UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

ARQUITETURA PDCCM EM HARDWARE PARA COMPRESSÃO/DESCOMPRESSÃO DE INSTRUÇÕES EM SISTEMAS EMBARCADOS

WANDERSON ROGER AZEVEDO DIAS

MANAUS
2009

UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

WANDERSON ROGER AZEVEDO DIAS

ARQUITETURA PDCCM EM HARDWARE PARA COMPRESSÃO/DESCOMPRESSÃO DE INSTRUÇÕES EM SISTEMAS EMBARCADOS

Dissertação apresentada ao Programa de Pós-Graduação em Informática do Departamento de Ciência da Computação da Universidade Federal do Amazonas, como requisito parcial para a obtenção do título de Mestre em Informática, área de concentração: Engenharia da Computação.

Orientador: Prof. Dr. Edward David Moreno Ordoñez

MANAUS

2009

WANDERSON ROGER AZEVEDO DIAS

ARQUITETURA PDCCM EM HARDWARE PARA COMPRESSÃO/DESCOMPRESSÃO DE INSTRUÇÕES EM SISTEMAS EMBARCADOS

Dissertação apresentada ao Programa de Pós-Graduação em Informática do Departamento de Ciência da Computação da Universidade Federal do Amazonas, como requisito parcial para a obtenção do título de Mestre em Informática, área de concentração: Engenharia da Computação.

BANCA EXAMINADORA

Prof. Dr. Edward David Moreno Ordoñez, Presidente

Universidade Federal de Sergipe – UFS

Prof. Dr. Carlos Humberto Llanos Quintero, Membro

Universidade de Brasília – UnB

Prof. Dr. Cícero Ferreira Fernandes Costa Filho, Membro

Universidade Federal do Amazonas – UFAM - FT

ARQUITETURA PDCCM EM HARDWARE PARA COMPRESSÃO/DESCOMPRESSÃO DE INSTRUÇÕES EM SISTEMAS EMBARCADOS

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Wanderson Roger Azevedo Dias e aprovada pela Banca Examinadora.

Manaus – AM, 30 de abril de 2009.

Prof. Dr. Edward David Moreno Ordoñez
Orientador

Prof. Dr. Carlos Humberto Llanos Quintero
Membro

Prof. Dr. Cícero Ferreira Fernandes Costa Filho
Membro

À Deus,
À minha esposa,
Aos meus pais,
Minhas irmãs,
Ao meu orientador
Aos meus professores,
Meus parentes e Amigos...

Agradecimentos

"*Em tudo e em todo lugar reconhecemo-lo com toda a gratidão...*" Atos 24:3ª.

A Deus o todo poderoso a qual nos concedeu a benção da vida, ao mesmo devo toda honra e toda glória por estar continuamente presente em meu viver, e me ajudar não só nos momentos pelo qual clamo em seu nome.

A minha querida esposa, amiga e companheira "Kattiussya Alves Oliveira Dias" por todo o seu amor, carinho, zelo, apoio, paciência, conselhos enfim, por existir em minha vida, me ajudar a crescer e me fazer feliz.

Ao meu querido papai "Nivaldo da Silva Dias" e minha querida mamãe "Suely Aparecida Azevedo Dias" por me ajudarem em todos os momentos do meu caminhar, dando-me forças, apoio e instruindo-me aos caminhos retos.

As minhas irmãs "Adrielly Larissa Azevedo Dias Santos e Anielly Laena Azevedo Dias", pessoas pelas quais sempre pude contar com seu apoio em todo o tempo e lugar.

Ao meu orientador e amigo, professor pós-doutor "Edward David Moreno Ordoñez", ser pelo qual tenho grande admiração e apreço, tornando-se para mim um exemplo a ser seguido pelo seu profissionalismo, inteligência e dedicação em instruir-nos pelos caminhos que levarão às conquistas satisfatórias.

Ao meu coordenador e co-orientador, professor doutor "Raimundo da Silva Barreto", por toda a sua ajuda a mim prestada e pela oportunidade concedida em cursar esse mestrado e assim escalar mais um degrau em minha vida.

Aos meus professores, desde o meu primeiro dia de aula até o presente momento. Pessoas pelas quais tiveram grandes parcelas em meu sucesso de estudo e não me negligenciaram informações, dicas, ajudas e outros, para a minha busca na soma de conhecimentos e informações.

Aos meus parentes e demais amigos que sempre me ajudaram e apoiaram em todos os momentos da minha vida. E também aos companheiros e companheiras do mestrado por todos os momentos vividos e sofridos juntos.

A secretaria do PPGI nas pessoas da Elienai, Marta e Adriana por todo o apoio concedido para mais essa conquista.

Ao INdT e FAPEAM pelas bolsas concedidas para o meu sustento nesse período.

Enfim, a todos só tenho que agradecer de todo o meu coração. Pois tudo isso não valeria a pena se não fosse assim.

“O coração do entendido adquire o conhecimento, e o ouvido dos sábios busca a ciência". Pv.18:15

Resumo

No desenvolvimento do projeto de sistemas embarcados vários fatores têm que ser levados em conta, tais como: tamanho físico, peso, mobilidade, consumo de energia, memória, refrescância, requisitos de segurança, confiabilidade e tudo isso aliado a um custo reduzido e de fácil utilização. Porém, à medida que os sistemas tornam-se mais heterogêneos os mesmos admitem maior complexidade em seu desenvolvimento. Existem diversas técnicas para otimizar o tempo de execução e o consumo de energia em sistemas embarcados. Uma dessas técnicas é a compressão de código, não obstante, a maioria das propostas existentes focaliza na descompressão e assumem que o código é comprimido em tempo de compilação. Portanto, este trabalho propõe o desenvolvimento de uma arquitetura, com respectiva prototipação em hardware (usando VHDL e FPGAs), para o processo de compressão/descompressão de código. Assim, propõe-se a técnica denominada de PDCCM (*Processor Decompressor Cache Compressor Memory*). Os resultados são obtidos via simulação e prototipação. Na análise usaram-se programas do *benchmark MiBench*. Foi também proposto um método de compressão, denominado de *MIC* (*Middle Instruction Compression*), o qual foi comparado com o tradicional método de compressão de *Huffman*. Portanto, na arquitetura PDCCM o método *MIC* apresentou melhores desempenhos computacionais em relação ao método de *Huffman* para alguns programas do *MiBench* analisados que são muito usados em sistemas embarcados, obtendo 26% a menos dos elementos lógicos do FPGA, 71% a mais na freqüência do *clock* em MHz e 36% a mais na compressão das instruções comparando com o método de *Huffman*, além de permitir a compressão/descompressão em tempo de execução.

Palavras-Chave: Sistemas Embarcados, Compressão/Descompressão de código, Processador, Memória, *Cache*.

Abstract

In the development of the design of embedded systems several factors must be led in account, such as: physical size, weight, mobility, energy consumption, memory, cooling, security requirements, trustiness and everything ally to a reduced cost and of easy utilization. But, on the measure that the systems become more heterogeneous they admit major complexity in its development. There are several techniques to optimize the execution time and power usage in embedded systems. One of these techniques is the code compression, however, most existing proposals focus on decompress and they assume that the code is compressed in compilation time. Therefore, this work proposes the development of an specific architecture, with its prototype in hardware (using VHDL and FPGAs), special for the process of compression/decompression code. Thus, it is proposed a technique called PDCCM (Processor Memory Cache Compressor Decompressor). The results are obtained via simulation and prototyping. In the analysis, benchmark programs such as MiBench had been used. Also a method of compression, called of MIC was considered (Middle Instruction Compression), which was compared with the traditional Huffman compression method. Therefore, in the architecture PDCCM the MIC method showed better performance in relation to the Huffman method for some programs of the MiBench analyzed that are widely used in embedded systems, resulting in 26% less of the FPGA logic elements, 71% more in the frequency of the clock MHz and in the 36% plus on the compression of instruction compared with Huffman, besides allowing the compression/decompression in time of execution.

Keywords: Systems Embedded, Compression/Decompression of code, Processor, Memory, Cache.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE SIGLAS

ADPCM	*Adaptive Differential Pulse Code Modulation*
AMD	*Advanced Micro Devices*
ARM	*Advanced RISC Machine*
ASIC	*Application Specific Integrated Circuit*
CAD	*Computer-Aided Design*
CCRP	*Compressed Code RISC Processor*
CDM	*Cache Decompressor Memory*
CI	*Circuit Integrated*
CLB	*Cache Line Address Look-aside Buffer*
CPI	*Instruction Per Cycle*
CPU	*Central Processing Unit*
CRC	*Cyclic Redundancy Check*
DRAM	*Dynamic Random Access Memory*
EEMBC	*EDN Embedded Microprocessor Benchmark Consortium*
EPROM	*Erasable Programmable Read Only Memory*
FDMA	*Frequency Division Multiple Access*
FPGA	*Field Programmable Gate Arrays*
GCC	*GNU Compiler Collection*
GNU	*GNU is Not Unix*
GSM	*Global Standard for Mobile*
HD	*Hard Disk*
HT	*Huffman Table*
IBC	*Instruction Based Compression*
IBM	*International Business Machines Corporation*

IEEE	*Institute of Electrical and Electronics Engineers*
ISA	*Instruction Set Architecture*
ISE	*Integrated Synthesis Environment*
JPGE	*Joint Photographic Experts Group*
LAT	*Line Address Table*
MIC	*Middle Instruction Compression*
MIPS	*Microprocessor without Interlocked Pipeline Stages*
MPEG	*Moving Picture Experts Group*
MSB	*Most Significant Bit*
PCM	*Pulse Code Modulation*
PDC	*Processor Decompressor Cache*
PDCCM	*Processor Decompressor Cache Compressor Memory*
POF	*Programmable Logic Object File*
RAM	*Random Access Memory*
RGB	*Red Green Blue*
RISC	*Reduced Instruction Set Computer*
RUU	*Register Update Unit*
SCE	*Execution Driven Simulation*
SCP	*Program Driven Simulation*
SCR	*Trace Driven Simulation*
SoC	*System-on-Chip*
SOF	*SRAM Object File*
SPARC	*Scalable Processor Architecture*
SPEC	*Standard Performance Evaluation Corporation*
SRAM	*Static Random Access Memory*
ST	*Sign Table*
TDMA	*Time Division Multiple Access*
TIFF	*Tagged Image File Format*
TLB	*Translation Look-aside Buffer*
ULA	*Arithmetic Logic Unit*
VHDL	*Very Hardware Description Language*

SUMÁRIO

CAPÍTULO 1

1. INTRODUÇÃO

Sistemas embarcados são quaisquer sistemas digitais que estejam incorporados a outros sistemas com a finalidade de acrescentar ou otimizar funcionalidades, conforme OLIVEIRA & ANDRADE [31]. Os sistemas embarcados têm por função monitorar e/ou controlar o ambiente no qual esteja inserido. Esses ambientes podem estar presentes, em dispositivos eletrônicos (tais como: celulares, jogos portáteis, aparelhos de som, DVD player, televisores, telefone sem-fio e outros), eletrodomésticos (tais como: forno microondas, geladeira, máquina de lavar roupa, ar-condicionado e outros), veículos (exemplos de freio ABS, computador de bordo, injeção eletrônica e outros), motores (exemplos de controladores de grupo geradores, controladores de pressão e outros), máquinas (tais como: controle de esteiras, braços mecânicos e outros), ambientes físicos (exemplo de módulos de sensoriamento em uma rede de sensores sem-fio monitorando um habitat) e muitos outros ambientes nos quais pode-se encontrar sistemas embarcados [31].

A crescente demanda pelo uso de sistemas embarcados tem se tornado cada vez mais comum à implementação de complexos sistemas em um único *chip*, os chamados *system-on-chip* (SoC), no entanto, o processador embarcado é um dos principais componentes dos sistemas computacionais embarcados, BENINI *et al* [4]. Esses processadores para aplicações embarcadas são extremamente simples (CPUs de apenas 8 ou 16 *bits*). Mas atualmente, muitos processadores embarcados encontrados no mercado são baseados em arquiteturas de alto-desempenho (arquitetura RISC de 32 *bits*) que garantem um melhor desempenho

computacional para as tarefas a serem executadas. Portanto, o projeto de sistemas embarcados para processadores de alto-desempenho não é uma tarefa simples.

Sabe-se que grande parte dos sistemas embarcados são alimentados por baterias, portanto, é de suma importância que estes sistemas sejam *power-aware*, isto é, capazes de controlar e gerenciar sua potência, possibilitando assim a redução no consumo de energia e no controle do aquecimento. Então, projetistas e pesquisadores concentraram-se no desenvolvimento de técnicas que diminuam o consumo de energia mantendo os requisitos de desempenho. Uma dessas técnicas é a compressão do código das instruções em memória.

A grande maioria das técnicas, metodologias e padrões de desenvolvimento de software, para o controle e o gerenciamento do consumo de energia, não se mostra viável para o desenvolvimento de sistemas embarcados, devido os mesmos sofrerem inúmeras limitações de recursos computacionais e físicos. As atuais estratégias concebidas para o controle e o gerenciamento no consumo de energia, foram desenvolvidas para sistemas de propósitos gerais, em que os custos adicionais de processadores ou memórias são geralmente insignificantes.

À medida que os sistemas tornam-se mais heterogêneos os mesmos admitem maior complexidade em seu desenvolvimento, entretanto, os softwares que neles operam também ampliam o seu grau de complexidade, causando assim aumento significativo em seu código na memória. Neste sentido, surgiu 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, NETTO *et al* [27, 28, 30].

A técnica de compressão foi desenvolvida com o intuito de reduzir o tamanho de um código [30]. Mas no decorrer do tempo, grupos de pesquisadores verificaram que essa técnica poderia trazer grandes benefícios para o desempenho e o consumo de energia nos sistemas de propósitos gerais e nos sistemas embarcados. A partir do momento que o código em memória está comprimido é possível em cada requisição do processador, buscar uma quantidade bem maior de instruções contidas na memória. Assim, haverá uma diminuição nas atividades de transição nos pinos de acesso à memória, levando a um possível aumento no desempenho do sistema e a uma possível redução no consumo de energia do circuito, NETTO *et al* [30].

Diante destas questões, surge a necessidade de criar um hardware capaz de aumentar a capacidade de armazenamento de instruções na memória *cache*, fazendo a compressão e descompressão das instruções em tempo de execução. Assim, desenvolvemos esta dissertação com o intuito de projetar e implementar um hardware compressor/descompressor, baseando-se nas arquiteturas PDC (Processador Descompressor *Cache*) e CDM (*Cache* Descompressor Memória) e em um novo método de compressão/descompressão denominado de *MIC* (*Middle Instruction Compression*); para isto utilizamos a linguagem VHDL (*Very Hardware Description Language*) [11] na descrição do hardware e uma FPGA (*Field Programmable Gate Arrays*) para a prototipação.

O diferencial neste trabalho refere-se a compressão/descompressão das instruções em tempo de execução e em hardware. Tendo em vista que os trabalhos desenvolvidos até aqui, utilizam apenas a descompressão das instruções em tempo de execução (feita pelo hardware), pois a compressão das instruções são feitas em nível de compilação. Com o hardware realizando as funções de compressão/descompressão poder-se-ia ter como benefícios uma possível redução do tempo de execução e do consumo de energia.

Portanto, a grande contribuição com esta dissertação foi o desenvolvimento de uma nova arquitetura (PDCCM) para compressão/descompressão dos códigos de instruções em tempo de execução e o desenvolvimento de um novo método denominado de *MIC* (*Middle Instruction Compression*) que realiza uma compressão de 50% no tamanho das instruções que são salvas na *cache* de instrução.

1.1. Trabalhos Correlatos

Nesta seção, são mostradas algumas arquiteturas para a execução de códigos de instruções comprimidas, encontrado nas literaturas.

WOLFE & CHANIN [39] desenvolveram o CCRP (*Compressed Code RISC Processor*), que foi o primeiro hardware descompressor implementado em um processador RISC (MIPS R2000) e também foi a primeira técnica a usar as falhas de acesso à *cache* para acionar o mecanismo de descompressão.

O CCRP tem uma arquitetura idêntica ao padrão do processador RISC e assim os modelos dos programas são inalterados. Isso implica em que todas as ferramentas de desenvolvimento existentes para a arquitetura RISC, incluindo compiladores otimizados, simuladores funcionais, bibliotecas gráficas e outras, também servem para a arquitetura CCRP.

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, devem ser descomprimidas e então alimentam a linha da *cache* onde houve a falha [39]. Segundo WOLFE & CHANIN [39] 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 [39]:

- Colocar hardware extra no processador para tratamento diferenciado dos saltos;
- Fazer *patches* de endereços de salto.

Conforme WOLFE & CHANIN [39] a técnica CCRP mostrou uma razão de compressão de 73%, em média para o pacote testado (composto pelos programas: *nasa1*, *nasa7*, *tomcatv*, *matrix25A*, *espresso*, *fpppp* e outros). Para modelos de memórias mais lentos do tipo DRAM (*Dynamic Random Access Memory*), o desempenho do processador foi na maioria das vezes suavemente melhorado. Para modelos mais rápidos de memória EPROM (*Erasable Programmable Read Only Memory*), o desempenho sofreu uma leve degradação.

BENINI *et al* [4] desenvolveram um algoritmo de compressão que é adaptado para execução eficiente do hardware (descompressor). As instruções são compactadas em grupos que têm o tamanho de uma linha da *cache* e a sua descompactação ocorre no instante que são extraídas da *cache*.

Os autores [4] fizeram experimentos com o processador DLX, devido o mesmo ter uma arquitetura simples de 32 *bits* e também ser uma arquitetura RISC. Além disso, o processador DLX é semelhante a vários processadores comerciais da família ARM [1] e MIPS.

Uma tabela de 256 posições foi utilizada para guardar as instruções mais executadas. Cada linha da *cache* é formada por 4 instruções originais ou um conjunto de instruções comprimidas e possivelmente intercaladas com outras não comprimidas, prefixado por uma palavra de 32 *bits*. A palavra não comprimida tem um posicionamento fixo na linha da *cache* e serve para diferenciar uma linha de *cache* com instruções comprimidas das outras linhas com as instruções originais. De fato, uma linha de *cache* comprimida não contém necessariamente todas as instruções comprimidas, mas sempre deve ter um número entre 5 e 12 instruções comprimidas na linha da *cache* para ser vantajoso o uso da compressão [4].

O algoritmo de compressão desenvolvido por BENINI *et al* [4], analisa o código seqüencialmente, a partir da primeira instrução (supondo que cada linha da *cache* já esteja alinhada) e tenta acondicionar instruções adjacentes em linhas comprimidas. Os experimentos realizados em vários pacotes de código C do *benchmark* fornecido pelo projeto *Ptolemy* [10], comprovaram que houve uma redução média no tamanho do código de 28% e uma economia média no consumo de energia de 30%.

LEKATSAS & WOLF [25] desenvolveram uma unidade de descompressão com um único ciclo. A descompressão pode ser aplicada para instruções de qualquer tamanho de um processador RISC (16, 24 ou 32 *bits*). A única aplicação específica é a parte do interfaceamento entre o processador e a memória (principal ou *cache*). O mecanismo de descompressão é capaz de descomprimir uma ou duas instruções por ciclo para atender a demanda da CPU sem aumentar o tempo de execução.

Os autores desenvolveram uma técnica para criar um dicionário que contém as instruções que aparecem com mais freqüência. O dicionário de código refere-se a uma classe de métodos de compressão que substitui seqüências de símbolos com os índices de uma tabela. Essa tabela é chamada de "dicionário" e os índices são os "*codewords*" no programa compactado [25]. A principal vantagem dessa técnica é que os índices geralmente são de comprimento fixo, e assim, simplifica a lógica da descompressão em acessar o dicionário e também reduz a latência da descompressão.

O algoritmo de descompressão é formado por uma árvore lógica. A ordem de descompressão das palavras pode ser esquematicamente representada por uma árvore. O nó raiz da árvore recebe instruções comprimidas de 32 *bits*.

O mecanismo de descompressão funciona da seguinte forma: o descompressor 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 tipo: *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*).

O mecanismo de descompressão foi projetado usando VHDL e Verilog e a simulação foi realizada usando o simulador *Vsystem* do *Model-Tech's* [25]. A CPU e o circuito do descompressor foram sintetizados usando a ferramenta *Synplicity's Synplify* para o dispositivo APEX20XE da família Altera®. A base de dados no formato *Synplify* foi utilizada pelas ferramentas Quartus II e Max-Plus II que geraram arquivos de saídas para o projeto no formato POF (*Programmable logic Object File*) e SOF (*SRAM Object File*).

Os resultados obtidos nos testes realizados demonstraram que houve um ganho médio de 25% de desempenho no tempo de execução dos aplicativos usando a compressão de código e uma média de 35% na redução do tamanho do código [25]. Segundo os autores essa tecnologia desenvolvida não está limitada a um único processador, pois pode ser aplicada e obter resultados similares em outros processadores.

Assim, são encontrados dois tipos básicos de arquiteturas de compressão de código, CDM (*Cache Decompressor Memory*) e PDC (*Processor Decompressor Cache*), identificando o posicionamento do descompressor em relação ao processador e subsistema de memória, como mostra a Figura 1.1. A arquitetura CDM indica que o descompressor está posicionado entre a *cache* e a memória principal, enquanto que a arquitetura PDC posiciona o descompressor entre o processador e a *cache*.

