimida e o seu tamanho). Após comprimido, é feito *patching* no código para correção dos *offsets* dos saltos, sendo que, para efeitos de simplificação, os alvos dos saltos são forçados a se manterem alinhados a palavras.

A ordem de descompressão das palavras pode ser esquematicamente representada por uma árvore (Figura 2.5). O nó *raiz* da árvore recebe instruções comprimidas de 32 bits. A partir daí, este *stream* de bits vai passando por nós da árvore, até chegar a uma *folha* da árvore, que contém de fato uma instrução descomprimida. Cada nó (ou bifurcação) gera duas ou mais instruções, ou seja, realiza de fato a descompressão (através de consultas ao dicionário). O processador espera receber *bitstreams* de 32 bits das instruções originais. A estrutura de decodificação de árvore proposta garante que sempre que um nó *folha* for atingido, pelo menos 32 bits de instruções estarão disponíveis para o processador. Os bits excedentes aos 32 bits requeridos pelo processador são armazenados num buffer para uso no próximo ciclo.

Baseado no que foi exposto, o diagrama de blocos do hardware descompressor proposto é mostrado na Figura 2.6. O mecanismo de descompressão é ativado pela chegada de um *bitstream* de 32 bits. Inicialmente, o descompressor analisa os quatro bits menos significativos para selecionar as linhas corretas do multiplexador e determinar o tipo de instrução. Após isso, a tabela de descompressão é acessada, fornecendo uma instrução de 24 bits descomprimida na saída. Existem alguns casos que o *bitstream* de 32 bits contém apenas parte de uma instrução, sendo que o resto da instrução está contido no próximo *bitstream*. Para estes casos existem alguns registradores.

- **pre-part inst**: Guarda a porção descomprimida no ciclo anterior.
- **pre-part set**: Indica que uma porção da instrução atual já foi descomprimida no ciclo anterior.
- **Bytes-next**: Indica se 8 ou 16 bits de *pre-part inst* foram decodificados no ciclo anterior (*pre-part inst*)

No diagrama, as linhas tracejadas indicam caminhos alternativos da entrada do *bitstream* de 32 bits, até a instrução descomprimida.

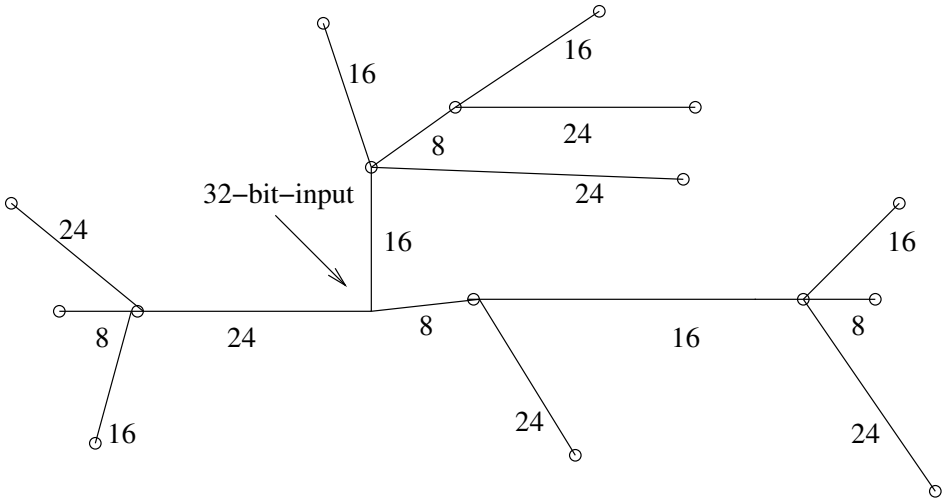


Figura 2.5: Árvore de descompressão para o descompressor de 1 ciclo de Lekatsas

Os resultados encontrados neste trabalho mostram que houve um ganho médio de desempenho de 25%, com diminuição do tamanho do código na média de 35%, sem considerar o descompressor.

Benini [26] propõe um método de compressão baseado em dicionário (256 palavras de 32 bits), utilizando como processador alvo o DLX. Contudo, a construção do dicionário é diferente dos métodos apresentados anteriormente. Em vez de levar em conta as estatísticas com que cada instrução aparece no código, este método leva em conta estatísticas de execução, ou seja, quantas vezes cada instrução foi executada. A unidade de compressão é a linha da *cache*, que o autor convenciona configurar com tamanho de 32 bytes. Para evitar o uso das tabelas de tradução de endereços vistas em técnicas anteriores, o autor exige que os endereços de destino estejam sempre alinhados a 32 bits (palavra), realizando *patching* para conversão dos *offsets* de saltos. A Figura 2.7 mostra a estrutura de uma linha comprimida. A primeira palavra da linha contém uma marca L e um conjunto de bits de *flag*. A marca é um código não usado pelo DLX, enquanto os *flags* são divididos em 12 grupos de 2 bits que servem para classificar como foi comprimido cada um dos bytes das três palavras restantes. Os *flags* podem assumir os seguintes valores: 00_2 se o byte correspondente contém uma instrução comprimida. 01_2 se o byte correspondente contém uma palavra não comprimida. 11_2 se o byte correspondente é de *padding*, para efeitos de alinhamento e 10_2 para sinalizar a última instrução comprimida

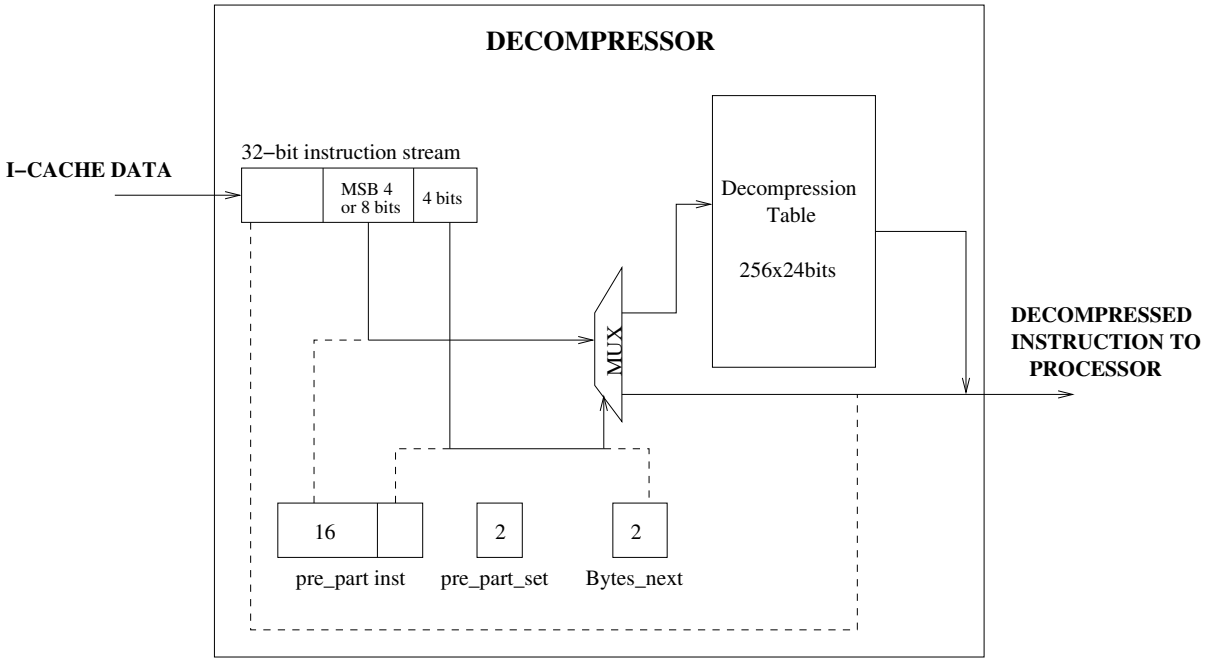


Figura 2.6: Descompressor de 1 ciclo sugerido por Lekatsas

da linha. O bit de *flag* L, indica se a linha é comprimida ou não.

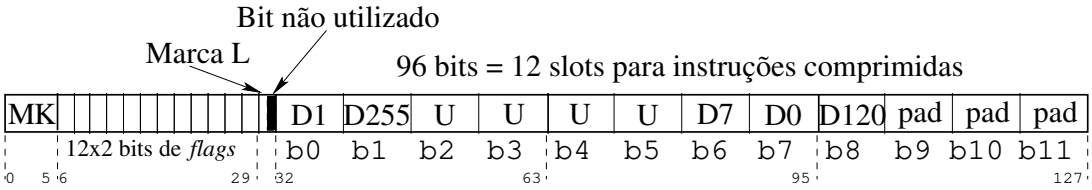


Figura 2.7: Estrutura da linha comprimida no método de compressão de Benini

O descompressor é integrado a I-cache do processador. O diagrama de blocos do descompressor proposto é mostrado na Figura 2.8. Ele contém os seguintes blocos:

- **controlador mestre**: é o controlador já conhecido em *caches*, responsável por analisar a TAG, verificar se houve acerto ou falha e fornecer a instrução correta ao processador. No caso de falha, o controlador mestre aciona o controlador escravo.
- **controlador escravo** (tratador de falhas): Responsável por buscar na memória principal as quatro palavras que alimentarão a linha da *cache* onde houve a falha.
- **lógica de atualização do PC**: Esta unidade contém um contador de 4 bits que

permite saber qual é a instrução que está sendo descomprimida da linha de *cache* comprimida.

- ***cache* de instruções**: É a memória de 256 linhas x 4 palavras da *cache*.
- **dicionário de instruções**: Contém o dicionário de instruções comprimidas.
- **MUX**: Seleciona entre a saída da memória de instruções (para instruções não comprimidas) ou dicionário (para instruções comprimidas).

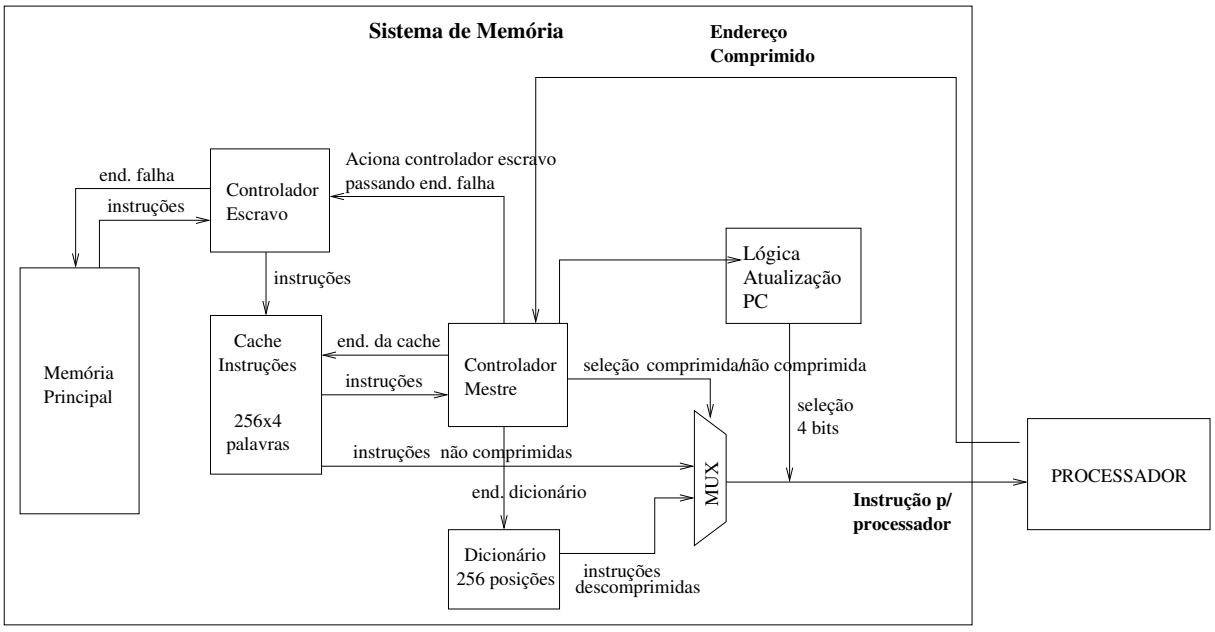


Figura 2.8: Descompressor para o DLX sugerido por Benini

Além do descompressor em si, o processador requer um hardware adicional de modo que o PC do processador seja congelado enquanto múltiplas instruções pertencentes a uma linha comprimida da *cache* seja descomprimida, de modo a manter sincronizado o PC do processador com a posição da instrução em execução.

O método proposto trouxe dimuinuição no tamanho do código de 28% em média e 30% de econômia de potência em média.

Lefurgy também propõe métodos baseados em dicionários. Entre os principais trabalhos, em [27] a compressão é realizada após a compilação do código fonte, examinando o programa em busca de seqüências de instruções que mais aparecem no código e as substituindo por índices (*codewords*) para um dicionário. O código final consiste de

codewords misturadas com instruções não comprimidas. O problema dos endereços dos saltos é tratado da seguinte forma: instruções com saltos relativos[1] não são comprimidas, o que facilita posteriormente o *patch* dos endereços não resolvidos. Saltos indiretos[2] são normalmente comprimidos, já que o alvo é obtido a partir de um registrador, porém, é necessário modificar as *jump tables* para que, em tempo de execução, o registrador contenha o valor correto do endereço alvo do salto. Além disso, o método permite que os endereços alvos estejam alinhados a 4 bits (tamanho de um *codeword*), e não ao tamanho da palavra do processador (32 bits), como usualmente feito pelos demais métodos propostos na bibliografia. Isto traz a vantagem de uma compressão melhor, mas a desvantagem de que o *core* do processador necessita de algumas alterações (hardware extra) para tratar saltos para endereços alinhados a 4 bits.

O autor não fornece muitos detalhes sobre o descompressor e sua integração com os processadores experimentados: PowerPC, ARM e i386. Tudo que é fornecido é o diagrama de blocos mostrado na Figura 2.9. Basicamente, a instrução é buscada da memória. Caso seja um *codeword*, a lógica de decodificação dos *codewords* obtem o deslocamento e tamanho do *codeword* que servirá como índice para acessar a instrução não comprimida no dicionário e repassar ao processador. No caso de instruções não comprimidas, elas são encaminhadas diretamente ao processador.

O método de Lefurgy consegue razões de compressão de 61%, 66% e 75% para as implementações no PowerPC, ARM e i386, respectivamente. Não são fornecidos valores de desempenho e consumo de energia.

2.4 Resumo das Técnicas

As Tabela 2.1 e Tabela 2.2 sumarizam a revisão feita neste capítulo. Estas tabelas serão novamente trabalhadas no Capítulo 5, visando uma comparação com os resultados encontrados adiante.

A técnica PDC-ComPacket, usada neste trabalho, não foi explicada neste momento, pois ela será descrita em detalhes no Capítulo 3.

[1] saltos relativos - instruções que fazem saltos relativos ao PC, ou branches

[2] saltos indiretos - instruções que saltam para o valor contino num registrador, chamado de registrador de indireção

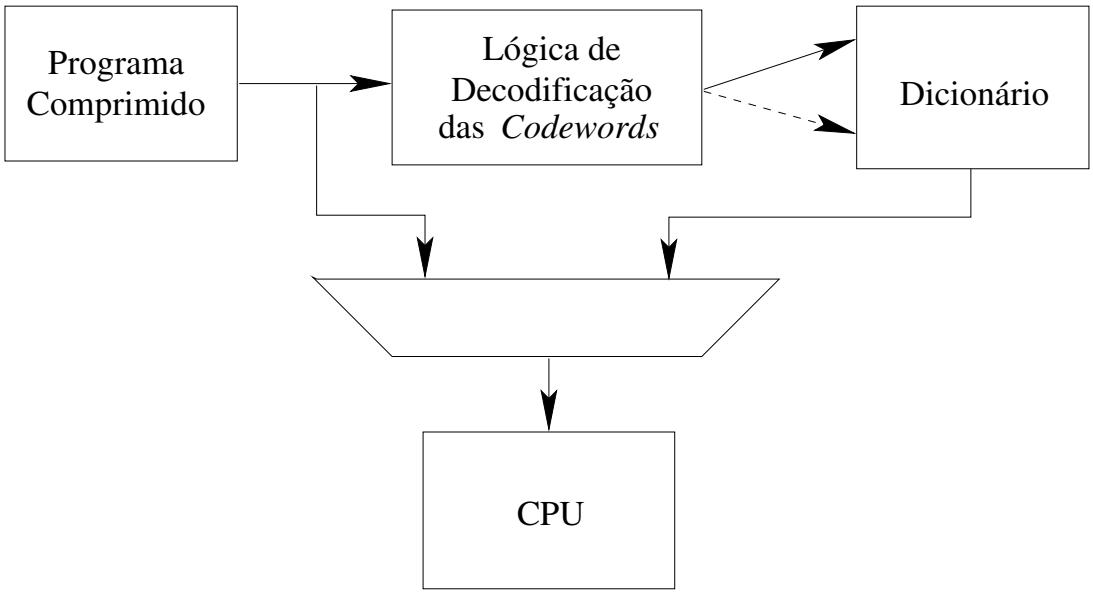


Figura 2.9: Descompressor sugerido por Lefurgy

Métodos CDM					
Autor	Arquitetura	*Benchmarks*	Razão de Compressão	Tempo de Execução	Redução Energia
Wolfe & Chanin[13]	MIPS	lex, pswarp, yacc, eightq, matrix25 lloop01, xlisp, espresso e spim	73%	–	–
IBM[17][18]	PowerPC	Mediabench SPEC95	60%	–	–
Azevedo[19]	SPARC	SPECint95	53,6%	+5,89%	–

Tabela 2.1: Resumo dos métodos CDM

Métodos PDC					
Autor	Arquitetura	*Benchmarks*	Razão de Compressão	Tempo de Execução	Redução Energia
Lekatsas[23]	SPARC	compress, diesel, i3d, key, mpeg smo, trick	65%	-25%	–
Lekatsas[24]	Xtensa 1040	compress, diesel, i3d, key, mpeg smo, trick	65%	-25%	-28
Benini[26]	MIPS	Ptolomy	72%	–	-30%
Lefurgy[27]	PowerPC	SPECint95	61%	–	–
Lefurgy[27]	ARM	SPECint95	66%	–	–
Lefurgy[27]	i386	SPECint95	75%	–	–

Tabela 2.2: Resumo dos métodos PDC

Capítulo 3

O Método PDC-ComPacket

Neste capítulo será descrito o método PDC-ComPacket, proposto como tese de Doutorado de Wanderley [1], alumni do LSC[1]. A referência principal é a própria tese de Wanderley, e artigos [28, 29, 30] publicados recentemente.

É importante que se faça um completo estudo do PDC-ComPacket, uma vez que a arquitetura de descompressão proposta neste trabalho baseia-se nele.

3.1 Análise do método

A idéia principal do PDC-ComPacket é identificar algumas instruções (instruções que mais aparecem no código, por exemplo) e inseri-las em um dicionário. Posteriormente, estas instruções são substituídas por índices para o dicionário, compactando assim o código. A seguir serão destacadas algumas características que levaram a escolha deste método.

- **Simplicidade na descompressão**: Diferentemente dos trabalhos mostrados na revisão bibliográfica do capítulo anterior, que usavam codificações aritméticas, codificações em formato de árvore, etc., este método faz uso de dicionários para compressão, exigindo assim, um hardware descompressor bastante simples. Além disso, ele não permite que hajam *codewords* que ultrapassem o limite de 32 bits, de modo que nunca é necessário fazer mais de um acesso à *cache* para obter a instrução que será enviada ao processador.

- **Arquitetura PDC**: Alguns autores já fizeram comparações entre as arquiteturas PDC e CDM, como [21, 26]. Em linhas gerais, eles concluíram que, em relação ao

[1]Laboratório de Sistemas de Computação - IC - Unicamp

consumo de energia e desempenho, arquiteturas PDC são mais eficientes, caso o *overhead* do descompressor seja pequeno. A explicação para isso é que as instruções estão armazenadas comprimidas na *cache*, aumentando sua capacidade. Por outro lado, implementações em silício geralmente escolhem arquiteturas CDM, indicando que a tarefa de tornar o *overhead* pequeno é muito difícil. Além disso, no caso de arquiteturas PDC, a descompressão é executada em cada busca de instrução pelo processador e não apenas nos *cache refills*. Estes motivos mostram a necessidade de uma implementação bastante otimizada do descompressor PDC, para que ele seja viável na prática. Levando em conta que o método PDC-ComPacket enfatiza a simplicidade de descompressão, acredita-se que um descompressor PDC possa ser projetado com *overhead* muito pequeno. Como será visto no capítulo posterior, os requisitos de projeto do descompressor sugerem que ele descomprima cada instrução em apenas um ciclo, sem alterar o tempo de ciclo do processador (caminho crítico).

- **Patching**: A escolha de usar *patching* em vez de uma ATT (tabela de tradução de endereços) depende geralmente do tamanho da ATT. Por estar localizado entre o processador e a *cache*, a ATT precisaria de uma entrada para cada endereço comprimido, ou em outras palavras, a ATT teria o tamanho do código original em número de entradas. Por este motivo, o uso de *patching* é mais interessante. Além disso, a descompressão é facilitada, já que não é necessário consultar um tabela de tradução, o que é uma grande vantagem, por simplificar o descompressor.

3.2 Tratamento dos saltos

O ComPacket se mostra bastante eficiente em relação ao tratamento dado aos saltos. Muitos dos métodos PDC evitam colocar saltos relativos no dicionário pois, se eles fossem permitidos, os deslocamentos das instruções de salto do dicionário mudariam depois do *patching*, o que levaria à necessidade de correção das *codewords* para apontar para outras posições no dicionário. Como geralmente as *codewords* têm tamanhos distintos, isso mudaria novamente os deslocamentos levando a necessidade de um novo *patch*. Em suma, nota-se que os passos tornam-se recorrentes, levando a um problema NP completo. O ComPacket, por sua vez, permite que instruções de salto relativos pertençam ao dicionário. Isto é conseguido fazendo com que apenas os bits semanticamente significativos dos saltos (tipo de salto: be, bne, etc.) fiquem no dicionário, enquanto que os bits de *offset* são

alocados junto à *codeword*, no código comprimido. Esta peculiaridade melhora a razão de compressão em cerca de 5%.

Os saltos indiretos, que são aqueles saltos feitos em relação ao valor contido no registrador, também devem ser corrigidos pelo compressor. Saltos indiretos são comumente usados, por exemplo, em tabelas de saltos (*jump tables*). O método também provê a correção destes endereços na etapa de *patching*.

Ao comprimir um programa, surgem problemas de alinhamento. Basta pegar o exemplo de um salto para uma palavra comprimida (*codeword*), considerando que esta palavra comprimida contém 4 instruções originais. Para qual das quatro instruções será feito o salto? Existem duas abordagens comumente encontradas na bibliografia. A primeira é usar bits de padding, de maneira que a instrução original sempre esteja na primeira posição do *codeword*. Esta abordagem, obviamente, traz perdas para compressão. Uma segunda abordagem é modificar o processador de modo que ele passe a saltar para *nibbles*, e não mais para palavras de 32 bits. Esta abordagem traz a desvantagem de que a distância máxima de salto fica menor, já que o *offset* dirá respeito ao deslocamento em *nibbles*. Ao contrário destas duas vertentes, o ComPacket permite que os alvos sejam desalinhados e faz uso de um conjunto de bits na codificação dos *codewords*, para indicar qual instrução dentro da *codeword* é o alvo.

3.3 Codificação proposta

A codificação foi desenvolvida em [1] baseada nos formatos de instrução do SPARC Versão 8 [31], processador escolhido para implementação da técnica. No SPARC V8, todos formatos de instruções têm 32 bits. No caso do formato de instrução 2, mostrado na Figura 3.1, o padrão reserva uma seqüência de bits para representar instruções inválidas para o processador: OP = 00_2 e OP2 = $00X_2$. É justamente esta seqüência de bits que é aproveitada pelo ComPacket. Ele a usa para identificar uma instrução comprimida. Isto permite que o descompressor, ao identificar que a seqüência acima NÃO foi usada, possa repassar ao processador diretamente a instrução sem necessidade de descompressão, já que se trata de uma instrução descomprimida, o que diminui consideravelmente o *overhead* imposto, minimizando também o consumo de energia, na medida que o hardware de descompressão só será ativado quando efetivamente necessário.

Palavras comprimidas (OP = 00_2, OP2 = $00X_2$), ou ComPackets, nada mais são do que conjuntos de índices para o dicionário de instruções (por razões práticas

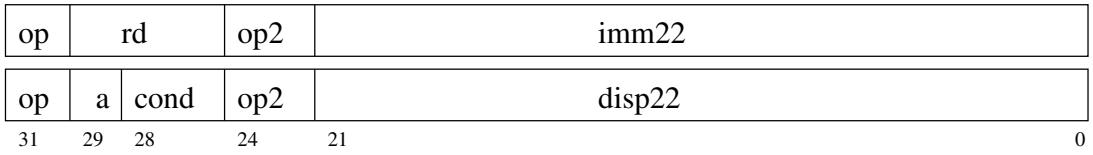


Figura 3.1: Formato 2 do padrão Sparc V8

explicadas adiante, o dicionário tem 256 posições). Existem quatro formatos de instruções comprimidas, permitindo de dois a quatro índices, como mostrado na Figura 3.2. Todos os formatos têm em comum uma seqüência de escape ESC que caracteriza o ComPacket. Além dos para identificação da compressão, OP e OP2 apresentados anteriormente, a seqüência de escape prevê o par de bits TT usado para identificar a instrução alvo, no caso de um salto para uma instrução comprimida. O bit S indica o tamanho dos índices. Para $S = 0_2$, os índices têm 6 bits. Para S = 1_2, são 8 bits. O bit B, indica se há alguma instrução de salto, dentre as instruções apontadas pelos índices do ComPacket.

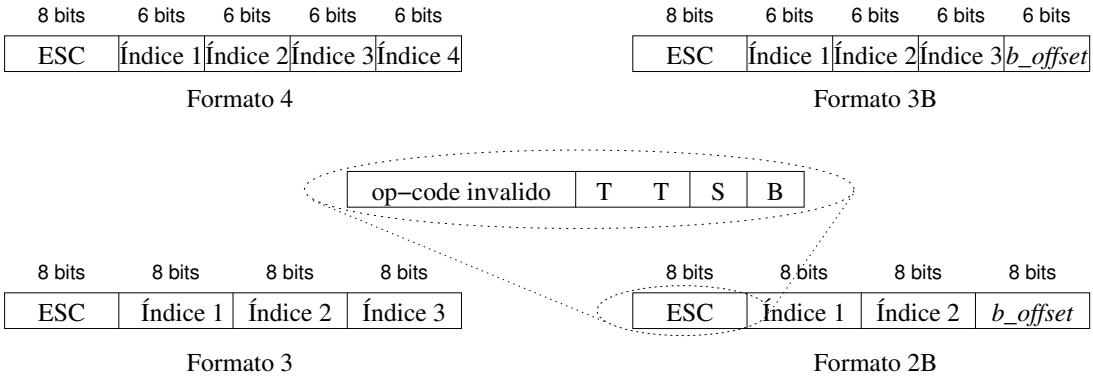


Figura 3.2: Codificação usada nos ComPackets

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

3.4 Algoritmo de compressão

Tendo descrito as principais características do método ComPacket, pode-se finalmente detalhar o algoritmo de compressão. Os passos do algoritmo de compressão são os seguintes:

1. **Construir o dicionário de instruções**: Na bibliografia, são comumente mencionadas duas maneiras de se construir o dicionários de instruções. Em uma delas, o dicionário é populado com aquelas instruções mais vezes encontradas no código (classificação estática). Na outra maneira, o dicionário é populado com aquelas instruções mais executadas (classificação dinâmica). A classificação estática oferece um maior ganho de compressão, já que é possível representar com menos bits as instruções que aparecem mais. A classificação dinâmica oferece melhores resultados em relação ao desempenho do sistema, por permitir um maior *throughput* de instruções entre a I-cache e o processador. Nesta técnica, é usado um misto destas duas abordagens. Acredita-se que, com isso, tenha-se o melhor custo / benefício no que se refere à compressão de código, desempenho do sistema e consumo de energia, requisitos indispensáveis em um sistema embutido. Inicialmente ordena-se as instruções tanto pela classificação estática quando dinâmica. Então insere-se uma porcentagem das instruções respeitando a classificação estática e as demais respeitando classificação dinâmica, até completar o dicionário. Obviamente, antes de inserir uma instrução no dicionário, é verificado se ela já foi inserida antes. Em caso positivo, a próxima instrução da classificação é usada. O percentual de instruções estáticas / dinâmicas inseridas no dicionário é um parâmetro fornecido pelo usuário. Desta forma, a técnica permite que o usuário configure o sistema de acordo com suas necessidades de compressão e desempenho.

2. **Marcar código**: Fazer a marcação no código original das instruções que pertecem ao dicionário, das instruções que são alvos de saltos e aquelas que precisam de *patching*.

3. **Atribuir os ComPackets**: Varre o código original e, de acordo com o dicionário e algumas restrições que serão vista num exemplo a seguir, procura marcar as instruções prioritariamente para compressão pelo primeiro formato (Formato 4). Se não for possível, tenta o segundo formato (Formato 3). E assim por diante, até o quarto formato (Formato 2B). Se ainda assim não conseguir, a instrução fica no

seu formato original, não comprimida. Ou seja, a preferência é dada aos índices que ocupam menos bits, para conseguir a melhor razão de compressão possível.

4. **Compactação do código**: Uma única passada do código e os ComPackets são atribuídos, substituindo as instruções originais. Nesta fase também é gerada uma tabela de mapeamento entre endereços convencionais e comprimidos.

5. ***Patching* dos endereços**: Finalmente é feito o *patch* das instruções de salto, com base na tabela de mapeamento gerada no passo anterior.

A seguir será mostrado um exemplo de compressão de código (figuras 3.3 a 3.11) extraído de [1].

Depois da construção do dicionário, as instruções do código original são marcadas (D: pertencentes ao dicionário; T: alvos de saltos; e P: necessitam de Patching) (Figura 3.3).

A primeira instrução do dicionário é procurada no código. Sua primeira ocorrência está no endereço 04_h. O algoritmo tenta formar o ComPacket 4 usando a instrução corrente e as 3 instruções anteriores ou 2 anteriores e 1 posterior ou 1 anterior e 2 posteriores ou 3 posteriores. Na Figura 3.4, apesar de haver uma seqüência de 4 instruções que pertencem ao dicionário, não é possível usar um formato 4 porque duas das instruções são alvos de uma instrução de salto, e apenas 1 alvo é permitido por ComPacket.

A segunda tentativa é formar um ComPacket 3. Ainda assim não é possível pelo mesmo motivo citado anteriormente. A próxima tentativa é para o Formato 3B, e mais uma vez não é possível formar um ComPacket.

Agora a tentativa é para formar um ComPacket 2B. Neste caso é possível formar um par com as instruções *a* e *b*. Elas são marcadas para um ComPacket 2B (Figura 3.5).

Figura 3.3: Exemplo do algoritmo de compressão: Passo 1

Figura 3.4: Exemplo do algoritmo de compressão: Passo 2

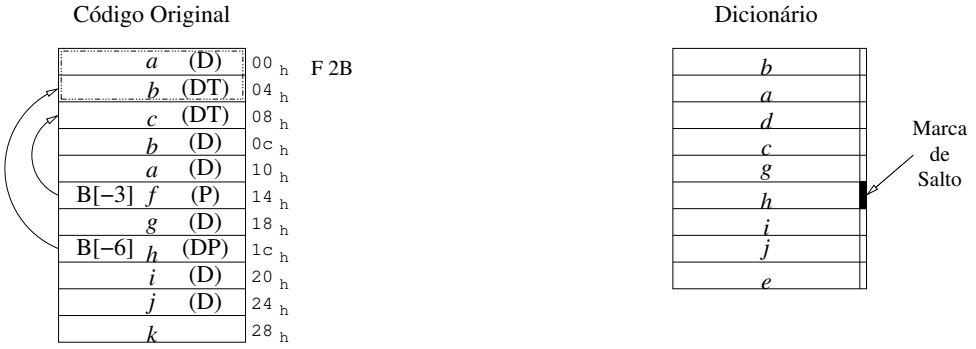


Figura 3.5: Exemplo do algoritmo de compressão: Passo 3

O algoritmo continua a percorrer o código original em busca da instrução b. Ela é encontrada no endereço $\mathtt{0c}_h$ (Figura 3.6). A primeira tentativa agora é um formato 4. Não há possibilidade de formação para um ComPacket 4 porque não há 4 instruções na circunvizinhança de b que ainda não estejam marcadas para um ComPacket e pertençam ao dicionário. O formato 3 pode ser utilizado pois c, b e a pertencem ao dicionário. Elas são marcadas para um ComPacket 3 (Figura 3.7).

Não há mais instruções b no restante do código. A instrução a do dicionário passa a ser procurada no código. Sua primeira ocorrência em $\mathtt{00}_h$, já está marcada para um ComPacket, bem como a segunda em $\mathtt{10}_h$. Não há mais instruções a no código. O passo se repete para c e d do dicionário e nenhum novo ComPacket é formado. Agora, a instrução g é averiguada. Sua ocorrência em $\mathtt{18}_h$ pode gerar um ComPacket 3B dado que g, h e i pertencem ao dicionário, apenas uma delas é um salto, apenas uma delas é um alvo e todas elas se encontram nas primeiras 64 instruções do dicionário (podem ser representadas com 6 bits). Elas são então marcadas para formar um ComPacket 3B (Figura 3.8). As demais instruções do dicionário são averiguadas no código, mas não

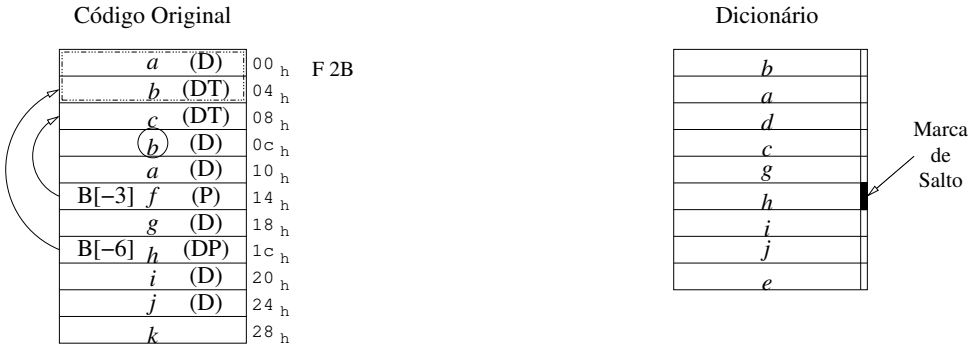


Figura 3.6: Exemplo do algoritmo de compressão: Passo 4

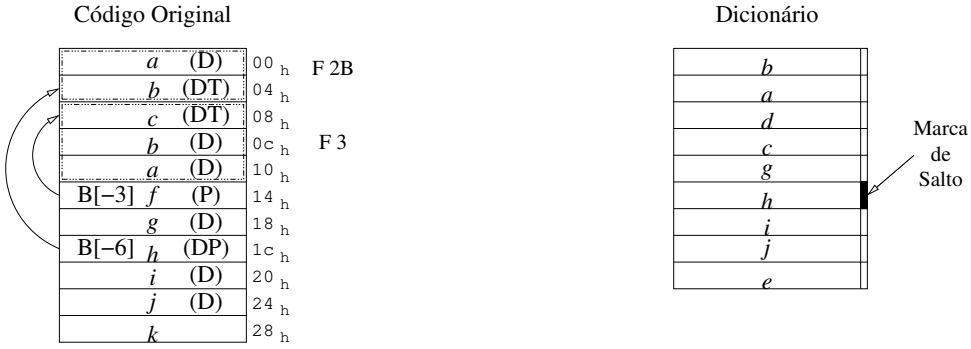


Figura 3.7: Exemplo do algoritmo de compressão: Passo 5

formam nenhum novo ComPacket.

Terminada esta fase de marcação dos ComPacket, resta montá-los e criar uma tabela de tradução de endereços. O código original é percorrido. Inicialmente, o ComPacket 2B é encontrado. A seqüência de escape é montada (Figura 3.9) levando em consideração que a instrução *b* é um alvo, que não há saltos e que o ComPacket é 2B. Neste caso, o *slot* do *offset* do salto é inutilizado (preenchido com zeros). O ponto no índice da instrução *b* é a indicação do alvo na figura (indicado pelos bits TT). A tabela de tradução de endereços é alimentada com as primeiras duas instruções do código.

O algoritmo continua até montar todos os ComPackets do Código (Figura 3.10). Montada a tabela de tradução de endereços e formado o código comprimido, agora é necessário atualizar os saltos (*patching*). O endereço 08_h comprimido contém um salto que originalmente tinha *offset* de -3. Depois de comprimido, ele passará a ter *offset* de -1, portanto será necessária uma intervenção no código comprimido. Outra instrução que necessita de *patching* é a *h*. De fato, por pertencer ao dicionário, sequer o *offset* original foi marcado no ComPacket correspondente. O novo valor do deslocamento é -3 (antes era

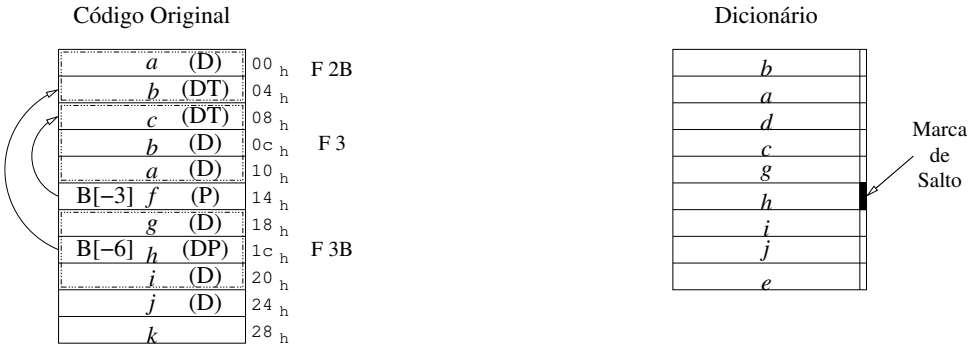


Figura 3.8: Exemplo do algoritmo de compressão: Passo 6

-6). O resultado final da compressão pode ser visto na Figura 3.11.

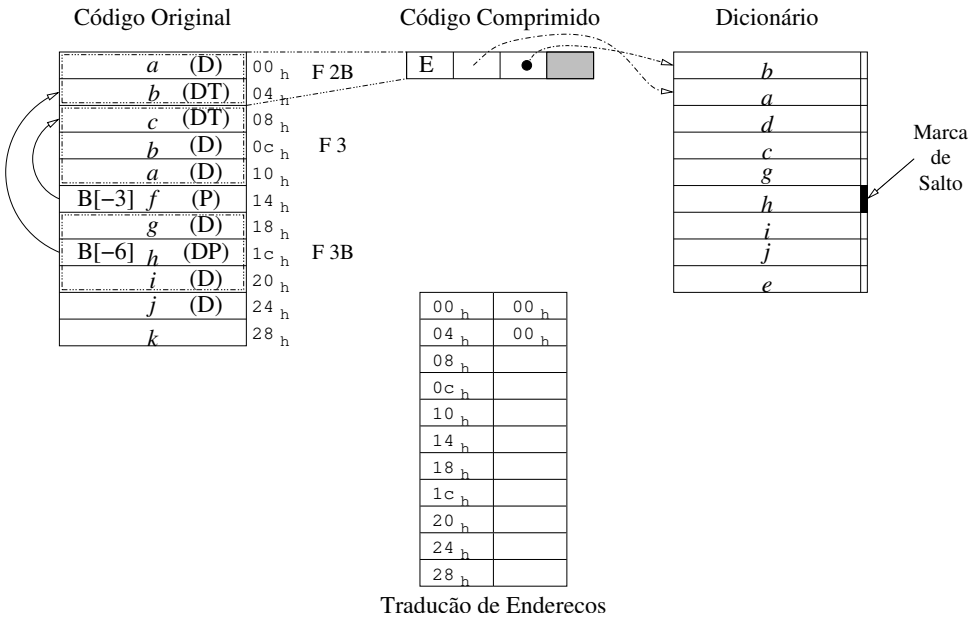


Figura 3.9: Exemplo do algoritmo de compressão: Passo 7

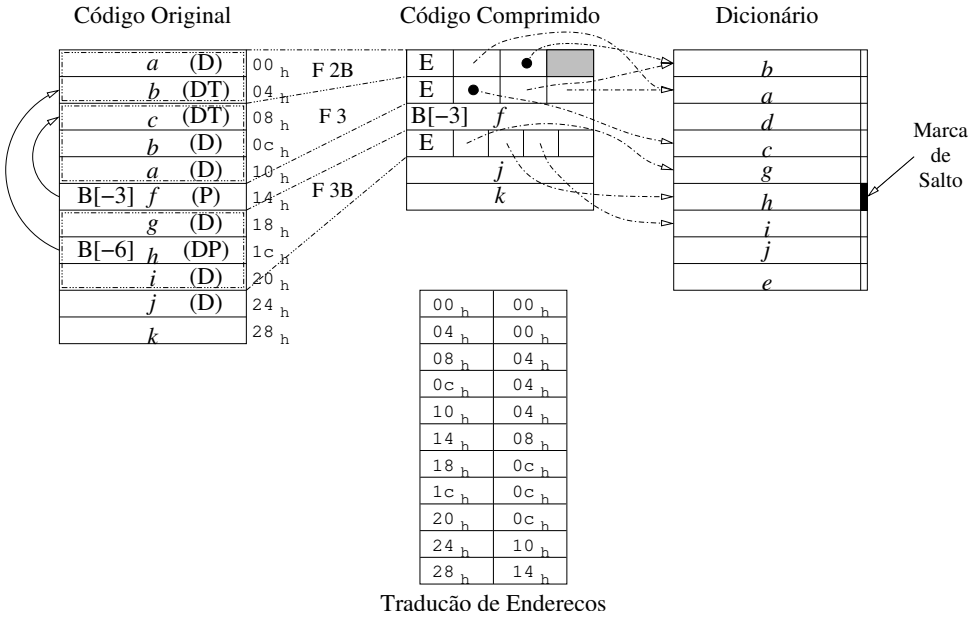


Figura 3.10: Exemplo do algoritmo de compressão: Passo 8

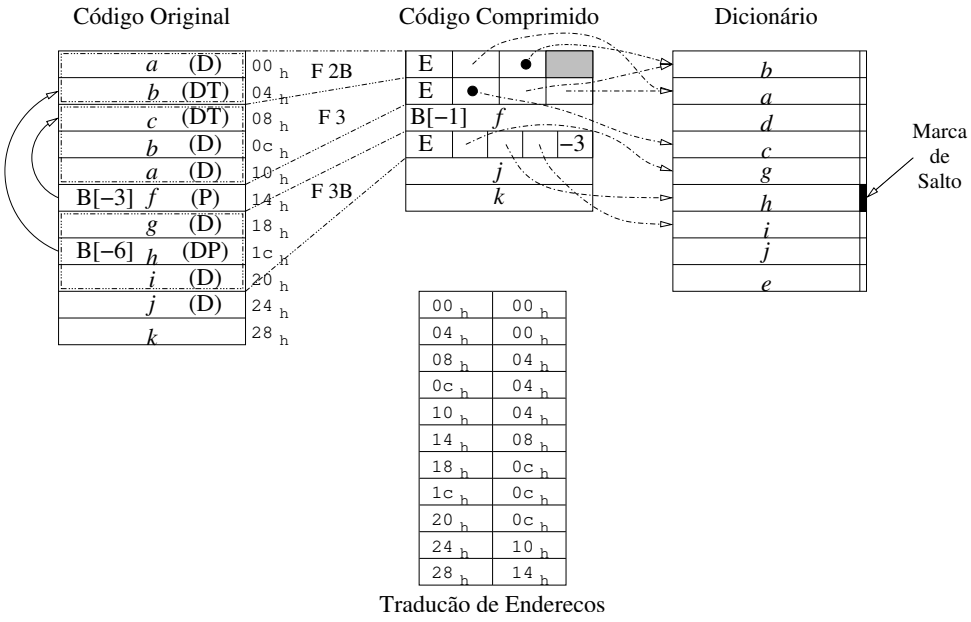


Figura 3.11: Exemplo do algoritmo de compressão: Final.

3.5 Fator dinâmico/estático *f*

Uma importante característica do método PDC-ComPacket é a forma como se contrói o dicionário de instruções. Se, por um lado, o uso de classificações estáticas (contagem de cada instrução do código) melhoram a razão de compressão, por outro, as classificações dinâmicas (contagem do número de vezes que cada instrução é executada, através de *profiling*) trazem vantagens em relação ao tempo de execução da aplicação. Isto porque, numa classificação estática, as instruções do dicionário aparecerão mais no código, enquanto que numa abordagem dinâmica, mais código comprimido é executado, o que leva a uma maior taxa de acertos na *cache*.

O PDC-ComPacket diferencia-se das demais técnicas no sentido que ele não segue completamente nenhuma das filosofias. Em vez disso, ele monta o dicionário baseado em ambas classificações. O usuário pode definir um fator percentual dinâmico/estático, chamado de *f*, que dirá a contribuição de cada uma das classificações na montagem do dicionário. Para f igual a 40%, por exemplo, no mínimo 40% das instruções vêm da classificação dinâmica, enquanto os demais 60% vem da estática.

Posteriormente, no Capítulo 5, o impacto de f será avaliado em relação a compressão e desempenho obtidos.

3.6 Proposta para Arquitetura de Descompressão

A Figura 3.12 mostra a sugestão original de arquitetura proposta por Wanderley [1]. Existem alguns motivos que levaram a modificações nesta arquitetura. Por exemplo, não é possível implementar o método PDC-ComPacket sem modificações no processador escolhido, devido ao controle do contador de instruções.

No Capítulo 4, onde será apresentada a Arquitetura de Descompressão projetada neste trabalho, estes problemas serão novamente levantados, com um maior detalhamento.

3.7 Considerações Finais e Conclusões

Este Capítulo descreveu o PDC-ComPacket e as vantagens que levaram a sua escolha. A simplicidade requerida para descompressão, a resolução de endereços através de *patching* e por ter sido projetado para funcionar como uma Arquitetura PDC são alguns motivos

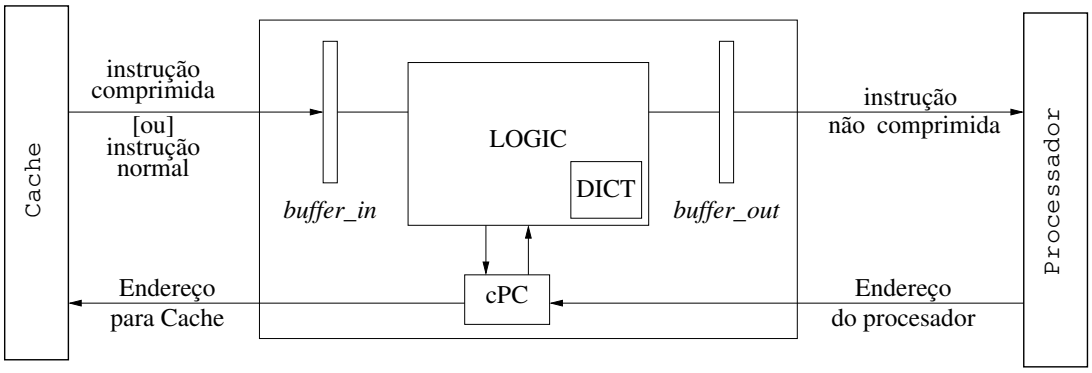


Figura 3.12: Arquitetura de descompressão inicial

que permitirão que se atinjam os objetivos iniciais deste trabalho, como chegar a uma versão funcional de uma Arquitetura de Descompressão.

Este Capítulo visou apenas uma primeira visão do PDC-ComPacket, em alguns pontos, sem muito aprofundamento. Seu objetivo foi criar a fundamentação necessária para entendimento dos Capítulos posteriores. Detalhes do método como, problemas na resolução de endereços, razão *f*, serão novamente levantandos no decorrer do trabalho.

Capítulo 4

Arquitetura de descompressão

Neste capítulo, será apresentado o descompressor para o método PDC-ComPacket. Procurou-se estruturar o capítulo de modo a torná-lo o mais compreensível possível, explicando passo a passo cada detalhe inferido até se chegar ao projeto completo.

Antes de iniciar o desenvolvimento de qualquer projeto, a primeira etapa é definir seus requisitos. Neste projeto, são eles:

- Chegar a uma versão funcional do descompressor, integrada ao processador Leon2 e prototipada em FPGA
- Inserir o descompressor com o mínimo de alterações no processador original e usando o mínimo de hardware possível
- O *cycle-time* do processador não deve ser degradado após a integração
- Nenhuma bolha (ciclos extras) devem ser inseridas devido ao descompressor.

Estes requisitos garantem o melhor aproveitamento possível do método PDC-ComPacket em termos de área e desempenho: o mínimo de área será usado (hardware extra) e o desempenho será o melhor possível (sem bolhas, nem degradação do *cycle-time*).

4.1 Processador LEON2

Esta seção objetiva apresentar o Leon2 [20], processador-alvo usado na implementação do método. O Leon2 é um processador RISC totalmente compatível com o padrão IEEE-1754

(SPARC Versão 8) [31], ideal para ser usado em projeto de sistemas embutidos SoC (*system-on-a-chip*). A Figura 4.1 mostra um diagrama do Leon2.

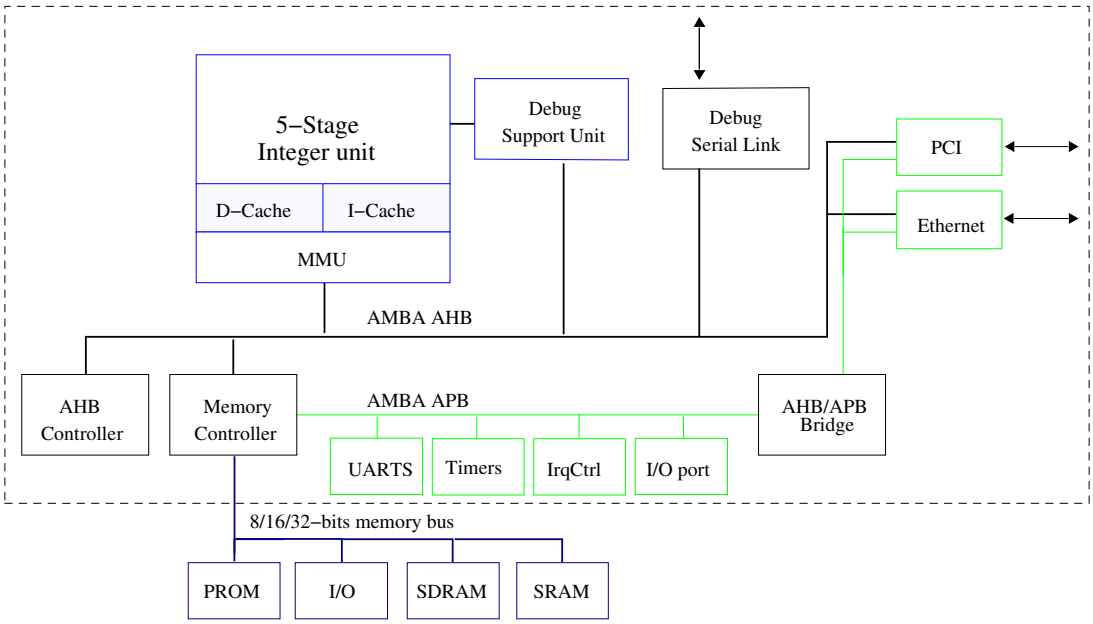


Figura 4.1: Diagrama do processador LEON2

Dentre suas características estão:

- Unidade de inteiros com 5 estágios de pipeline;
- *Cache* de intruções e dados separadas (Harvard);
- Sistema de memória virtual (*Memory Management Unit* - MMU);
- Multiplicação e divisão implementadas em hardware, por meio de instruções como *umul* e *udiv*;
- Alguns periféricos implementados on-chip: *uarts*, temporizadores, porta de entrada-saída de 16 bits, Ethernet MAC, *interface* PCI;
- Controlador de Interrupção;
- Controlador de memória para: *SRAM/SDRAM/PROM/IO*;
- Unidade de depuração implementada em hardware (*Debug Support Unit*): Esta é uma funcionalidade bastante interessante, sendo essencial para o desenvolvimento

deste trabalho. Através do GRMON, programa que roda em um computador *host* conectado serialmente através do *debug serial link* ao Leon2, é possível baixar programas para o processador e testar a execução do programa no hardware sintetizado em FPGA. Além disso, o GRMON possui suporte para o *GDB* (*GNU Project Debugger*, permitindo que seja feita depuração no hardware real a partir do código fonte do programa, escrito em C (como é feito para uma aplicação de alto nível);

- Barramento para dispositivos rápidos (AHB) e lentos (APB), totalmente compatível com a especificação AMBA [32].