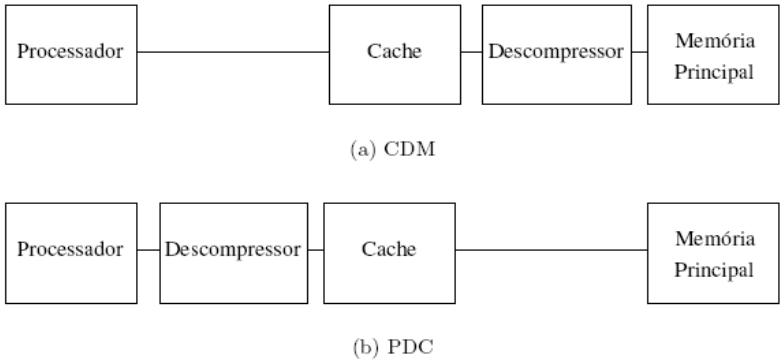

Figura 1.1 – Arquiteturas de descompressão de código: (a) CDM e (b) PDC [27]

Existem algumas vantagens e desvantagens de cada uma das arquiteturas. 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. Já as arquiteturas CDM podem ter tempo de descompressão mais lento e em contra partida costuma ter uma razão de compressão melhor.

Em ambas as arquiteturas, existem códigos de instruções já comprimidos em tempo de compilação, seja na memória principal (arquitetura CDM) ou na memória *cache* (arquitetura PDC). Então essas arquiteturas (CDM e PDC) só trabalham com a parte de descompressão das instruções, uma vez que o compilador já fez todo o processo de compressão das instruções.

1.2. Objetivos da Dissertação

1.2.1. Objetivo Geral

O objetivo principal desta dissertação é desenvolver um método de compressão/descompressão e projetar um hardware (em FPGA) capaz de comprimir e descomprimir (em tempo de execução) os códigos de instruções armazenadas na memória de sistemas embarcados e que seja compatível com processadores embarcados de arquitetura RISC (ARM, PowerPC, XScale e outros).

1.2.2. Objetivos Específicos

- Conhecer o estado da arte em termos de compressão e descompressão de código em memória;
- Realizar uma caracterização de sistemas embarcados sob a visão de desempenho, consumo de memória, tempo de execução e outras métricas usando programas do *benchmark MiBench* [16];
- Obter uma caracterização arquitetural de processadores embarcados nos quesitos de tamanho das *caches* de dados e instruções; quantidade de instruções decodificadas por ciclos; largura do barramento de memória; latência das *caches* de dados e instruções; número de ULAs em inteiros e em pontos flutuantes e outros;
- Prototipar um hardware em FPGA para comprimir e descomprimir códigos de instruções em tempo de execução;
- Desenvolver um novo algoritmo de compressão/descompressão em hardware;
- Projetar uma versão em hardware do algoritmo de compressão *Huffman*.

1.3. Metodologia

A metodologia empregada no desenvolvimento deste trabalho envolveu as seguintes etapas:

- Cursar as disciplinas;
- Definição e análise do projeto;
- Embasamento teórico em compressão/descompressão de código;
- Definição da arquitetura do módulo de compressão e descompressão;
- Definição do método de compressão e descompressão;
- Simulação arquitetural de processadores embarcados usando a ferramenta SimpleScalar [45];
- Implementação do método de compressão e descompressão em VHDL;
- Projeto e prototipação do hardware compressor/descompressor em uma FPGA;
- Realização de testes;
- Análises dos testes;
- Escrita e submissão de artigos científicos;

- Escrita da dissertação;
- Defesa da dissertação.

1.4. Recursos

Para elaboração deste trabalho foi necessário a utilização dos seguintes recursos:

- Computador;
- Plataforma para prototipação do hardware compressor/descompressor (FPGA);
- Ferramentas:
  - descrição e simulação do hardware compressor/descompressor (ferramenta da ALTERA® "Quartus II Web *Edition*" [43]);
  - mensurar os desempenhos arquiteturais de processadores embarcados usando o *benchmark MiBench* (ferramenta SimpleScalar [45]).

Os recursos necessários para o desenvolvimento desse trabalho foram fornecidos pelo Programa de Pós-Graduação em Informática (PPGI) da Universidade Federal do Amazonas (UFAM).

1.5. Contribuições do Trabalho

Com o desenvolvimento desse trabalho obtiveram-se as seguintes contribuições:

- Foi desenvolvido uma nova arquitetura PDCCM (Processador Descompressor *Cache* Compressor Memória) que realiza tanto a compressão quanto a descompressão dos códigos de instruções em tempo de execução;
- Foi projetado um módulo compressor e descompressor de código, o qual foi implementado em hardware (prototipado em FPGA) usando a nova arquitetura PDCCM compatível com o processador ARM [44], muito usado em sistemas embarcados;
- Foi feito uma caracterização arquitetural de processadores embarcados executando os programas do *benchmark MiBench*;

- Foi gerado um novo método para compressão/descompressão (nesta dissertação denominado de método *MIC*) de código de instrução.

1.6. Organização da Dissertação

Este trabalho está organizado em 6 capítulos, da seguinte forma:

- O capitulo 1 apresenta a importância da compressão/descompressão de código de instrução e enfatiza nos objetivos e contribuições desta dissertação;
- O capítulo 2 apresenta os principais conceitos de compressão/descompressão de código, arquitetura CDM, arquitetura PDC, juntamente com a análise dos trabalhos correlatos;
- O capítulo 3 apresenta as metodologias de avaliação (simulação, prototipação e modelagem) juntamente com os detalhes da metodologia utilizada nesta dissertação, focando nas características do *benchmark MiBench* para medição de desempenho e também na caracterização das arquiteturas de processadores embarcados; logo em seguida é visto a simulação com os *benchmark MiBench* usando a ferramenta de SimpleScalar;
- O capítulo 4 apresenta a descrição detalhada da arquitetura do hardware compressor/descompressor e a sua simulação e implementação em VHDL. Esse hardware é compatível com instruções provenientes de um sistema embarcado baseado em processadores ARM de 32 *bits*;
- O capítulo 5 apresenta as simulações realizadas na arquitetura PDCCM usando alguns programas do *benchmark MiBench* para os métodos de compressão/descompressão *MIC* e *Huffman* obtendo os resultados mensurados tais como: as estatísticas de recursos utilizados do FPGA e respectiva temporização, assim como a taxa de compressão e tempo de execução;
- E finalmente, o capítulo 6 apresenta o fechamento do trabalho destacando as principais contribuições, assim como sugerindo possíveis trabalhos futuros.

CAPÍTULO 2

2. ARQUITETURAS PARA COMPRESSÃO DE CÓDIGO

Neste capítulo, é descrito o conceito de compressão de código e em seguida os conceitos das arquiteturas existentes, tais como: arquitetura CDM e arquitetura PDC, juntamente com os trabalhos correlatos encontrados na literatura. Portanto, no final do capítulo também aparecem tabelas comparativas entre as arquiteturas mencionadas.

2.1. Compressão de Código

A compressão de código constitui-se na criação de uma imagem do programa original, ou seja, uma cópia do programa formada com menos *bits*. Porém, para a descompressão do código é necessário usar uma rotina de descompressão (via *software*) ou um mecanismo implementado em hardware que realiza a descompressão do código comprimido (via *hardware*).

Existe um grande número de métodos de compressão proposto nas literaturas. A compressão de código pode ser obtida no próprio projeto do processador, utilizando campos menores para codificar instruções mais freqüentes e campos maiores para codificar as instruções pouco freqüentes, o projeto *Burroughs* de WILNER [38] é um exemplo desse método. Na compressão/descompressão de código é essencial a escolha de um método de compressão no qual não será gerado um código descomprimido errôneo, ou seja, sem perdas

de instruções.

A compressão de código geralmente é realizada em tempo de compilação e a descompressão de código é realizada em tempo de execução, NETTO *et al* [29, 30]. Porém, na descompressão de código não é possível aplicar os métodos de compressão de dados (tais como: *Shannon-Fano*, *Huffman*, *Aritmético*, *Elias-Bentley*, *Lempel-Ziv* e outros) [36, 15, 18, 12, 6, 41]. Pois, os métodos de compressão de dados só permitem a descompressão do código de forma seqüencial e não permitem a descompressão do código de forma aleatória.

A descompressão de código nem sempre pode ser realizada de forma seqüencial devido às instruções de desvios (controle de fluxos) existentes nos programas. A Figura 2.1 mostra um exemplo de um trecho de código que contém instruções de desvios.

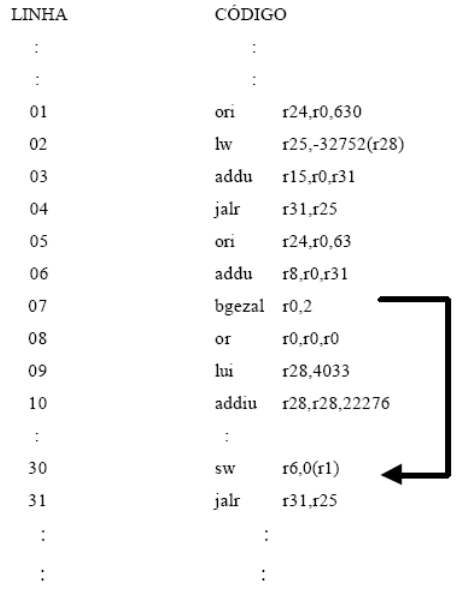

Figura 2.1 – Exemplo de trecho de código que contém instrução de desvios [2]

No exemplo da Figura 2.1, supondo que a instrução de desvio da linha 07 tenha que ser executada e que a instrução alvo do desvio seja a instrução da linha 30, para que a execução do programa seja feita corretamente é necessário enviar ao processador as instruções a partir da linha 30. Porém, se o trecho de código da Figura 2.1 fosse comprimido utilizando qualquer método de descompressão de dados, a próxima linha a ser descomprimida seria a linha de instrução subseqüente a linha da instrução 07, passando ao processador uma informação errônea.

No restante deste capítulo serão mostradas algumas arquiteturas para execução de códigos comprimidos obtidas através de técnicas de compressão/descompressão de códigos encontradas nas literaturas, artigos, internet e outros.

2.2. Arquitetura CDM

A arquitetura de compressão de código CDM (*Cache Decompressor Memory*) é identificada devido o posicionamento do descompressor em relação ao processador e subsistema de memória, como mostra a Figura 2.2. Na arquitetura CDM o descompressor está posicionado entre a *cache* e a memória principal.

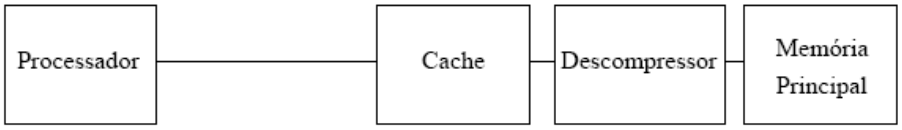

Figura 2.2 – Exemplo da arquitetura CDM [27]

Baseando-se na arquitetura CDM, WOLFE & CHANIN [39], desenvolveram um sistema de descompressão chamado de CCRP (*Compressed Code RISC Processor*). O CCRP foi desenvolvido a fim de realizar simulações e a sua implementação reflete na tecnologia RISC (*Reduced Instruction Set Computer*) para processadores embarcados. A Figura 2.3 mostra a arquitetura do CCRP.

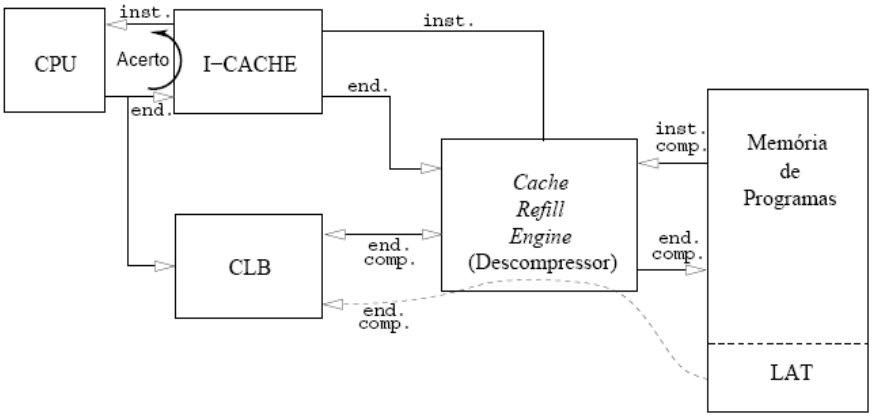

Figura 2.3 – Arquitetura do *Compressed Code RISC Processor* [25]

No que se refere à descompressão, o mecanismo proposto é inovador em alguns aspectos, pois é o primeiro hardware descompressor implementado em um processador RISC (MIPS R2000) e também é a primeira técnica que usa as falhas de acesso a *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 [39]:

- Colocar hardware extra no processador para tratamento diferenciado dos saltos;
- Fazer *patches* de endereços de salto.

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 foi resolvido usando LAT (*Line Address Table*).

A LAT foi implantada no hardware de reenchimento da *cache*, e contém os mapas dos programas com os endereços não comprimidos da *cache* em endereços na memória principal, [39]. Os dados contidos na LAT são gerados por ferramentas de compressão e armazenado em conjunto com o programa. A Figura 2.4 exemplifica a arquitetura da LAT de uma linha de *cache* de 32 *bits*.

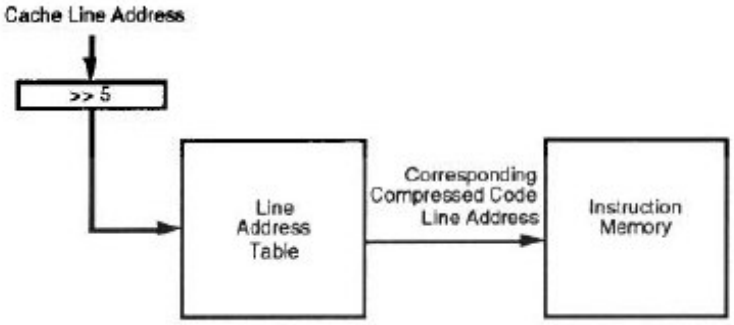

Figura 2.4 – Arquitetura da *Line Address Table* [39]

Uma restrição imposta para que o 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.

Segundo WOLFE & CHANIN [39] 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 Look-aside Buffer*), responsável por armazenar as últimas entradas consultada da LAT, minimizando significativamente o *overhead* causado nas falhas de acesso à *cache*.

O sistema descompressor CCRP foi posicionado entre a *cache* de instrução (*I-cache*) e a memória principal (RAM - *Random Access Memory*), 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 a *I-cache* e em seguida para o 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 a *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 *clock*, podendo este número aumentar no caso de memórias lentas, onde o mecanismo de descompressão não é o gargalo.

WOLFE & CHANIN [39] utilizaram o código de *Huffman* [18] gerado através de um histograma de ocorrências de *bytes* do programa. A desvantagem deste método está na

descompressão, pois, geralmente a codificação *Huffman* gera códigos com palavras grandes e com tamanho variável, fazendo com que o hardware responsável pela descompressão se torne caro e complexo. As palavras de código grandes fazem com que, em alguns casos, não possam ser armazenadas em uma única palavra de memória. Isto acontecendo, a máquina de descompressão teria que ir à memória principal mais de uma vez para poder descomprimir uma instrução. O tamanho variável da palavra de código aumenta a complexidade do hardware descompressor, pois ele tem que detectar o tamanho da palavra de código antes de descomprimi-la.

Os autores fizeram testes no CCRP e demonstraram uma razão de compressão de 73% em média, no tamanho do código, mas nos testes não foi levado em conta o tamanho da LAT, da CLB nem do descompressor. Observa-se ainda que o tamanho do símbolo utilizado é de 1 *byte*, sendo que para os símbolos maiores implicará em um hardware de descompressão mais lento.

Seguindo a arquitetura CDM, a IBM (*International Business Machines Corporation*) [19, 14] desenvolveu um novo método de descompressão chamado de CodePack e tendo como plataforma alvo o processador PowerPC. Os autores do projeto verificaram diferenças significativas na compressão de código, aplicando na parte alta das instruções (16 *bits* mais significativos) em relação à parte baixa das instruções (16 *bits* menos significativos) tirando assim proveitos do formato das instruções do processador PowerPC. Então, foram criados dois dicionários de códigos um para cada parte da instrução, obtendo assim melhores resultados na compressão de código.

A Figura 2.5 mostra um exemplo da codificação usando o método de compressão CodePack. Cada símbolo consiste em um ou dois campos (cada *bit* é representado por *n*). Os oitos símbolos mais freqüentes recebem a codificação mais densa, 00_2. Os outros 32 *bits* recebem a codificação 01_2 e assim por diante, até os menos freqüentes, que recebem sua codificação original (16 *bits* não comprimidos).

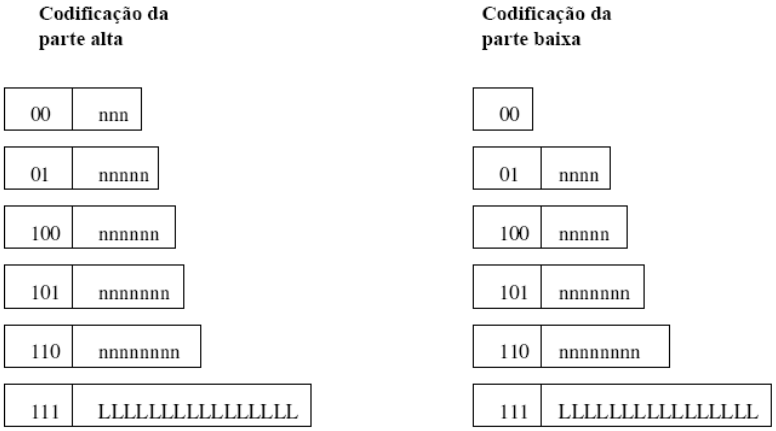

Figura 2.5 – Codificação de símbolos do CodePack da IBM [19]

A arquitetura de descompressão proposta pela IBM [19] é muito parecida com a arquitetura CCRP proposta por WOLFE & CHANIN [39], fazendo parte do sistema de memória. Uma LAT é responsável por fazer o mapeamento entre os endereços não comprimidos (*I-cache*) em endereços comprimidos (memória principal).

Segundo os autores [19], a implementação dessa arquitetura sem considerar otimizações, fornece ao processador uma instrução a cada três *cycle-time*, sendo uma taxa muito baixa. Uma possível otimização para o método é fazer com que as tabelas de mapeamento apontem para blocos de instruções (16 instruções por bloco) ao invés de deixar uma entrada na tabela a cada instrução. Uma outra otimização é partindo do princípio que o processador fica grande parte do tempo executando trechos de códigos seqüenciais (sem salto), então, o descompressor previamente já conhece a próxima linha da *cache* a ser buscada na memória, sem haver a necessidade de realizar consultas na LAT, causando assim a economia de um *cycle-time* em cada recarga da *cache*.

Em [19, 14] não é fornecido um valor exato sobre o desempenho obtido, sendo citado apenas que para taxas altas de acertos na *I-cache*, a queda de desempenho é relativamente pequena, mas que esse valor é expressivo à medida que a taxa de acertos cai, devido ao *delay* fornecido pelo hardware de descompressão. Já a razão de compressão obtida nos teste alcançou em torno de 60%.

Outro trabalho interessante realizado com arquitetura CDM é o trabalho do AZEVEDO [2], onde foi proposto um método chamado de IBC (*Instruction Based Compression*), que tem por função realizar a divisão do conjunto de instruções do processador em classes, levando em consideração a quantidade de ocorrências juntamente com número de elementos de cada classe. Pesquisas mostraram melhores resultados na compressão de 4 classes de instruções. A técnica de compressão desenvolvida consiste em agrupar pares no formato [prefixo, *codeword*] que substituem o código original. Nos pares formados, o prefixo indica a classe da instrução e o *codeword* serve como um índice para a tabela de instruções. Como nos outros métodos visto anteriormente, este também necessita de uma LAT.

Em [2] é vista a implementação do método para dois tipos de processadores sendo eles MIPS (*Microprocessor without Interlocked Pipeline Stages*) e SPARC (*Scalable Processor Architecture*), usando a arquitetura de descompressão semelhante. A Figura 2.6 mostra a arquitetura do descompressor de código IBC proposto por Azevedo (2002) para o processador SPARC V8 LEON.

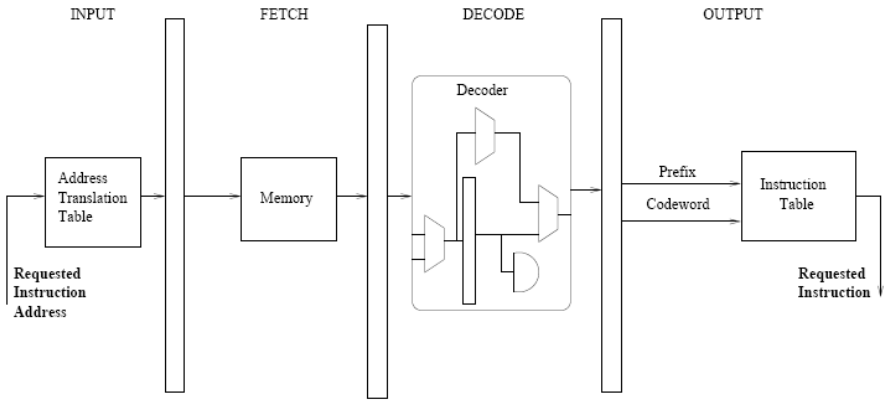

Figura 2.6 – Arquitetura do descompressor IBC para processador SPARC V8 LEON [2]

O processo de descompressão é realizado em 4 estágios de *pipeline*. O primeiro estágio é chamado de *INPUT* onde é convertido o endereço do processador (código não comprimido) em endereço da memória principal. O segundo estágio é chamado de *FETCH*, que é responsável pela busca da palavra comprimida na memória principal. O terceiro estágio é conhecido como *DECODE* onde verdadeiramente é realizada a decodificação dos *codewords*. E finalmente no quarto estágio, chamado de *OUTPUT*, é realizada a consulta no

dicionário de instrução para ser fornecida a instrução ao processador.