O processador Leon2 já vinha sendo usado por outros alunos do LSC em seus trabalhos, o que levou a uma experiência considerável em cima deste processador. Além disso, outros motivos que levaram a sua escolha foram:

- ***Arquitetura RISC***: Os processadores RISC são bastante completos, conseguindo atender as necessidades de complexidade dos sistemas embutidos atuais. Além disso, os processadores RISC são conhecidos por oferecerem um conjunto de instruções que não fogem de uma certa regularidade (as instruções têm 32 bits, independente da complexidade da instrução). Com isso, a densidade de código se torna 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 Leon2 foi recentemente certificado pela organização SPARC International como sendo um processador totalmente compatível com a arquitetura SPARC versão 8 [31]. Para obter tal 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 Leon2 é aberto, podendo ser obtido gratuitamente, tanto no meio acadêmico quanto comercial, segundo os termos da licença GNU LGPL. O código está escrito em VHDL, uma linguagem de alto nível para descrever hardware.

- ***Facilidades para o desenvolvimento e depuração***: Todas as ferramentas de desenvolvimento e depuração em cima do Leon2 estão disponíveis gratuitamente,

permitindo que se tenha todo o ambiente configurado: *cross-compiler*, *testbenches*, VHDL para simulação RTL do modelo, scripts de síntese, ambiente para depuração a partir de um computador host (DSU/Grmon/GDB, descrito anteriormente).

4.2 Descompressor ComPacket

Nesta seção será apresentado o hardware para o descompressor numa versão *stand-alone*, sem preocupação com detalhes de integração. De acordo com o que foi mostrado do método PDC-ComPacket, no capítulo anterior, não fica muito difícil inferir como deve ser o hardware para descompressão. A Figura 4.2 mostra como ficou o projeto final, já com várias otimizações de *timing* e utilização de recursos de hardware.

Na parte superior é mostrada a lógica de busca da instrução no dicionário. O bit *size* identifica o tipo de ComPacket, com índices de 8 bits (4 ou 3B) ou 6 bits (3 ou 2B). De acordo com o tipo de ComPacket e de acordo com o *cpc*, o multiplexador *MUX 1* seleciona o índice correspondente para endereçar o dicionário de instruções. No caso de índices de 6 bits (ComPacket 4 e 3B), os 2 bits mais significativos são complementados com 00_2 para formar 8 bits. Tendo o endereço de acesso do dicionário, busca-se a instrução, bem como o bit de *flag* (*branch*) correspondente, indicando se aquela é uma instrução de salto.

Na parte central é gerada a palavra descomprimida. O sinal de controle *DecompSel* faz a seleção segundo a Tabela 4.1. São 4 seleções possíveis: caso *deci.inst* apresente um *opcode* inválido, isto é, 0000_2, e o bit de *branch* vindo do dicionário seja 0_2 (não se trata de uma instrução de salto), significa que a instrução descomprimida é exatamente a saída do dicionário (*dic_data*). No segundo caso, onde os bits de *branch* e *size* de *deci.inst* indicam um ComPacket3B e além disso, o bit de *branch* vindo do dicionário indica que é um salto, então a instrução descomprimida será uma composição entre os 10 primeiros bits de *dic_data* e os 22 bits previamente estendidos a partir do *offset* de 6 bits localizado em *deci.inst*. O terceiro caso é análogo ao segundo, porém, em se tratando de um ComPacket2B, seleciona-se o desvio estendido a partir do *offset* de 8 bits. Finalmente, o último e quarto caso, quando existe um *opcode* válido, trata-se de uma instrução descomprimida e, portanto, ela é diretamente repassada para saída.

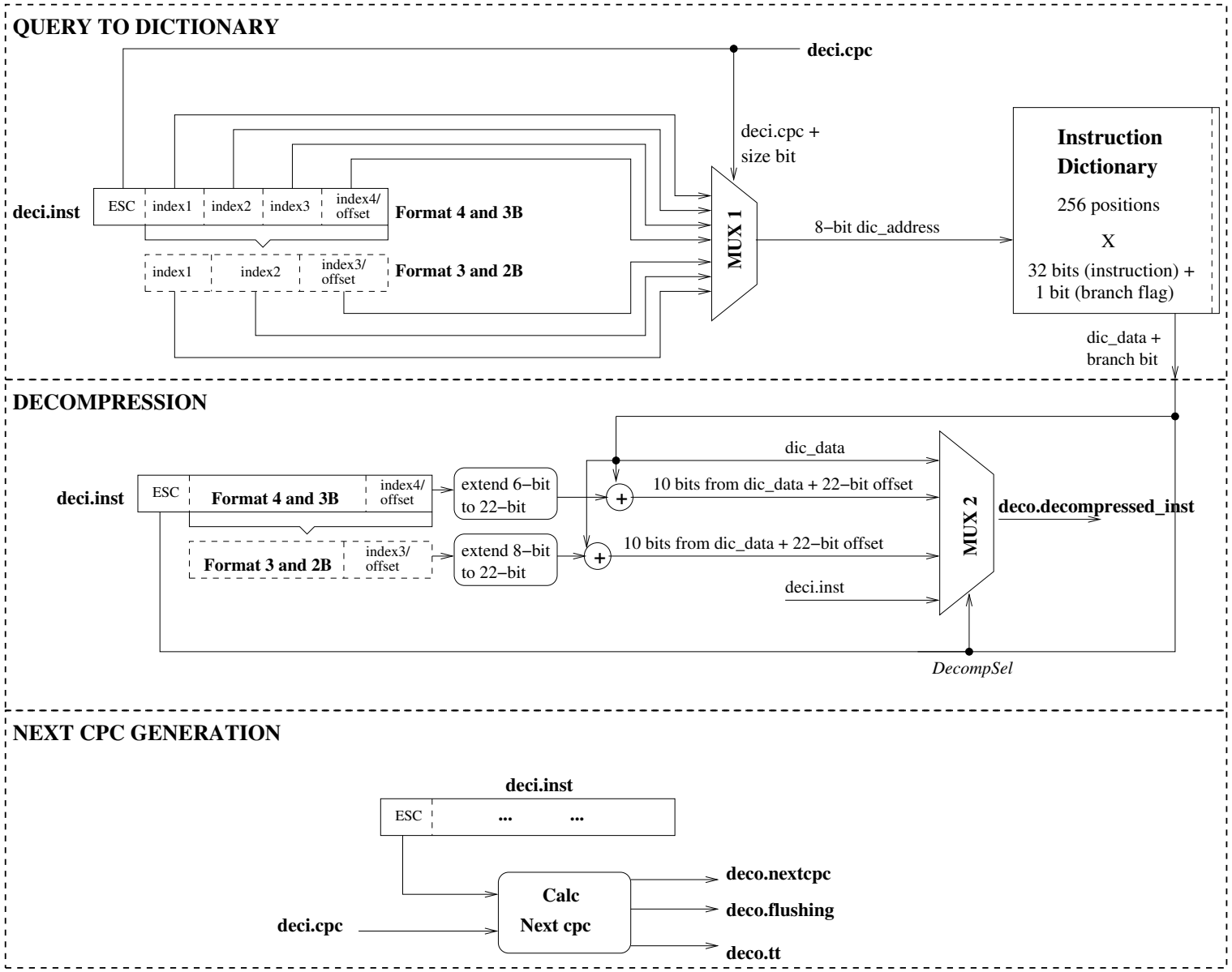


Figura 4.2: Diagrama do descompressor ComPacket (*stand-alone*)

deci.inst			Dicionário	Descompressão	
Opcode	Branch	Size	Branch	Tipo	*DecompSel*
0000	0	0	0	ComPacket4	dic_data
0000	0	1	0	ComPacket3	
0000	1	0	0	ComPacket3B	
0000	1	1	0	ComPacket2B	
0000	1	0	1	ComPacket3B	10 bits dic_data + 22 desvio (ext. de 6 bits)
0000	1	1	1	ComPacket2B	10 bits dic_data + 22 desvio (ext. de 8 bits)
XXXX	X	X	X	Não comprimida	deci.inst

Tabela 4.1: Obtenção da instrução descomprimida

Na parte inferior é mostrada a lógica de geração do próximo *cpc*. Para simplificar a visualização, o hardware é representado por uma caixa preta, devendo oferecer uma lógica combinacional que gere saídas como ilustrado na Figura 4.3. Da figura, pode-se inferir a função de cada sinal de saída:

- ***nextcpc***: É o próximo *cpc* seqüencial, de acordo com o *cpc* de entrada, usado para saber qual instrução deve-se descomprimir dentro de um ComPacket. O retorno para o 0_2 (índice 0), deve ser feito levando em conta o número de índices do ComPacket (2, 3 ou 4 índices).

- ***flushing***: Assume nível lógico baixo apenas quando o último índice do ComPacket está sendo analisado. Este sinal é usado, por exemplo, para saber quando o PC do processador deve permanecer interrompido. Da mesma forma que o *nextcpc*, a geração deste sinal deve ser feita levando em conta o número de índices do ComPacket.

- ***tt***: Este campo não depende do índice sob análise. É o *tt* contido na seqüência de escape de *deci.inst*, usado para definir a instrução alvo em um salto para um ComPacket.

Cabe ressaltar que o descompressor apresentado nesta seção é puramente combinacional. Ou seja, dada algumas entradas, assinaladas como *deci*, são geradas suas saídas *deco*. Aspectos seqüenciais serão mostrados posteriormente, no momento que este hardware for integrado ao processador.

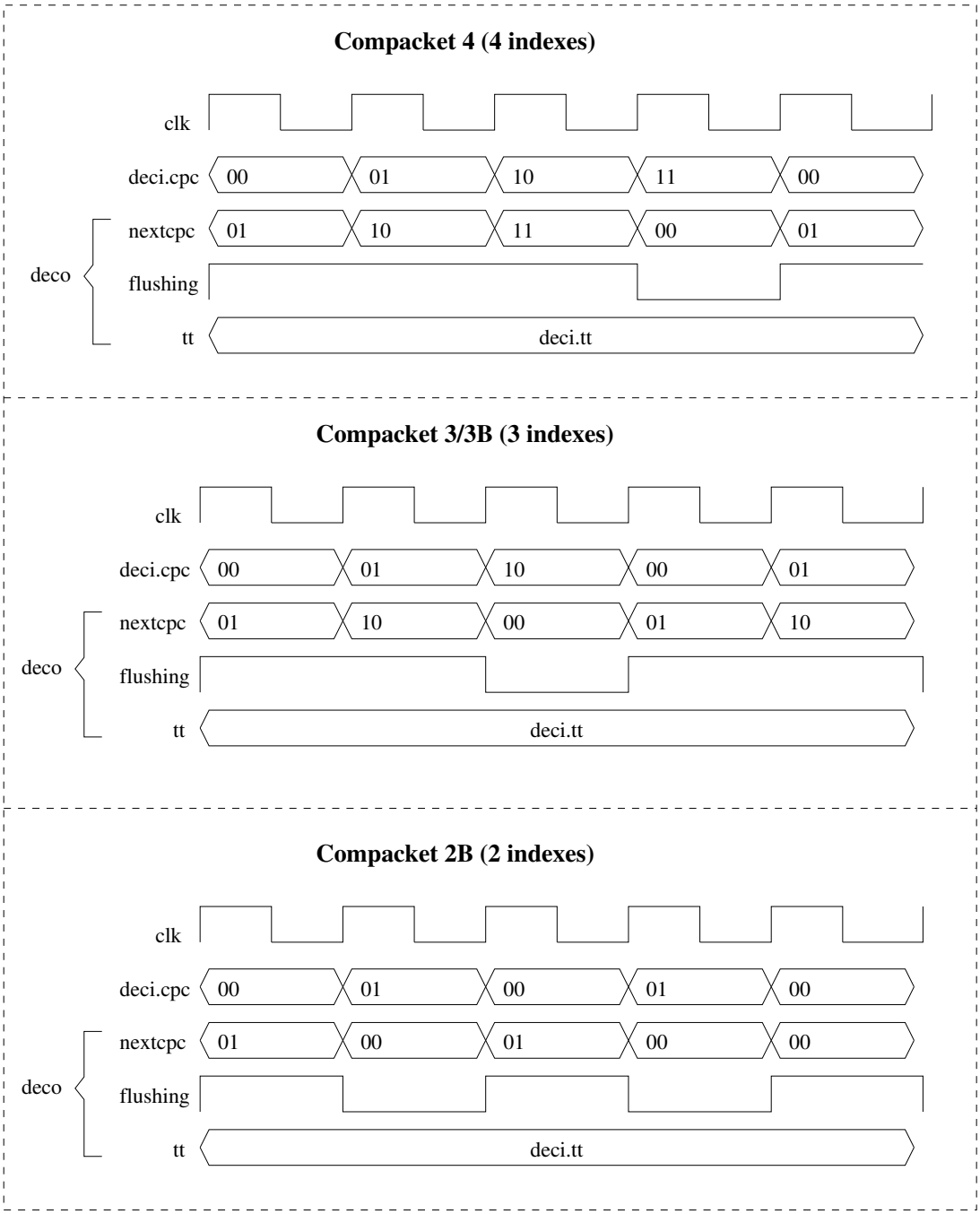


Figura 4.3: Formas de onda: lógica de cálculo do *nextcpc*

4.3 Arquitetura de Descompressão

No capítulo anterior, quando foi apresentado o método PDC-ComPacket, mostrou-se uma sugestão inicial do que seria uma arquitetura para o método ComPacket. Esta seção visa analisar esta abordagem, bem como as demais que surgiram posteriormente, até se chegar a arquitetura final que será detalhada na seção posterior.

4.3.1 Descompressor como estágio de Pré-busca

A Figura 4.4 replica a sugestão apresentada no capítulo anterior (Figura 3.12), que não foi implementada por dois motivos:

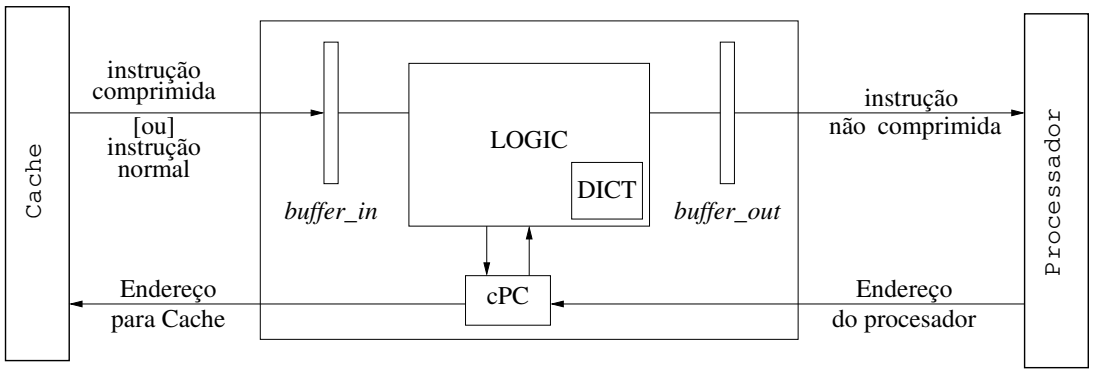


Figura 4.4: Arquitetura de descompressão inicial

- *Lógica de PC*: Durante a descompressão de uma palavra comprimida, um ComPacket 4, por exemplo, o incremento do PC do processador deve ser interrompido de modo que a contagem fique sincronizada com a execução. Para isso, são necessárias mudanças na lógica de PC **dentro** do processador, o que contraria a arquitetura acima que não preve nenhuma alteração no processador.
- *Inserção de bolha nos saltos*: Partiu-se do requisito de se chegar a uma implementação que não inserisse bolhas no *pipeline* sob nenhuma condição. Deste modo, a inserção de bolhas passou também a ser um problema desta abordagem.

4.3.2 Descompressor dentro do processador

Em virtude dos problemas apresentados pela abordagem anterior, o descompressor foi reprojetado para atuar dentro do processador. Um requisito do projeto era que o

descompressor fosse inserido sem alterar o *cycle-time*. Para isso, foram cogitadas três alternativas, como mostrado na Figura 4.5. A primeira alternativa, posicionando o descompressor no estágio de busca, após a saída da I-cache, aumentou o *cycle-time* em cerca de 25%. A segunda alternativa, posicionando o descompressor no início do estágio de decodificação, aumentou o *cycle-time* em cerca de 30%. A terceira alternativa, que buscava balancear os tempos entre ambos estágios, implementando parte da lógica de descompressão no estágio de busca e parte no estágio de decodificação, aumentou em 25%. Pela lógica, a terceira alternativa deveria ser a melhor das três, mas não foi, pois em se tratando de FPGA's, o *cycle-time* alcançado depende muito de aspectos físicos, sobretudo varia sensivelmente de acordo com o *P&R*[1].

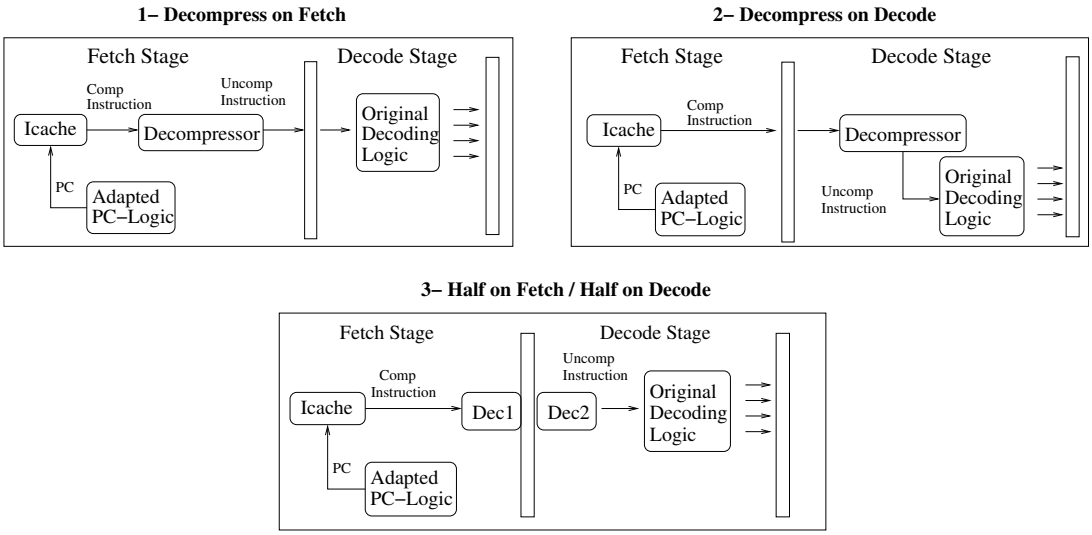


Figura 4.5: Opções de posicionamento do descompressor no *pipeline*

A falha nas três alternativas acima ocorreu em uma implementação bastante inicial do descompressor, sem otimizações de *timing*. Baseado nisto, o objetivo seguinte foi melhorar o *timing*, usando ferramentas da Xilinx com esta finalidade. O primeiro passo foi deixar de lado as *SRAMs*[2] e usar lógica distribuída para o dicionário de instruções, ou seja, usar as próprias unidades programáveis da FPGA. Desta forma, a ferramenta passou a inferir vários multiplexadores em série: de acordo com o endereço sendo requisitado (de 0 a 255), os multiplexadores são selecionados de uma forma que suas saídas apresentam a instrução correspondente ao endereço requisitado. Fisicamente, multiplexadores oferecem um tempo

[1]Place and Route - Fase da síntese de hardware onde são selecionados os elementos físicos da FPGA que sintetizarão o *design* (*Place*), bem como a interconexão entre estes elementos (*Route*)

[2]SRAMs: *Static-RAMS* (Memórias estáticas)

de propagação dos sinais bastante baixo, melhorando assim o caminho crítico. Outro passo foi paralelizar ao máximo a lógica de decodificação, criando hardwares independentes (cálculo do próximo cpc, acesso ao dicionário, etc.).

Após estas otimizações de *timing*, tentou-se novamente a primeira alternativa proposta anteriormente (descompressor no estágio de busca), constatando-se agora que o *cycle-time* do processador não foi mais degradado.

Com o problema de *timing* resolvido, tudo parecia estar funcionando, porém, ao longo do desenvolvimento surgiram novamente problemas em relação ao controle do PC. Verificou-se que alterações apenas no processador não seriam suficientes e que alterações na I-cache também seriam necessárias. Isto devido a funcionalidade de *bursting* mostrada na Figura 4.6. No momento que há uma falha de acesso na *cache*, as instruções são buscadas seqüencialmente na memória externa até encherem a linha da *cache*. No exemplo dado, são buscadas 7 instruções (posição 1 a 7 da linha da *cache*). Enquanto a linha da *cache* é alimentada, as instruções vão sendo diretamente repassadas ao processador, não havendo assim, necessidade de parar o *pipeline*. Mas como fazer se uma das instruções buscadas da memória for comprimida, como no exemplo, onde a segunda palavra é um Compacket 4 (c4). A I-cache, que não diferencia instruções comprimidas de descomprimidas, irá entregar as instruções ao processador incorretamente (Tabela 4.2). Além disso, após um longo estudo do Leon2, verificou-se que o melhor posicionamento do descompressor seria embuti-lo na I-cache. Todo o controle de *bursting*, controle de andamento/interrupção do *pipeline* de acordo com a busca de instruções, disponibilidade do barramento, saltos, etc., é feito pela I-cache. Deste modo, concluiu-se que colocando o descompressor dentro da I-cache não ofenderia a concepção de projeto do Leon2, exigindo o mínimo de hardware para fins de integração. Na seção posterior, será feito um completo detalhamento do hardware projetado.

4.4 Projeto do descompressor

Antes de integrar o descompressor ao processador, convém entender um pouco do estágio de busca, mais especificamente do controlador da I-cache, onde será embutido o descompressor. A Figura 4.7 detalha o sistema de busca de instruções do Leon2. Nesta figura, assim como nas demais, sempre que possível, os nomes dos sinais serão semelhantes aqueles usados no código VHDL, de modo que este trabalho sirva também como uma documentação para o código. Os três elementos básicos da figura são: processador,

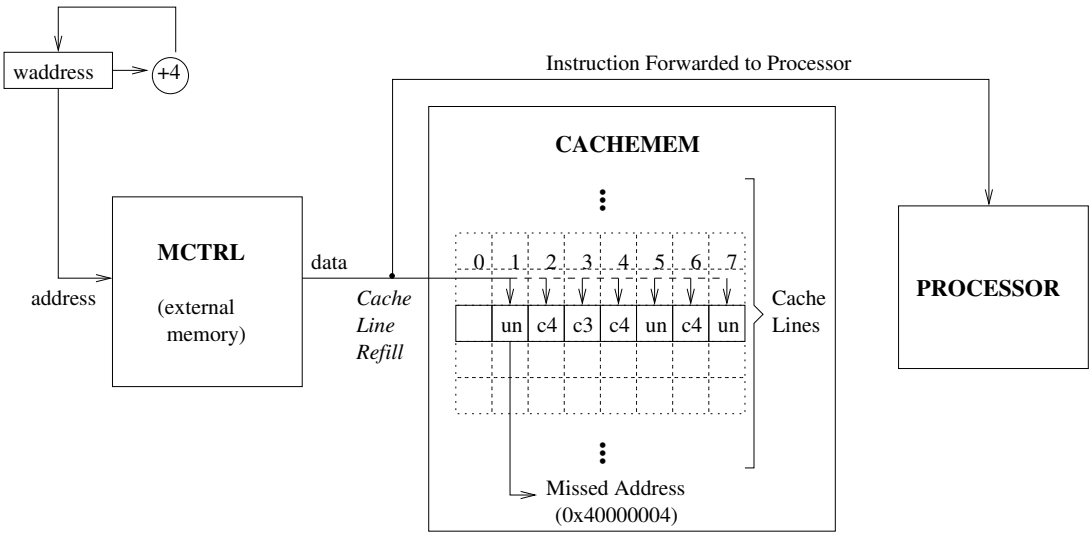


Figura 4.6: Esquema de bursting da *I-cache*

Ciclo	**Ordem Incorreta**/Atual (compressão não considerada)	**Ordem Correta** (compressão considerada)
0	Descomprimida (slot 1)	Descomprimida (slot 1)
1	ComPacket 4 (slot 2)	ComPacket 4, índice 0 (slot 2)
2	ComPacket 3 (slot 3)	ComPacket 4, índice 1 (slot 2)
3	ComPacket 4 (slot 4)	ComPacket 4, índice 2 (slot 2)
4	Descomprimida (slot 5)	ComPacket 4, índice 3 (slot 2)
5	ComPacket 4 (slot 6)	ComPacket 3, índice 0 (slot 3)
6	Descomprimida (slot 7)	ComPacket 3, índice 1 (slot 3)
...	...	...

Tabela 4.2: Ordem de entrega das instruções ao processador

I-cache e memória externa. O processador requisita a instrução à I-cache através de *fpc* (PC atual) e *rpc* (próximo PC). Na I-cache, o multiplexador *MUX 1* seleciona entre o fpc e rpc, através do sinal *taddrsel*. A saída do MUX 1, fornece o endereço de acesso à *cache*, chamado de *Taddr*. Este endereço permite localizar a linha da *cache* (TAG + instruções).

O *TAG* de uma linha da I-Cache é ilustrado na Figura 4.8. O primeiro campo, *ATAG*, contém o endereço guardado na linha da *cache*. O bit *LRR* é usado para armazenar um histórico de substituição. O bit *LOCK*, quando setado, permite que a linha da cache seja congelada, não podendo ser sobreescrita. Os últimos 8 bits, *valid*, indicam a validade de cada uma das 8 instruções da linha da *cache*. Destes campos, *ATAG* e *valid* são usados

para determinar um acerto/falha de acesso. O acerto se dá quando a *ATAG* é igual a parte alta (bits mais significativos) de *fpc* e o bit de validade correspondente a instrução sendo buscada é 1_2 (instrução válida).

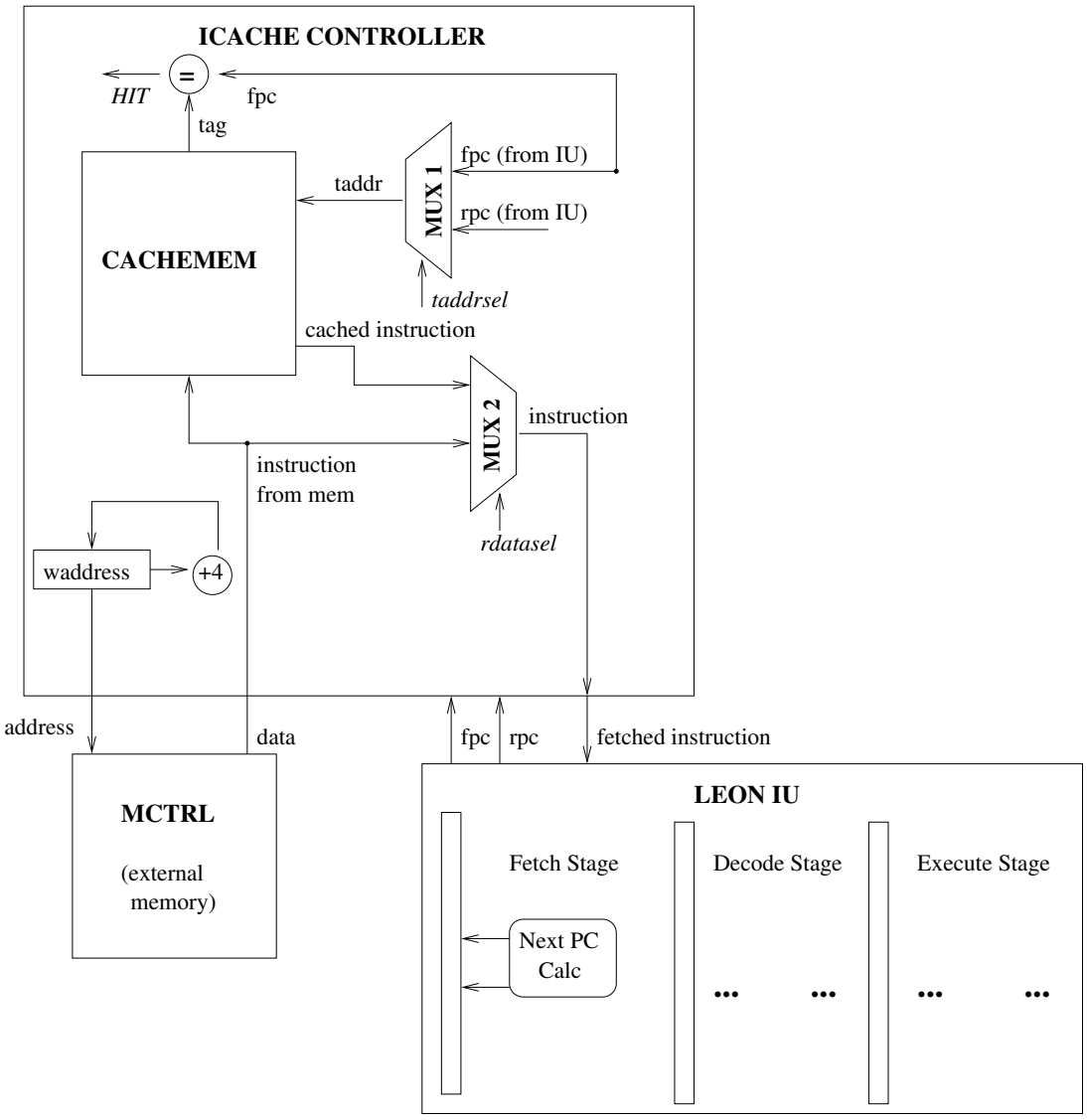


Figura 4.7: Subsistema de memória do LEON2

Voltando à Figura 4.7, em caso de acerto, o *MUX 2* seleciona a instrução saindo da I-cache para repassá-la ao processador. Note que todo esse processo de busca de uma instrução, onde há acerto, leva um ciclo de relógio, sem necessidade de interrupção do *pipeline*. Numa segunda situação, onde acontece uma falha de acesso, a I-cache entra em modo de *bursting*, como descrito na seção anterior. Neste caso, *MUX 2* faz com

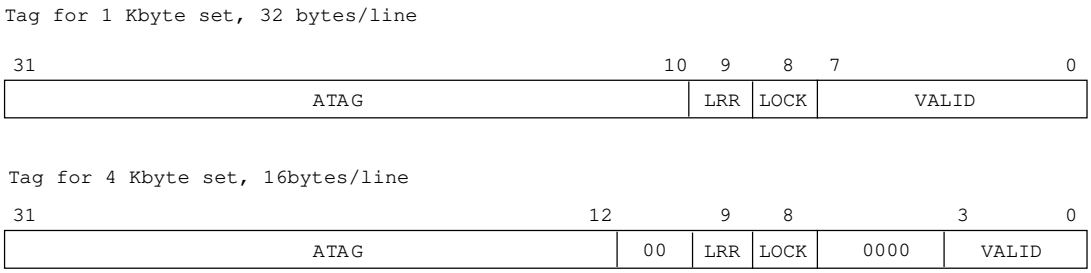


Figura 4.8: Formato da TAG da linha da I-cache

que as instruções que vão enchendo a linha da *cache*, sejam repassadas diretamente ao processador. O sinal *rdatasel* faz a seleção do *MUX 2*, abordado novamente a seguir.

4.4.1 Primeira versão do descompressor

Agora que foi apresentado o mecanismo de busca do Leon2, pode-se entrar nos detalhes de integração do descompressor ao processador. Lembrando que, por ser uma arquitetura PDC, o descompressor deve estar localizado entre a I-cache e o processador, de maneira a melhorar o *throughput* de instruções, já que elas estarão comprimidas na I-cache.

A Figura 4.9 apresenta uma primeira versão do descompressor integrado ao sistema de *Fetch* do Leon2. Observe que existe uma linha tracejada dividindo o controlador da I-cache em dois. O descompressor, bem como todo hardware usado para integração está representado ao lado direito da linha tracejada (exceto quando se tratar de modificação de um hardware já existente no processador). A caixa preta *descompressor* refere-se ao que foi detalhado na Figura 4.2. Além desta caixa preta, existe um buffer *r.buf* com a função de armazenar um ComPacket durante os ciclos nos quais suas instruções são descomprimidas.

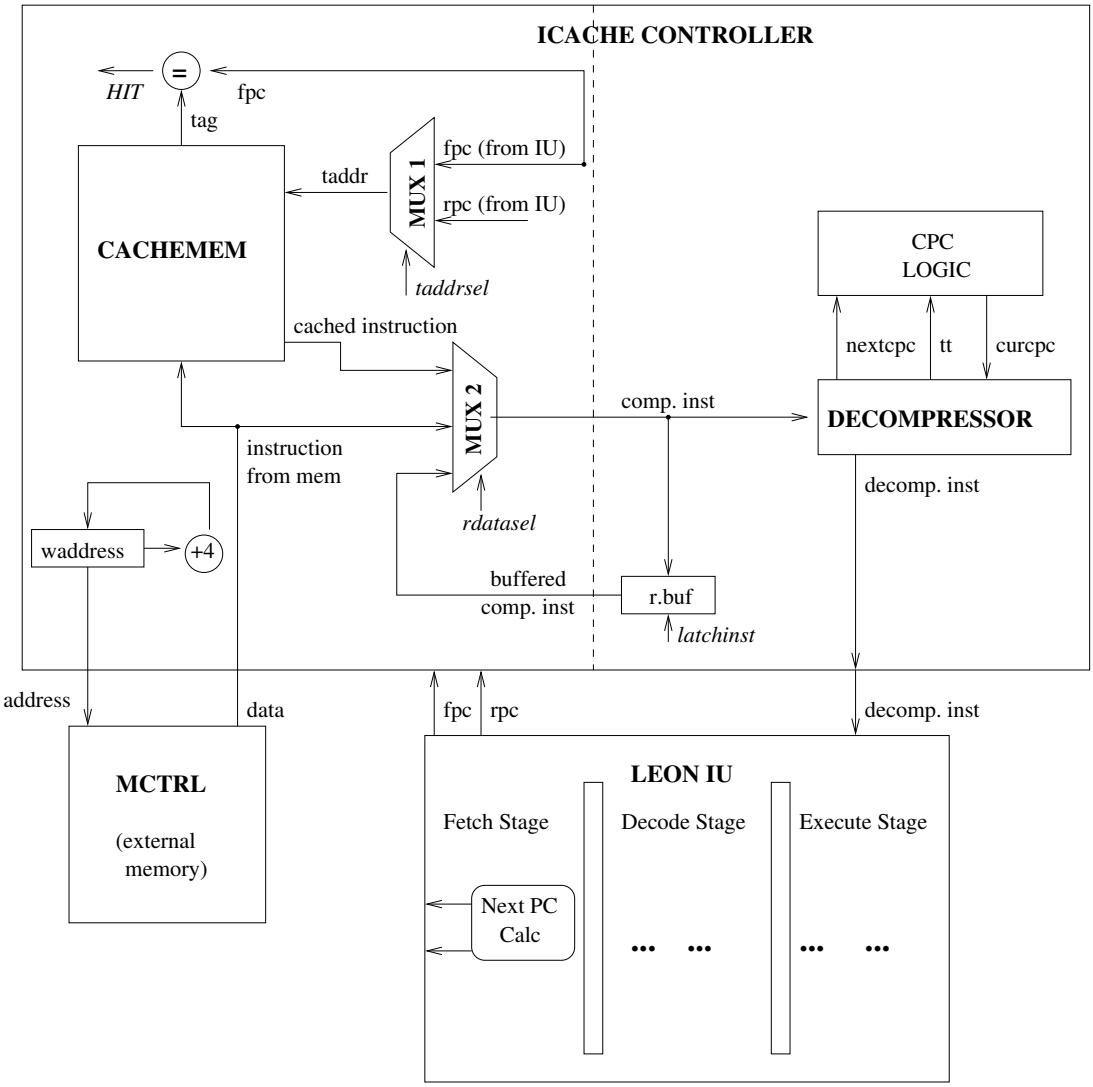


Figura 4.9: Descompressor integrado ao LEON2

Em relação a alterações no hardware já existente, pode-se notar que uma nova entrada foi adicionada ao *MUX 2*. Esta nova entrada seleciona *r.buf*, já que a descompressão das instruções de um ComPacket leva dois ou mais ciclos. Desta forma, a partir da segunda instrução (segundo índice), *r.buf* fornece a palavra comprimida. A partir daí, a saída deste multiplexador passa pela unidade de descompressão e a palavra descomprimida é fornecida ao processador.

A lógica seqüencial para controle do *cpc* (*cpc logic*), será detalhada posteriormente.

4.4.2 Estudo de caso - programa seqüencial

Para verificar a validade do modelo proposto, bem como incrementá-lo, será realizado um estudo de caso, inicialmente considerando um programa puramente seqüencial (sem saltos). A Figura 4.10 mostra um trecho de código real (parte superior) e o comportamento dos sinais ao longo do tempo (parte inferior). Os sinais são os seguintes:

- *fpc*: PC atual, fornecido pelo processador;
- *inst*: Palavra comprimida (saída do *MUX 2*);
- *cpc*: CPC atual, fornecido pela lógica do CPC embutida a I-cache;
- *nextcpc, flushing, decompressed_inst*: Saídas fornecidas pelo descompressor;
- *wait*: Este é o único sinal que ainda não havia sido comentado. Sua composição é ***wait = not (eholdn and not ici.nullify)***. Ambos os sinais, *eholdn* e *ici.nullify* já fazem parte da I-cache. *Eholdn* assume nível lógico baixo para solicitar que o processo de busca seja interrompido, podendo ser, por exemplo, porque a D-cache (cache de dados) está fazendo uma operação de leitura/escrita. Isto porque a I-cache e D-cache compartilham o controlador de acesso ao barramento AHB. Outro motivo que pode levar o *eholdn* a assumir um valor baixo é quando a unidade de ponto flutuante está realizando uma operação demorada, necessitando, com isso, que o *pipeline* do processador seja interrompido e, consequentemente, a operação de busca também. O sinal *nullify*, por sua vez, assume nível lógico alto também para indicar a necessidade de interrupção do processo de busca, porém, por outros motivos, tais como: *data hazards* e *control hazards*. Ou seja, o sinal de *wait* vai para nível lógico alto, quando o processo de busca deve ser interrompido, seja pelo sinal de *eholdn* ou *ici.nullify*.

Sequential code (without branches)

Address	Compressed Code / Type	Uncompressed Code / Assembly	
4000150c	207DFDFC ComPacket 3	9E100003	mov %g3, %o7
		87480000	mov %psr, %g3
		8628EF00	andn %g3, 0xf00, %g3
40001510	8188C000 Uncompressed	8188C000	mov %g3, %psr
40001514	00000007 ComPacket 4	01000000	nop
		01000000	nop
		01000000	nop
		81C3E008	retl
40001518	01000000 Uncompressed	01000000	nop

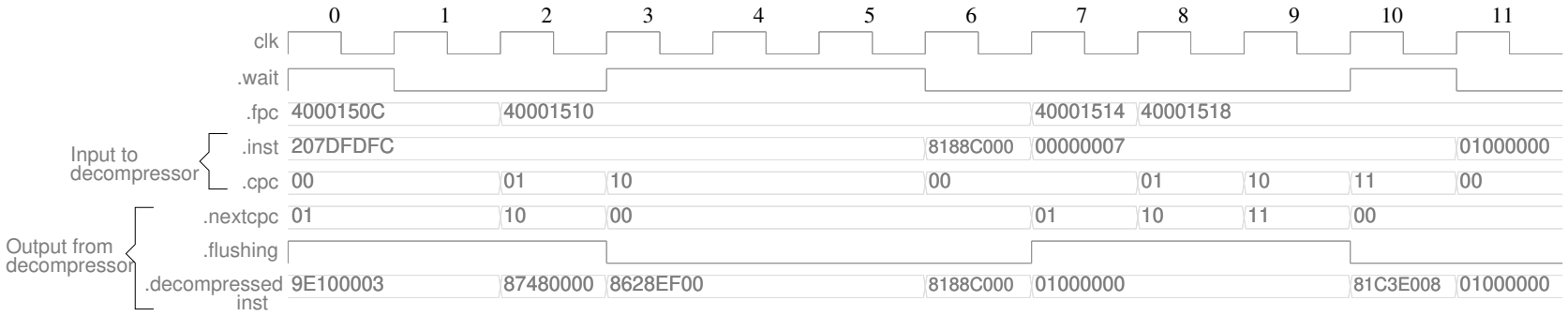


Figura 4.10: Programa seqüencial: primeira análise

Do ciclo 0 ao 5 é descomprimida a primeira palavra comprimida (0x4000150c), com 3 ciclos (ciclos 0, 2 e 3) de descompressão das três instruções compondo o ComPacket3 e outros 3 ciclos de interrupção devido ao sinal de *wait*. Entre 0 e 5, observe que o *cpc* varia de 00_2 a 10_2, contemplando assim os 3 índices, sendo que a contagem só é feita quando o sinal *wait* está baixo.

No ciclo 6 é buscada a palavra referente ao endereço 0x40001510, que se trata de uma instrução não comprimida, sem necessidade de descompressão.

Do ciclo 7 ao 10 é descomprimida a terceira palavra, referente ao endereço 0x40001514 e, finalmente, no ciclo 11, a palavra não comprimida referente ao endereço 0x40001518 é buscada, sem necessidade de descompressão.

A partir desta primeira análise, já se pode esboçar um hardware inicial para lógica de CPC, como mostrado na figura Figura 4.11. Existe um registrador (*r.cpc*) que contém o índice do ComPacket sendo processado, repassado à entrada do descompressor (*deci.cpc*). Este registrador é atualizado para *deco.nextcpc* toda vez que o sinal de *wait* não estiver ativo.

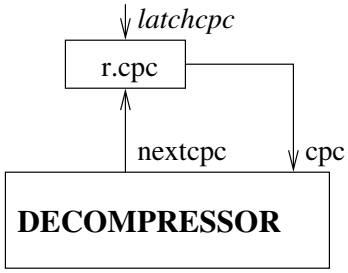


Figura 4.11: Lógica de CPC seqüencial

Continuando este estudo de caso, a Figura 4.12 apresenta outras formas de onda. O trecho de análise é exatamente o mesmo do caso anterior, porém, os sinais sob análise são outros. Desta vez não é mais usado o *fpc*, mas sim *taddr*. Além deste, as formas de onda contêm o sinal *taddrsel* usado para definir a seleção do *MUX 1*, *fpc* ou *rpc*. Outra diferença é que não é mais mostrada *deci.inst*, saída do *MUX 2*, mas sim, *ico_data*, que é a instrução da saída da *cachemem*. Isto será usado para definir *rdatasel*.

Sequential code (without branches)

Address	Compressed Code / Type	Uncompressed Code / Assembly	

4000150c	207DFDFC
	ComPacket 3
		9E100003
		87480000
		8628EF00
			mov %g3, %o7
			mov %psr, %g3
			andn %g3, 0xf00, %g3

40001510	8188C000 Uncompressed	8188C000	mov %g3, %psr
40001514	00000007 ComPacket 4	01000000 01000000 01000000 81C3E008	nop nop nop retl
40001518	01000000 Uncompressed	01000000	nop

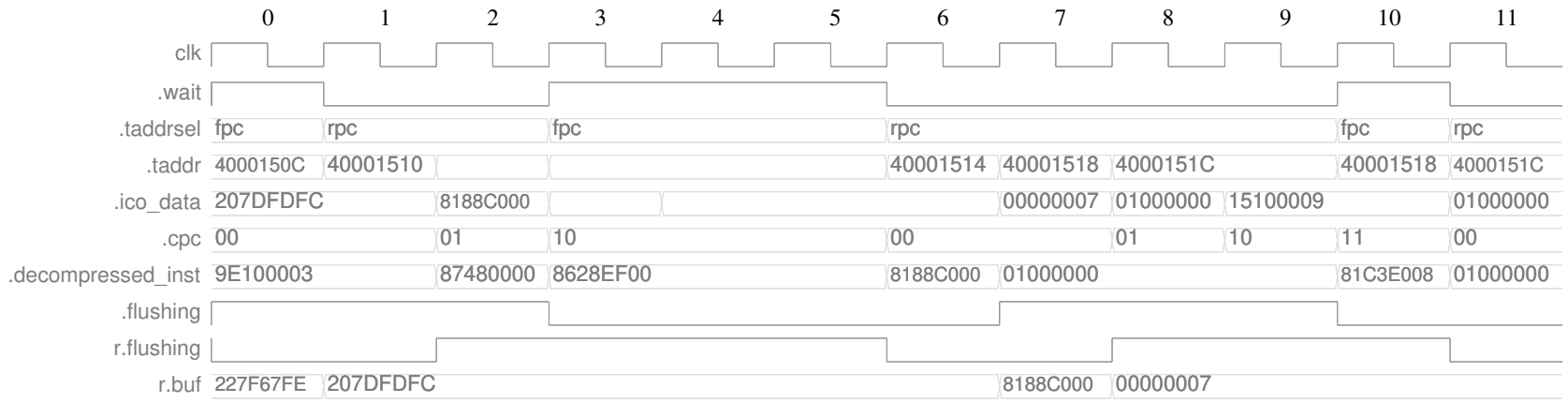


Figura 4.12: Programa seqüencial: segunda análise

O registrador *r.buf*, já citado anteriormente, serve para guardar uma palavra comprimida durante os ciclos em que suas instruções serão descomprimidas. Em um ComPacket 4, por exemplo, durante a descompressão do primeiro índice, a I-cache apresenta em sua saída a palavra correta. Porém, nos demais ciclos necessários para descomprimir o segundo, terceiro e quarto índices, a palavra não será mantida estável na saída da I-cache, pois no projeto original do processador, a I-cache já deveria estar fornecendo a palavra posterior. O *r.buf* soluciona este problema. Nos ciclos 0, 6 e 7, onde o primeiro índice dos endereços 0x4000150c, 0x40001510 e 0x40001514 estão sendo processados, a palavra deve vir diretamente da saída da memória da *cache* (*cachemem*), referenciada pelo sinal ico_data e, além disso, a palavra é gravada no registrador *r.buf*, para descompressão dos demais índices. Estes ciclos são identificados pelo nível baixo de *r.flushing*, que é sinal registrado de *deco.flushing* (só é registrado quando o sinal *wait* não estiver ativo). Desta análise, pode-se inferir os seguintes sinais de controle:

- **latchflushing = not wait**
- **latchinst = not r.flushing**
- **rdatasel = r.flushing** (*rdatasel* faz a seleção do MUX. Se *rdatasel* = 0, então o MUX assume a saída da I-cache (*ico_data*), caso contrário assume *r.buf*).

Outro ponto que pode ser observado a partir das formas de onda da Figura 4.12 é o sinal *taddrsel*, que define qual o endereço é enviado para *cachemem*, *fpc* ou *rpc*. Antes de obter a expressão que define *taddrsel*, cabe levantar um detalhe que ainda não foi explicado. Observe que a instrução de saída da I-cache (*ico_data*), refere-se ao endereço *taddr* calculado no ciclo anterior. Por exemplo, *taddr* vale 0x40001510 no ciclo 1, porém, a instrução referente a este endereço (0x8188C000) aparecerá apenas no ciclo 2.

Originalmente, o sinal *taddrsel* era definido pela expressão: **taddrsel = wait**. Ou seja, quando o sinal *wait* contiver nível lógico baixo, significa que no próximo ciclo a I-cache deverá apresentar a próxima instrução e, portanto, *rpc* deve ser selecionado. Caso *wait* apresente nível alto, significa que o processo de busca foi interrompido por algum motivo e, neste caso, no próximo ciclo a I-cache precisa manter sua saída, o que é feito selecionando *fpc*. Esta lógica seria suficiente se não houvessem palavras comprimidas. Observe que no ciclo 10, mesmo *wait* estando baixo, o *taddrsel* deve valer *fpc* (e não *rpc*). Neste ciclo, está sendo descomprimida o quarto e último índice da palavra localizada no endereço 0x40001514. Desta forma, no próximo ciclo (11), a palavra que deve aparecer na

saída é aquela referente ao endereço 0x40001518, portanto *fpc* (embora não mostrado na figura, *fpc* está valendo 0x40001518 e *rpc* está valendo 0x4000151C). Este caso particular ocorre pois um ComPacket leva alguns ciclos para ser consumido. Durante este período, a unidade de busca do processador incrementou um PC que nem chegou a ser consumido (se não houvesse compressão, este PC teria sido consumido). Este problema foi facilmente resolvido com uma pequena extensão da expressão que define *taddrsel*, assinalada em negrito:

- *taddrsel* = *wait* ***or (r.flushing and not deco.flushing)***

A expressão indica que, além do sinal de *wait*, o fato de estar processando uma instrução comprimida ($r.flushing = 1_2$) e estar na última posição ($deco.flushing = 0_2$) faz com que o *taddrsel* valha *fpc*.

A Figura 4.13 apresenta o diagrama com a incorporação das modificações descritas nesta subseção.

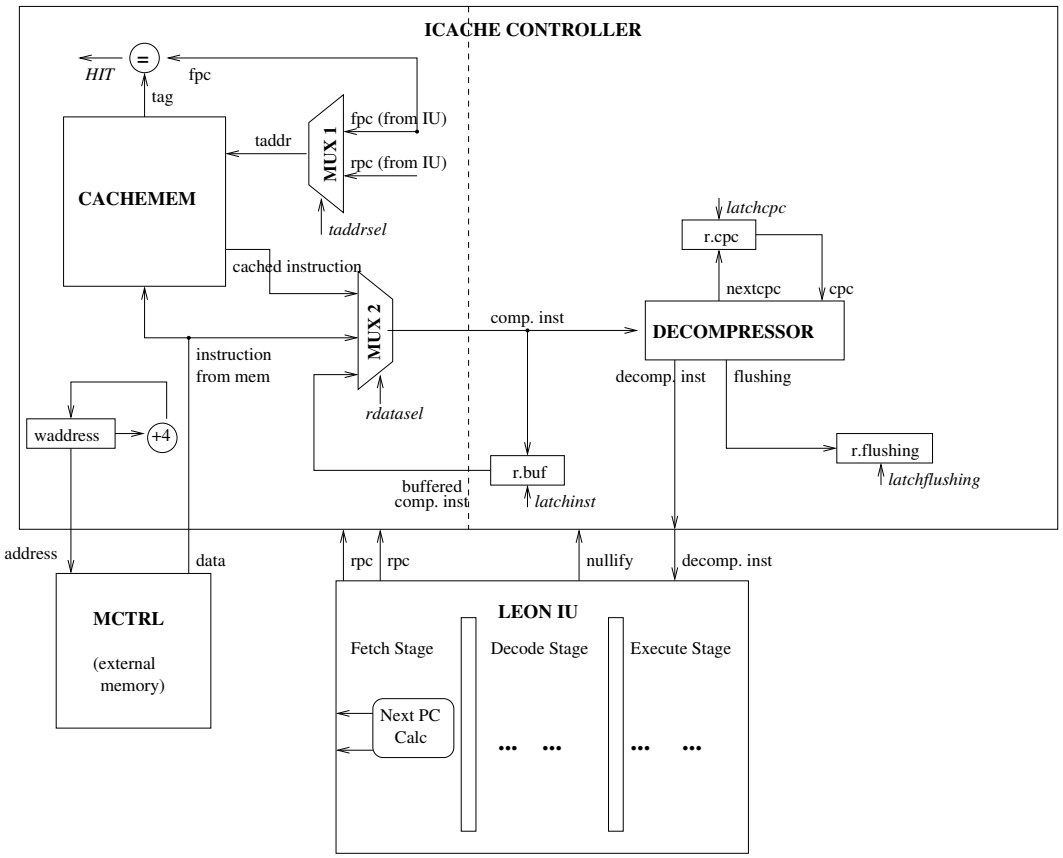
4.4.3 Estudo de caso - programa com saltos

Resolvido o problema dos programas seqüenciais, pode-se agora analisar programas com saltos. A Figura 4.14 mostra a execução de um trecho de código com um salto (parte superior) e algumas formas de ondas (parte inferior). Dos sinais tratatos, os únicos que ainda não foram apresentados são: *ici.fbranch* e *ici.rbranch*, os quais serão apresentados a seguir.