Nos testes realizados por Azevedo (2002) obteve-se uma taxa de compressão de 53,6% para o processador MIPS e 61,4% para o processador SPARC. Quanto ao desempenho, constatou-se uma perda de 5,89% utilizando o método IBC [2].

2.3. Arquitetura PDC

A arquitetura de compressão de código PDC (*Processor Decompressor Cache*) é identificada devido ao posicionamento do descompressor em relação ao processador e subsistema de memória, como mostra a Figura 2.7. Porém, na arquitetura PDC o descompressor está posicionado entre o processador e a *cache*.

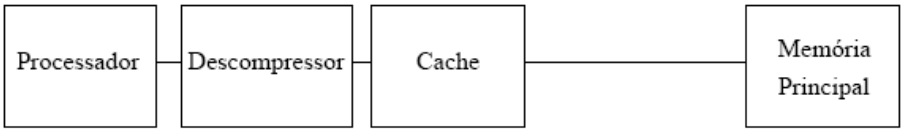

Figura 2.7 – Exemplo da arquitetura PDC [27]

Uma das vantagens que pode ser citada quanto ao uso da arquitetura PDC em comparação com a arquitetura CDM, é que na arquitetura PDC geralmente haverá um melhor desempenho devido a um possível aumento na taxa de acertos na *cache*. Por outro lado, esta arquitetura é mais difícil de ser implementada sem o aumento no *cycle-time* do processador, sendo assim uma das suas desvantagens.

Mediante a esta abordagem, LEKATSAS *et al* [23] desenvolveram um método de compressão de código onde as instruções foram agrupadas separadamente por suas características próprias. As instruções da plataforma SPARC, foram utilizadas para a criação dos grupos de instruções, sendo [23]:

- Grupo 1 – Instruções com Imediatos: é utilizada a codificação aritmética. Sabe-se que este tipo de codificação possui vantagens significativas sobre o método de compressão de *Huffman*, especialmente apropriada quando as ocorrências das instruções com imediatos são bastante espaçadas. Na maioria das vezes, a tabela

de codificação é criada de forma mais genérica e produz uma razão de compressão melhor do que a codificação de *Huffman*, [3];

- Grupo 2 – Instruções de Salto: são comprimidas através da compactação do campo de deslocamento. Para obter a compressão, codifica-se apenas a quantidade mínima de *bits* necessários que representa o deslocamento (salto) em questão;
- Grupo 3 – Instruções de Acesso Rápido: são os índices para um dicionário de instruções, portanto, essas instruções podem ser descomprimidas com um simples acesso a um dicionário de código e usando apenas um único ciclo do processador;
- Grupo 4 – Instruções Não Comprimidas: são as instruções do código que não foram comprimidas.

A identificação dos grupos é realizada através de uma seqüência de *bits*, sendo que [23]:

- Grupo 1 = 0_2;
- Grupo 2 = 11_2;
- Grupo 3 = 100_2;
- Grupo 4 = 101_2.

Desta forma, o hardware descompressor consegue realizar de forma correta a descompressão do código de acordo com o seu grupo de instrução, visto que cada grupo possui a sua particularidade, ou seja, uma codificação própria.

Dando continuidade ao método de compressão por grupos, LEKATSAS *et al* [24] sugeriram uma nova arquitetura de descompressão para o método proposto em [23]. O descompressor é implantado entre o processador e a *cache*, chamando assim de arquitetura PDC.

A intenção de implantar o descompressor entre o processador e a *cache* foi de aumentar o número de acertos na *cache*, e assim conseguir obter dois grandes objetivos, sendo eles [24]:

- Redução no consumo de energia;
- Ganho de desempenho.

O desenvolvimento do descompressor é ajustado com o *pipeline* do processador conforme mostra a Figura 2.8. São adicionados quatro *pipelines*, um para cada grupo de instruções descrito em [23]. Conforme a Figura 2.8, observa-se que antes do hardware descompressor existe um *buffer* de entrada (*I-Buf*), que tem por função armazenar as instruções comprimidas vindas da *cache* de instruções (*I-cache*) e após o hardware descompressor existe um *buffer* de saída (*O-Buf*), que tem por função armazenar as instruções descomprimidas e fornecê-las ao *pipeline* do processador à medida que forem requisitadas.

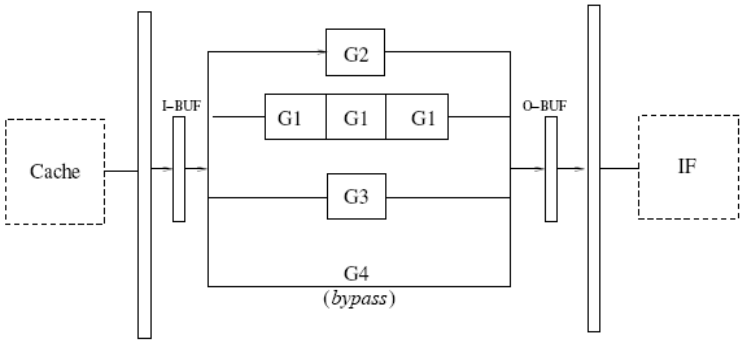

Figura 2.8 – Arquitetura do descompressor *pipelined* proposto por LEKATSAS *et al* [24]

Uma característica marcante nesta arquitetura é que ela permite a execução dos quatro *pipelines* ao mesmo tempo, causando assim certa dificuldade na projeção da modelagem do hardware descompressor. Exemplo desta característica pode ser visto no caso de uma instrução Y (que vem após uma instrução X no programa) ser descomprimida mais rápido que a instrução X, pois Y é do grupo 3 (descompressão em apena um ciclo do processador). Isto faz com que as instruções chegem ao *O-Buf* fora de ordem, necessitando assim de um hardware extra para realizar o controle da chegada e entrega das instruções descomprimidas ao processador.

LEKATSAS *et al* obtiveram os seguintes resultados [24]: 53% das instruções pertencem ao grupo 1; 26,7% das instruções ao pertencem ao grupo 2; 19,7% das instruções formam o grupo 3 e apenas 0,6% das instrução faz parte do grupo 4, ou seja, são instruções que não são comprimidas. Quanto à razão de compressão ficou em torno de 65%. Ainda vale ressaltar que nestes valores obtidos não foi levado em conta o *overhead* do hardware descompressor, que é bastante complexo, o que impossibilita usar esta métrica para

comparação com outros métodos. Já as estimativas: redução no consumo de energia apontou em média um índice de 28% de economia e um ganho de desempenho de 25% em média.

Em outro trabalho [22], LEKATSAS *et al* ainda sugeriram outra arquitetura de descompressão. Neste trabalho, o grande objetivo foi desenvolver um hardware descompressor capaz de descomprimir de uma a duas instruções por ciclo para atender a demanda do processador sem nenhuma penalidade no tempo de execução da descompressão. Para isto foi utilizado como plataforma alvo o processador *Xtensa*-1040 [20].

LEKATSAS *et al* [22] definiram algumas estratégias arquiteturais necessárias para permitir que o hardware de descompressão realize a descompressão em apenas um ciclo, tais estratégias são:

- Nenhuma modificação ao processador é necessária. Isso significa que o hardware de descompressão tem interface com o processador e à memória hierárquica. Portanto, ambos não estão cientes da existência do hardware descompressor;
- O hardware de descompressão deverá ser consideravelmente menor do que o tamanho do processador.

Para esta arquitetura [22], é utilizado um método de compressão completamente baseado em dicionários. Mediante as estatísticas do programa, as instruções que aparecem com mais freqüência no código são substituídas por um índice do dicionário de código (que contém 256 entradas). Porém, o dicionário de código refere-se a uma classe de métodos de compressão que substitui seqüências de símbolos com os índices de uma tabela. Essa tabela é chamada de "dicionário" e os índices são os "*codewords*" no programa comprimido. A principal vantagem dessa técnica é que os índices geralmente são de comprimento fixo, e assim, simplifica a lógica da descompressão em acessar o dicionário e também reduz a latência da descompressão. A Figura 2.9 mostra a arquitetura básica proposta por LEKATSAS *et al* em [22] e a Figura 2.10 ilustra a substituição de uma instrução por um *codewords* do dicionário de compressão.

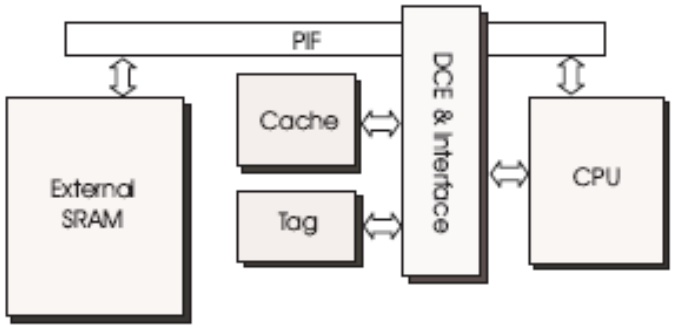

Figura 2.9 – Arquitetura básica proposta por LEKATSAS *et al* [22]

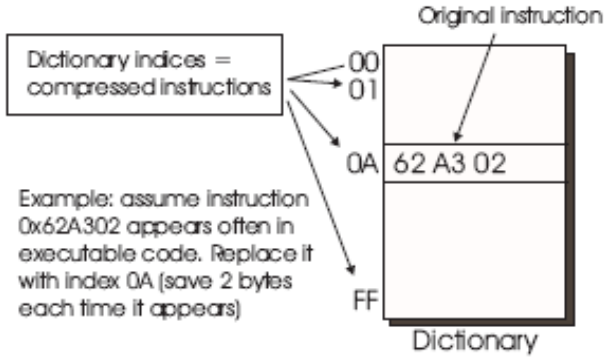

Figura 2.10 – Substituição de instrução por *codewords* do dicionário de compressão [22]

As instruções de 24 *bits* do processador *Xtensa*-1040 podem receber dois tipos de codificação, sendo elas: 8 *bits* e 16 *bits*. Quando a codificação é 8 *bits*, o código inteiro é usado para endereçar o dicionário. No caso da codificação de 16 *bits*, apenas os 8 *bits* mais significativos são usados para endereçamento do dicionário, enquanto os outros 8 *bits* menos significativos 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 parte mais significativa do algoritmo de descompressão para essa arquitetura é formada por uma árvore lógica. Porém, na Figura 2.11 é representada a ordem de descompressão das palavras. O nó raiz da árvore recebe instruções comprimidas de 32 *bits* e é apontado pela seta de entrada (*32-bit input*). Daí em diante, este *stream* de *bits* vai percorrendo os nós da árvore, até chegar a uma ponta da árvore, ou seja, uma folha da árvore

que contém realmente uma instrução comprimida. Em cada nó (ou bifurcação) é possível gerar duas ou mais instruções, realizando assim a descompressão (mediante as consultas ao dicionário) e em cada nó da árvore também será mostrado o tamanho das instruções que possivelmente poderá ocorrer. Portanto, o processador espera receber um *bitstreams* de 32 *bits* das instruções originais.

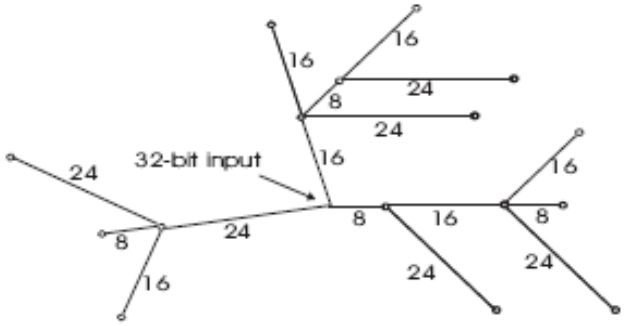

Figura 2.11 – Árvore lógica para hardware de descompressão de 1 ciclo [22]

Desta forma, a estrutura de decodificação da árvore proposta em [22], garante que sempre que um nó folha for atingido, pelo menos 32 *bits* de instruções estarão disponíveis para o processador. Então, os bits que forem excedentes aos 32 *bits* requeridos pelo processador são armazenados em um *buffer* para serem usado no próximo ciclo.

Tendo como base tudo aquilo que foi citado anteriormente, a Figura 2.12 mostra o diagrama de blocos da arquitetura do hardware descompressor proposto por LEKATSAS *et al* [22]. O hardware de descompressão é ativado com a chegada de um *bitstream* de 32 *bits*. Primeiramente, o hardware descompressor verifica os 4 *bits* menos significativos para selecionar as linhas corretas do multiplexador e determinar qual o tipo de instrução. Em seguida, a tabela de descompressão é acessada repassando uma instrução de 24 *bits* descomprimida na saída. Em alguns casos, o *bitstream* de 32 *bits* contém apenas parte de uma instrução, sendo que o restante da instrução está contida no próximo *bitstream*. Na ocorrência destes casos existem alguns registradores específicos para os tais, como [22]:

- ***Pre_part_inst***: guarda o pedaço da instrução descomprimida no ciclo anterior;
- ***Pre_part_set***: indica que um pedaço da instrução atual já foi descomprimido no ciclo anterior;
- ***Bytes_next***: indica se 8 ou 16 *bits* de *Pre_part_inst* foram decodificados no ciclo

anterior (*Pre_part_inst*).

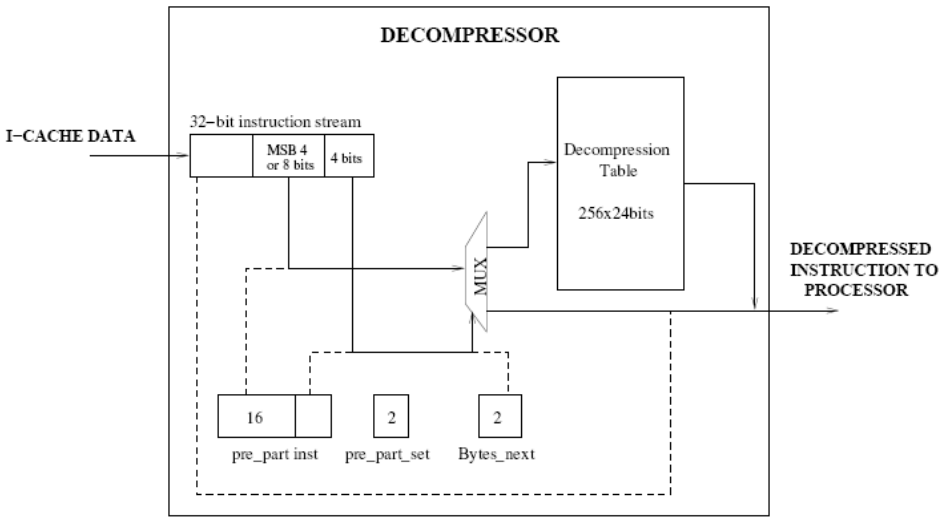

Figura 2.12 – Diagrama de blocos da arquitetura do hardware descompressor proposto por LEKATSAS *et al* [22]

Para a realização dos testes, os autores escolheram várias aplicações para demonstrar as vantagens da arquitetura proposta para o hardware descompressor mostrado em [22]. Esse é o primeiro protótipo que executa a descompressão de instruções em apenas um ciclo e sem nenhum atraso em relação ao tempo de execução da descompressão. Os resultados obtidos nos testes demonstraram que houve um ganho médio de desempenho de 25% e a diminuição do tamanho do código na média de 35%. LEKATSAS *et al*, afirmam que essa tecnologia desenvolvida não está limitada em apenas um processador, mais sim pode ser aplicada e obter resultados similares há outros processadores.

BENINI *et al* [4, 5] propuseram um método de compressão baseado em dicionário de código (256 palavras de 32 *bits*), utilizando como plataforma alvo um processador DLX que se assemelha muito com vários processadores comerciais da família ARM (*Advanced RISC Machine*) e MIPS. As instruções são comprimidas em grupos que tem o tamanho de uma linha da *cache* (128 *bits*) e a sua descompressão ocorre no instante em que são extraídas da *cache*. Este método destaca-se na construção do dicionário de códigos que será formado com as instruções mais executadas no programa.

A unidade usada para a compressão é a linha da *cache*. Porém, em cada linha da *cache* existe 4 instruções originais ou senão um conjunto de instruções comprimidas e possivelmente intercaladas com outras não comprimidas, prefixado por uma palavra de 32 *bits*. A palavra não comprimida tem um posicionamento fixo na linha da *cache* e serve para diferenciar uma linha de *cache* com instruções comprimidas das outras linhas com as instruções originais. De fato, uma linha de *cache* comprimida não contém necessariamente todas as instruções comprimidas, mas sempre deve ter um número entre 5 e 12 instruções comprimidas na linha da *cache* para ser vantajoso o uso da compressão, [5]. A estrutura detalhada de uma linha comprimida de *cache* é mostrada na Figura 2.13.

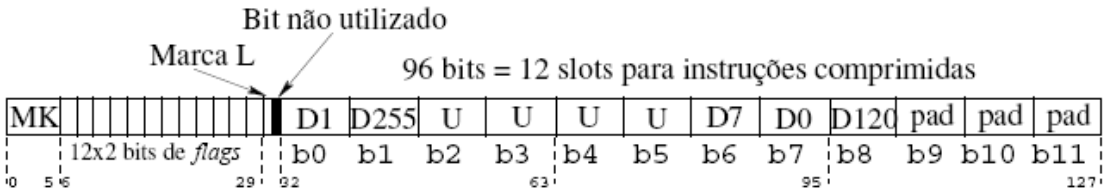

Figura 2.13 – Estrutura da linha comprimida sugerido por BENINI *et al* [4]

Para evitar o uso das tabelas de tradução de endereços, BENINI *et al* [4] exigiram que os endereços de destino estejam sempre alinhados a 32 *bits* (palavra), realizando *patching* para conversão dos *offsets* de saltos. A primeira palavra (32 *bits*) da linha de *cache* contém uma marca L e um conjunto de *bits* de *flags*. A marca é um *opcode* de instrução não utilizada, ou seja, um *opcode* inválido que sinaliza uma linha comprimida (no processador DLX os *opcodes* são de 6 *bits*). Segue-se um conjunto de 12 *bits* de *flags*, que forma um subconjunto com pares de 2 *bits* para sinalizar se o restante, ou seja, os 3 *bytes* correspondentes (b0 a b11) contém instruções comprimidas ou não.

Os valores do conjunto dos *bits* de *flags* são atribuídos da seguinte forma [4]:

- 00_2 se o *byte* correspondente contém uma instrução comprimida;
- 01_2 se o *byte* correspondente contém 8 *bits*, ou seja, parte de uma instrução não comprimida;
- 11_2 se o *byte* correspondente é deixado vazio por motivo do alinhamento de instruções;
- 10_2 serve para assinalar a última instrução comprimida da linha;
- O *bit* de *flag* L indica se a linha é comprimida (L = 0) ou não (L = 1).

Conforme BENINI *et al* [4], o algoritmo de compressão analisa o código seqüencialmente a partir da primeira instrução e tenta acondicionar instruções adjacentes em linhas comprimidas. O procedimento da compressão garante que o código comprimido nunca seja maior do que o código original. Além disso, o número dos *bits* transferidos da memória para a *cache* ao ser executada a compressão do código, nunca seja maior do que para o caso não comprimido.

Um detalhe adicional é que nenhuma instrução pode ultrapassar o limite da linha de *cache*. E ainda, nem todas as instruções que pertencem ao dicionário estarão representadas de forma comprimida no código, dado que a compressão depende da sua vizinhança e da viabilidade de formação de uma linha comprimida.

O hardware de descompressão proposto em [4] é executado *on-the-fly* e está embutido no controlador principal. A Figura 2.14 mostra à arquitetura do hardware descompressor proposto por BENINI *et al* [4].

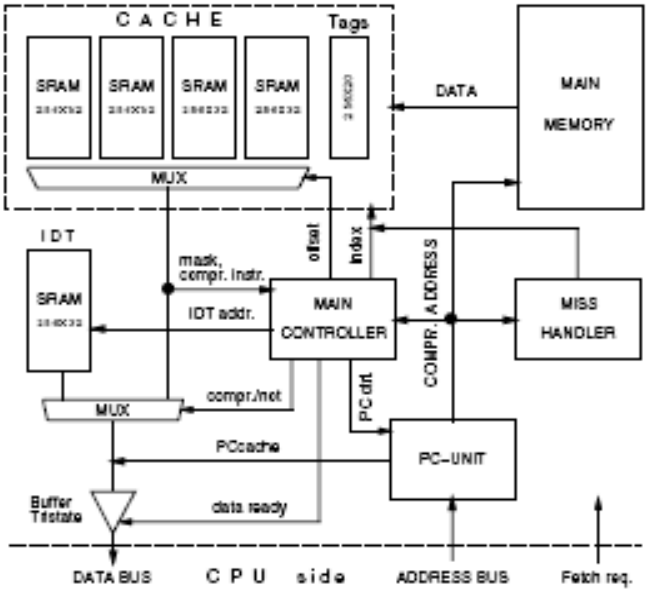

Figura 2.14 – Arquitetura do hardware descompressor sugerido por BENINI *et al* [4]

O hardware descompressor contém os seguintes blocos [4]:

- **Controlador Principal** (*Main Controller*): é o controlador já conhecido em *caches*, responsável pelas analisas das *tags* da *cache* para verificar se houve acerto

ou falha e fornecer a instrução correta ao processador. O *Main Controller* também é conhecido como Controlador Mestre;

- **Matriz de *Cache*** (*CACHE* e *Tags*): é a memória de 256 linhas x 32 *bits*, ou seja, 4 palavras de *cache* (256x32) mais uma *tag* de 20 *bits*;
- **Dicionário de Instrução** (*IDT*): contém o dicionário de instrução comprimida e é uma memória de 256 linha e 4 palavras de *cache* (256x32);
- **MUX de *Cache***: é colocado na saída da Matriz de *Cache*;
- **MUX**: seleciona entre a saída do dicionário de instrução (*IDT*) onde estão as instruções comprimidas e a saída do MUX de *Cache* onde as instruções não estão comprimidas;
- **Lógica de Atualização do PC** (*PC-UNIT*): está unidade contém um controlador de 4 *bits* que permite saber qual é a instrução que está sendo descomprimida da linha de *cache* comprimida.