Do ciclo 0 ao 3, são executadas os índices 1 e 2 da palavra contida no endereço 0x4000115c. Cabe lembrar que a palavra comprimida está vindo de *r.buf* (e não da saída da I-cache), portanto, o *taddr*, mesmo valendo 0x40001160, não está sendo usado.

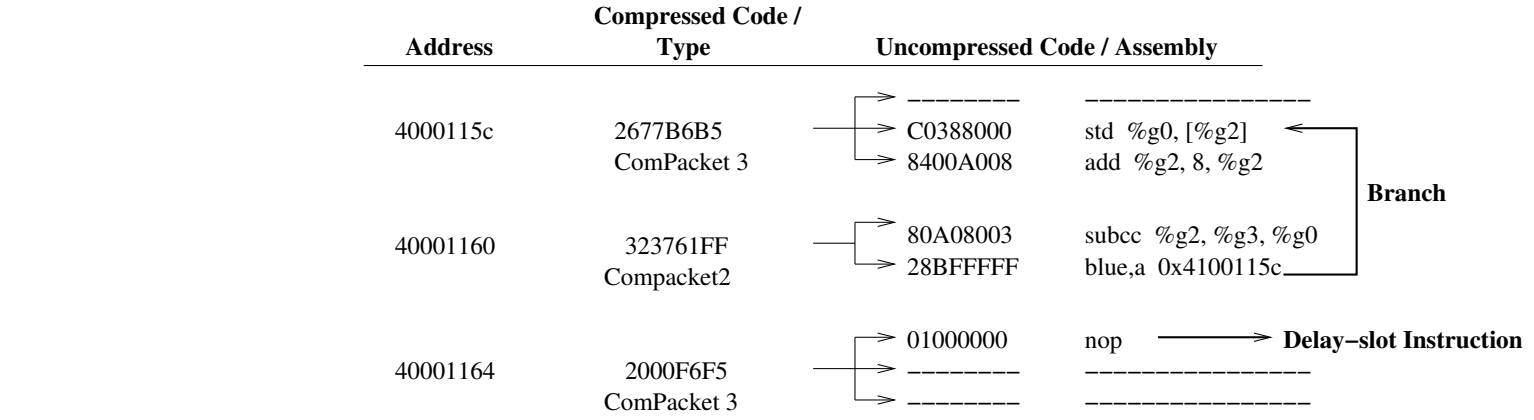
Nos ciclos 4 e 5, são executadas as instruções referentes à palavra comprimida do endereço 0x40001160. O segundo índice desta palavra é justamente o salto condicional para o endereço 0x4000115c/índice 01_2.

Finalmente, nos ciclo 6 e 7 é executada a instrução de *delay-slot*, que é o primeiro índice (00_2) da palavra referente ao endereço 0x40001164 (lembrando novamente que num dos ciclos a I-cache fica em estado de espera devido ao sinal de *wait*).



wait = not (eholdn and not ici.nullify) *latchinst = not r.flushing* *taddrsel = (wait) or (r.flushing and not deco.flushing)*
latchflushing = not wait *rdatasel = r.flushing* *latchcpc = not wait*

Figura 4.13: LEON2 com descompressor: apenas programas seqüenciais



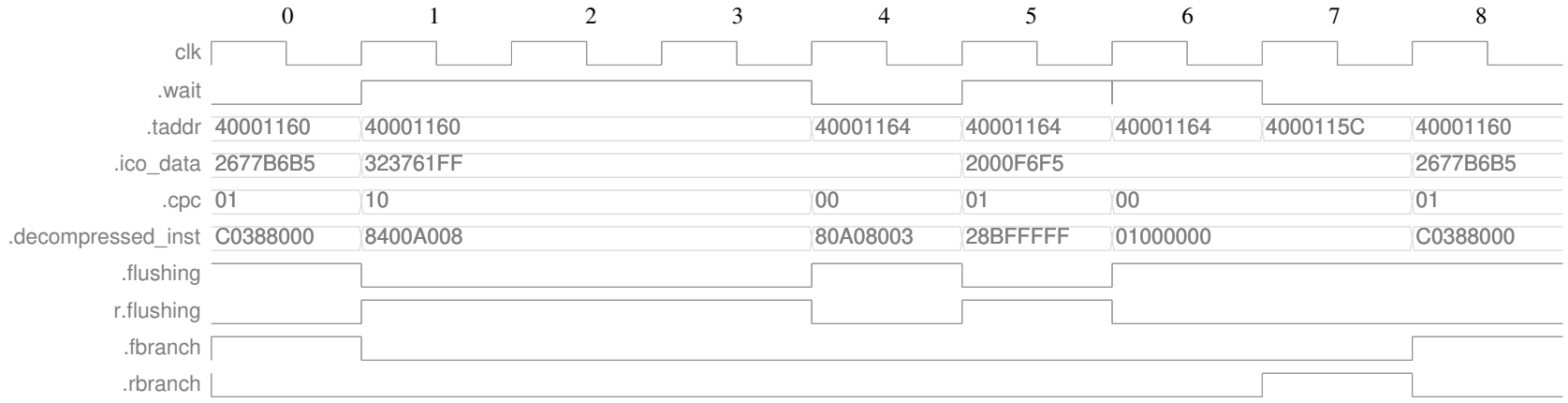


Figura 4.14: Programa com saltos

O sinal *rbranch* é gerado pelo processador, possuindo nível lógico alto quando existir uma instrução de salto no estágio de decodificação do *pipeline*. Isto pode ser visto no ciclo 7, quando a instrução de salto (endereço 0x40001160, índice $\mathtt{01}_2$), está sendo decodificada. O sinal *fbranch* é o sinal *rbranch* defasado de 1 estágio (não necessariamente defasado de 1 ciclo, pois este sinal só é registrado quando o *pipeline* caminha). Estes dois sinais serão usados para definir o hardware que executa programas com saltos.

Para permitir saltos a partir de ComPackets, é necessário fazer uma primeira modificação no hardware do descompressor. Observe que apenas a primeiro índice do endereço 0x40001164 deve ser executado (ciclos 6 e 7), referente ao *delay-slot*. Se o sinal *r.flushing* fosse sempre registrado a partir de *deco.flushing*, como definido anteriormente, a palavra seria inteiramente descomprimida e os índices 2 e 3 seriam incorretamente executados. Para contornar esta situação, *r.flushing* deve ser zerado no caso de haver o salto durante a execução de um ComPacket. O hardware da Figura 4.15 contempla este caso, usando o sinal *rbranch* para detectar o salto.

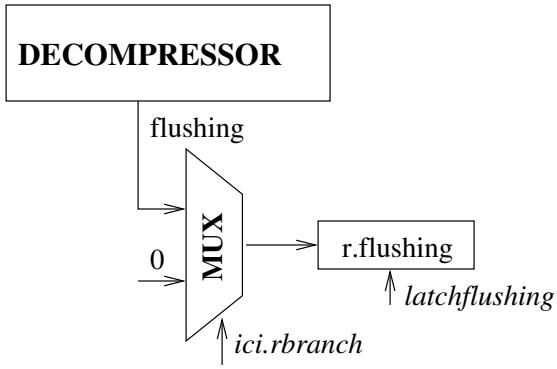


Figura 4.15: Lógica de flushing

No caso mostrado, o cancelamento na descompressão é feito durante a execução da palavra no *delay-slot*. Em um outro caso, supondo um ComPacket 4 com índices apontando para as instruções: *BRANCH*, *NOP*, *NOP*, *NOP*. Analogamente ao caso anterior, haveria necessidade de se cancelar a descompressão, sendo que apenas os índices $\mathtt{00}_2$ e $\mathtt{01}_2$ do ComPacket deveriam ser executados (supondo que o salto foi tomado, evidentemente). O hardware descrito acima contempla também este caso.

A segunda modificação necessária no hardware descompressor diz respeito aos alvos dos saltos. No exemplo dado, o salto é feito para o endereço 0x4000115C, mas para a instrução referente ao índice $\mathtt{01}_2$. Ou seja, o índice $\mathtt{00}_2$ não deve ser executado. Este caso é previsto pelo método de compressão, sendo que o campo *tt* da palavra comprimida

define o índice alvo do salto. Neste caso, no ciclo 7, embora o *cpc* valha 00_2, o que é efetivamente considerado é o campo *tt* da palavra referente ao endereço 0x4000115C, que vale 01_2. A Figura 4.16 mostra o hardware remodelado para cálculo do próximo *cpc*.

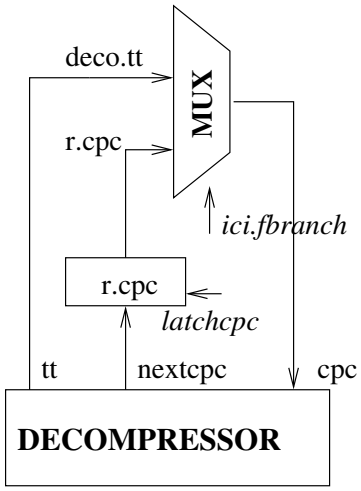


Figura 4.16: Lógica de CPC não seqüencial

Além dessas alterações na I-cache, algumas pequenas modificações devem ser feitas no *pipeline*. A alteração mais evidente é interromper o cálculo do próximo PC, enquanto uma instrução comprimida estiver sendo processada. Isto é feito facilmente no estágio de busca, sendo que a carga dos registradores deste estágio é controlada pelo sinal *holdpc*, que é uma saída que foi criada na I-cache indicando se uma instrução comprimida está sendo processada (*r.flushing*) e, neste caso, o PC deve ser interrompido.

Uma segunda alteração no *pipeline* refere-se aos saltos. No capítulo anterior, quando foi apresentado o método PDC-ComPacket, levantou-se algumas restrições em relação aos saltos. Uma delas era evitar que instruções de *CALL* fossem comprimidas, de modo a facilitar o cálculo do PC de retorno de uma função. No entanto, existe um caso excepcional que não é tratado pelo método durante a compressão e, portanto, deve ser tratado pelo descompressor. Este caso está ilustrado na Figura 4.17. Observe que no primeiro caso é executado uma instrução de salto (*JMP 0x40001000*) seguido da instrução de *delay-slot* (*NOP*). Neste primeiro caso, a instrução de retorno da chamada à função seria 0x40000008. Já no segundo caso, a instrução de *delay-slot* é um *ComPacket4* e, o endereço de retorno é 0x40000004 (e não 0x40000008). Para solucionar este problema, a modificação é feita no estágio de execução, como mostrado na parte inferior da mesma figura. Ao multiplexador que calcularia o endereço de resultado alvo, é adicionada uma nova entrada com o PC de

retorno correto, ou seja, *EX.PC - 4*. O sinal de controle *alusel* é modificado de modo a selecionar esta entrada, caso a instrução presente no estágio de decodificação (anterior), referente ao *delay-slot*, seja comprimida.

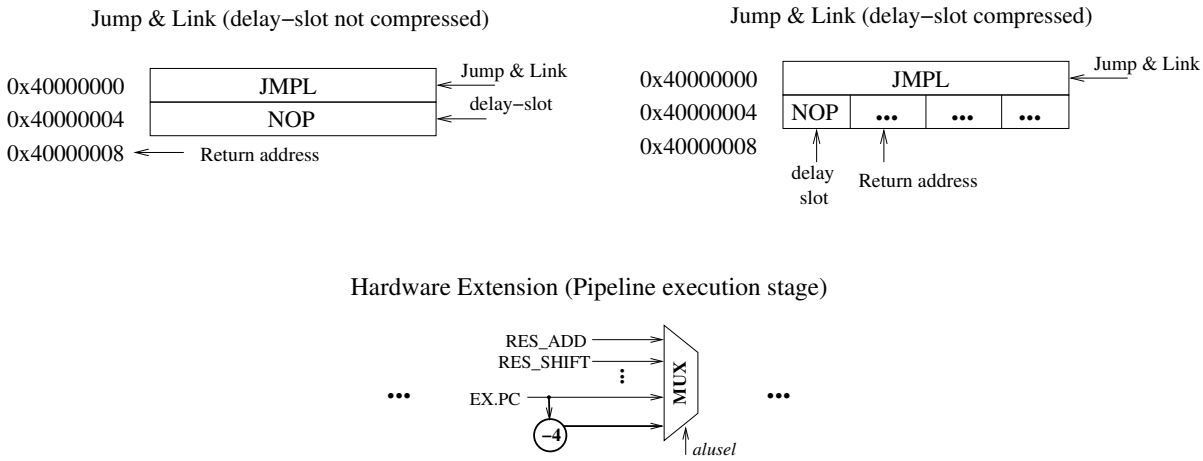


Figura 4.17: *Jump-and-link* com instrução de *delay-slot* comprimida

Mecanismo de Bursting

Até o momento o hardware descompressor contempla aqueles casos onde há acerto no acesso a I-cache, sem levar em conta os casos de falha em que as instruções são buscadas na memória e, ao mesmo tempo, encaminhadas ao processador, através do mecanismo de *bursting*, já introduzido na Figura 4.6. Neste momento, será detalhado um pouco mais o funcionamento do *bursting* para posteriormente fazer as modificações/extensões necessárias para finalmente chegar a versão completa do hardware descompressor.

Como já mencionado, a I-cache do Leon2 é responsável por controlar o andamento/interrupção do *pipeline* de acordo com a disponibilidade/não disponibilidade da instrução requisitada. Isto é feito através do sinal *ico.holdn*, que é uma saída da I-cache ao processador. Quando *ico.holdn* tem valor lógico alto, significa que a instrução está disponível e que o *pipeline* pode andar. Do contrário, o *pipeline* deve permanecer interrompido. Ele não havia sido mencionado até agora, pois só se tratou de situações de acerto e, portanto, nestes casos *ico.holdn* sempre tinha nível alto.

Quando acontece uma falha de acesso, a I-cache passa do estado de **hit** para o estado de **miss**, o que é indicado pelo sinal *r.istate*. Neste caso, como houve uma falha de acesso, o sinal de *holdn* vai para nível baixo e só retorna para nível alto quando a instrução

buscada na memória estiver disponível, para fazer o *pipeline* andar e o PC do processador ser incrementado. Ou seja, durante o *bursting*, o sinal de *holdn* varia entre baixo e alto, de acordo com a disponibilidade das instruções buscadas na memória.

Este mecanismo descrito funciona perfeitamente para códigos seqüenciais, já que as instruções vão sendo buscadas seqüencialmente na memória até encherem a linha da I-cache. Desta forma, cada vez que uma instrução estiver disponível, o *pipeline* pode andar e o PC ser incrementado de modo que a próxima instrução seja buscada. Mas o que fazer quando houver um salto durante o *bursting*? A solução adotada no Leon2 é continuar a busca das instruções na memória até encher a linha da *cache*, porém, elas não são mais repassadas ao processador. Ou seja, executa-se a instrução de salto e *delay-slot* e, após isto, *holdn* fica com valor baixo, já que a instrução referente ao alvo do salto ainda não está disponível. Terminado o *bursting*, a I-cache volta ao estado de *hit* e, caso a instrução alvo do salto realmente não esteja presente na I-cache, o estado é novamente modificado para *miss*, repetindo o processo de *bursting* descrito.

A idéia por traz de permitir compressão, continuando com a funcionalidade de *bursting* é análoga ao caso de saltos. Se, durante a fase de *bursting*, houver uma instrução comprimida, o preenchimento da linha da I-cache continua, mas o sinal de *holdn* vai para nível baixo, pois o PC do processador não pode continuar sendo incrementado. Desta forma, é executada apenas a primeira instrução do ComPacket, sendo que as demais instruções são executadas durante o estado de *hit*, fazendo com que o modelo proposto até aqui continue válido.

O mecanismo de *bursting* do Leon2 é bastante complexo. Além do caso de salto citado anteriormente, o processo de busca da instrução pode ser interrompido durante o *bursting* por vários outros fatores: *ici.nullify*, *eholdn*, etc. Existe toda uma lógica que identifica estes casos e interrompe o *bursting* através do sinal de *underrun*. Quando *underrun* estiver alto significa que as instruções não serão mais enviadas ao processador (*holdn* baixo). A seguir a expressão que define este sinal. Em negrito, a parte que foi adicionada para permitir compressão:

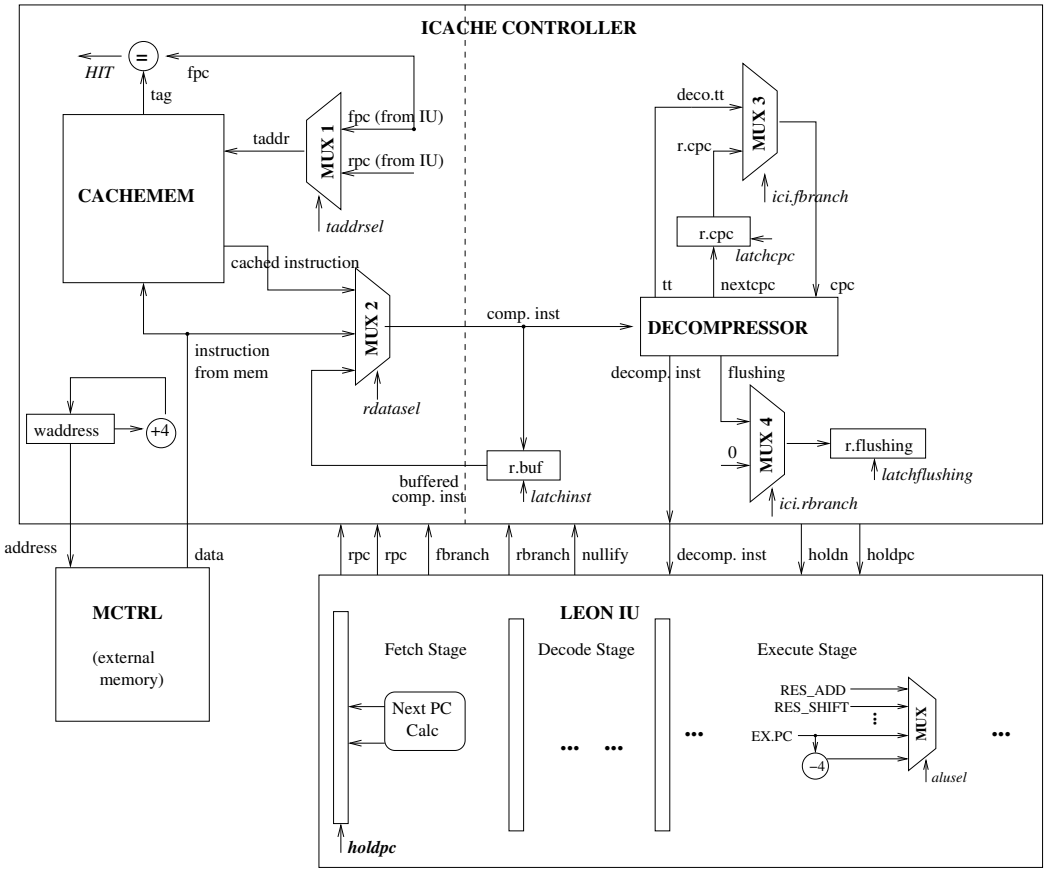
- *underrun := r.underrun or (write and ((ici.nullify or not eholdn) and (mcio.ready and not (r.overrun and not r.underrun)))) or (branch) or (**mcio.ready and deco.flushing**)*

Ou seja, se houver uma palavra disponível na saída da memória (*mcio.ready*) e for comprimida (*deco.flushing*), o processo de envio das instruções ao processador é

interrompido.

Continuando as alterações necessárias para permitir o *bursting*, o sinal *rdatasel*, definido anteriormente, precisa ser estendido, já que em *bursting* as instruções enviadas ao processador provêm diretamente da saída da memória. Para isso o sinal é redefinido da seguinte forma: **rdatasel = r.istate,r.flushing** (concatenação, formando 2 bits). Ou seja, o estado da I-cache (*hit* ou *miss*) será levado em conta para definir a seleção a palavra comprimida.

Feitas todas as alterações, pode-se finalmente apresentar o diagrama completo do hardware descompressor na Figura 4.18.



wait = not (eholdn and not ici.nullify) *latchinst = not r.flushing* *taddrsel = (wait) or (r.flushing and not deco.flushing)*
latchflushing = not wait *rdatasel = r.flushing, r.istate* *latchcpc = not wait*

Figura 4.18: LEON2 com descompressor: diagrama completo

4.4.4 Descompressor Completo: exemplo

Para ilustrar o funcionamento do descompressor (projeto completo), será exercitado o código mostrado na Figura 4.19, já estudado anteriormente.

A Figura 4.20 mostra a execução da primeira instrução. Considera-se que já foi realizada uma iteração do laço. Ou seja, esta é uma instrução alvo de um salto, portanto, *cpc* assume o valor *tt*. Neste ciclo também, a palavra já está gravada em *r.buf*, de onde é fornecida ao descompressor. A Figura 4.21 mostra a execução da próxima instrução. A única diferença em relação ao ciclo anterior, é que, neste caso, o *cpc* é fornecido pela lógica seqüencial, valendo 10_2.

A Figura 4.22 mostra a execução da primeira instrução da palavra comprimida referente ao endereço 0x40001160. Note que, neste caso, a palavra vem da I-cache e é gravada em *r.buf* para uso posterior. A Figura 4.23 ilustra a execução da segunda instrução do ComPacket, que é um salto. Desta vez a palavra comprimida vem de *r.buf* e *cpc* está valendo 01_2.

Finalmente a Figura 4.24 mostra a execução da instrução de *delay-slot*. Note que, neste momento, a instrução de salto chegou ao estágio de decodificação do *pipeline*, fazendo *r.branch* ir para nível lógico alto, o que zera o registrador *r.flushing* (interrupção do processo de descompressão da palavra devido a um salto).

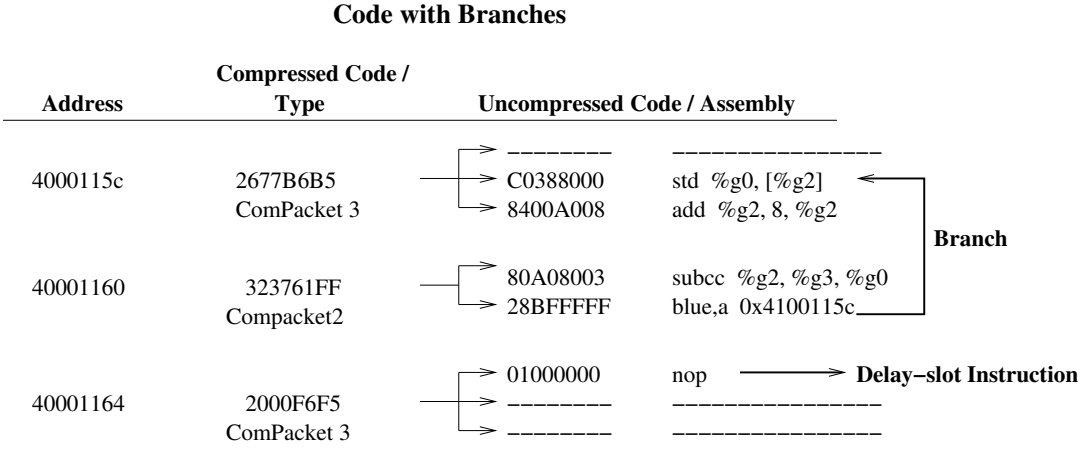


Figura 4.19: Código com saltos usado no exemplo

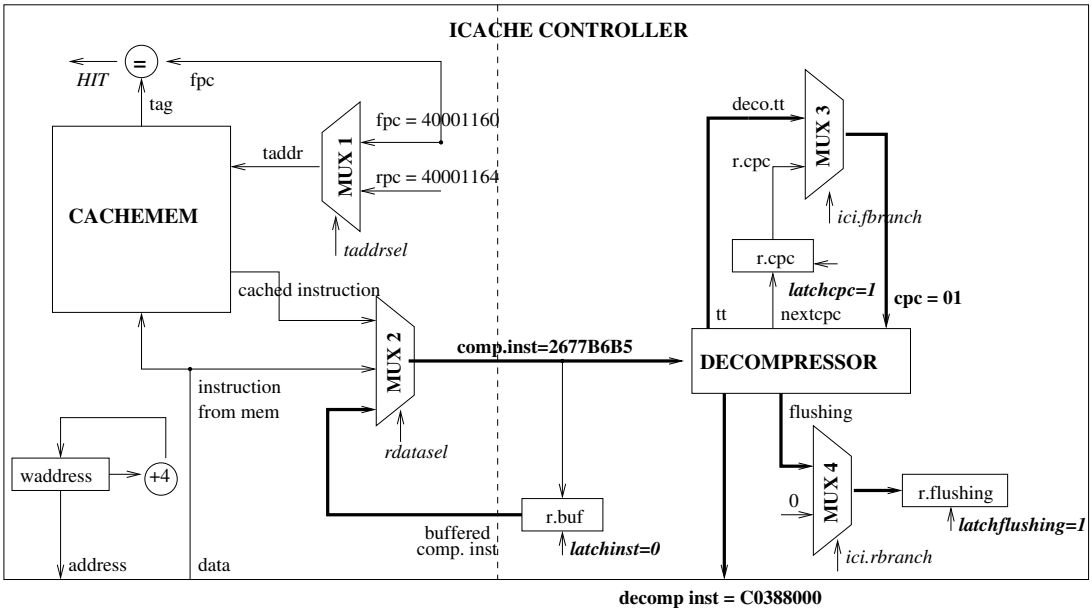


Figura 4.20: Exemplo de Execução no Descompressor: Instrução 1

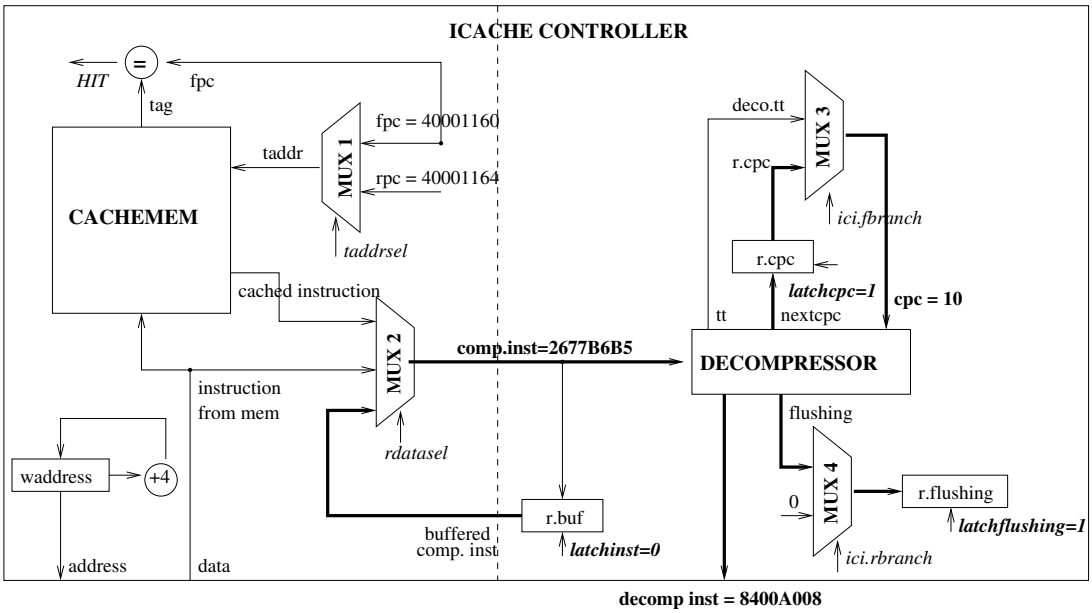


Figura 4.21: Exemplo de Execução no Descompressor: Instrução 2

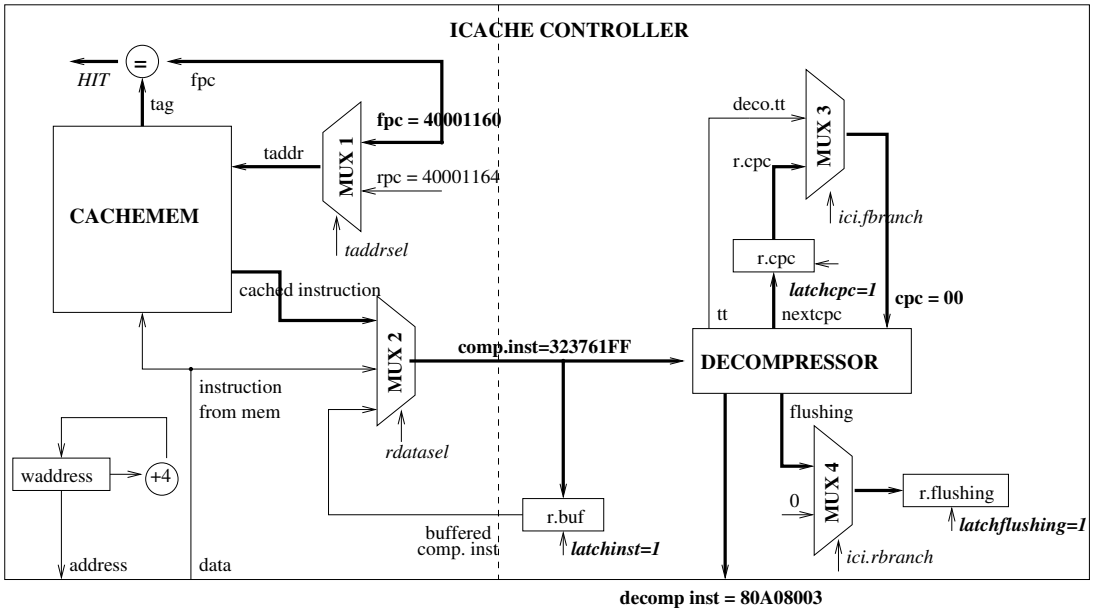


Figura 4.22: Exemplo de Execução no Descompressor: Instrução 3

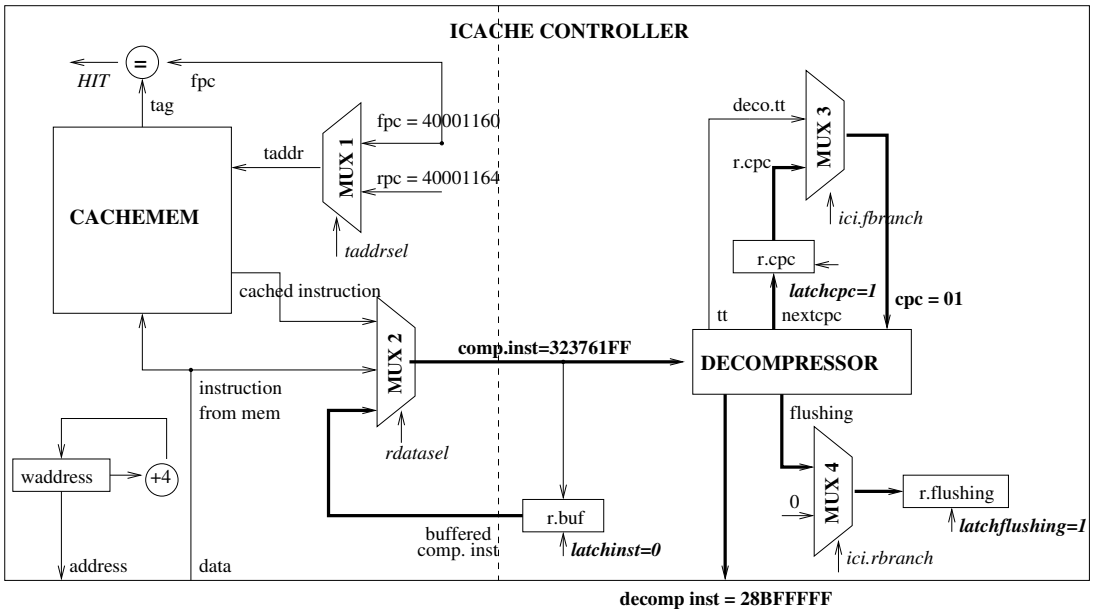


Figura 4.23: Exemplo de Execução no Descompressor: Instrução 4

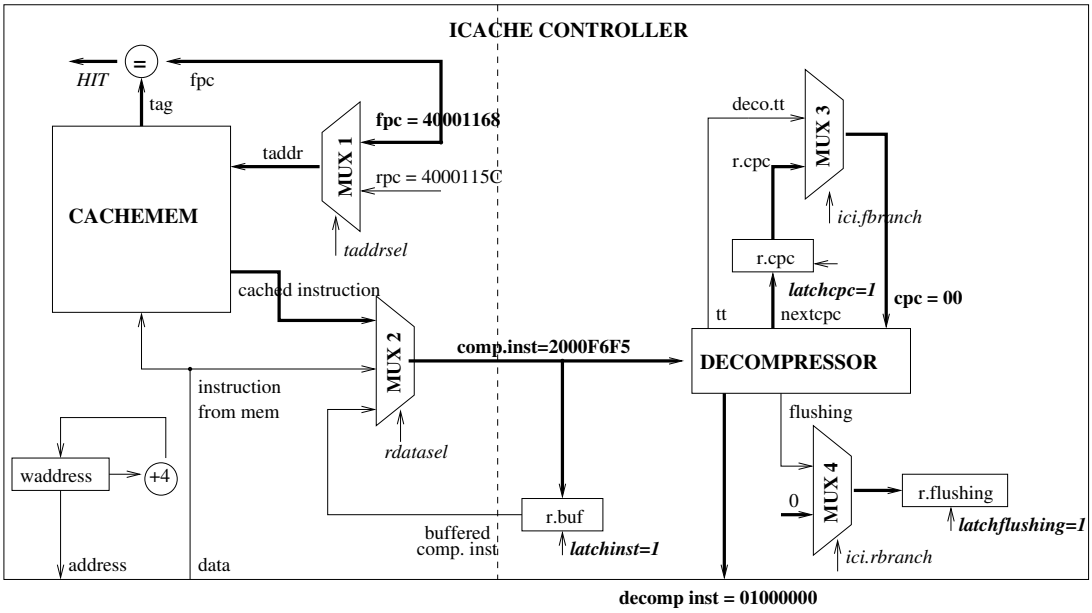


Figura 4.24: Exemplo de Execução no Descompressor: Instrução 5

4.5 Considerações Finais e Conclusões

O projeto apresentado atendeu todos requisitos levantados no início do capítulo. O requisito de modificar o projeto original do processador o mínimo possível ajudou no cumprimento de outro requisito que era usar o mínimo de hardware. Isto porque, a partir do momento que a concepção inicial de projeto do processador era obedecida, os detalhes de integração foram todos resolvidos com soluções simples: adição de um multiplexador, adição de algumas portas lógicas na expressão lógica de um sinal, etc. Isto levou a um hardware usado para fins de integração praticamente desprezível.

No caso do hardware usado para descompressão, também houve bastante economia. Fora o dicionário de instruções, todo o resto do hardware resume-se a algumas portas lógicas, multiplexadores e *shifters*. O fato de usar lógica distribuída no lugar das *SRAM's* acaba sendo uma vantagem em termos de uso de hardware, pois estas memórias estariam disponíveis para outro uso.

Os requisitos de *timing* também foram alcançados. Para isso vários cuidados tiveram que ser tomados: achar a melhor localização arquitetural do descompressor, paralelizar logicamente ao máximo a propagação dos sinais e, finalmente, fazer uso de sínteses físicas mais rápidas (lógica distribuída no lugar de *SRAM's*).

Em comparação aos trabalhos relacionados apresentados no capítulo 2, existe uma série de pontos positivos neste trabalho. Este foi o primeiro que ofereceu uma implementação de descompressão permitindo a funcionalidade de *bursting* na I-cache. O trabalho de Benini [26] é o único que comenta sobre permitir a funcionalidade de *bursting* junto com descompressão, porém, como um trabalho futuro. Testes realizados mostram que a funcionalidade de *bursting* aumenta consideravelmente o desempenho do processador. Sendo assim, se esta funcionalidade fosse inibida, este trabalho tornar-se-ia inválido em relação ao requisito de melhorar o desempenho do processador, pois, se por um lado a compressão traz ganhos, por outro existem perdas ao tirar esta funcionalidade. O segundo ponto positivo está em não alterar o *cycle-time* do processador, já que a maioria dos trabalhos apresentados o fazem. O terceiro ponto positivo está no fato de se ter chegado a uma implementação funcional. A maioria dos trabalhos não chegam a uma implementação real (em hardware) do descompressor, obtendo dados apenas através de simulações ou estimativas. A validação de um método através de simulações é questionável, tanto do ponto de vista dos resultados obtidos que podem ser bastante diferentes dos resultados reais (uso de simuladores não *cycle-accurate*), quando do ponto de vista da viabilidade

na implementação do método. Como foi visto no decorrer do capítulo, existe uma série de aspectos que podem inviabilizar a implementação. Finalmente, vale a pena levantar um ponto negativo que está na falta de generalidade deste projeto, já que ele é bastante específico ao processador Leon2. O descompressor até poderia ser projetado em outro processador, porém, as maiores dificuldades deste projeto, que são justamente aquelas referentes a aspectos de integração, na maioria dos casos seriam específicas do Leon2. Como foi mostrado ao longo deste capítulo, infelizmente a arquitetura ideal (genérica) sugerida para o método PDC-ComPacket seria **impossível** para o caso deste processador. Isto pode ser visto com um *tradeoff*: se por um lado perde-se em generalidade, já que este processador exige uma implementação bastante específica, por outro, ofereceu-se a implementação do método em um processador real, consagrado nos meios acadêmcio e industrial. Isto aumenta as chances de este método continuar a ser efetivamente estudado/utilizado no futuro.

Capítulo 5

Resultados Obtidos

Este capítulo visa apresentar os resultados obtidos com a implementação da arquitetura de compressão.

Primeiramente será detalhada toda a infraestrutura utilizada para realização dos experimentos. Depois disso, são relatados resultados de compressão, desempenho e consumo de energia, obtidos com variações de parâmetros como: tamanho das *caches*, *f*, etc. Finalmente, será apresentada uma demonstração prática do uso da técnica de compressão envolvendo a decodificação e reprodução de MP3.

5.1 Infraestrutura Utilizada

A infraestrutura necessária para os experimentos envolve recursos de Software para permitir a compilação, compressão dos programas e síntese do modelo do processador, bem como recursos de hardware, para a prototipagem do processador em FPGA.

5.1.1 Compilação e Compressão

Na Figura 5.1, são mostrados os recursos necessários para a compilação e compressão do programa fonte. O compilador utilizado foi o LECCS (*Leon/ERC32 GNU Cross-Compiler System*), desenvolvido pela *Gaisler Research*, que é uma integração de alguns pacotes públicos como: GCC, binutils, GDB, newlib, etc., devidamente adaptados para características particulares do Leon2. Os programas foram compilados com o LECCS com as seguintes opções:

- *-mv8*: Indicam que as operações de multiplicação e divisão estão implementadas em hardware como previsto na versão 8 do padrão SPARC e, desta forma, o compilador gera as instruções *umul* e *udiv* em vez de implementá-las em software;
- *-msoft-float*: Indica que o processador não tem unidade de ponto flutuante e, portanto, o compilador deve gerar código que faça operações deste tipo;
- *-O2*: Esta opção indica que devem ser feitas otimizações na geração do código.

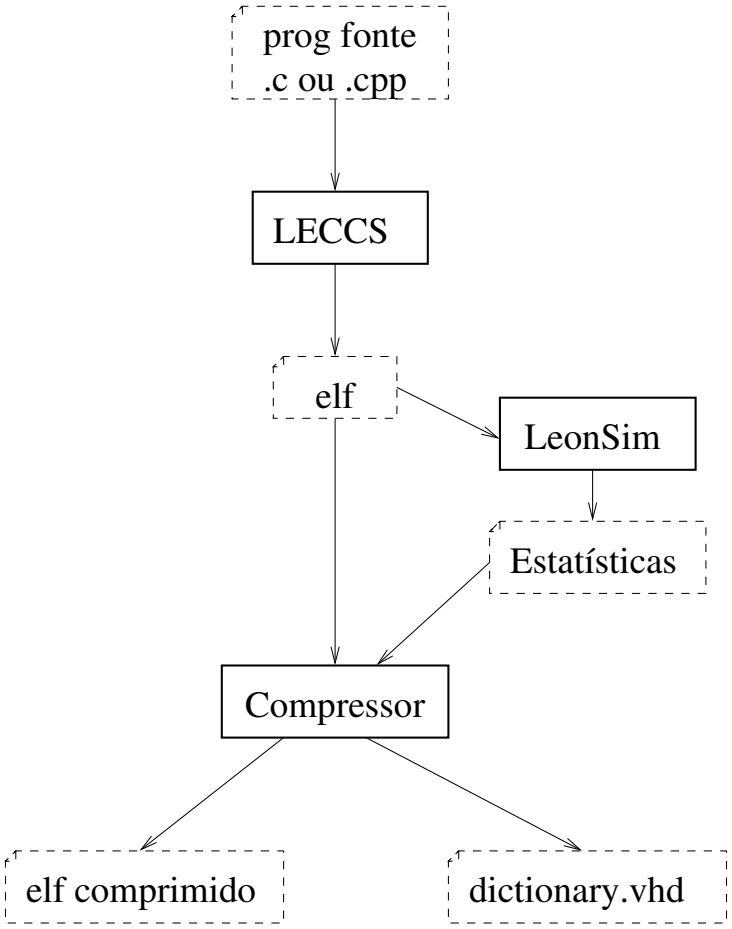


Figura 5.1: Infraestrutura: compilação e compressão

Depois de compilado, o arquivo ELF é executado no simulador (LeonSim) para obtenção de estatísticas dinâmicas (quantas vezes cada instrução é executada), bem como estatísticas estáticas (quantas vezes cada instrução aparece no programa). Estas estatísticas são passadas ao compressor, que gera o código ELF comprimido de acordo

com a razão f (dinâmico/estático) definida pelo usuário. Além do código comprimido, o compressor gera o código VHDL representando o dicionário de instruções, que posteriormente será anexado ao modelo do processador. Existem, também, outros arquivos gerados pelo compressor que são usados para depuração, como: arquivo com mapeamento entre código original e comprimido; arquivo com a marcação das instruções que são saltos e as que são alvos, entre outros.

O simulador do Leon (LeonSim), bem como o compressor, foram desenvolvidos como parte do trabalho de Wanderley [1], sendo que algumas extensões foram necessárias para tornar o ambiente de protipagem mais completo, tais como: permitir que o LeonSim aceitasse um arquivo ELF de entrada (antes havia um formato próprio de entrada) e fazer a geração automática do dicionário de instruções.

5.1.2 Desenvolvimento/Síntese/Validação do Descompressor

Continuando a apresentação da infraestrutura utilizada, a Figura 5.2 apresenta os recursos necessários para desenvolver o descompressor. Num primeiro nível, para testar o modelo do processador em VHDL que foi estendido com o descompressor, faz-se simulações de modo a validá-lo logicamente, usando o ModelSim da Mentor Graphics.

Uma vez validado logicamente, é feita a síntese lógica através do XST, que consiste em descrever o modelo em termos de uma lista de elementos lógicos (*netlist*), que posteriormente são traduzidos para portas lógicas, multiplexadores, RAMs, flip-flops pela ferramenta *NgdBuild*. Neste momento, pode ser feita a simulação *gate-level* (nível de portas), que valida a síntese lógica. Além disso, durante a simulação, pode ser gerado um arquivo com a anotação de todos as transições de sinais, a ser usado posteriormente para obtenção de estimativas de potência.

Um passo posterior na síntese física é a etapa de mapeamento, que consiste em mapear as portas lógicas para elementos físicos da tecnologica alvo (FPGA xc2v3000, por exemplo), o que é feito através da ferramenta *map*.

Após o mapeamento, vem a etapa de *Place & Route*, onde os elementos físicos devem ser atribuídos (*place*) e interconectados (*route*), de forma que o *cycle time* definido como requisito seja respeitado. Esta etapa é feita pela ferramenta *Place_Route*.

Finalmente, o arquivo gerado após a etapa de *Place & Route* é traduzido para um arquivo de programação através da ferramenta *Bitgen*. Este arquivo é a síntese física em si, pronta para ser carregada na FPGA.

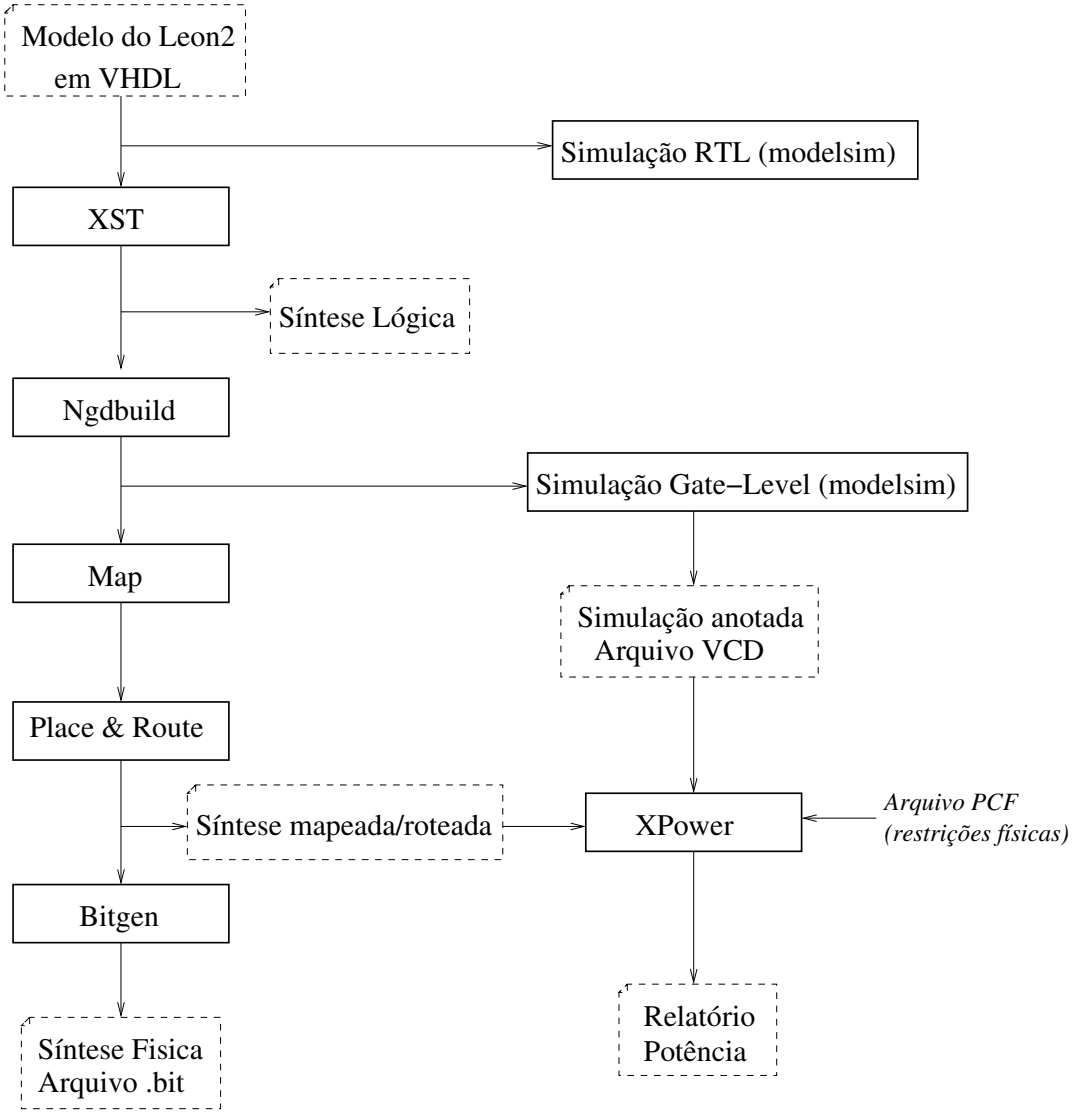


Figura 5.2: Infraestrutura: Desenvolvimento/Síntese/Validação do Descompressor

Um detalhe que não foi explicado até aqui é a obtenção das estimativas de potência. Isto é feito através da ferramenta *Xpower* a partir de três arquivos: arquivo anotado da simulação *gate level*, arquivo gerado após a etapa de *Place & Route* e o arquivo de restrições físicas (também usado pelo *bitgen*). Em linhas gerais, as anotações de transições de sinais são aplicadas à síntese física e, desta forma, as estimativas de potência são bastante confiáveis, muito próximas dos valores reais.

As ferramentas de desenvolvimento citadas: *Xst*, *Ngdbuild*, *Map*, *Place_route*, *Bitgen* e *Xpower* são da *Xilinx*, fazendo parte do *Xilinx ISE 6.1*.

5.1.3 Ambiente de Prototipagem

Para completar a infraestrutura utilizada, falta o ambiente de prototipagem, mostrado na Figura 5.3. A partir de uma estação de trabalho comum (PC), pode-se carregar a síntese para placa através da interface JTAG, com o uso da ferramenta *Impact* da *Xilinx*. Depois de carregada a síntese, pode-se depurar o processador através da aplicação *Dsumon* rodando no *host*, que se comunica serialmente com a interface de depuração do processador. Através desta aplicação, é possível carregar programas para as memórias da placa, executar programas, inserir *breakpoints*, obter *traces* de execução, ler/escrever na memória externa da placa (SRAM ou SDRAM), ler/escrever nos registradores do Leon, etc.

Durante a execução de um programa, a sua saída é direcionada à segunda serial e, assim, é possível visualizá-la em um terminal (*minicom*, por exemplo). Além disso, o *dsumon* tem uma interface GDB, permitindo que se faça depuração de alto nível (código fonte em C) de um programa executando no processador. Contudo, esta depuração só pode ser feita para código não comprimido. Para que fosse possível depuração também de código comprimido , seria necessário modificações na unidade de depuração do processador (DSU), no compilador GCC e no Dsumon, sendo que este último não tem código aberto, o que impede que isto seja feito, levando a um nível de depuração para códigos comprimidos bastante baixo. Esta foi uma das grandes dificuldades deste trabalho, já que a depuração tinha que ser feita no nível de *assembly*, sendo possível armazenar, no máximo, até 1024 instruções no *trace* de execução. Deste modo, diversas vezes, durante o desenvolvimento aconteciam saltos para posições incorretas (por algum problema na lógica de PC do descompressor ou por algum endereço de salto não resolvido pelo compressor) e, analisando apenas as instruções do *trace* não era possível descobrir

onde houve o salto incorreto. A solução era inserir *breakpoints* e verificar se eles eram atingidos, o que tornava a depuração de problemas bastante árdua e demorada.