BENINI *et al* [4] ainda constataram que há uma perda de sincronismo entre o *PC* do processador central com a posição da instrução em execução, portanto, foi desenvolvido um hardware adicional de modo que o *PC* do processador seja congelado enquanto múltiplas instruções pertencentes a uma linha comprimida da *cache* sejam descomprimidas, e assim manteve-se a sincronização do *PC* do processador com a posição da instrução em execução.

No método proposto foram realizados vários experimentos no qual resultou uma diminuição no tamanho do código de 28% em média e uma economia média no consumo de energia de 30%.

LEFURGY *et al* [21] propuseram uma técnica de compressão de código baseado na codificação do programa usando um dicionário de códigos. A Figura 2.15 mostra a arquitetura desta abordagem.

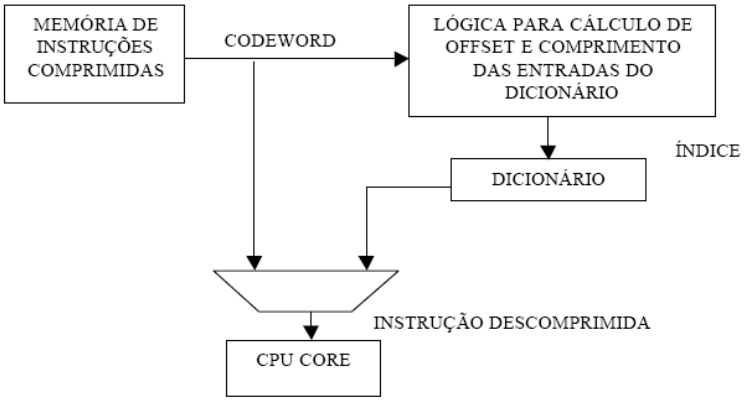

Figura 2.15 – Arquitetura do descompressor sugerido por LEFURGY *et al* [21]

Usando esta técnica, a compressão é realizada após a compilação do código fonte, porém, o código objeto é analisado e as seqüências comuns de instruções são substituídas por uma palavra codificada (*codewords*), como na compressão de texto. Apenas as instruções mais freqüentes são comprimidas. Um *bit* (e*scape bit*) é utilizado para distinguir uma instrução comprimida (codificada) de uma instrução não comprimida. As instruções correspondentes às instruções comprimidas são armazenadas em um dicionário no hardware de descompressão. As instruções comprimidas são usadas para indexar as entradas do dicionário. O código final consiste de *codewords* misturadas com instruções não comprimidas. A Figura 2.16 exemplifica o mecanismo de compressão e indexação no dicionário.

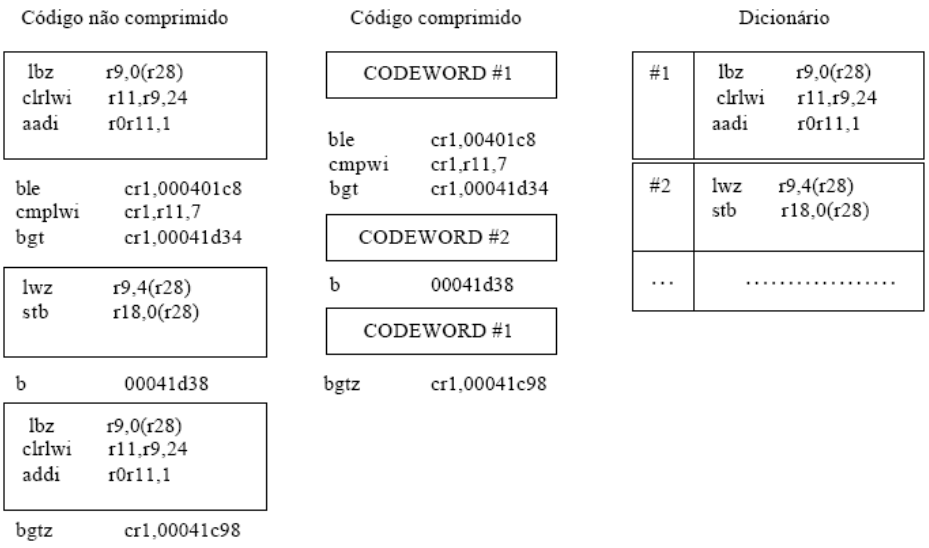

Figura 2.16 – Esquema de compressão de LEFURGY *et al* [21]

Observa-se que um dos problemas mais comuns encontrados na compressão de código se refere à determinação dos endereços alvo das instruções de salto. Normalmente este tipo de instrução (desvio direto) não é codificado para evitar a necessidade de reescrever as palavras de códigos que representam estas instruções [18]. Já os desvios indiretos podem ser codificados normalmente, pois, como seus endereços alvos estão armazenados em registradores, apenas as palavras de códigos necessitam ser rescritas. Neste caso, é necessário apenas uma tabela para mapear os endereços originais armazenados no registrador para os novos endereços comprimidos.

Este método diverge dos demais métodos vistos na literatura no sentido que, os endereços alvos estejam sempre alinhados a 4 *bits* (tamanho de um *codeword*) e não ao tamanho da palavra do processador (32 *bits*). Como vantagem destaca-se uma melhor compressão; mas como desvantagem verifica-se a necessidade de alterações no *core* do processador (um hardware extra) para tratar desvios para endereços alinhados a 4 *bits*.

Em [21], não fica claro os detalhes sobre a interação do hardware descompressor da Figura 2.15 com os processadores experimentados (PowerPC, ARM e i386). O funcionamento do hardware descompressor é realizado basicamente da seguinte maneira: A instrução é buscada da memória, caso seja um *codeword*, a lógica de decodificação dos *codewords* obtém o deslocamento e o 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 repassadas diretamente ao processador.

Com o método proposto em [21], foram obtidas taxas de compressões de 61% para o processador PowerPC, 66% para o processador ARM e 75% para o processador i386. As métricas de desempenho e consumo de energia não foram expressas por LEFURGY *et al* [21].

2.4. Comparação das Técnicas de Compressão

A sumarização da revisão realizada neste capítulo pode ser observada nas Tabelas 2.1 (Arquitetura CDM) e 2.2 (Arquitetura PDC).

Tabela 2.1 – Resumo das arquiteturas CDM

Arquitetura CDM					
Autor	**Plataforma**	***benchmarks***	**Razão de Compressão**	**Tempo de Execução**	**Redução de Energia**
Wolfe *et al* [39]	MIPS	lex, pswarp, yacc, eightq, matrix25, lloop01, xlisp, espresso e spim	73%	----	----
IBM [19, 14]	PowerPC	MediaBench, SPEC95	60%	----	----
Azevedo [2]	SPARC	SPECint95	61,4%	+ 5,89%	----
Azevedo [2]	MIPS	SPECint95	53,6%	+ 5,89%	----

Tabela 2.2 – Resumo das arquiteturas PDC

Arquitetura PDC					
Autor	**Plataforma**	***benchmarks***	**Razão de Compressão**	**Tempo de Execução**	**Redução de Energia**
Lekatsas *et al* [24]	SPARC	Compress, diesel, i3d, key, mpeg, smo, trick	65%	- 25%	- 28%
Lekatsas *et al* [22]	Xtensa-1040	Compress, diesel, i3d, key, mpeg, smo, trick	65%	- 25%	----
Benini *et al* [4]	DLX	Ptolomy	72%	----	- 30%
Lefurgy *et al* [21]	PowerPC	SPECint95	61%	----	----
Lefurgy *et al* [21]	ARM	SPECint95	66%	----	----
Lefurgy *et al* [21]	i386	SPECint95	75%	----	----

2.5. Resumo do Capítulo

Este capítulo descreveu os principais trabalhos correlatos existentes, separando entre as arquiteturas CDM e PDC. Dentro das arquiteturas foram mostrados os métodos existentes para a compressão juntamente com a arquitetura do hardware descompressor para tal método. Os resultados obtidos nos testes dos trabalhos correlatos também foram descritos neste capítulo. Mediante aos resultados definiu-se que esta dissertação desenvolverá uma nova arquitetura que realizará tanto a compressão quanto a descompressão das instruções em tempo

de execução e em hardware. O hardware de descompressão se posicionará entre o processador e a *cache*, já o hardware de compressão se posicionará entre a *cache* e a memória principal. Essa nova arquitetura será intitulada de PDCCM, descrita em VHDL e prototipada em um FPGA.

O próximo capítulo aborda aspectos da metodologia de avaliação (simulação, prototipação e modelagem) juntamente com as características específicas de cada uma delas. Para isto foi realizado uma caracterização com processadores embarcados (ARM, PowerPC e XScale) a fim de diagnosticar/verificar qual a melhor plataforma para utilizar na simulação da arquitetura PDCCM. No decorrer do capítulo observa-se também os aspectos dos *benchmark MiBench* usados para medições de desempenhos; e para finalizar, análises dos *benchmark MiBench* nas arquiteturas dos processadores embarcados (ARM, PowerPC e XScale).

CAPÍTULO 3

3. METODOLOGIA DE AVALIAÇÃO

Este capítulo mostra as diferentes possibilidades que podem ser levadas em consideração com o objetivo de analisar as metodologias de avaliação (simulação, prototipação e modelagem). Inicialmente, são apresentadas as características básicas das técnicas de avaliações mais usadas e discutidas suas vantagens e desvantagens. Portanto, para execução desta dissertação foi inicialmente utilizada a metodologia de simulação com a ferramenta SimpleScalar e posteriormente a prototipação em hardware usando FPGAs. Também ressalta-se as características do *benchmark MiBench* que são utilizados para realizar as medições de desempenho da compressão/descompressão dos códigos de instruções e a caracterização de processadores embarcados.

3.1. Técnicas de Avaliação

Em geral, o estudo e avaliação de um sistema computacional são baseados na descrição da arquitetura e na definição dos programas aplicativos que estimulam o sistema, conforme ORDOÑEZ [32]. As técnicas de avaliação existentes são [32]: técnicas de simulação, técnica de prototipação e técnica de modelagem, as quais são detalhadas a seguir.

3.1.1. Técnicas de Simulação

De acordo com [32], um simulador emula o comportamento dos programas aplicativos num ambiente específico, levando em consideração as características do sistema que determinam como ele se comportará quando submetido a um determinado programa. Características da rede de interconexão, da arquitetura do processador, do sistema de memória e até do software do sistema podem ser incluídas no ambiente de simulação. Segundo ORDOÑEZ [32], as técnicas de simulação mais usadas atualmente são: Simulação Comandada por Rastro (SCR - *Trace Driven Simulation*), Simulação Comandada por Execução (SCE - *Execution Driven Simulation*) e Simulação Comandada por Programa (SCP - *Program Driven Simulation*). Esta dissertação utilizará a técnica de simulação comandada por execução que será comentada a seguir.

O simulador controla a execução das tarefas, fazendo com que a intercalação (*interleaving*) das referências à memória seja similar à da máquina simulada (máquina paralela). Além disso, não requer o armazenamento de rastros. No caso do SimpleScalar, uma ferramenta que permite a simulação comandada por execução, só faz necessário o pré-processamento e a recompilação da aplicação depois de cada mudança no sistema de memória, ou em geral, de qualquer parâmetro da arquitetura.

Geralmente um simulador comandado por execução pode ser visto como uma ferramenta composta por duas partes principais: um gerador de eventos e um simulador do sistema alvo, conforme ORDOÑEZ [32].

O gerador de eventos modela a execução de um aplicativo com um número determinado de processadores. Quando o programa realiza uma operação de interesse (exemplo uma referência à memória), o gerador transmite o evento ao simulador do sistema alvo.

O simulador do sistema alvo modela a rede de interconexão, os módulos de memória do sistema e todos os recursos de hardware ou software que oferecem contenções ou sobrecargas, segundo [32]. De modo geral, são considerados todos os módulos de interesse a serem estudados e avaliados, determinando quais processos podem continuar em execução com base no momento em que cada evento se completa. A Figura 3.1 mostra esse processo.

Outro componente não visível nessa figura são as bibliotecas de simulação que gerenciam os eventos e os processos do programa em execução.

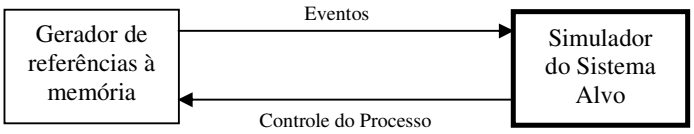

Figura 3.1 – Simulação Comandada por Execução (SCE) [32]

3.1.2. Técnicas de Prototipação

De acordo com [32], a construção de protótipos é outro mecanismo possível para avaliar o desempenho de novas arquiteturas. A sua principal vantagem é o fornecimento de dados sobre o custo de construção do sistema e detalhes do hardware.

Porém, muitas vezes sistemas digitais devem ser prototipados o mais rápido possível, seja para atender ao mercado, ou para verificação das especificações iniciais, conforme citado em CARRO [8]. Um protótipo antecipado do hardware é importante para não ocorrer os atrasos na elaboração e desenvolvimento dos softwares. Quando for necessária a utilização de um determinado circuito com alta freqüência de operação ou uma interface rápida e assíncrona com o mundo real, apenas simulação em ferramentas CAD (*Computer-Aided Design*) não é o suficiente, já que estes fatores não são simuláveis. Portanto, faz-se necessário a prototipação do hardware em um meio físico, tais como: FPGAs.

Tecnologicamente falando, existe uma vasta gama de opções para a implementação em hardware. As soluções mais interessantes para hardware dedicados são os *Gate Array* e os FPGAs [8]. Estes tipos de componentes proporcionam a integração de lógica e memória em um único circuito, acelerando a integração e a prototipação dos sistemas.

3.1.3. Técnicas de Modelagem

A técnica de modelagem também conhecida como modelo analítico, descreve as operações do sistema e os programas aplicativos em termos matemáticos. As estimativas de desempenho são obtidas mediante uma solução analítica ou numérica do modelo matemático

resultante. Esses modelos podem ser determinísticos ou probabilísticos. Os determinísticos são usados para estimar a ordem de magnitude ou limites de determinados índices de desempenho, mas não são muito empregados já que não representam a variabilidade dos programas aplicativos. Por outro lado, os modelos probabilísticos permitem a representação de certos aspectos da variabilidade, conforme ORDOÑEZ [32].

Em [32], é visto a existência de muitos trabalhos onde a avaliação de um determinado sistema é feito com modelos analíticos. São exemplos disso, uma técnica baseada em redes de Petri generalizada, modelos em redes de filas para sistemas chaveados por pacotes com rede baseada em barramento, em múltiplos barramentos, modelos para determinar o *throughput* do sistema, modelamento de protocolos de coerência, modelos aproximados na avaliação de sistemas multiprocessadores com múltiplos barramentos e etc.

Assim, nesta dissertação se realizou o estudo usando as técnicas de simulação (via ferramenta SimpleScalar) e prototipação (via FPGA).

3.2. Simulação com SimpleScalar

A ferramenta *SimpleScalar Tool Set* [7] foi desenvolvida na Universidade de *Wisconsin* (*Madison*) como parte do projeto *MultiScalar*. O SimpleScalar é um ambiente de simulação que agrupa um conjunto de ferramentas (simuladores, compiladores, *assemblers* e *linkers* para a arquitetura simulada) que suportam simulações detalhadas para algumas características encontradas nos processadores modernos.

O SimpleScalar é uma das principais ferramentas de simulação utilizada no meio acadêmico e profissional [7], as simulações das arquiteturas de processadores são focadas principalmente nas características de arquiteturas escalares e super-escalares. Esses simuladores são amplamente utilizados para a pesquisa de novos mecanismos e comportamento de arquiteturas de processadores devido à flexibilidade e possibilidade de extensão do seu código, bem como o detalhamento nos resultados. Essas mesmas características tornam a ferramenta atraente também para o ensino de arquiteturas de

processadores, sendo que esses simuladores são usados em disciplinas de diversas universidades e em diferentes enfoques.

A arquitetura implementada pelos simuladores é muito similar à do MIPS, a qual é amplamente usada no ensino de arquiteturas de computadores [17], com versões *Big-Endian* e *Little-Endian*. A semântica do conjunto de instruções do SimpleScalar é um super-conjunto do MIPS-IV ISA (*Instruction Set Architecture*), sendo chamada PISA (*Portable ISA*). Seis simuladores orientados à execução são incluídos. Além disso, a ferramenta fornece códigos binários pré-compilados (incluindo o SPEC95) e uma versão do compilador GCC (GNU *Compiler* C) modificado que compila códigos fonte FORTRAN ou C, gerando códigos binários executáveis para a arquitetura específica do simulador. Um depurador e um visualizador de *pipeline* (em modo texto) também estão disponíveis. A Figura 3.2 mostra a arquitetura geral do simulador SimpleScalar definido por BURGER & AUSTIN [7].

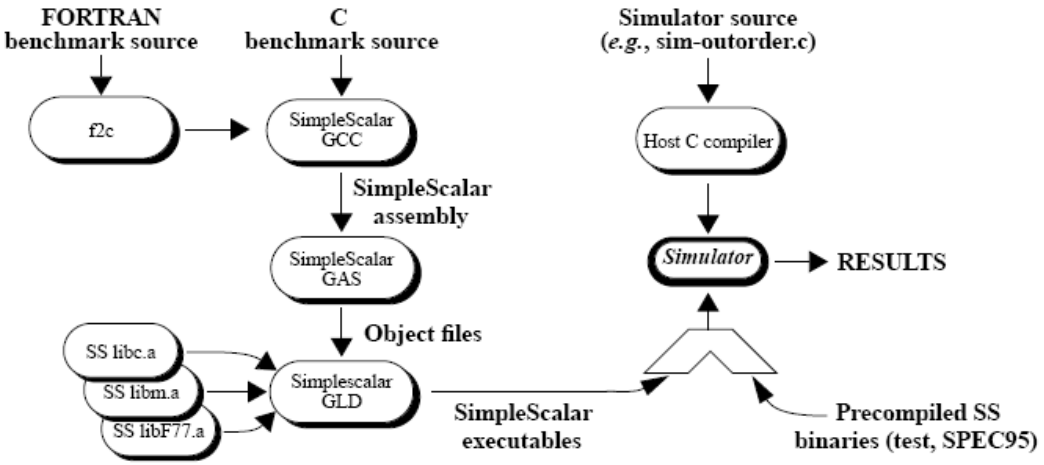

Figura 3.2 – Arquitetura do simulador SimpleScalar definida por BURGER & AUSTIN [7]

As chamadas de sistema feitas pelos binários executados em simulação são interceptadas por um tratador que decodifica a chamada, copia os argumentos e faz a chamada correspondente ao sistema operacional da máquina hospedeira, e copia os resultados da chamada para a memória do programa simulado. O SimpleScalar utiliza espaço de endereçamento de 31 *bits*.

Os seis simuladores orientados à execução inclusos na ferramenta SimpleScalar são [7]:

- ***Sim-Fast***: cada instrução é executada seqüencialmente;
- ***Sim-Safe***: cada instrução é executada seqüencialmente e é verificado o alinhamento e a permissão de acesso a cada referência de memória, o que diminui o seu desempenho;
- ***Sim-Cache***: faz a simulação de hierarquia de memória;
- ***Sim-Cheetah***: faz a simulação de hierarquia de memória juntamente com uma nova política de substituição;
- ***Sim-Profile***: gera o *profile* de cada classe de instruções (desvio, cálculo inteiro/ponto-flutuante, acesso à memória);
- ***Sim-Outorder***: modela um processador completo, incluindo simulação de ciclos e suporta execução fora de ordem, baseada na RUU (*Register Update Unit*) e em uma fila de *loads* e *stores*. Este simulador implementa um *pipeline* de seis estágios: busca, despacho, escalonamento, execução, escrita de resultados e graduação.

3.3. *Benchmark MiBench* para Medição de Desempenho

Atualmente existe uma grande variedade de *benchmarks*, tais como: *Dhrystone*, *Linpack*, *HPL* e *MediaBench*, sendo, em sua maioria, desenvolvidos para uma área específica da computação. Dentre os *benchmarks* existentes, um dos mais utilizados é o SPEC (*Standard Performance Evaluation Corporation*), que atualmente está na versão SPEC2006 [46].

O SPEC é um *benchmark* para computadores de uso geral, que agrupa um conjunto de programas e dados divididos em duas categorias: inteiro e ponto flutuante [46]. Esse *benchmark* ganhou grande popularidade no mundo comercial e acadêmico, por ser um ótimo medidor de desempenho. Essa popularidade tem influenciado o desenvolvimento de microprocessadores de uso geral, empregados em computadores pessoais e servidores, de modo que esse processadores são projetados para ter alta performance no SPEC, segundo GUTHAUS *et al* [16].

De acordo com [16], apesar do SPEC ser uma referência para processadores de propósito geral, não é uma boa opção para avaliar desempenho de microprocessadores

dedicados. A principal característica dessa classe de processadores é a diversidade de domínios de aplicação, e, além disso, não existe uma padronização para o conjunto de instruções das arquiteturas, como acontece nos processadores de uso geral, o que torna difícil a utilização da maioria dos *benchmarks*. Então, surgiram algumas pesquisas para desenvolver *benchmarks* para processadores especializados. Umas das principais é a suíte desenvolvida e conhecida como EEMBC (*EDN Embedded Microprocessor Benchmark Consortium*). Este *benchmark* reúne vários domínios de aplicação e disponibiliza um conjunto de *benchmarks* para avaliar cinco domínios de processadores embarcados.

Neste contexto, GUTHAUS *et al* [16], apresentam o *MiBench*, um *benchmark* desenvolvido para microprocessadores de uso específico, que possui trinta e cinco aplicações divididas em seis domínios de aplicações (categorias): *Automotive and Industruial Control*, *Consumer Devices*, *Office Automation*, *Networking*, *Security* e *Telecommunications*.

O *MiBench* é um *benchmark* livre, sendo basicamente implementado utilizando a linguagem de programação C. Todos os programas que compõem o *MiBench* estão disponíveis em código fonte, sendo possível executar em qualquer arquitetura de microprocessador que possua um compilador C.

No *MiBench* são consideradas as classes de instrução: controle, aritmética inteira, aritmética em ponto flutuante e acesso à memória. Nas categorias *Telecommunications* e *Security* todas as aplicações têm mais de 50% de operações inteiras na ULA (*Arithmetic Logic Unit*). A categoria *Consumer Devices* tem mais operações de acesso à memória por causa do processamento de grandes arquivos de imagem e áudio. A categoria *Office Automation* faz uso intenso de operações de controle e acesso à memória. Os programas dessa categoria usam muitas chamadas de funções para bibliotecas de *string* para manipular os textos, o que causa o uso de várias instruções de desvio. Além disso, os textos ocupam bastante espaço na memória, o que requer muitas instruções de acesso à memória. A categoria *Automotive and Industruial Control* faz o uso intenso de todas as classes de instrução devido a sua aplicabilidade e por fim, a categoria *Networking* faz muito uso de operações aritméticas de inteiros, pontos flutuantes e acesso à memória, devido as tabelas de IPs para roteamento em rede, ordenação de dados e outros. Em comparação, o *benchmark* SPEC apresenta, aproximadamente, a mesma distribuição de operações para todas as categorias.

A Tabela 3.1 apresenta o número de instruções executadas por cada aplicação contida no *MiBench*, para cada conjunto de dados. Os conjuntos de dados padrão são divididos em duas classes para cada categoria do *benchmark*: conjunto de dados reduzido e conjunto de dados expandido. O primeiro representa o uso moderado da aplicação do *benchmark*, enquanto que o segundo representa a sobrecarga da aplicação.

Tabela 3.1 – Número de instruções executadas para cada aplicação contida no *MiBench* [16]

Aplicação	Conjunto Reduzido	Conjunto Expandido	Aplicação	Conjunto Reduzido	Conjunto Expandido
basicmath	65.459.080	1.000.000.000	ispell	8.378.832	640.420.106
bitcount	49.671.043	384.803.644	rsynth	57.872.434	85.005.687
quicksort	43.604.903	595.400.120	stringsearch	158.646	38.960.051
susan.corners	1.062.891	586.076.156	blowfish.decode	52.400.008	737.920.623
susan.edges	1.836.965	732.517.639	blowfish.encode	42.407.674	246.770.499
susan.smoothing	24.897.492	1000.000.000	pgp.decode	85.006.293	259.293.845
jpeg.decode	6.677.595	990.912.065	pgp.encode	38.960.650	824.946.344
jpeg.encode	28.108.471	543.976.667	rijndael.decode	23.706.832	140.889.705
lame	175.190.457	544.057.733	rijndael.encode	3.679.378	24.910.267
mad	25.501.771	272.657.564	sha	13.541.298	20.652.916
tiff2bw	34.003.565	697.493.266	crc32	52.839.894	61.659.073
tiff2rgba	36.948.939	1.000.000.000	fft.inverse	65.667.015	377.253.252
tiffdither	273.926.642	1.000.000.000	fft	52.625.918	143.263.412
tiffmedian	141.333.005	817.729.663	adpcm.decode	30.159.188	151.699.690
typeset	23.395.912	84.170.256	adpcm.encode	37.692.050	832.956.169
dijkstra	64.927.863	272.657.564	gsm.decode	23.868.371	548.023.092
patricia	103.923.656	1.000.000.000	gsm.encode	55.361.308	472.171.446
ghostscript	286.770.117	673.391.179			

As seis categorias do *MiBench* oferecem estruturas de programas que permitem explorar as funcionalidades do *benchmark* e do compilador para se ajustarem melhor a um determinado microprocessador.

Os *benchmark* utilizados nas simulações nesta dissertação são do pacote *MiBench*, específicos para sistemas embarcados e de categorias diferentes, sendo [16]:

- **Dijkstra** (*Networking*): é um algoritmo que calcula o menor caminho em um grafo;
- **Patrícia** (*Networking*): é utilizado para representar tabelas de roteamento em aplicações de redes de computadores;
- **JPEG** (*Consumer Devices*): é um compressor e descompressor de imagens com o formato JPEG (*Joint Photographic Experts Group*);
- **Lame** (*Consumer Devices*): é um codificador de MP3 que suporta a codificação da razão de *bits* constante, médio e variável;

- **Mad** (*Consumer Devices*): é um decodificador de áudio (MPEG - *Moving Picture Experts Group*) de alta qualidade;
- **Tiffmedian** (*Consumer Devices*): é um algoritmo usado para reduzir o número de cores em uma imagem;
- **Tiffrgba** (*Consumer Devices*): é um algoritmo que converte uma imagem colorida no formato TIFF (*Tagged Image File Format*) em TIFF RGB (*Red Green Blue*);
- **Bitcount** (*Automotive and Industruial Control*): é um algoritmo que testa a habilidade de manipulação de *bits* do processador, contando o número de *bits* em um vetor de inteiros;
- **QuickSort** (*Automotive and Industruial Control*): é um algoritmo que faz ordenação de dados;
- **Susan** (*Automotive and Industruial Control*): é um algoritmo que faz o reconhecimento de imagens através de suas bordas;
- **Blowfish** (*Security*): é um cifrador simétrico com chave variável de 32 até 448 *bits*, é ideal para encriptação doméstica;
- **Rijndael** (*Security*): é um algoritmo que faz criptografia com chaves do tamanho de 128, 192 e 256 *bits*;
- **SHA** (*Security*): é um algoritmo que gera chaves de criptografia para troca segura de dados e assinaturas digitais;
- **ADPCM** (*Telecommunications*): *Adaptive Differential Pulse Code Modulation* é uma variação do PCM (*Pulse Code Modulation*). Uma implementação freqüente usa uma amostragem PCM de 16 *bits* e a transforma em amostragem de 4 *bits* com uma compressão de 4:1;
- **CRC32** (*Telecommunications*): é um algoritmo utilizado para detectar erros na transmissão de dados e a redundância do CRC (*Cyclic Redundancy Check*) em um arquivo;
- **FFT** (*Telecommunications*): é um algoritmo que executa a *Transformação de Fourier* usado no tratamento de sinais digitais para encontrar as freqüências em um sinal de entrada;
- **GSM** (*Telecommunications*): *Global Standard for Mobile* é um padrão para a codificação e decodificação na Europa e em vários países. Usa uma combinação de acesso múltiplo por divisão de tempo e freqüência (TDMA/FDMA) para codificar e decodificar *streams* de dados;

- **Rsynth** (*Office Automation*): é um sintetizador de texto para voz que inclui várias partes de código de domínio público em um só programa;
- **Stringsearch** (*Office Automation*): é um algoritmo que faz a busca de uma *string* em uma frase ou texto;
- **Sphinx** (*Office Automation*): é um decodificador de fala que pode trabalhar com comandos de voz ou discursos.

3.4. Caracterização Arquitetural de Processadores Embarcados

Para esta dissertação foram realizadas simulações, configuradas na ferramenta SimpleScalar, com três arquiteturas distintas para sistemas embarcados sendo elas: ARM [44], PowerPC [34] e XScale [40].

A escolha dessas três arquiteturas, dentre as diversas arquiteturas existentes para sistemas embarcados (exemplos: MIPS, MSP, FreeScale, DLX, ARM, PowerPC, XScale e outras), deu-se devido a forte aceitação no mercado, pois compreendem em arquiteturas do tipo RISC e arquiteturas com o conjunto de instruções fixas em 32 *bits* cada.

Nas configurações de entrada definidas na ferramenta de simulação SimpleScalar tentamos nos aproximar o máximo possível da sua configuração real nos quesitos: tamanho das *caches* de dados e instruções; quantidade de instruções decodificadas por ciclos; largura do barramento de memória; latência das *caches* de dados e instruções; número de ULAs em inteiros e em pontos flutuantes. A Tabela 3.2 mostra as características específicas das arquiteturas (ARM, PowerPC e XScale). Essas configurações foram obtidas através de estudos detalhados sobre as referidas arquiteturas.

Tabela 3.2 – Configuração das arquiteturas simuladas na ferramenta SimpleScalar

	Arquiteturas		
Características das arquiteturas	**ARM**	**PowerPC**	**XScale**
Cache de dados	16 Kb	16 Kb	8 Kb
Cache de instruções	16 Kb	16 Kb	8 Kb
Instruções decodificadas por ciclos	4	4	4
Executa instruções em ordem	true	false	true
Largura do barramento de memória	8	32	8

Número de remessa de instruções em um ciclo	4	4	4
Latência da *cache* de dados	1	8	1
Latência da *cache* de instruções	1	8	1
Número de ULAs de inteiros	4	1	4
Número de ULAs de pontos flutuantes	4	2	4

Na Tabela 3.3 mostra-se a quantidade de instruções mensuradas pela ferramenta SimpleScalar em cada *MiBench* nas arquiteturas ARM, PowerPC e XScale, conforme as configurações arquiteturais da Tabela 3.2.

Tabela 3.3 – Número de instruções para as simulações

Benchmarks		**Arquiteturas**		
Categoria	***MiBench***	**ARM**	**PowerPC**	**XScale**
Networking	Dijkstra	64926066	64927640	64926291
	Patrícia	1803242	1803285	1803186
Consumer Devices	JPEG	6752716	6753142	6752758
	Lame	115177294	115177412	115177334
	Mad	91212692	91265409	91217062
	Tiffmedian	131138928	131187769	131157466
	Tiffrgba	34442181	34475555	34452835
Automotive and Industruial Control	Bitcount	66311260	66315337	66310711
	QuickSort	42206853	42207818	42207260
	Susan	29776135	29776151	29776052
Security	Blowfish	46491591	46491770	46491640
	Rijndael	101430045	101435250	101431053
	SHA	55116591	55119044	55116611
Telecommunications	ADPCM	37694023	37705658	37696596
	CRC32	15422562	15423082	15422381
	FFT	31400015	31399892	31399828
	GSM	23865393	23868177	23865946
Office Automation	Rsynth	54984320	54984717	54984442
	Stringsearch	3677500	3677703	3677484
	Sphinx	35354798	35360183	35354837

O que se observa na Tabela 3.3 é que nas três arquiteturas simuladas, a categoria de aplicações *Consumer Device* apresentou a maior média no número de instruções para a simulação ficando em torno de 75.756.037 instruções emitidas de seus *benchmarks* para as simulações. Isto deve-se ao fato de que essa categoria faz muitos acessos à memória ao processar arquivos grandes de imagem e áudio. Já a categoria *Telecommunications* foi a que

obteve a menor média no número de instruções para as simulações, tendo sido emitidas aproximadamente 27.096.963 instruções dos seus *benchmarks* para essa categoria, sendo 64,24% menor que a categoria *Consumer Devices*, o que nos indica que a categoria de *Telecommunications* faz mais acesso em operações de ponto inteiro na ULA.

A categoria *Office Automation* apresentou uma média de 31.339.554 instruções emitidas para a simulação dos seus *benchmarks*, enquanto que para a categoria *Networking* a média foi de 33.364.952 instruções emitidas para a simulação dos seus *benchmarks*. A categoria *Automotive and Industruial Control* proporcionou a emissão de 39,15% instruções a menos em relação à categoria *Consumer Devices*, ou seja, uma média de 46.098.620 instruções emitidas para os seus *benchmarks.* E por fim a categoria *Security* emitiu 67.680.399 instruções para a simulação dos seus *benchmarks*.

A Figura 3.3 mostra o gráfico com o tempo de simulação em ciclos do processador para todos os programas do *MiBench* nas três arquiteturas simuladas (ARM, PowerPC e XScale).

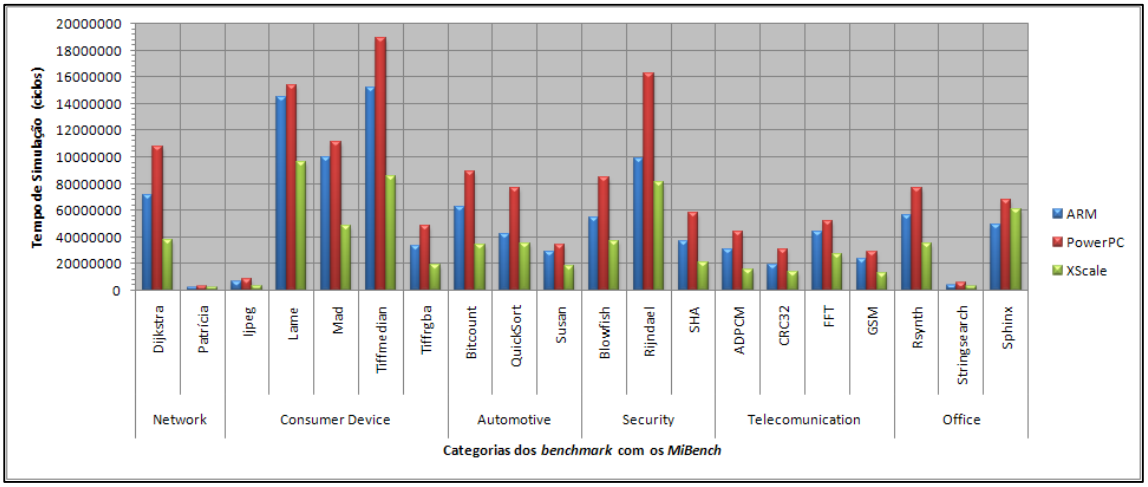

Figura 3.3 – Tempo de simulação em ciclos

Observa-se que o tempo de simulação foi maior com a arquitetura PowerPC e a categoria *Consumer Devices* apresentou o maior tempo de simulação, sendo o *benchmark* Tiffmedian o que teve sua simulação executada em maior tempo; ainda vale ressaltar que observando a Tabela 3.3 esse *benchmark* (Tiffmedian) gerou o maior número de instruções

nas três plataformas simuladas, pois se trata de um algoritmo que trabalha com cores nas imagens.

A Figura 3.4 mostra o gráfico da emissão de ciclos por instrução (CPI) nas arquiteturas simuladas.

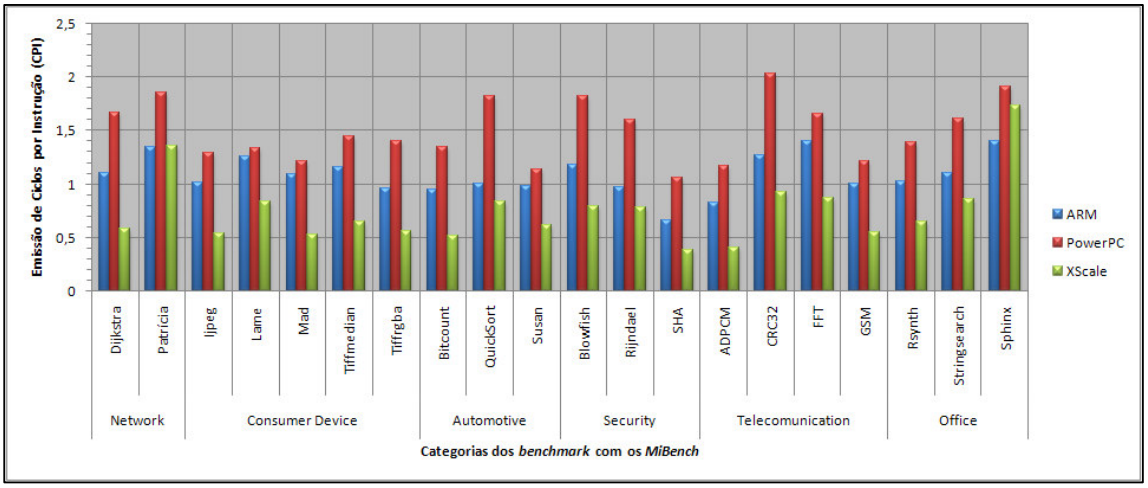

Figura 3.4 – Emissões de ciclos por instruções (CPI)

Conforme a Figura 3.4 nota-se que a arquitetura XScale apresentou melhor performance na emissão de ciclos por instrução (em média 0,75), ou seja, são necessários menos ciclos de processador em relação às outras duas arquiteturas embarcadas; no ARM a média foi de 1,08 e o PowerPC obteve uma média de 1,49, para a execução das instruções dos programas dos *MiBench* analisados.

Já a categoria *Consumer Devices* precisou de menos ciclos para a execução das instruções ficando em média 1,02 ciclos por instrução. No entanto, a categoria *Networking* apresentou o maior número de ciclos por instrução em média 1,31, este fato pode ser justificado pelo uso em operações aritméticas de inteiros, pontos flutuantes e acesso a memória. O *benchmark* SHA, para as três arquiteturas analisadas, foi o que apresentou a melhor performance entre os demais *benchmarks* utilizados para as análises.

A Figura 3.5 mostra o gráfico com a taxa de perdas nas buscas da *cache* de dados e a Figura 3.6 mostra o gráfico com a taxa de perdas nas buscas da *cache* de instruções. São visíveis as diferenças incididas com a taxa de perda nas *caches* de dados e instruções ocorrida

nos *MiBench*; a categoria *Telecommunications* apresentou uma melhor performance nas três arquiteturas analisadas (ARM, PowerPC e XScale), tanto para as perdas na *cache* de dados (em média 0,19) quanto para as instruções (em média 0,03).

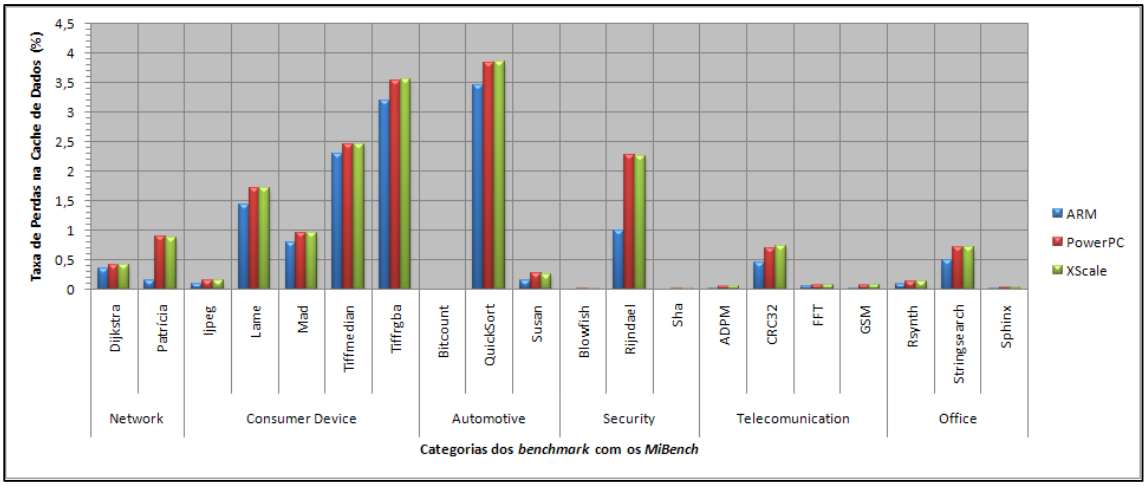

Figura 3.5 – Percentual de perdas na *cache* de dados

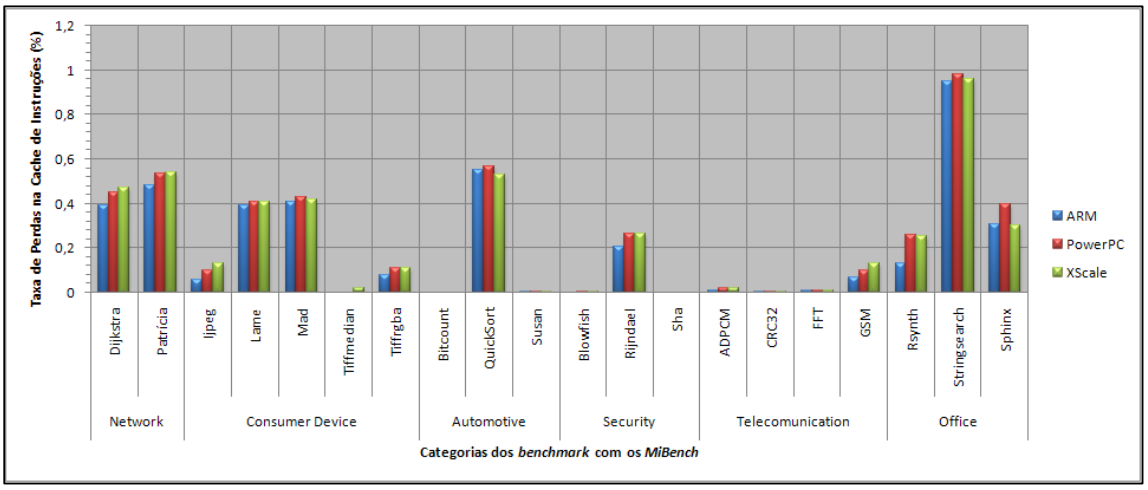

Figura 3.6 – Percentual de perdas na *cache* de instruções

Contudo, nota-se que de acordo com a configuração das arquiteturas dos processadores embarcados, ou seja, com os dados da Tabela 3.2, a arquitetura ARM apresenta *caches* de dados e instruções igual da arquitetura PowerPC e maior que da arquitetura XScale. Então, justifica-se o seu melhor desempenho computacional nas simulações realizadas com os *benchmark MiBench* em relação às outras duas arquiteturas também simuladas.