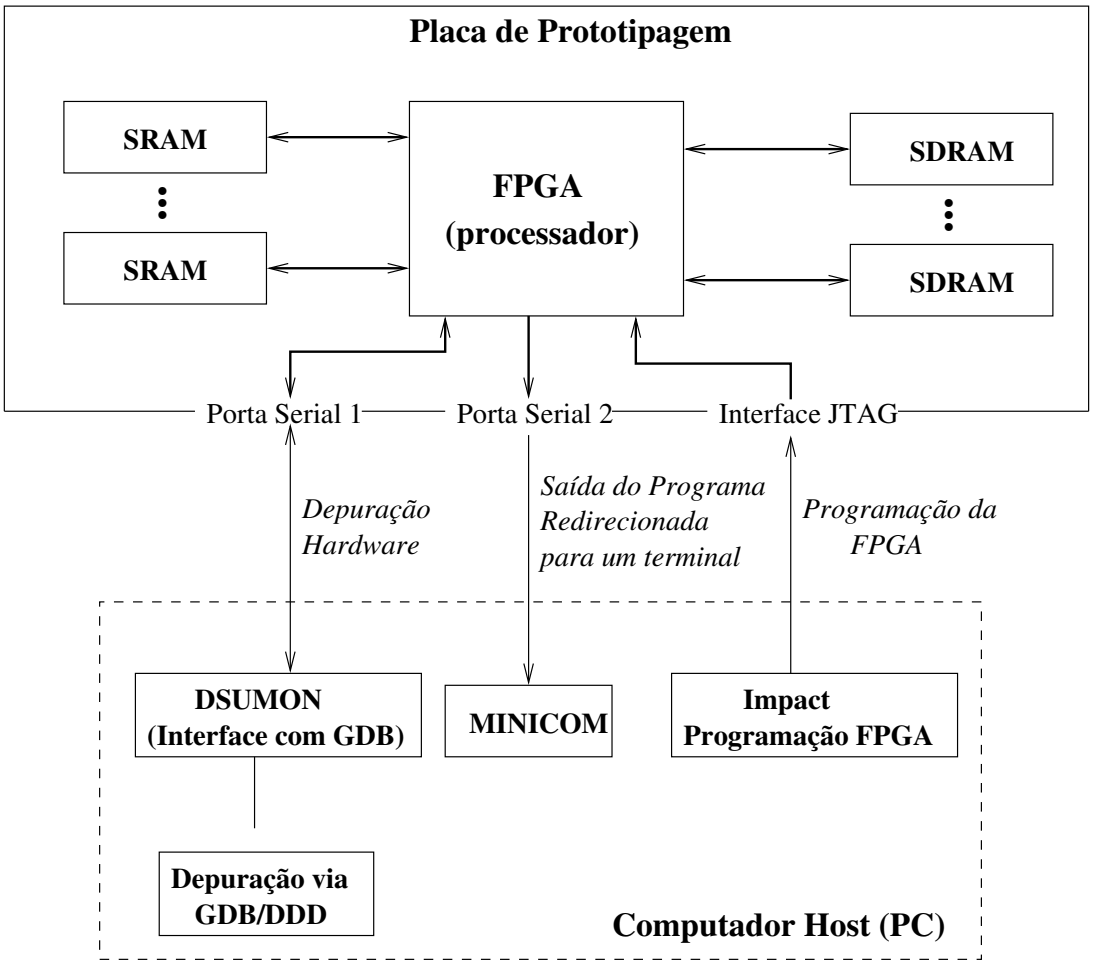


Figura 5.3: Infraestrutura: Ambiente de Prototipagem

Foram usadas duas placas de prototipagem para realização dos experimentos. A primeira foi a *GR-PCI-XC2V* [33], desenvolvida pela *Gaisler Research*. Esta placa foi usada para obtenção de todos os resultados que serão apresentados a seguir, exceto para a implementação da demostração prática a ser descrita no final do capítulo, onde foi usada a *Xilinx Microblaze & Multimedia*, [34], desenvolvida pela *Xilinx*, pois necessitava de interface física de som, não presente na *GR-PCI-XC2V*.

Dentre as características relevantes na placa *GR-PCI-XC2V* estão:

- 1 MB de memória SRAM;

- 64 MB de memória SDRAM (PC133);
- JTAG interface;
- FPGA Virtex-II XC2V3000-FG676-4;
- Interface PCI;
- Interface Ethernet 10/100.

Uma característica interessante mostrada acima é a interface PCI. Com isso, a comunicação entre *dsumon* e módulo de depuração do processador se dá pela interface PCI, em vez de serialmente. Isto é bastante útil para análise de programas grandes, já que a velocidade de comunicação é muito maior.

A placa *Xilinx Microblaze & Multimedia* é semelhante a *GR-PCI-XC2V*, com algumas pequenas diferenças:

- 10 MB de memória SRAM;
- Não há SDRAM (PC133);
- FPGA Virtex-II XC2V2000-FF896-4;
- Sem Interface PCI.

5.1.4 Programas avaliados

Os experimentos foram feitos em cima de programas para avaliação de desempenho, chamados *benchmarks*. Optou-se por utilizar os mesmos programas usados por Wanderley [1], exceto o JPEG (programa para compressão e descompressão de imagens) que foi substituído pelo *minimad/libmad* que é um programa para descompressão e reprodução de arquivos MP3 e será descrito ao final do capítulo, na seção de demonstração. Os programas foram retirados de duas *suites* de *benchmarks* bastante conhecidos: *MediaBench*[35] e *MiBench* [36], descritos a seguir:

MiBench

- **Susan**: É usado para reconhecimento de imagens de ressonância magnética, através do reconhecimento de cantos e bordas.
- **StringSearch**: Faz a busca por palavras em frases.
- **Dijkstra**: Faz a busca por caminhos mínimos entre vértices de um grafo.

MediaBench

- **Adpcm Coder/Decoder**: O *coder* codifica/comprime arquivos do formato *pcm* para o formato *adpcm*. O *decoder* faz o inverso.
- **Pegwit**: Programa para criptografia/decriptografia a partir de chaves pública/privada.

A maioria dos programas, citados acima, fazem acesso a arquivos. É o caso, por exemplo, do *Adpcm Coder*, que recebe como entrada um *pcm* e gera um arquivo *adpcm* de saída. Contudo, o ambiente de prototipagem acima não tem disco rígido e, portanto, as leituras e escritas a descritores de arquivos não têm efeito algum. Para resolver este problema, foi necessário fazer com que estas leituras e escritas fossem feitas na memória (SRAM ou SDRAM), o que foi feito de uma maneira bastante transparente, envolvendo mínimas alterações nos códigos fontes das aplicações. Foi desenvolvida uma biblioteca chamada de *leonlib.h*, que implementa funções como: *fopen*, *fwrite*, *fread*, *fseek*, *fprintf*, etc., através de acessos a memória. Os programas fontes, por sua vez, passam a incluir esta biblioteca e, após isto, uma chamada a uma destas funções não será mais ligada (do inglês: *linked*) com a *libgcc.a*, mas sim, com esta biblioteca adaptada. A Figura 5.4 ilustra o que foi explicado.

A Tabela 5.1 mostra como cada um destes programa foi utilizado para a obtenção dos resultados, indicando também o número de instruções geradas após a compilação.

5.1.5 Metodologia de Validação

Para validar a arquitetura de descompressão é necessário certifica-se de que o processador modificado continua executando os programas da mesma maneira que o processador original. No nível de simulação RTL, isto fica fácil de fazer, já que é possível obter *tracings*

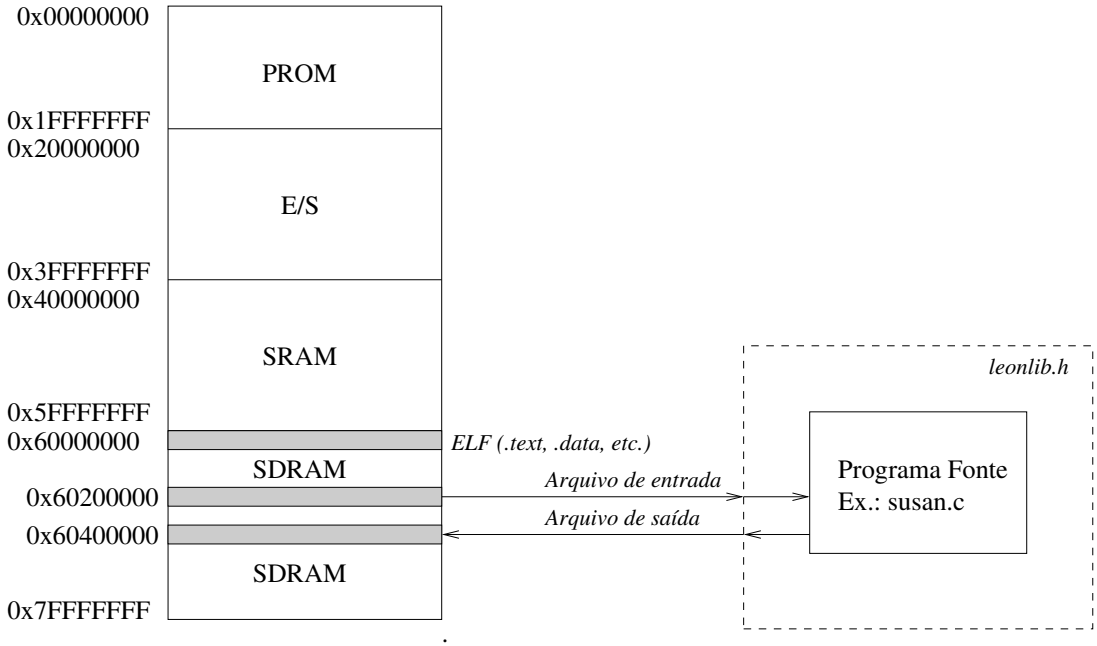


Figura 5.4: Mapa de memória do Leon, incluindo regiões onde são mapeados os arquivos de entrada e saída

de toda execução, bastanto assim, comparar a versão modificada e original. Contudo, neste tipo de simulação, a velocidade de execução das instruções fica na ordem de milhares por minuto. Como a maioria das execuções está na ordem de milhões, algumas chegando até a ordem de bilhões, fica evidentemente impossível validar os programas desta forma.

A solução está em validar as execuções na própria placa de prototipagem, usando a interface de depuração em hardware e o aplicativo *dsumon*, descrito anteriormente. Para realizar a validação, o programa é carregado/executado e, posteriormente, verifica-se se a saída gerada está igual a esperada, distribuída junto ao *benchmark*.

A checagem acima já é um grande indício da correta execução do código comprimido, mas ainda não garante sua total validade. Uma segunda checagem é comparar o número de instruções da execução original e comprimida. Se a execução for válida, os valores estarão muito próximos, não estando iguais pois existem existem *traps* que são mais executadas em um caso do que no outro (*trap* que trata o estouro do registrador de tempo, por exemplo). Para fazer esta segunda validação, foi necessário estender o processador e colocar um contador de instruções.

Com o que se tem de ferramentas, essas são as únicas validações vislumbradas. Apesar de não serem formais, pode-se dizer, sem exageros, que elas já validam o protótipo de

Programa	Utilização	Número de Instruções
Susan	Suavização das bordas de uma imagem preto e branco	62960
StringSearch	Busca de 1332 palavras em 1332 frases	16528
Dijkstra	Encontrar os menores caminhos em 100 grafos de 100 vértices	13264
Adpcm Coder	Codificar o arquivo clinton.pcm	2304
Adpcm Decoder	Decodificar o arquivo clinton.adpcm	2295
Pegwit	Decriptografar um arquivo de texto	70464

Tabela 5.1: Dados sobre a compilação dos *benchmarks*

forma satisfatória.

5.2 Área usada com o descompressor

A Tabela 5.2 mostra a área adicional por conta do descompressor, mostrando também as configurações usadas para cada placa. Observe que o acréscimo é muito pequeno, praticamente desprezível, já que se tem um hardware descompressor bastante simples.

	GR-PCI-XC2V	*Xilinx M. M.*
I-cache	2 sets de 8 Kbytes	1 set de 4 KBytes
D-cache	2 sets de 8 Kbytes	1 set de 4 Kbytes
DSU	SIM	SIM
Ethernet	SIM	SIM
PCI	SIM	NÃO
Trace buffer	256 linhas	256 linhas
Mul/Div em hardware	NÃO	NÃO
Unidade de Ponto Flutuante	NÃO	NÃO
MMU	NÃO	NÃO
Número de Portas Equiv. (Original)	**1.923.763**	**998.051**
Número de Portas Equiv. (com Descompressor)	**1.930.236**	**1.004.710**
Aumento da Área (%)	**0,01**%	**0,01**%

Tabela 5.2: Utilização da área da FPGA: processador original *versus* modificado

5.3 Experimentos com Benchmarks: MiBench e MediaBench

Nesta seção serão apresentados os resultados envolvendo os programas dos *benchmarks* *MiBench* e *MediaBench*. Antes de mostrar os resultados, cabe apresentar os parâmetros que foram fixados para a realização dos experimentos.

O primeiro parâmetro é o tamanho da I-cache. Segundo Wanderley [1], que fez um estudo na variação na taxa de acertos em função do tamanho na *cache*, os tamanhos de *cache* que oferecem uma boa relação custo/benefício (custo = tamanho da cache, benefício = taxa de acertos) são mostrados na Tabela 5.3.

Programa	Tamanho da I-cache
susan	256 bytes
search	2 Kbytes
dijkstra	128 bytes
adpcm_e	128 bytes
adpcm_d	128 bytes
pegwit	1 Kbyte

Tabela 5.3: Tamanhos de I-cache com melhor custo-benefício (Tamanho/Taxa de Acertos)

Tento em vista que o Leon2 aceita um tamanho mínimo de *cache* de 1 Kbyte e que o objetivo desta seção não é avaliar o impacto da variação da *cache*, resolveu-se fixar o tamanho da *cache* em 1 Kbyte para todos os programas. O estudo sobre a variação da *cache* será feito posteriormente neste capítulo.

Um segundo parâmetro é o tamanho do dicionário de instruções. Resolveu-se neste caso usar exatamente aquele tamanho tido como melhor (em termos de custo/benefício) no trabalho de Wanderley [1], que é 256 instruções, tanto para os experimentos desta seção quanto para os demais deste capítulo.

Um terceiro parâmetro, bastante importante, é a taxa f. Como já explicado, sabe-se que valores baixos de f (estatísticas estáticas tem um peso maior que dinâmicas) geralmente levam a códigos com razões de compressão melhores e tempos de execução piores, enquanto que valores altos de levam a razões de compressão piores e tempos de execução melhores. A Figura 5.5 mostra as razões de compressão obtidas com a variação de f para os programas experimentados. Observe que as razões de compressão pioram bastante para valores de f maiores que 90%. Como o objetivo desta seção é ter uma idéia

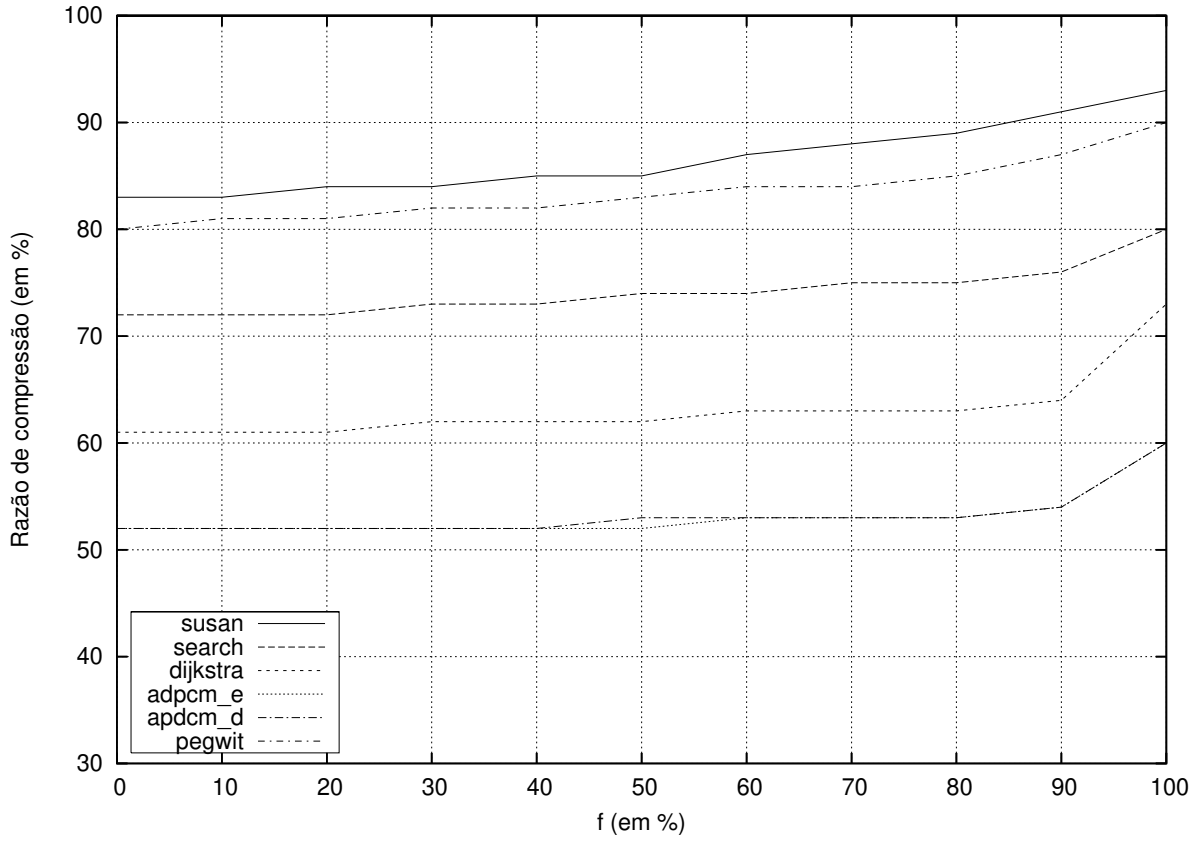


Figura 5.5: Razões de compressão do PDC-ComPacket em função de f

do ganho de desempenho na execução de códigos comprimidos, mas por outro lado tendo uma razão de compressão razoável, escolheu-se f igual a 90%.

A Tabela 5.4 mostra os resultados dos experimentos em cima dos dois *benchmarchs*. Um ponto interessante é que o ganho na execução de código comprimido está diretamente relacionado à taxa de acertos a I-cache. Isto pode ser constatado no programa *search*, com maior ganho (38,46%), onde também houve o maior aumento na taxa de acertos, de 91,06% a 98,93%. No outro extremo, o *adpcm_e* e *adpcm_d* têm taxas de acertos ótimas mesmo sem compressão. Em outras palavras, todas as instruções necessárias para execução do programa são buscadas na memória uma única vez e, assim, a compressão não traz vantagens. Pelo contrário, acontece uma pequena queda de desempenho devido ao *burst* de instruções da I-cache, pois o envio de instruções ao processador é interrompido ao encontrar um *ComPacket*, o que não acontece com o código original. Mas cabe ressaltar que, mesmo com essa pequena queda de desempenho nestes dois casos, continua sendo válido o uso de compressão, já que o código é bastante comprimido (64% para o adpcm_e

Programa	Compressão (%)	Ciclos: orig ↦ comp (**ganho em %**)	Taxa de Acertos na I-cache(%): orig ↦ comp
susan	88	609.506.076 ↦ 576.693.724 (5,69)	96,80 ↦ 99,45
search	76	**10.460.236 ↦ 7.554.628 (38,46)**	91,06 ↦ 98,93
dijkstra	72	670.595.866 ↦ 624.739.059 (7,34)	96,70 ↦ 98,60
adpcm_e	64	17.404.118 ↦ 17.415.625 (-0,07)	99,99 ↦ 99,99
adpcm_d	65	15.551.559 ↦ 15.560.237 (-0,06)	99,99 ↦ 99,99
pegwit	89	81.952.630 ↦ 72.311.316 (13,33)	78,04 ↦ 83,72

Tabela 5.4: Resultados de compressão e desempenho (*MiBench* e *MediaBench*)

e 65% para o adpcm_d), diminuinto os requisitos de memória.

Além dos resultados mostrados, todos os programas foram devidamente validados segundo a metodologia apresentada em seção anterior.

5.4 Análise da tríade: compressão, desempenho e energia

Agora, que um número razoável de programas já foi experimentado, com bons resultados de compressão e desempenho, pode-se partir para uma análise mais aprofundada, incluindo também aspectos como: variação da taxa f e análise do consumo de energia.

Para isto, foi escolhida uma versão compactada do programa *search*, chamada *search_small*. Foi necessário a escolha de um programa que executasse em poucos ciclos, já que as estimativas de potência são tiradas a partir de simulações *gate-level*, ainda mais lentas que as simulações *RTL*.

A Tabela 5.5 e a Figura 5.6 mostram os resultados de compressão, desempenho e consumo de energia, sendo que a tabela traz alguns detalhes a mais como: potência e taxa de acertos na I-cache. A potência do processador (em mW), foi obtida através do *XPower*. Note que ela começa com 809mW (código original) e cai sutilmente a medida que aumenta a taxa *f*. A queda não é maior pois, mesmo com a diminuição nas atividades

nos acessos à memória externa ao longo do tempo, devido ao aumento na taxa de acertos na I-cache, existe a energia consumida pelo hardware descompressor, que está ativo todos os ciclos de relógio. Porém, potência mede o consumo de energia ao longo do tempo. Considerando-se que existe uma queda considerável no tempo de execução (entre 28,29% e 44,35%), isto faz com que o consumo de energia também diminua, já que:

$$\textbf{Energia = Potência x Tempo} \tag{5.1}$$

Na tabela, além da energia para executar o programa, dada em mJ (Joule x 10^{-3}), mostra-se a redução da energia consumida, variando entre 22,53% e 32,27%.

Um ponto importante nos dados mostrados é que, nem sempre, o melhor tempo de execução e consumo de potência estará com f igual a 100%. Neste caso, os melhores valores são encontrados quando f é igual a 70%. Isto porque existem outras variáveis em um processador real, como por exemplo, ocupação de barramento, *stalls* no *pipeline*, que podem levar as estas pequenas diferenças. Na prática, caberia ao projetista obter e analisar os dados, determinando qual o melhor valor de f em termos de compressão, desempenho e consumo de energia.

f (%)	Taxa de compressão(%)	Ciclos & Ganho(%)	Taxa de Acertos na I-cache(%)	Potência(mW)	Energia(mJ) & Redução(%)
Orig.	-	113.505 (-)	88,55	809	2,30 (-)
0	72	87.563 (29,63)	95,54	803	1,76 (23,43)
10	72	88.475 (28,29)	95,36	804	1,78 (22,53)
20	72	88.178 (28,72)	95,47	802	1,77 (22,99)
30	73	87.506 (29,71)	95,68	802	1,75 (23,57)
40	73	87.067 (30,37)	95,79	802	1,75 (23,96)
50	74	86.944 (30,55)	95,80	800	1,74 (24,25)
60	74	86.215 (31,65)	95,97	801	1,73 (24,79)
70	**75**	**78.631 (44,35)**	**98,03**	**791**	**1,55 (32,27)**
80	75	80.888 (40,32)	97,43	792	1,60 (30,23)
90	76	82.365 (37,81)	97,06	791	1,63 (29,05)
100	80	78.684 (44,25)	98,06	791	1,56 (32,22)

Tabela 5.5: Resultados de compressão/desempenho/consumo energia do *search_small*

Um exemplo prático de como tirar proveito da redução de depempenho imposta pela compressão está em diminuir a freqüência de operação do circuito e, ainda assim, possibilitar que o programa continue executando no mesmo tempo que o programa

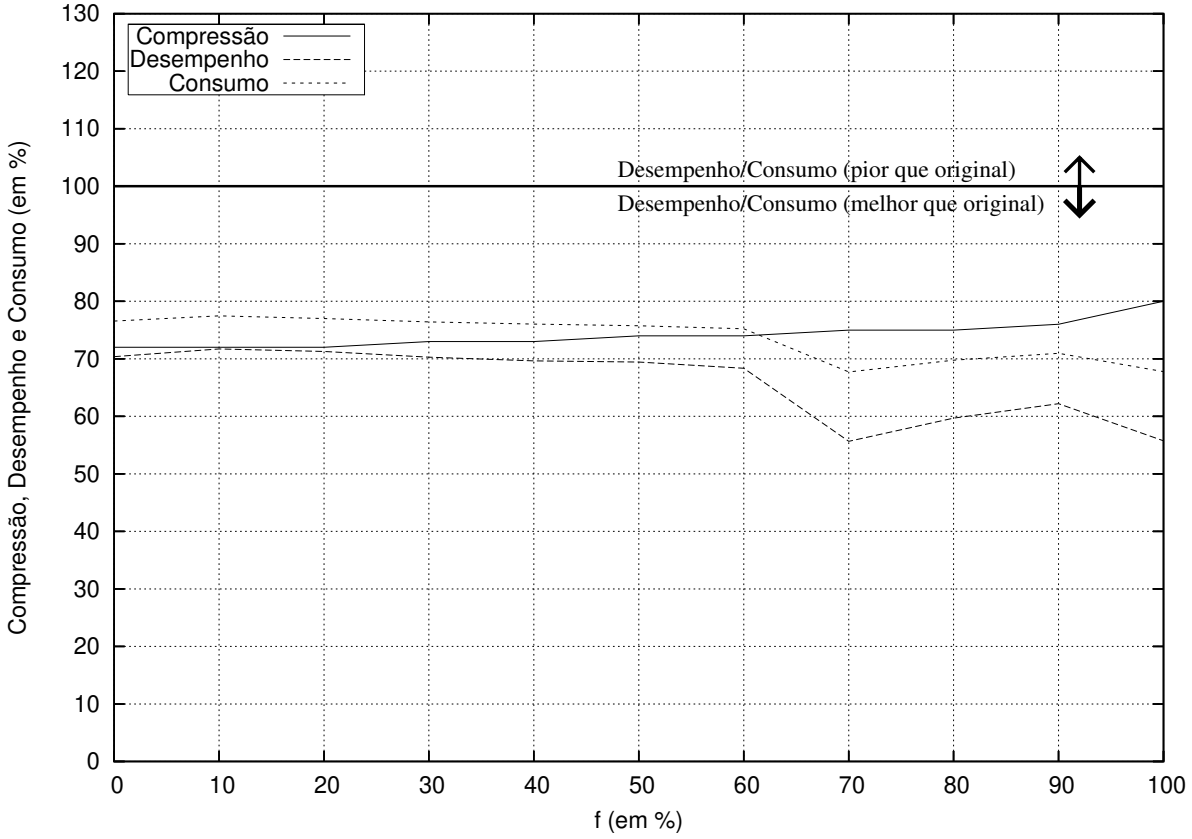


Figura 5.6: Comportamento da tríade: compressão, desempenho e energia para um caso selecionado: *search_small*

original, já que existe uma considerável redução no número de ciclos com compressão. Foi feito um experimento neste sentido, sendo que o código comprimido ($f = 70\%$) foi executado em um processador com freqüência de operação igual a 30Mhz (a execução dos programas acima havia sido feita em 40Mhz). Com isso, o programa foi executado em 2,62ms (78.631 ciclos / 30Mhz), ou seja, 8,01% mais rápido do que o original 2,83ms (113.505 ciclos / 40Mhz). Outro ponto interessante é que, mesmo diminuindo a freqüência e, com isso, o tempo de execução, o consumo de energia ainda é reduzido, passando para 1,93mJ (738mW x 2,62ms), o que equivale a uma redução de 15,91% sobre o consumo de energia original, que era 2,30mJ.