Para a perda na *cache* de dados (Figura 3.5) o *MiBench* QuickSort foi o que apresentou o pior desempenho (com taxa média de perda de 3,71 instruções emitidas pelo processador), pelo fato desse *benchmark* ser um algoritmo que faz muitas manipulações com os dados por ele analisados. Quanto a perda na *cache* de instrução (Figura 3.6), o *benchmark* Stringsearch foi o que apresentou o maior número de perdas de instruções na *I-cache* (com taxa média de perda de 0,96 instruções emitidas pelo processador).

Ressalta-se que a arquitetura ARM apresentou algumas vantagens em relação as arquiteturas PowerPC e XScale, tais como: uma maior aceitação e utilização no mercado, *caches* de dados e instruções maior/igual, execução de instruções em ordem, número de ULAs de inteiros e pontos flutuantes maiores e outras. Então, devido às *caches* serem maiores ocorreram menos acessos à memória principal, o que causou uma redução no tráfego dos dados e, como conseqüência, um melhor desempenho da arquitetura que nos permite compreender que os sistemas embarcados possuem características de alto desempenho. Assim, conforme as vantagens apresentadas foi escolhido o processador embarcado ARM para utilizar nas simulações da arquitetura PDCCM.

Buscando uma análise mais detalhada sobre os impactos da compressão e descompressão dos códigos de instruções, foram realizados testes de compressão dos *benchmark MiBench* usando o método de *Huffman* (implementado em software). A Figura 3.7 mostra a razão de compressão dos *MiBench* analisados.

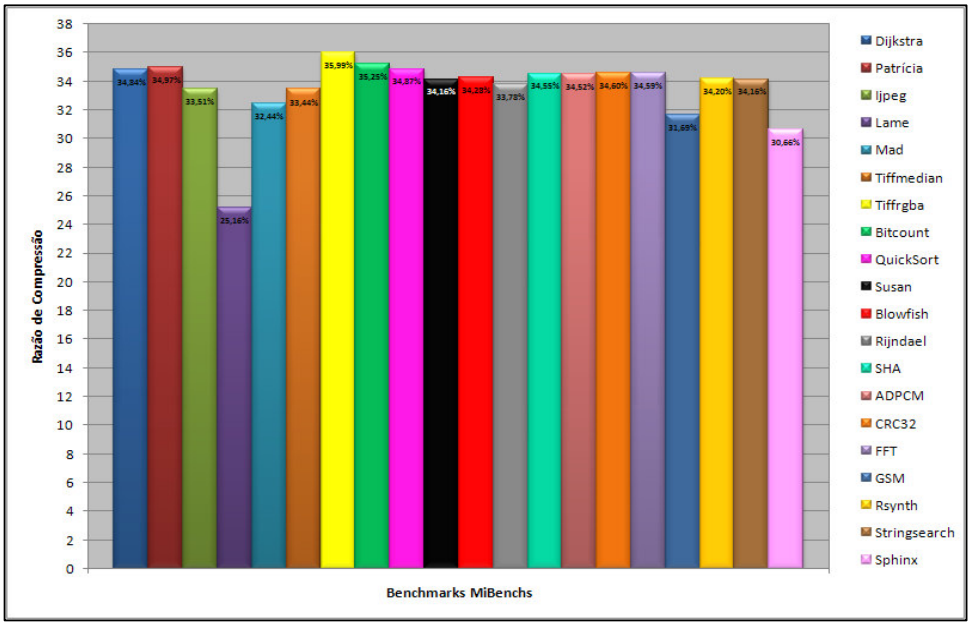

Figura 3.7 – Razão de compressão dos *MiBench* analisados

O método de *Huffman* implementado em software, foi o método tradicional encontrado na maioria das literaturas. Portanto, o método realizou uma compressão média de 33,58% no tamanho do código dos *benchmarks MiBench* analisados. A Figura 3.8 mostra os tamanhos dos *benchmark MiBench* não comprimidos e após suas compressões através do método de *Huffman*.

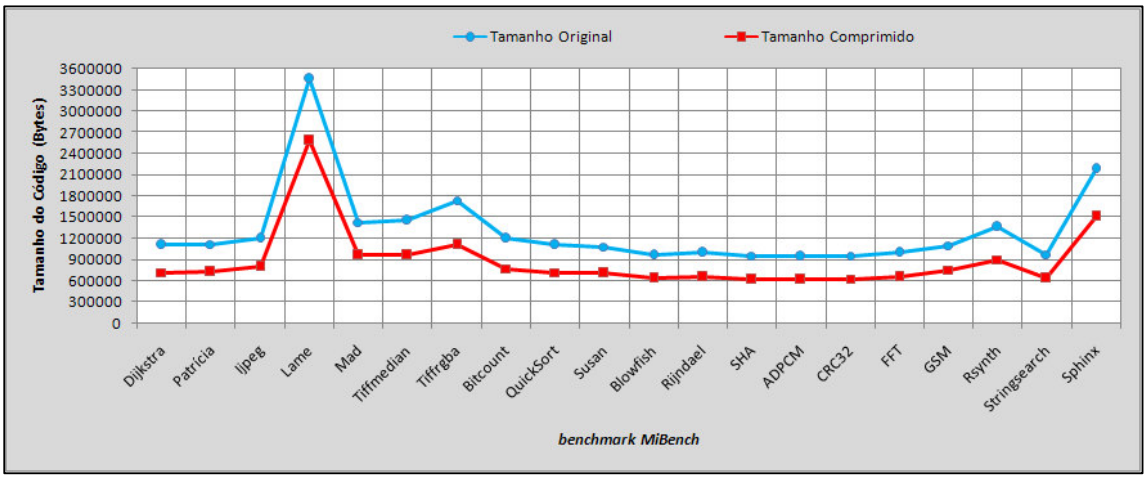

Figura 3.8 – Tamanho dos *benchmark MiBench* não comprimidos e comprimidos

3.5. Prototipação em Hardware usando FPGAs

Muitos sistemas computacionais exigem a integração de todos os componentes de hardware em um único circuito integrado, por razão de desempenho e precisão (controle de míssil, por exemplo) ou de potência (exemplo: equipamentos biomédicos) ou ambos (computadores, palmtop, telefones celulares), de acordo com CARRO [8]. Portanto, nesta seção será estudado conceitos de FPGA, sua emulação em hardware, a programação dos FPGAs e por fim a prototipação de hardware usando FPGAs.

3.5.1. O que é FPGA

FPGA é um circuito integrado digital que contém uma estrutura regular de células configuráveis e de interligações programáveis pelo usuário final, e que pode ser usado para prototipar sistemas de propósitos gerais ou sistemas embarcados, segundo COSTA [9]. Os

FPGAs são limitados apenas pelo número de células e ligações disponíveis. Através desse componente (FPGA) pode-se implementar um sistema embarcado praticamente em tempo de simulação [8]. A Figura 3.9 apresenta a estrutura básica de um FPGA e a Figura 3.10 mostra a arquitetura interna do FPGA.

Conforme ERCEGOVAC *et al* [13], FPGA é um hardware que pode ser programado para implementar um sistema digital que consiste em dezenas de milhares de portas lógicas, permitindo a realização de sistemas complexos em um único chip.

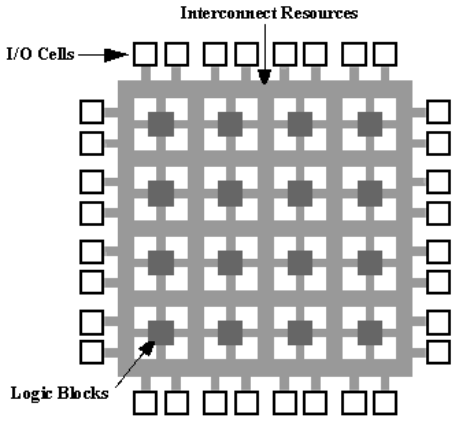

Figura 3.9 – Estrutura básica de um FPGA [35]

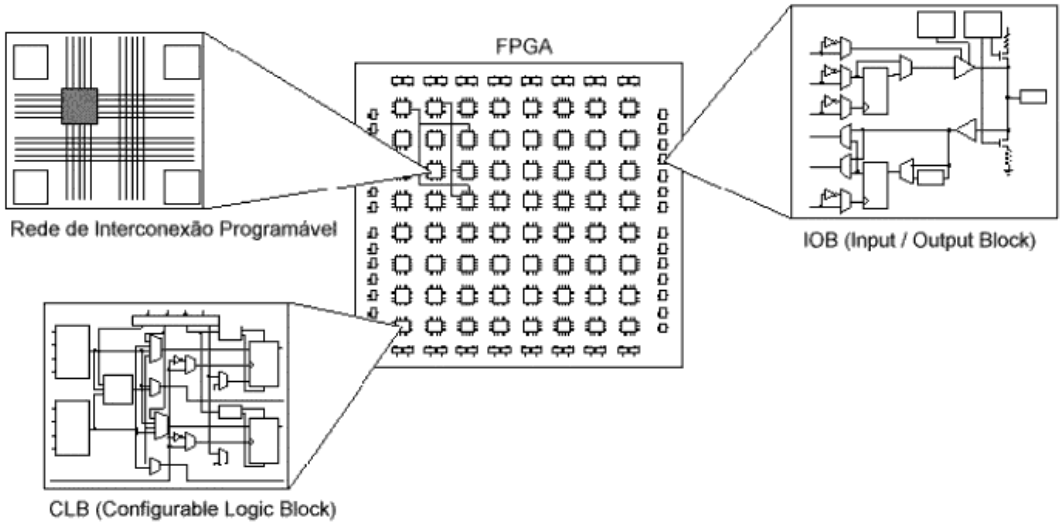

Figura 3.10 – Arquitetura interna do FPGA [9]

A estrutura básica de um FPGA, citada por ORDOÑEZ *et al* [33], pode variar de fabricante, família ou até em uma mesma família pode existir variações. As Figuras 3.11, 3.12 e 3.13 apresentam estruturas básicas dos FPGAs dos fabricantes ALTERA [43], XILINX [47] e ACTEL [42] respectivamente. O fundamental é que exista sempre os seguintes recursos nos FPGAs [35]:

- **Bloco Lógico Configurável** (**CLB**): é a unidade lógica de um FPGA;
- **Bloco de Entradas e Saídas** (**IOB**): localizado nas periferias dos FPGA, são responsáveis pela interface com o ambiente;
- **Caixa de Conexão** (**SB**): responsável pela interconexão entre os CLBs, através dos canais de roteamento.

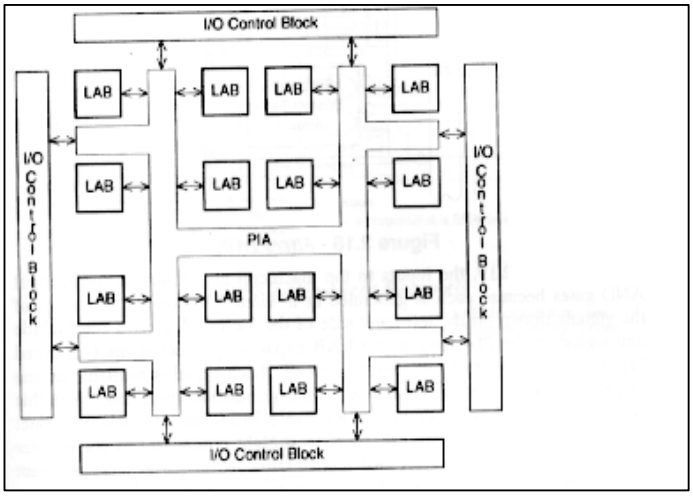

Figura 3.11 – Estrutura geral do FPGA da Altera [35]

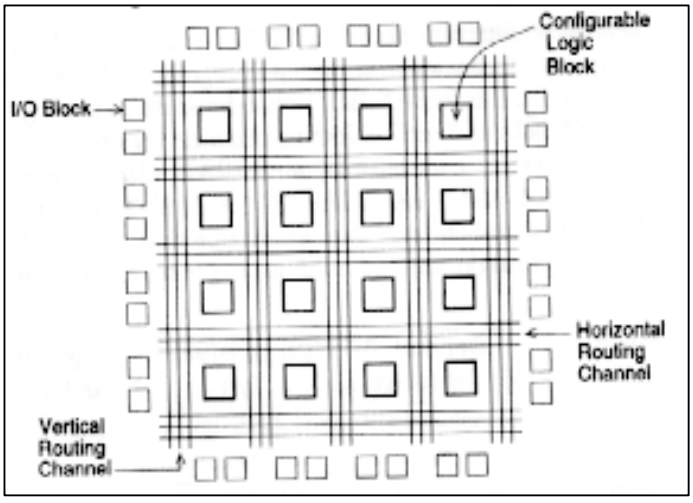

Figura 3.12 – Estrutura geral do FPGA da Xilinx [35]

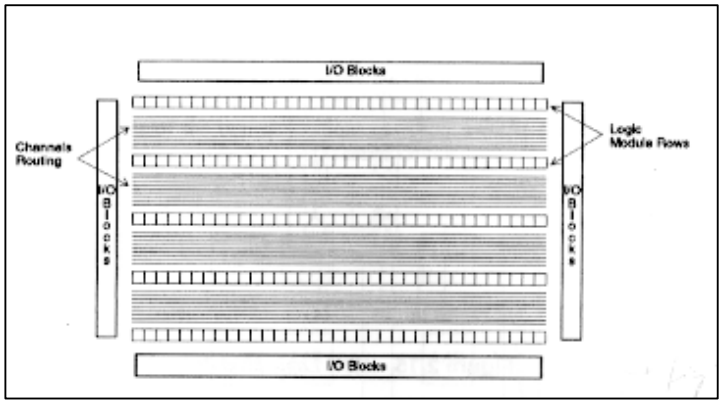

Figura 3.13 – Estrutura geral do FPGA da Actel [35]

Pelo fato de poderem ser programáveis pelo usuário final, os FPGAs trouxeram ao desenvolvimento de hardware quase a mesma agilidade do mercado de software, ou seja, sobre a base fixa (microprocessador ou FPGA) pode-se personalizar o sistema para uma aplicação específica (através de um programa ou pela programação da conexão entre portas), conforme citado em [8].

Devido aos custos que envolvem a elaboração e desenvolvimento de hardware, além da tendência de sempre ter de alterar o projeto a posteriori por modificações de especificações ou até mesmo, necessidades de novos projetos, a opção pelo uso de um FPGA como tecnologia alvo de prototipação para hardware é comum. As Figuras 3.14 e 3.15 exemplificam FPGAs da Altera® e Xilinx® respectivamente.

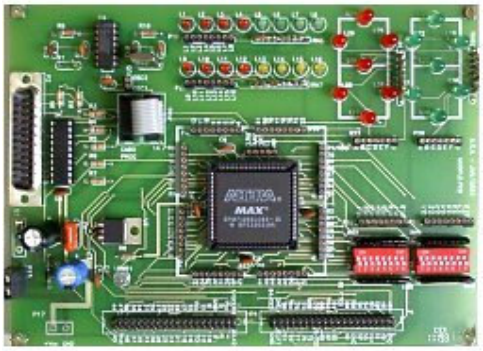

Figura 3.14 – Exemplo de FPGA da ALTERA® [43]

Figura 3.15 – Exemplo de FPGA da XILINX® [47]

3.5.2. FPGA e Emulação de Hardware

Pode-se utilizar um FPGA ou um conjunto destes para prototipar não somente um único sistema, mas sim um ambiente completo. Por exemplo, a prototipação completa de um processador da AMD (*Advanced Micro Devices*), executando o sistema operacional Windows. Portanto, existente a hipótese de se colocarem pequenos e médios sistemas dentro de um único FPGA, conforme citado em [8].

Com a existência de modernos FPGAs que disponibilizam memória não é descartada a hipótese de emular hardwares tão complexos como processadores. Porém, os FPGAs colaboram para o rápido desenvolvimento do hardware final, conforme CARRO [8].

MARTINS *et al* [26] afirmam que os FPGAs possuem desempenho de processamento equivalente ou superior a circuitos integrados (CIs) de propósito geral, após desenvolvido um programa que represente a lógica funcional do circuito geral, bastará programar o FPGA e este sendo um chip único, resulta em hardware menor, com maior capacidade de processamento, menor consumo de energia e direcionado a resolver problemas específicos.

3.5.3. Programação para FPGA

Os FPGAs podem ser utilizados em diferentes funções de um projeto de hardware, segundo [8]. Por exemplo, como um circuito em si, um controlador de barramento, como uma função de hardware a ser executada em paralelo com o processador para acelerar um certo algoritmo e outros.

De acordo com [26], dentre os principais fabricantes de FPGAs destacam-se a ALTERA, XILINX, ACTEL, QUICKLOGIC, ALGOTRONIX e ATMEL. Cada fabricante possui várias famílias de dispositivos. Exemplos de famílias: Clássica, MAX 5000, MAX 7000, MAX 9000, FLEX 8000, FLEX 10K, Cyclone (I, II e III) sendo todos do fabricante ALTERA® e as famílias XC4010XL, XC4000XL, XC4010E, Virtex todos do fabricante XILINX®, onde cada qual dispositivo (FPGA) é voltado para tipos diferentes de aplicações, colocando à disposição do usuário vários métodos de programação pelo protocolo IEEE

(*Institute of Electrical and Electronics Engineers*) 1149.1. Por este protocolo serial pode-se não somente programar o circuito desejado, mas efetuar testes sobre a programação e o comportamento do mesmo.

3.5.4. Prototipação com FPGA

A utilização do FPGA na indústria tem um amplo horizonte, conforme citado em [26], podendo-se implementar desde circuitos básicos como um rádio relógio até circuitos complexos como uma CPU de alto desempenho. O Quadro 3.1 apresenta possíveis hardwares prototipados com FPGAs.

Quadro 3.1 – Possíveis hardwares prototipados com FPGAs [26]

Áreas de Aplicações	Exemplos de hardware prototipado usando FPGAs
Consumo	vídeo games, sistemas de karaokê e decodificador de áudio digital.
Transportes	sistemas de estradas de ferro, controle de tráfego e semáforos.
Industrial	equipamentos de testes e medidas, equipamentos médicos, controle remoto, robótica, emulador ASIC e sistema de visão.
Comunicação de Dados	multiplexadores, roteadores, vídeo conferência, criptografia, conectores, modems, compressão de dados, LANs, FDDI e wireless.
Telecomunicações	interfaces de fibra óptica, controlador de voice-mail, equipamentos de PABX, multiplexadores e compressores de dados.
Militar	sistemas de computadores e comunicação, controle de fogo.
Periféricos	controladores de discos, controladores de vídeo, fax, máquina de caixa, modems, terminais, impressoras, scanners e copiadoras.
Computadores	interface de memória, controladores de *cache*, multimídia e gráficos.

Uma área de aplicação que vem se destacando com o uso da prototipação em FPGAs é a de processadores embarcados, em que um processador é integrado a um sistema maior com objetivo de auxiliar no controle e execução de tarefas, conforme [33].

As Figuras 3.16 e 3.17, exemplificam o projeto de um circuito lógico prototipado na ferramenta CAD Quartus II da Altera®. Na Figura 3.16 é mostrado um projeto de um circuito prototipado através de digrama esquemático, pois o circuito é composto por três portas de entrada (A, B e C), uma de saída (Z) e duas primitivas lógicas (XOR). Já na Figura 3.17 é mostrada a fase de simulação em forma de ondas da prototipação do circuito projetado.

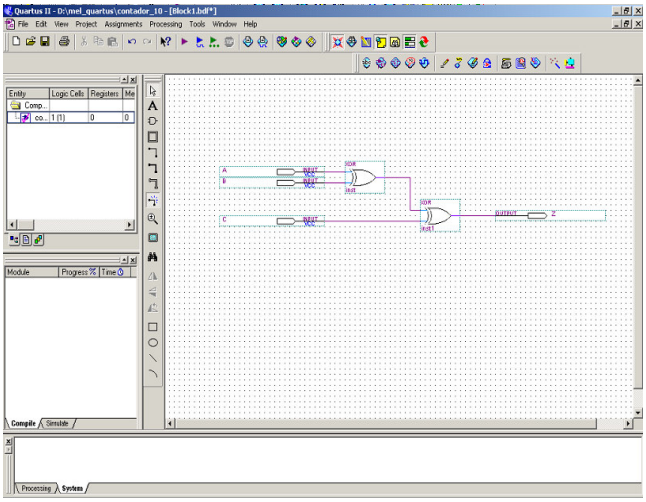

Figura 3.16 – Exemplo de circuito prototipado por diagrama esquemático

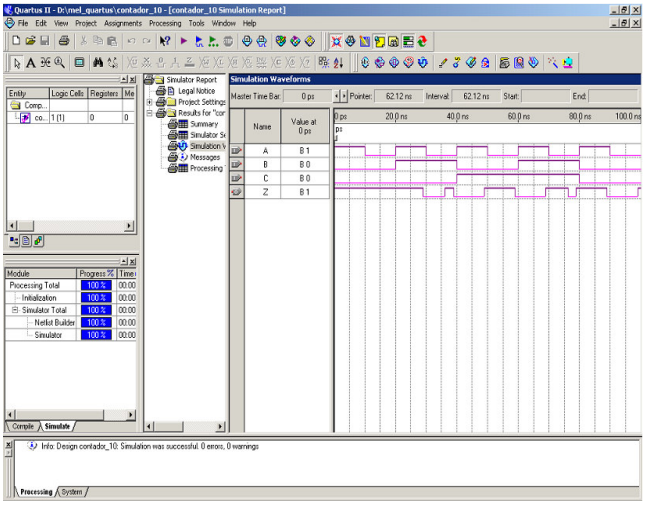

Figura 3.17 – Exemplo de simulação em forma de ondas da prototipação do circuito projetado

3.6. Resumo do Capítulo

Este capítulo descreveu as diferentes técnicas de avaliação mais usadas (simulação, prototipação e modelagem). Mas adiante foi descrito detalhadamente as duas metodologias usadas nesta dissertação sendo elas: simulação e prototipação. Ainda dando continuidade foi descrito os *benchmark MiBench* para medição de desempenho e em seguida foi realizado uma caracterização dos processadores embarcados, no qual através de simulações com a

ferramenta SimpleScalar e usando os *benchmark MiBench* conclui-se que a plataforma ARM apresentou melhores desempenhos computacionais e será utilizada na simulação da arquitetura PDCCM.

No próximo capítulo apresenta-se a nova arquitetura desenvolvida nesta dissertação juntamente com a sua variação, também será descrito e mostrado a implementação em VHDL do novo método desenvolvido *MIC* e do método de *Huffman* que fazem a compressão/descompressão dos códigos de instruções.

CAPÍTULO 4

4. PROJETO DA ARQUITETURA PDCCM DO COMPRESSOR/DESCOMPRESSOR

Neste capítulo, apresenta-se a arquitetura do compressor/descompressor desenvolvida nesta dissertação, denominada de PDCCM (Processador Descompressor Cache Compressor Memória), juntamente com o novo método *MIC* (*Middle Instruction Compression*) e o tradicional método de *Huffman*. Procurou-se estruturar o capítulo de modo a torná-lo o mais compreensível possível, explicando passo a passo cada detalhe da arquitetura, sua variação e o funcionamento do método *MIC*.

4.1. Descrição da Arquitetura

Como foi visto nos trabalhos correlatos, o desenvolvimento de arquiteturas para compressão ou descompressão de código de instrução é feito de forma separada, ou seja, na grande maioria dos trabalhos desenvolvidos só é tratado o hardware descompressor porque a compressão das instruções geralmente é feita por meio de modificações nos compiladores. Assim, a compressão é realizada em tempo de compilação e a descompressão é feita em tempo de execução usando um hardware específico.

Esta dissertação tem como grande diferencial, a implementação de uma nova arquitetura em hardware que realiza a compressão e a descompressão dos códigos das

instruções em tempo de execução. A arquitetura criada foi intitulada de PDCCM (**P**rocessador **D**escompressor **C**ache **C**ompressor **M**emória) no qual é mostrado que o hardware de compressão foi inserido entre as memórias *cache* e principal e o hardware de descompressão foi inserido entre o processador e a memória *cache*. A arquitetura PDCCM foi descrita em VHDL e prototipada em uma FPGA do fabricante ALTERA®. A Figura 4.1 mostra o modelo da arquitetura PDCCM proposta e desenvolvida nesta dissertação.

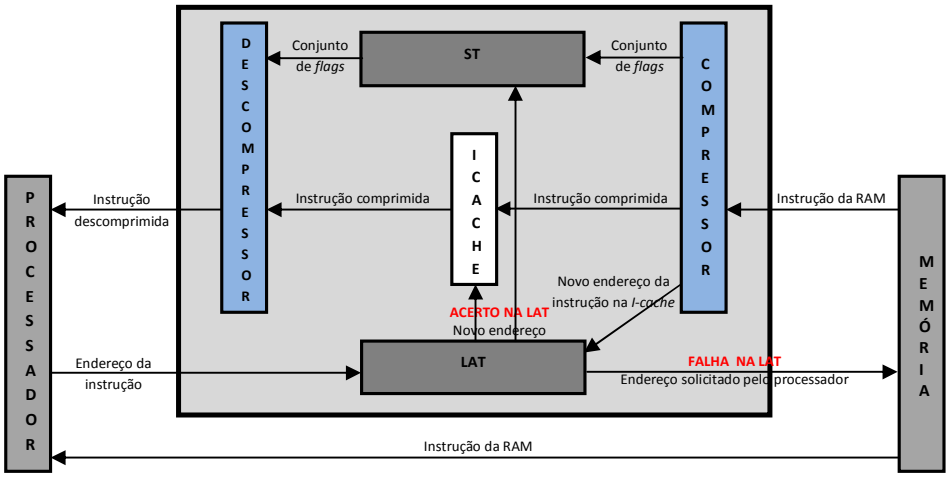

Figura 4.1 – Arquitetura PDCCM

A arquitetura PDCCM trabalha com instruções do tamanho de 32 *bits*, ou seja, cada linha da *I-cache* é composta por 4 *bytes*. Assim, a arquitetura proposta é compatível com sistemas que usam o processador ARM como núcleo do seu sistema embarcado, pois este processador tem o conjunto de instruções formada por 32 *bits* cada (arquitetura RISC). Nesta arquitetura, usando o método de compressão/descompressão *MIC*, todas as instruções que serão gravadas na *I-cache* sofrerão uma compressão de 50% no seu tamanho original. Maiores detalhes sobre o método de compressão e descompressão desenvolvido nesta dissertação serão descritos na seção "Descrição do Método *MIC*".

Para um melhor esclarecimento e sempre que possível, os nomes dos componentes serão semelhantes aqueles usados no código em VHDL, de modo que este trabalho sirva também como uma documentação para o código fonte. Os quatros componentes básicos desenvolvidos para a arquitetura PDCCM são: LAT, ST, Compressor e Descompressor. Na seção posterior é feito um completo detalhamento destes componentes.

De forma geral, a arquitetura mostrada na Figura 4.1 funciona da seguinte maneira: o processador solicita uma instrução para a *I-cache* passando um endereço de memória. Primeiramente será pesquisado na LAT se existe ou não esse endereço de instrução na *I-cache*. Caso a pesquisa na LAT retorne um ACERTO, significa que a instrução encontra-se na *I-cache*; então a LAT fornece o novo endereço da instrução na *I-cache*, o endereço do conjunto de *flags* na ST e em qual dupla de *bytes* encontra-se a instrução a ser descomprimida. Assim, é repassada para o descompressor a instrução comprimida juntamente com o conjunto de *flags* e o descompressor retorna para o processador a instrução de forma descomprimida.

Mas, se na pesquisa da LAT for retornada uma FALHA, então a instrução é buscada na memória RAM. Uma cópia da instrução buscada na RAM é passada para o processador e outra cópia é passada para o compressor, que irá comprimir a instrução e salvará a instrução comprimida na *I-cache* e o conjunto de *flags* na ST e em seguida atualizará a LAT com o novo endereço da instrução na *I-cache* e em qual dupla de *bytes* foi salva a instrução.

4.1.1. Descrição dos Componentes da Arquitetura PDCCM

Para a implementação da arquitetura PDCCM foram criados os seguintes componentes em VHDL:

- **LAT** (Tabela da Linha de Endereços - *Line Address Table*): é uma tabela que tem por função fazer o mapeamento dos endereços das instruções com o seu novo endereço na *I-cache*. Essa tabela também contém um *bit flag* que sinaliza em qual dupla de *bytes* (primeira ou segunda) está armazenado o código da instrução. A LAT deve ter o dobro do tamanho da *I-cache*. A Figura 4.2 mostra um trecho da LAT, onde se observa que no campo *addr_old* a primeira instrução encontrava-se armazenada no endereço $000fff_h$ e após o processo de compressão essa mesma instrução passou a ser armazenada no novo endereço (*addr_new*) $00c9_h$ da *I-cache*. No campo *pair_bytes* observa-se que para a primeira instrução o *bit flag* foi setado com valor 0 (zero), significando que a instrução encontra-se salva na primeira dupla de *bytes* na linha da *I-cache*.

addr_old	*addr_new*	*pair_bytes*
0x000fff	0x00c9	0
0x000ffe	0x00c8	1
0x000ffd	0x003a	0
0x000ffc	0x0039	1
...	...	...

Figura 4.2 – Exemplo de um trecho da LAT

O Código Fonte 1 apresenta a descrição da LAT em VHDL, onde pode-se notar os tipos de dados juntamente com os seus respectivos tamanhos. Os tipos de dados são: "natural" e "bit" e as variáveis de mais interesse neste componente são: `addr_old`, `addr_new` e `pair_bytes`.

Código Fonte 1 - Descrição da LAT - VHDL

```
type reg_LAT is record
      indice_LAT  : natural range ((2 * SIZE_CACHE) - 1) downto 0;
      addr_old    : natural range SIZE_ADDR_R downto 0;
      addr_new    : natural range SIZE_ADDR_C downto 0;
      pair_bytes  : bit;
end record;

type vetor_LAT is array (0 to ((2 * SIZE_CACHE) - 1)) of reg_LAT;
```

- **ST** (Tabela de Sinais - *Sign Table*): é uma tabela que contém *bits* que servem como *flags* para indicar ao descompressor qual dupla de *bits* deverá ser reconstituída, ou seja, descomprimida. A ST deve ter o mesmo tamanho da *I-cache*. A Figura 4.3 montra um trecho da ST;

Quando uma dupla de *bits* iniciar com o valor 0 (exemplo 00 ou 10), esta será substituída pelo *bit* 0, e quando uma dupla de *bits* iniciar com o valor 1 (exemplo 11 ou 01), será substituída pelo *bit* 1. Vale ressaltar que o modo de endereçamento das linhas de instrução para essa arquitetura é *Big-Endian*.

Na Figura 4.3 o campo *addr_ST* indica o novo endereço da instrução na *I-cache*. Então tomando como exemplo a primeira linha da ST, ou seja, *addr_ST* igual a $00c9_h$, temos no campo conjunto de *flags* dois conjuntos de *flags* de instruções (de 16 *bits* cada) que foram

gerados através do processo de compressão. Os primeiros 16 *bits* são de uma instrução comprimida e os últimos 16 *bits* são de outra instrução também comprimida. E assim, é formada toda a ST.

addr_ST	conjunto de *flags* (instruções com 16 *bits*) - *sinal_ST*
0x00c9	0000000011111111 0000000000000000
0x00c8	0000000100000001 0000000000000001
0x003a	0011110000111100 1111111100000000
...	...

Figura 4.3 – Exemplo de um trecho da ST

O Código Fonte 2 apresenta a descrição da ST em VHDL, onde pode-se notar os tipos de dados juntamente com os seus respectivos tamanhos. Os tipos de dados para a ST são: "natural" e "wordT", sendo que o tipo wordT é um subtipo do tipo "bit_vector", ou seja, vetor de *bits*. As variáveis de mais interesse neste componente são: `addr_ST` e `sinal_ST`.

Código Fonte 2 - Descrição da ST - VHDL

```
type reg_ST is record
      indice_ST : natural range (SIZE_CACHE - 1) downto 0;
      addr_ST   : natural range SIZE_ADDR_C downto 0;
      sinal_ST  : wordT;
end record;

type vetor_ST is array (0 to (SIZE_CACHE - 1)) of reg_ST;
```

- **Compressor**: tem por função fazer a compressão de todos os códigos das instruções que serão salvas na *I-cache*. O compressor é acionado toda vez que a memória RAM tiver um acesso e uma nova instrução for repassada para ser salva na *I-cache*.

O processo de compressão é realizado da seguinte forma: a instrução vinda da memória RAM é formada por 32 *bits* e é dividida em 16 duplas de 2 *bits* cada. Então, de forma seqüencial para cada dupla de *bits* serão feitas duas análises. A primeira análise é se os *bits* da dupla são iguais (00 e 11) ou diferentes (01 e 10). Se os *bits* da dupla forem iguais então são substituídos pelo *bit* 0 e se os *bits* da dupla forem diferentes então são substituídos

pelo *bit* 1. Após a análise de cada dupla de *bits* é salvo na *I-cache* esse novo valor, ou seja, o valor de substituição de cada dupla de *bits*. A segunda análise feita pelo compressor, diz se a dupla de *bits* inicia com o *bit* 0 (exemplo 00 e 10) ou com o *bit* 1 (exemplo 11 e 01). Então, para a dupla de *bits* que inicia com o *bit* 0 será salvo na ST o valor 0 e para a dupla de *bits* que inicia com o *bit* 1 será salvo na ST o valor 1.

O Código Fonte 3 apresenta a função de compressão descrita em VHDL que faz todo o processo de compressão das instruções. Os tipos de dados para esta função são: "natural" e "bit_vector" e os subtipos "vetor_iCACHE" e "vetor_ST". A variável de mais interesse nesta função é a `dupla` (forma as duplas de *bits* para serem inseridas na iCACHE e na ST).

Código Fonte 3 - Descrição da Função de Compressão - VHDL

```
function Compressor(iCACHE   : vetor_iCACHE;
                    ST       : vetor_ST;
                    dado_ram : wordT;
                    ind1     : natural;
                    incr_ind : natural)
return vetor_iCACHE is

variable cach     : vetor_iCACHE;
variable tb_s     : vetor_ST;
variable id1      : natural range 0 to (SIZE_CACHE - 1);
variable dupla    : bit_vector(2 downto 1);
variable cont     : natural := 1;
variable c_ic     : natural;

begin
      cach  := iCACHE;
      tb_s  := ST;
      id1   := ind1;

      if (id1 > SIZE_CACHE) then
            id1 := 0;
      end if;

      if ((incr_ind mod 2) = 0) then
            c_ic := 1;
            pt_dp <= '0';
      else
            c_ic := 17;
            pt_dp <= '1';
      end if;

      for i in (SIZE_WORD/2) downto 1 loop
            for x in 2 downto 1 loop
                  dupla(x) := dado_ram(cont);
                  cont := cont + 1;
            end loop;

            if (dupla = "00") then
```

```
                cach(id1).dado_memoria_icache(c_ic) := '0';
                tb_s(id1).sinal_ST(c_ic) := '0';
          elsif (dupla = "11") then
                cach(id1).dado_memoria_icache(c_ic) := '0';
                tb_s(id1).sinal_ST(c_ic) := '1';

          elsif (dupla = "01") then
                cach(id1).dado_memoria_icache(c_ic) := '1';
                tb_s(id1).sinal_ST(c_ic) := '0';

          else
                cach(id1).dado_memoria_icache(c_ic) := '1';
                tb_s(id1).sinal_ST(c_ic) := '1';
          end if;
          c_ic := c_ic + 1;
    end loop;

    returnC_icache <= cach(id1).dado_memoria_icache;

    return cach;
end function;
```

- **Descompressor**: tem por função fazer a descompressão de todas as instruções que estão armazenadas na *I-cache* e serão repassadas ao processador. O descompressor é acionado toda vez que a LAT for consultada e retornar um acerto.

O processo de descompressão é realizado da seguinte forma: a LAT indica qual o novo endereço da instrução na *I-cache*, qual o endereço do conjunto de *flags* na ST e em qual dupla de *bytes* a instrução encontra-se salva. O descompressor recebe todas essas informações e começa o processo de descompressão da instrução comprimida.

O primeiro *bit* da instrução comprimida vinda da *I-cache* é lido e se o mesmo for igual a 0 (zero), significa que a dupla de *bits* a ser reconstituída será 00 ou 11, o *bit flag* da ST definirá qual será essa dupla de *bits*. Se o *bit flag* for 0 (zero), então a primeira dupla de *bits* da instrução a ser reconstituída assume o valor 00, caso contrário assume o valor 11. Agora se o primeiro *bit* da instrução comprimida vinda da *I-cache* for igual a 1 (um), então significa que a dupla de *bits* a ser reconstituída será 01 ou 10. Novamente será usado o *bit flag* para definir qual será a dupla de *bits* a ser reconstituída. Se o *bit flag* for 0 (zero), então a primeira dupla de *bits* da instrução a ser reconstituída assume o valor 10, caso contrário assume o valor 01. Para cada instrução a ser descomprimida, repte-se esse mesmo processo com os 16 *bits* da

instrução salva na *I-cache*, transformando assim as instruções comprimidas de 16 *bits* em instruções descomprimidas de 32 *bits*.

O Código Fonte 4 apresenta a função de descompressão descrita em VHDL que faz todo o processo de descompressão das instruções. Os tipos de dados para esta função são: "natural" e "bit_vector" e os subtipos "vetor_iCACHE", "vetor_ST" e "wordT". A variável de mais interesse nesta função é a `instruction` (forma a instrução descomprimida, ou seja, instrução com 32 *bits*).

Código Fonte 4 - Descrição da Função de Descompressão - VHDL

```
function Descompressor(iCACHE      : vetor_iCACHE;
                       ST          : vetor_ST;
                       new_addr    : natural;
                       parte_dupla : bit)
return bit_vector is

variable cach     : vetor_iCACHE;
variable tb_s     : vetor_ST;
variable cont     : natural := 1;
variable acerto   : bit := '0';
variable instruction : wordT;

begin
      cach   := iCACHE;
      tb_s   := ST;

      for i in 0 to (SIZE_CACHE - 1) loop
            if (cach(i).addr_memoria_icache = new_addr)
            and (acerto = '0') then
                  acerto := '1';
                  i1_iCACHE_ST <= cach(i).indice_icache;
                  nw_ad <= cach(i).addr_memoria_icache;
                  if (parte_dupla = '0') then
                        for k in 1 to 16 loop
                              if ((cach(i).dado_memoria_icache(k)= '0')
                              and (tb_s(i).sinal_ST(k) = '0')
                              and (cont < 33)) then
                                    instruction(cont) := '0';
                                    cont := cont + 1;
                                    instruction(cont) := '0';
                                    cont := cont + 1;

                              elsif ((cach(i).dado_memoria_icache(k) = '0')
                              and (tb_s(i).sinal_ST(k) = '1')
                              and (cont < 33)) then
                                    instruction(cont) := '1';
                                    cont := cont + 1;
                                    instruction(cont) := '1';
                                    cont := cont + 1;
```

```
                              elsif ((cach(i).dado_memoria_icache(k) = '1')
                              and (tb_s(i).sinal_ST(k) = '0')
                              and (cont < 33)) then
                                    instruction(cont) := '0';
                                    cont := cont + 1;
                                    instruction(cont) := '1';
                                    cont := cont + 1;

                              else
                                    instruction(cont) := '1';
                                    cont := cont + 1;
                                    instruction(cont) := '0';
                                    cont := cont + 1;
                              end if;
                        end loop;
                  else
                        for k in 17 to 32 loop
                              if ((cach(i).dado_memoria_icache(k)= '0')
                              and (tb_s(i).sinal_ST(k) = '0')
                              and (cont < 33)) then
                                    instruction(cont) := '0';
                                    cont := cont + 1;
                                    instruction(cont) := '0';
                                    cont := cont + 1;

                              elsif ((cach(i).dado_memoria_icache(k) = '0')
                              and (tb_s(i).sinal_ST(k) = '1')
                              and (cont < 33)) then
                                    instruction(cont) := '1';
                                    cont := cont + 1;
                                    instruction(cont) := '1';
                                    cont := cont + 1;

                              elsif ((cach(i).dado_memoria_icache(k) = '1')
                              and (tb_s(i).sinal_ST(k) = '0')
                              and (cont < 33)) then
                                    instruction(cont) := '0';
                                    cont := cont + 1;
                                    instruction(cont) := '1';
                                    cont := cont + 1;

                              else
                                    instruction(cont) := '1';
                                    cont := cont + 1;
                                    instruction(cont) := '0';
                                    cont := cont + 1;
                              end if;
                        end loop;
                  end if;
            end if;
      end loop;
      return instruction;
end function;
```

Cabe ressaltar que o Compressor apresentado no Código Fonte 3 e o Descompressor apresentado no Código Fonte 4 são puramente combinacionais, ou seja, dadas algumas

entradas tipo `dado_ram`, `new_addr` e `parte_dupla` são geradas suas saídas em `returnC_icache`, `returnC_inst_proc` e `returnD_inst_proc`.