Na prática, este exemplo é útil naqueles casos onde o *cycle-time* do circuito não é suficiente para a demanda de desempenho requerida pela aplicação. Neste caso, a compressão surge para auxiliar a atingir este requisito.

5.5 Estudo de casos - Libmad

Nesta seção serão apresentados alguns experimentos em cima do *libmad* [37]. *Libmad* é uma biblioteca de decodificação de arquivos MP3, especialmente interessante por ser bastante rápida e projetada para operar em sistemas sem unidade de ponto flutuante.

5.5.1 Variação do Tamanho da I-cache e D-cache

Um primeiro experimento foi variar os tamanhos da I-cache e D-cache para verificar o desempenho do sistema para o código original do *libmad versus* código comprimido com f igual a 90%, o que é mostrado na Tabela 5.6. Note que, numa mesma coluna da tabela, ou seja, mesmo tamanho de I-cache e variado-se a D-cache, existe uma redução do tempo de execução tanto para o caso original, quanto comprimido. Contudo, o desempenho relativo entre eles é muito pouco afetado. Por exemplo, para I-cache e D-Cache de 1 Kbyte, tem-se um tempo de execução em ciclos de 805.852.330 e 689.653.240 para o caso original e comprimido, respectivamente, o que faz um ganho de 16,84%. Para o mesmo tamanho de I-cache, mas D-cache de 8 Kbytes, o número de ciclos cai para 766.120.912 e 649.997.074, porém a ganho relativo sofre apenas um pequeno aumento, indo para 17,86%. Este fato se repete para os demais tamanhos de I-cache, mostrando que a D-cache tem pouca influência sobre o ganho de desempenho relativo trazido com a compressão.

Outro ponto interessante é que os valores de desempenho mudam muito pouco para I-cache de 4 Kbytes para 8 Kbytes, o que pode ser explicado pela Tabela 5.7, que mostra as taxas de acertos de acordo com a variação no tamanho na I-cache. Observe que a variação na taxa de acertos para esses dois tamanhos de I-cache é muito pequena. Desta maneira, para os demais experimentos desta seção, optou-se pela escolha da I-cache e D-cache de 4Kbytes, as quais parecem ter o melhor custo-benefício. O tempo de execução do código comprimido para uma I-cache e D-cache de 4Kbytes (541.405.070) estão abaixo do tempo de execução do código original para I-cache e D-cache de 8 Kbytes (656.638.720), o que mostra a eficácia da técnica de compressão. Isto que dizer que, com menos hardware, consegue-se um melhor desempenho do sistema.

5.5.2 Variação da razão dinâmico/estático (f)

Neste segundo experimento, serão mostrados resultados referentes à variação da razão f, da mesma maneira como foi feito para o *search_small*, só que desta vez para o *libmad*.

	I-cache			
	1 Kbytes	2 Kbytes	4 Kbytes	8 Kbytes
D-cache	orig (Kciclos) $\mapsto$ comp (Kciclos) (ganho em %)			
1 Kbytes	805.852 $\mapsto$ 689.653 (**16,84**)	749.256 $\mapsto$ 668.926 (**12,00**)	712.097 $\mapsto$ 577.991 (**23,20**)	699.711 $\mapsto$ 564.738 (**23,90**)
2 Kbytes	786.914 $\mapsto$ 670.774 (**17,31**)	729.956 $\mapsto$ 649.622 (**12,36**)	692.954 $\mapsto$ 558.572 (**24,01**)	677.898 $\mapsto$ 547.442 (**23,83**)
4 kbytes	769.726 $\mapsto$ 653.601 (**17,76**)	712.782 $\mapsto$ 632.447 (**12,70**)	675.523 $\mapsto$ 541.405 (**24,77**)	660.256 $\mapsto$ 526.982 (**25,29**)
8 kbytes	766.120 $\mapsto$ 649.997 (**17,86**)	709.160 $\mapsto$ 628.843 (**12,77**)	671.900 $\mapsto$ 537.802 (**24,93**)	656.638 $\mapsto$ 522.801 (**25,60**)

Tabela 5.6: Resultados de desempenho com variação do tamanho da I-cache e D-cache (*libmad*) (Ciclos divididos por 1000)

I-cache	Taxa de Acertos na I-cache (Orig.)	Taxa de Acertos na I-Cache (Comp)
1 Kbytes	71,40%	82,58%
2 Kbytes	76,75%	84,56%
4 Kbytes	82,33%	95,30%
8 Kbytes	89,26%	96,05%

Tabela 5.7: Taxas de acertos na I-cache variando seu tamanho (*libmad*)

A Tabela 5.8 mostra os resultados de compressão e desempenho encontrados na forma tabular, enquanto a Figura 5.7 sintetiza essas informações por meio de um gráfico. Dois pontos são interessantes de serem observados. O primeiro é constatar, novamente, que existe uma piora abrupta da taxa de compressão em f=90%, o que sugere que se escolha um valor abaixo deste. O segundo ponto está no desempenho, em comparação com o *search_small* (Tabela 5.5 e Figura 5.6), neste caso o aumento do desempenho com a variação da taxa f se dá de maneira bem mais suave. Com f entre 0% e 70% o valores variam muito pouco, entre 21,04% e 22,31%. Isto pode ser explicado pelo grande tamanho do código (79.312 instruções), o que traz um grande número de instruções repetidas, fazendo com que, mesmo em se optando por instruções estáticas (f baixos), exista uma grande chance destas instruções serem comprimidas e freqüentemente executadas. Por outro lado, para taxas acima de 70%, o desempenho melhora mais rapidamente, o que sugere que se escolha um f acima deste valor.

Juntando com o primeiro comentário, pode-se dizer que, para esta aplicação, o melhor custo-benefício em termos de compressão e desempenho estaria para um f entre 70% e

90%.

f (%)	Taxa de Compressão	Ciclos & Ganho (%)	Taxa de Acertos na I-cache (%)
Orig.	-	675.523 (-)	80,33
0	84	558.078 (21,04)	91,67
10	84	555.462 (21,61)	91,94
20	84	556.333 (21,42)	91,85
30	85	552.312 (22,31)	92,24
40	85	557.441 (21,18)	91,75
50	85	557.567 (21,16)	91,73
60	86	556.764 (21,33)	91,80
70	87	553.178 (22,12)	92,16
80	87	544.199 (24,13)	93,03
90	89	541.405 (24,77)	93,30
100	94	535.483 (26,15)	93,90

Tabela 5.8: Resultados de compressão e desempenho do *libmad*. (Ciclos divididos por 1000)

5.5.3 Demonstração Prática - *libmad*

Tendo-se estudado o comportamento do *libmad*, pode-se agora, finalmente, descrever a demonstração prática do uso de compressão.

A aplicação consiste em decodificar e tocar arquivos de MP3 na placa *Xilinx Microblaze & Multimedia*, descrita no início do capítulo. Para isto, foi necessário a implementação de uma infra-estrutura que permitisse a realização do trabalho:

- Porte do Leon2 para a placa *Xilinx Microblaze & Multimedia*
- Infraestrutura para tocar MP3 incluindo: módulo de som AC97 conectado no barramento APB AMBA do Leon2, *drivers* para o módulo, adaptação do aplicativo *minimad* para o Leon2.

Ambos os trabalhos foram publicados e divulgados à comunidade do Leon2, sendo que o primeiro trabalho (Porte do Leon2), atualmente, faz parte da distribuição oficial do processador. A documentação associada encontra-se *online* em [38] e [39], recebendo dezenas de visitas diariamente.

Devidamente descrita a infraestrutura, pode-se voltar à descrição da demonstração. O *libmad* foi comprimido com f=90%. Depois disso, reproduziu-se um arquivo de

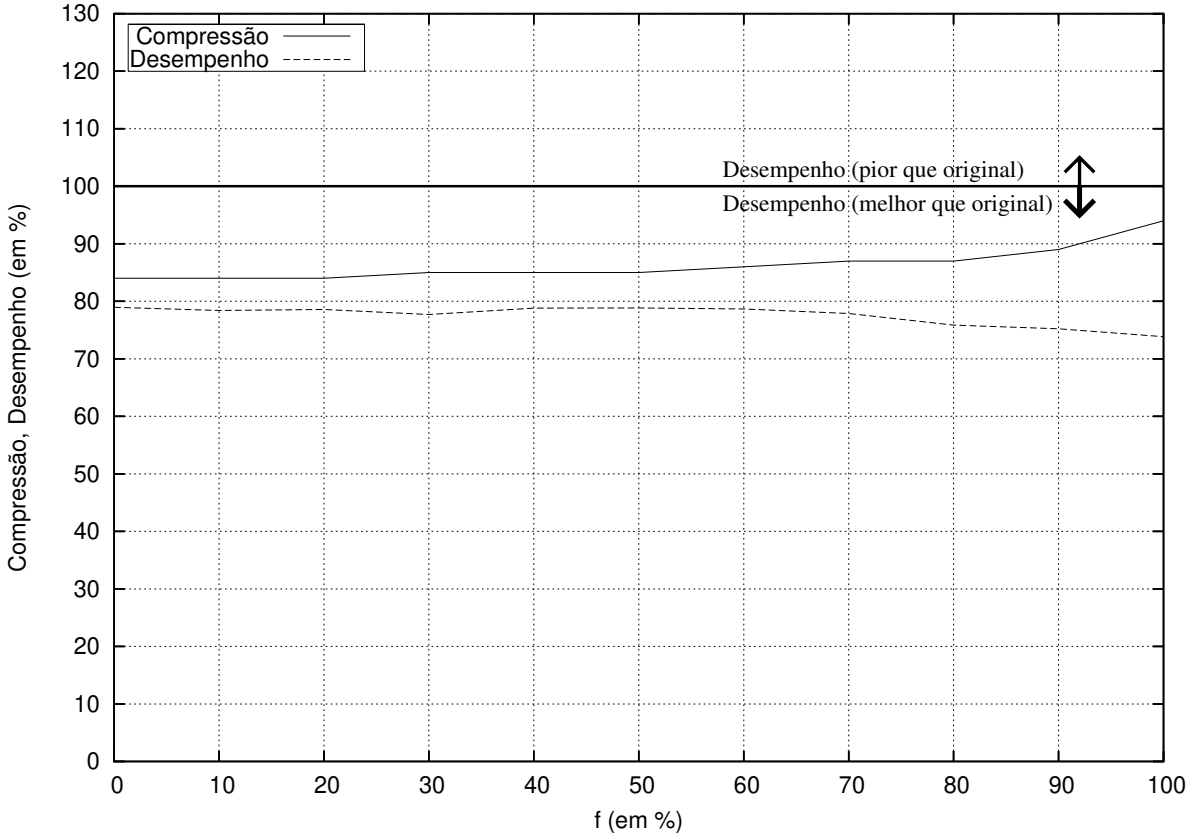


Figura 5.7: Comportamento da compressão e desempenho para o *minimad*

128 kbps (44Khz/16 bits) no processador original e com descompressor, ambos com freqüência de operação = 54Mhz. No processador original, a velocidade de execução da aplicação permitiu a decodificação/reprodução de 40.090 amostras por segundo (abaixo das 44K amostras por segundo necessárias!), enquanto que, no caso comprimido, a decodificação/reprodução chegou a 44.500 amostras por segundo. Ou seja, o processador com compressor permitiu a correta reprodução da música, enquando o original perdeu amostras, levando a uma reprodução não contínua.

Um detalhe relevante é que o *libmad* foi executado a partir de memórias estáticas (SRAM), tidas como memórias bastante rápidas. Isto mostra que ganhos de desempenho são conseguidos tanto para memórias rápidas (SRAM, por exemplo), quanto lentas (SDRAM, por exemplo).

5.6 Considerações Finais e Conclusões

Os resultados encontrados com os experimentos foram bastante satisfatórios, de acordo com as expectativas iniciais. Chegou-se a ganhos de desempenho maiores que 40% e redução de energia maior do que 30% o que, num processador real, é bastante. Porém, não se pode chegar a um número médio de aumento de desempenho ou redução de energia, pois, como foi mostrado, estes números variam muito de uma aplicação para outra e qualquer número médio que fosse atribuído não significaria nada.

Uma funcionalidade importante, não presente neste trabalho, e que será citado como trabalho futuro, é permitir que durante o *bursting* de instruções, o processador continue caminhando, mesmo que haja uma instrução comprimida. Esta funcionalidade envolverá uma lógica de controle bastante complexa, mas, certamente, trará ganhos ainda maiores que os apresentados. Além disso, é provável que quedas de desempenho em casos muito particulares como o *adpcm* não fossem mais vistas após a implantação desta nova funcionalidade.

Outro ponto importante, que deve ser ressaltado, é o consumo de energia, que foi satisfatório tanto por manter a potência do circuito mesmo com o descompressor, quanto pela energia final reduzida, por conta, principalmente, do menor tempo para execução da aplicação.

Os experimentos mostraram também que a obtenção de bons resultados está fortemente atrelada a escolha correta de parâmetros de entrada: tamanhos de *cache*, *f*, etc. Um bom projetista deve ser capaz de analisar os dados e estipular os melhores parâmetros para o seu projeto, levando em conta os requisitos de: tamanho de código, desempenho e consumo de energia.

Uma análise comparativa entre os resultados deste trabalho e os demais encontrados na literatura é bastante difícil, no sentido de que os ambientes de experimentação variam muito de trabalho para trabalho. Se for comparado com o trabalho de Wanderley [1], em que a mesma técnica foi usada, porém os dados foram tirados a partir de estimativas em um simulador, já há várias discrepâncias. No seu trabalho são obtidos ganhos de desempenho de mais de 60% e reduções no consumo de energia de mais de 50%. Observe, entretanto, que as condições são muito diferentes:

- As estimativas de consumo de energia são feitas apenas em cima da I-cache, neste trabalho são feitas em cima do processador inteiro;

- São usadas I-caches ideais (128 bytes, por exemplo), impossíveis de serem usadas neste ambiente de prototipagem;
- É um ambiente para simulação funcional. Esta é certamente a maior diferença, pois existem várias questões existentes num ambiente real (*stalls* no *pipeline*, acesso a barramentos internos, etc.), não levados em conta no ambiente simulado.

A Tabela 5.9 e Tabela 5.10 fazem a comparação em termos de compressão, potência e desempenho com os trabalhos considerados mais importantes da literatura. Note que, se por um lado, o método ComPacket-PDC apresenta razões de compressão piores que os demais, por outro, ele é o método com melhores ganhos de desempenho. Em termos de consumo de energia, ele está próximo dos demais trabalhos (Benini e Lekatsas). Mas ainda assim, não se sabe em que condições estes valores foram obtidos: energia apenas na I-Cache? processador inteiro? D-Cache? Com simulação ou prototipagem?

Em suma, mesmo tendo informações bastante vagas na literatura, pode-se dizer, a partir dos dados encontrados, que os resultados deste trabalho atenderam as expectativas. Além disso, um item que não foi analisado acima (por falta de informação nos trabalhos relacionados), é a área utilizada na FPGA para implementação do *descompressor*. Entretanto, é certo que este trabalho estaria bastante competitivo neste ponto, tendo em vista a simplicidade do hardware projetado.

Métodos CDM					
Autor	Arquitetura	*Benchmarks*	Razão de Compressão	Tempo de Execução	Energia
Wolfe & Chanin[13]	MIPS	lex, pswarp, yacc, eightq, matrix25 lloop01, xlisp, espresso e spim	73%	–	–
IBM[17][18]	PowerPC	Mediabench SPEC95	60%		–
Azevedo[19]	SPARC	SPECint95	53,6%	+5,89%	–

Tabela 5.9: Comparação dos resultados com métodos CDM

Métodos PDC					
Autor	Arquitetura	*Benchmarks*	Razão de Compressão	Tempo de Execução	Energia
Lekatsas[23]	SPARC	compress, diesel, i3d, key, mpeg smo, trick	65%	-25%	–
Lekatsas[24]	Xtensa 1040	compress, diesel, i3d, key, mpeg smo, trick	65%	-25%	-28
Benini[26]	MIPS	Ptolomy	72%	–	-30%
Lefurgy[27]	PowerPC	SPECint95	61%	–	–
Lefurgy[27]	ARM	SPECint95	66%	–	–
Lefurgy[27]	i386	SPECint95	75%	–	–
Billo/Wanderley	SPARC	MediaBench MiBench	72% $\sim$ 88%	0% $\sim$ -45%	0% $\sim$ -35%

Tabela 5.10: Comparação dos resultados com métodos PDC

Capítulo 6

Conclusões e Trabalhos Futuros

Este trabalho atingiu os objetivos traçados inicialmente, chegando ao projeto, implementação e validação de uma Arquitetura de Compressão. O projeto e implementação, bastante dificultada pela complexidade associada ao LEON2, além de cobrir todos os requisitos iniciais, estendeu-se de modo a permitir que a funcionalidade de *bursting* da *cache* de instruções do Leon2 continuasse ativa, mesmo com a integração do descompressor. Esta foi uma das tarefas mais difíceis de serem concebidas, mas inevitável, já que uma implementação sem *bursting* e com descompressor teria desempenho equivalente ou pior que o processador original, o que levaria a resultados bem menos expressivos. Outro ponto bastante difícil na implementação, foi ter conseguido eliminar o ciclo de penalidade, inicialmente previsto por Wanderley[1], nos saltos. Estima-se que, com esta funcionalidade, o desempenho aumentou em cerca de 5%.

A arquitetura foi suficientemente validada através de programas pertencentes a duas *suites* de *benchmarks*: *MediaBench* e *MiBench*, além de uma demonstração prática envolvendo a decodificação e reprodução de arquivos MP3 (*libmad*). Os resultados foram bastante satisfatórios, com ganhos de desempenho chegando até 45% e redução de energia de até 35% e, naturalmente, permitindo que o código fosse comprimido, com razões de compressão variando entre 72% e 88%. Os dados de desempenho e consumo de energia estão próximos ou superam os resultados encontrados na literatura, o que compensa não se chegar a razões de compressões ótimas, já que esse foi um *trade-off* levado em conta durante o projeto do PDC-ComPacket.

O trabalho contou, também, com duas importantes contribuições para a comunidade do Leon: o porte do processador para uma *kit* de desenvolvimento da Xilinx, que foi posteriomente incorporado a sua distribuição oficial e um tutorial incluindo todos os

fontes implementados, mostrando como implementar MP3 neste processador.

Juntando com a tese de Wanderley [1], acredita-se que este trabalho é uma importante contribuição na área, por ter chegado a uma implementação funcional em um processador muito utilizado, com resultados expressivos.

6.1 Trabalhos Futuros

Embora todos objetivos iniciais tenham sido atingidos, alguns melhoramentos podem ser feitos na implementação atual. Alguns itens dizem respeito ao trabalho de Wanderley, mas também serão citados, pois existe uma interdependência entre a técnica para compressão do código e a Arquitetura de Descompressão:

- Extender o compressor de maneira que ele resolva todos endereços de salto, inclusive os saltos indiretos. Talvez a única maneira para se conseguir esse item, seja integrar o compressor ao *back-end* do compilador, antes da geração do código objeto. Essa funcionalidade é crucial para um projetista que fosse usar o PDC-ComPacket. Ele deve ser capaz de comprimir o código, por exemplo, com a simples adição de uma opção à linha de compilação.

- Permitir que, durante o *bursting* de instruções, o processador continue caminhando, mesmo que haja uma instrução comprimida. Esta funcionalidade envolverá uma lógica de controle bastante complexa, mas certamente trará ganhos ainda maiores que os apresentados. Além disso, é provável que as quedas de desempenho em casos muito particulares, como aconteceu no *adpcm*, não fossem mais vistas após a implantação desta nova funcionalidade.

- Estudar a possibilidade de deixar o descompressor mais isolado do Leon2, já que atualmente ele está muito integrado ao processador, gerando uma dependência muito grande, não desejável.

- Seguindo a idéia do item anterior, possibilitar que a arquitetura seja generalizada e viabilizada para outros processadores. Talvez para o Leon2, seja difícil isolar o descompressor, mas para outros processadores isto seja possível.

- Chegar a uma implementação em ASIC da Arquitetura de Descompressão, de modo que o trabalho transcendesse o âmbito acadêmico, servindo como uma contribuição, também, para o meio industrial.

Referências Bibliográficas

[1] E. Wanderley Netto. *Compressão de Código Baseada em Multi-Profile*. PhD thesis, Instituto de Computação, Universidade Estadual de Campinas, May 2004.

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

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

[4] P. Weiner. Linear pattern matching algorithms. In *Proc. Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[5] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[6] E. Ukkonen. Constructing suffix-trees on-line in linear time. *Information Processing*, I(92):484–492, 1992.

[7] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

[8] J. Ernst, C. Fraser, W. Evans, S. Lucco, and T. Proebsting. Code compression. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 358–365, June 1997.

[9] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Addison Wesley, 1988.

[10] S. 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*, pages 272–285, March 1995.

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

[12] K. Kissell. MIPS16: High-density MIPS for the embedded market. In *Proc. of Real Time Systems*, pages 559–571, 1997.

[13] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *Proc. Int'l Symp. on Microarchitecture*, pages 81–91, November 1992.

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

[15] M. Beneš, A. Wolfe, and S. Nowick. A high-speed asynchronous decompression circuit for embedded processors. In *Proc. Conf. on Advanced Research in VLSI*, pages 219–236, September 1997.

[16] M. Beneš, S. Nowick, and A. Wolfe. A fast asynchronous Huffman decoder for compressed-code embedded processors. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 43–56, September 1998.

[17] T. Kemp, R. Montoye, J. Harper, J. Palmer, and D. Auerbach. A decompression core for PowerPC. *IBM Journal of Research and Development*, 42(6):807–812, September 1998.

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

[19] R. Azevedo. *Uma arquitetura para Código Comprimido em Sistemas Dedicados*. PhD thesis, Instituto de Computação, Universidade Estadual de Campinas, June 2002.

[20] J. Gaisler. Leon2 processor user's manual 1.0.24. [OnLine], September 2004. Available: `http://www.gaisler.com`.

[21] H. Lekatsas, Joerg Henkel, and Wayne Wolf. Code compression for low power embedded system design. In *Proc. ACM/IEEE Design Automation Conference*, pages 294–299, 2000.

[22] T. Bell, J. Cleary, and I. Witten. *Text Compression*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[23] 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*, pages 63–68, October 2001.

[24] H. Lekatsas, J. Henkel, and W. Wolf. Design of an one-cycle decompression hardware for performance increase in embedded systems. In *Proc. ACM/IEEE Design Automation Conference*, pages 34–39, June 2002.

[25] E. Killian and F. Warthman. *Xtensa Instruction Set Architecture Reference Manual*. Tensilica, 2001.

[26] L. Benini, A. Macii, and A. Nannarelli. Code compression for cache energy minimization in embedded systems. *IEE Proceedings on Computers and Digital Techniques*, 149(4):157–163, July 2002.

[27] C. Lefurgy, P. Bird, I-C. Chen, and T. Mudge. Improving code density using compression techniques. In *Proc. Int'l Symp. on Microarchitecture*, pages 194–203, December 1997.

[28] E. Wanderley Netto, R. Azevedo, P. Centoducatte, and G. Araujo. Mixed static/dynamic profiling for dictionary based code compression. In *Proc. of the International Symposium on System-on-Chip*, pages 159–163, November 2003.

[29] E. Wanderley Netto, R. Azevedo, P. Centoducatte, and G. Araujo. Code compression to reduce cache accesses. Technical Report IC-03-23, Instituto de Computação - UNICAMP, November 2003.

[30] E. Wanderley Netto, R. Azevedo, P. Centoducatte, and G. Araujo. Multi-profile based code compression. In *Proc. ACM/IEEE Design Automation Conference*, June 2004.

[31] SPARC International Inc. Sparc v8 architecture manual. [OnLine], December 1991. Available: `http://www.sparc.org`.

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

[33] Gaisler Research. *LEON-PCI-XC2V Development Board User Manual*, 6 2004.

[34] Xilinx. *Microblaze and Multimedia Development Board User Guide*, 8 2002.

[35] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia communication systems. In *Proc. Int'l Symp. on Microarchitecture*, pages 330–337, December 1997.

[36] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, and T. Mudge. Mibench: a free, commercially representative embedded benchmark suite. In *Proc. of the IEEE 4th Annual Workshop on Workload Characterization*, pages 3–14, December 2001.

[37] Vários autores (SourceForge). Mad. [OnLine], 2005. Available: `http://sourceforge.net/projects/mad/`.

[38] Eduardo Billo. Leon2 on the xilinx microblaze & multimedia development board. [OnLine], November 2004. Available: `http://www.lsc.ic.unicamp.br/~billo/leon2\_on\_mblazeboard`.

[39] Eduardo Billo. Playing mp3 on leon2/uclinux. [OnLine], November 2004. Available: `http://www.lsc.ic.unicamp.br/~billo/mp3onleon.htm`.