4.1.2. Instruções na *I-cache*

Todas as instruções que são gravadas na *I-cache* reduzirão o seu tamanho em 50%, ou seja, as instruções que são compostas originalmente por 32 *bits*, quando passarem pelo estágio de compressão, reduzirão o seu tamanho para 16 *bits*, e os outros 16 *bits* restantes da instrução são salvos em uma tabela (ST) que será utilizada apenas pelo descompressor para reconstituir a instrução original. A Figura 4.4 mostra um trecho de uma *I-cache* sem compressão e a Figura 4.5 mostra um trecho de uma *I-cache* comprimida.

address I-cache	Instruções Normais (32 *bits*)
0x00c9	00000000000000001010101010101010
0x00c8	10101010101010101111111111111111
0x003a	00000000000000000000000000000011
0x0039	10000000000000011000000000000001
...	...

Figura 4.4 – *I-cache* não comprimida

address I-cache	Instruções Comprimidas (16 *bits*)
0x00c9	1111111100000000 0000000011111111
0x00c8	1000000110000001 0000000000000000
0x003a	...
0x0039	...
...	...

Figura 4.5 – *I-cache* comprimida

O que pode-se observar na Figura 4.5 é que as instruções comprimidas ocuparam apenas 16 *bits* nas linhas da *I-cache*, possibilitando assim em cada linha da *I-cache* armazenar duas instruções. Exemplo, as linhas de endereço 0x00c9 e 0x00c8 da Figura 4.4 encontram-se representadas apenas na linha de endereço 0x00c9 da Figura 4.5, e as linhas de endereço 0x003a e 0x0039 da *I-cache* não comprimida estão representadas apenas na linha de endereço 0x00c8 da *I-cache* comprimida. Então, verifica-se que em apenas duas linhas de endereço da

I-cache comprimida (em 64 *bits*) consegui-se salvar quatro instruções normais da memória RAM (128 *bits* o tamanho das quatro instruções).

Uma das vantagens de aplicar esse tipo de compressão é justamente a possibilidade de dobrar o tamanho da capacidade da *I-cache*, lembrando que em sistemas embarcados esse tipo de recurso é muito limitado.

Citando como exemplo uma plataforma ARM *device* ARM922T em que a *I-cache* é de apenas 8Kbytes conforme SLOSS [37], aplicando a compressão será possível ter na *I-cache* até 16Kbytes de instruções comprimidas utilizando apenas os 8Kbytes de memória *I-cache* da plataforma.

4.2. Descrição do Método *MIC*

O método *MIC* (***M****iddle* ***I****nstruction* ***C****ompression*) é um método de compressão que tem por função reduzir em 50% o tamanho das instruções que são salvas na *I-cache*, passando então o tamanho dessas instruções de 32 *bits* (tamanho original) para 16 *bits* (tamanho comprimido).

Para compressão, a cada instrução que for lida na memória RAM e salva na *I-cache* será dividida em duplas de *bits* sendo cada dupla de *bits* formada por: 00, 01, 10 e 11. O compressor realiza a seguinte lógica: duplas com valores iguais são substituídas pelo *bit* 0 (zero) e duplas com valores diferentes são substituídas pelo *bit* 1 (um). Então os *bits* 00 e 11 na compressão são substituídos pelo *bit* 0 e os *bits* 01 e 10 são substituídos pelo *bit* 1. Assim, uma dupla de *bits* é reduzindo a um *bit* único.

Uma tabela auxiliar (ST) é criada para guardar o conjunto de *flags* da dupla de *bits* comprimidos. Em duplas de *bits* que iniciam com o valor 0 (zero), como no caso 00 ou 10, é gravado na ST o *bit* 0 e em duplas de *bits* que iniciam com o valor 1 (um), como no caso 11 ou 01, é gravado na ST o *bit* 1.

4.2.1. Algoritmo de Compressão e Descompressão do Método *MIC*

O processador requisita uma instrução à *I-cache* através de `end_inst_proc` (PC atual). Será verificada na LAT a existência ou não do endereço fornecido pelo processador. Se a instrução for encontrada na *I-cache* a LAT sinalizará com um ACERTO. Então, a LAT fornecerá o novo endereço da instrução na *I-cache*, o endereço do conjunto de *flags* da instrução na ST e o posicionamento da dupla de *bytes* (primeira ou segunda) onde encontra-se a instrução e os *flags* na *I-cache* e na ST, respectivamente. Todas essas informações são repassadas ao descompressor que reconstituirá a instrução e a retornará de forma descomprimida ao processador através da variável `returnD_inst_proc`.

A **descompressão das instruções** é realizada da seguinte forma:

- O novo endereço da instrução que foi repassado pela LAT, foi localizado na *I-cache* e na ST;
- A *I-cache* e a ST retornam para o descompressor os 16 *bits* da instrução comprimida e os 16 *bits* do conjunto de *flags*;
- Se o *bit* lido da instrução comprimida na *I-cache* for 0 (zero), a dupla de *bits* a ser reconstituída será 00 ou 11. O que definirá qual será a dupla de *bits* é o *bit flag*, ou seja, se o *bit flag* for 0 a dupla de *bits* a ser reconstituída será 00 e se o *bit flag* for 1 a dupla de *bits* a ser reconstituída será 11;
- Mas se o *bit* lido da instrução comprimida na *I-cache* for 1 (um), a dupla de *bits* a ser reconstituída será 10 ou 01. Então, novamente o *bit flag* que definirá qual será a dupla de *bits*, ou seja, se o *bit flag* for 0 a dupla de *bits* a ser reconstituída será 10 e se o *bit flag* for 1 a dupla de *bits* a ser reconstituída será 01;
- Para cada instrução a ser descomprimida, serão analisados os 16 *bits* da instrução salva na *I-cache*, transformando assim as instruções comprimidas de 16 *bits* em instruções descomprimidas de 32 *bits*.

Agora, se o endereço fornecido pelo processador não se encontrar na LAT, significa que não existe essa instrução na *I-cache*. A LAT sinalizará uma FALHA na linha da *I-cache*. O endereço fornecido pelo processador será repassado para a memória RAM, onde a mesma será consultada e verificada se existe ou não essa instrução. Caso a pesquisa na RAM indique uma FALHA, a instrução será buscada no HD (*Hard Disk*). Agora se a pesquisa indicar um

ACERTO significa que a instrução encontra-se na RAM. Em seguida, a RAM retornará uma cópia da instrução no formato original (não comprimida) para o processador através da variável `returnC_inst_proc` e outra cópia para o compressor, que realizará todo o processo de compressão.

A **compressão das instruções** é realizada da seguinte forma:

- A instrução é localizada na memória RAM e uma cópia da mesma é repassada para o processador e outra para o compressor;
- A instrução no compressor é dividida em 16 duplas de *bits*, sendo que cada dupla é formada no instante que é lida pela função de compressão. O início da leitura da instrução vinda da memória RAM é pelo MSB (*bit* mais significativo);
- O compressor sempre analisará em qual parte da dupla de *bytes* (primeira ou segunda) deverá ser salva a instrução comprimida na *I-cache* e na ST;
- Se a dupla de *bits* lida para a compressão for 00 ou 11, então essa dupla de *bits* será substituída pelo *bit* 0 e salva na *I-cache*. Agora se a dupla de *bits* lida for 10 ou 01 então essa dupla de *bits* será substituída pelo *bit* 1 e salva na *I-cache*;
- O conjunto de *flags* da ST será formado através da seguinte lógica: se o primeiro *bit* da dupla de *bits* que está sendo comprimido for 0, então o *bit flag* salvo será 0. Agora se o primeiro *bit* da dupla de *bits* que está sendo comprimido for 1, então o *bit flag* salvo será 1;
- Após o compressor fazer toda a substituição dos 32 *bits* da instrução original nos 16 *bits* comprimidos e seus respectivos *bits flags*, o compressor irá salvar na dupla de *bytes* (primeira ou segunda) a instrução comprimida na *I-cache* e o conjunto de *flags* na ST;
- A tabela LAT será atualizada através do novo endereço da instrução salva na *I-cache* e na ST;
- Para cada instrução que for buscada na memória RAM, repetirá esse processo de compressão.

Importante salientar que esses mecanismos de compressão e descompressão são realizados em tempo de execução, via hardware específico que foi prototipado em FPGAs, o que torna este trabalho como sendo um dos inéditos nessa linha de pesquisa. A arquitetura PDCCM sofreu uma pequena perda de desempenho devido ao ciclo adicional em seu *pipeline*,

mas uma vez que tratando tudo isso como sendo hardware, então a perda de desempenho, não foi muito grande. Portanto, na implementação da arquitetura PDCCM com o método *MIC* verificou-se que é similar ao trabalho do LEKATSAS [22, 25], obtendo um componente que precisa de apenas um único ciclo para o processo de compressão ou descompressão, além dos benefícios mostrados na próxima seção.

4.3. Resultados das Simulações

Para a realização das simulações foi descrito, no código fonte em VHDL da arquitetura desenvolvida, um trecho da memória RAM contendo os endereços e seus respectivos valores de instruções exemplos que foram mostrados na Figura 4.6. E ainda, para facilitar as simulações de compressão e descompressão também foi descrito um trecho da memória *I-cache* já contendo algumas linhas preenchidas com instruções comprimidas (de 32 para 16 *bits* cada instrução). A mesma está exemplificada na Figura 4.7.

addr_RAM	Valor da instrução (32 *bits*)
0x000fff	00000000000000001010101010101010
0x000ffe	10101010101010101111111111111111
0x000ffd	00000000000000000000000000000011
0x000ffc	10000000000000011000000000000001
0x000ffb	01010101010101011010101010101010
0x000ffa	00001111111100000000111111110000
0x000ff9	10101111010100000101101011110000
0x000ff8	01111111111111110000000000000001
0x000ff7	10000011011111001111111100000000
0x000ff6	00000000000000001111111111111110
0x000ff5	10101010000011111111000001010101
0x000ff4	00000000010101011010101011111111
...	...

Figura 4.6 – Exemplo de trecho da memória RAM

addr_I-cache	Valor da instrução (16 *bits*)
0x00ab	1111111100000000 0000000011111111
0x00aa	1000000110000001 0000000000000000
0x00a9	0000000000000000 1111111111111111
0x00a8	0000111111110000 1111000000001111
...	...

Figura 4.7 – Exemplo de trecho da *I-cache*

Para que a arquitetura realize a descompressão em nível de simulação, é necessário que a LAT também já esteja previamente preenchida com os endereços velhos das instruções e os seus novos endereços na *I-cache* e na ST, juntamente com o posicionamento da dupla de *bytes*. A Figura 4.8 mostra um trecho da LAT preenchida.

addr_old	*addr_new*	*pair_byte*
0x000fff	0x00ab	0
0x000ffe	0x00ab	1
0x000ffd	0x00aa	0
0x000ffc	0x00aa	1
0x000ffb	0x00a9	0
0x000ffa	0x00a9	1
0x000ff5	0x00a8	0
0x000ff4	0x00a8	1
...	...	...

Figura 4.8 – Exemplo de trecho da LAT

Mediante as informações das Figuras 4.6, 4.7 e 4.8 é possível realizar uma exemplificação mais clara e concisa do funcionamento da arquitetura PDCCM usando o método *MIC*. A Figura 4.9 mostra de forma geral as ondas da simulação de todos os pinos da arquitetura PDCCM. As simulações foram obtidas usando a ferramenta Quartus II Web Edition da empresa ALTERA® [43].

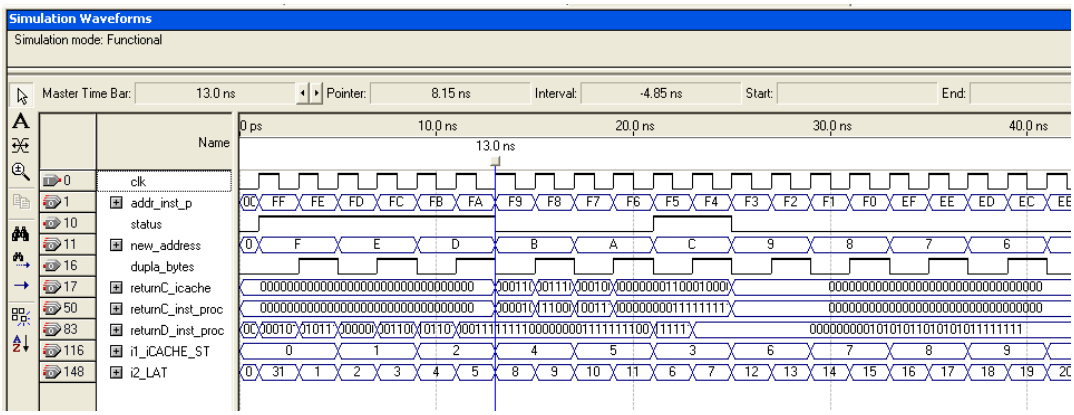

Figura 4.9 – Simulação em forma de ondas da arquitetura PDCCM

A arquitetura simulada tem como pino de entrada o **clock** (`clk`) com formação de ondas com o período de 1 ns. Os pinos de saída são:

- **addr_inst_p**: esse conjunto de pinos mostra os valores dos endereços da memória RAM;
- **status**: esse pino indica quando está havendo uma compressão e uma descompressão. O pino `status` estando setado com o valor 0, ou seja, nível baixo, indica que está havendo uma compressão, já estando setado com o valor 1, ou seja, nível alto, indica que está havendo o processo inverso (uma descompressão);
- **new_address**: esse pino indica qual o novo posicionamento da instrução na *I-cache* e na ST;
- **dupla_bytes**: indica em qual dupla de *bytes* a instrução encontra-se na *I-cache* e na ST. O pino *dupla_bytes* estando setado com o valor 0 indica que a instrução está na primeira dupla de *bytes*. Já o pino `dupla_bytes` estando setado com o valor 1 indica que a instrução está na segunda dupla de *bytes*;
- **returnC_icache**: mostra a instrução que será salva na *I-cache* após o processo de compressão (instrução comprimida);
- **returnC_inst_proc**: mostra a instrução que foi buscada na memória RAM e repassada uma cópia para o processador e outra para o compressor;
- **returnD_inst_proc**: mostra a instrução que foi descomprimida e será repassada ao processador;
- **i1_iCACHE_ST**: serve apenas para demonstrar o índice do posicionamento da instrução na *I-cache* e na ST;
- **i2_LAT**: indica qual a posição do índice do endereço da instrução na LAT.

Nas Figuras 4.10 e 4.11 são mostradas as simulações das operações de descompressão e compressão passo a passo.

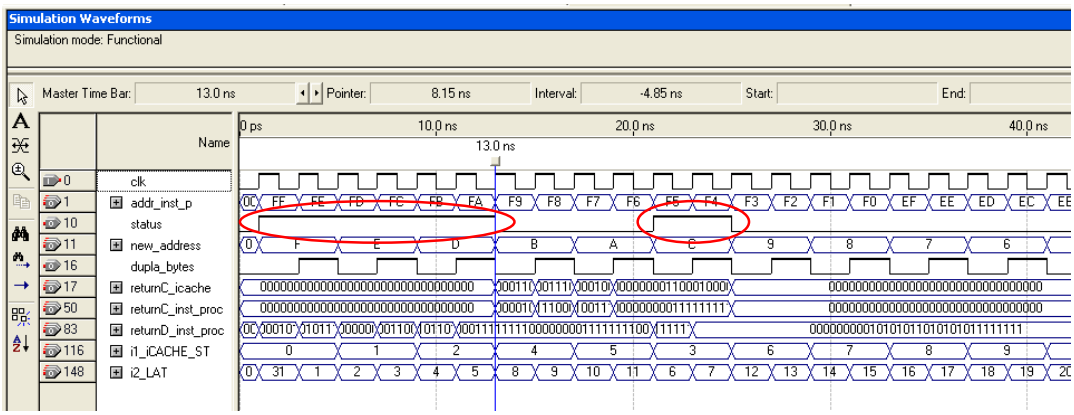

Figura 4.10 – Exemplo 1 da simulação em forma de ondas - Descompressão

Como pode-se observar na Figura 4.10 os círculos em vermelho no pino `status` indicam pontos da simulação onde ocorreu a descompressão de instruções. Através do pino `new_address` destaca-se que as instruções que foram descomprimidas estão contidas nos endereços F_h, E_h, D_h e C_h.

Neste exemplo, o processador solicitou uma instrução e passou o endereço FF_h. A LAT foi pesquisada e retornou um ACERTO indicando que a instrução solicitada encontra-se comprimida na *I-cache*. Então, a LAT indicou o novo endereço da instrução na *I-cache* que para esse exemplo é F_h e pode ser observado no pino `new_address` e a instrução encontra-se salva na primeira dupla de *bytes* da linha da *I-cache* e da ST que também pode ser observada no pino `dupla_bytes`.

A fórmula para o cálculo do tempo de descompressão de cada instrução está descrita a seguir:

Tdescomp = número de *clocks*

Em nosso exemplo, 1 *clock* é igual a 2 ns. Então, **Tdescomp = 2 ns**

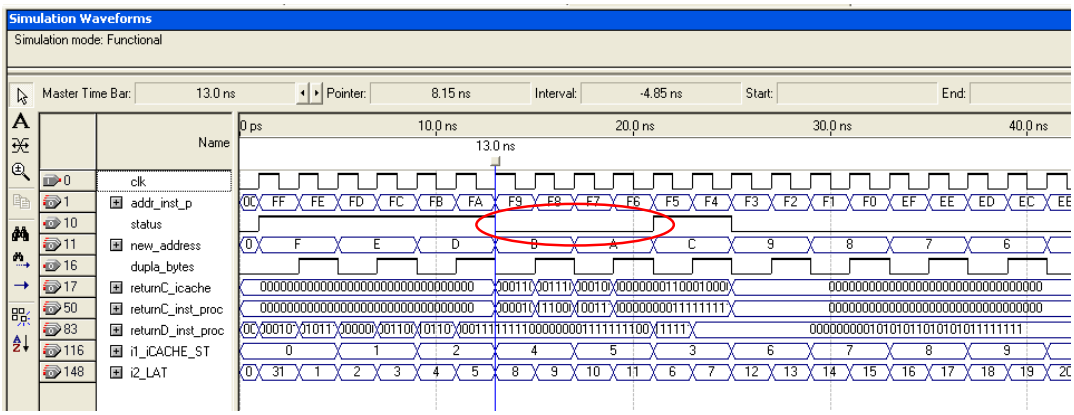

Figura 4.11 – Exemplo 2 da simulação em forma de ondas - Compressão

Observa-se na Figura 4.11, que o círculo em vermelho no pino `status` indica pontos da simulação onde ocorreu a compressão de instruções. Através do pino `addr_inst_p` destacamos que as instruções, que estão na memória RAM nos endereços $F9_h$ até $F6_h$, foram comprimidas e agora ocupam os novos endereços na *I-cache*: B_h e A_h.

Neste exemplo, o processador solicitou uma instrução e passou o endereço $F9_h$. A LAT foi pesquisada e retornou uma FALHA indicando que a instrução solicitada não encontra-se comprimida na *I-cache*. Então, a LAT repassou para a memória RAM o endereço solicitado pelo processador e agora uma nova busca da instrução foi realizada na RAM. Achando a instrução, foi repassada uma cópia para o processador onde podemos ver através do pino `returnC_inst_proc` e uma cópia da instrução para o compressor.

O compressor realiza todo o processo de compressão e salva a instrução comprimida na *I-cache* e o conjunto de *flags* na ST e em seguida atualiza a LAT. Então, através do pino `returnC_icache` é possível ver o código da instrução comprimida (16 *bits*), e para esse exemplo no pino `new_address` observa-se que a instrução foi salva no endereço B_h e na primeira dupla de *bytes* o que pode ser notado com o pino `dupla_bytes`.

A fórmula para o cálculo do tempo de compressão de cada instrução está descrita a seguir:

Tcomp = número de *clocks*

Em nosso exemplo, 1 *clock* é igual a 2 ns. Então, **Tcomp = 2 ns**

4.3.1. Estatísticas de Recursos utilizados do FPGA

Observa-se na Tabela 4.1 informações sobre as características do FPGA e do ambiente de simulação e na Tabela 4.2 visualizam-se estatísticas de desempenho dos componentes da arquitetura PDCCM e da arquitetura completa usando o método *MIC*.

Tabela 4.1 – Características do FPGA e do ambiente de simulação

FPGA	
Fabricante	Altera®
Família	Cyclone II
Dispositivo	EP2C20F484C7
Ambiente de Simulação	
Fabricante	Altera®
Ferramenta de simulação e síntese	Quartus II Web Edition
Versão	8.0 build 231 SP 1 SJ

Tabela 4.2 – Estatística de desempenho da arquitetura PDCCM

FPGA / Componentes	Elementos Lógicos	Registradores	Pinos
LAT	74 (<1%)	43 (<1%)	15 (5%)
ST	63 (<1%)	10 (<1%)	70 (22%)
Compressor	65 (<1%)	12 (<1%)	128 (41%)
Descompressor	67 (<1%)	11 (<1%)	134 (43%)
Arquitetura PDCCM método *MIC*	**1.753 (9%)**	**644 (3%)**	**177 (56%)**

Na Tabela 4.3 é possível visualizarmos os tempos dos componentes da arquitetura PDCCM juntamente com a arquitetura final.

Tabela 4.3 – Temporização da arquitetura PDCCM

FPGA / Componentes	Tempo no pior caso	Freqüência do *clock* em MHz	Freqüência do *clock* em Tempo
LAT	10.055 ns	154.01 MHz	6.493 ns
ST	9.842 ns	181.03 MHz	5.524 ns
Compressor	8.537 ns	175.26 MHz	5.567 ns
Descompressor	7.797 ns	177.43 MHz	5.636 ns
Arquitetura PDCCM método *MIC*	**9.065 ns**	**40.05 MHz**	**24.967 ns**

Como podemos observar na Tabela 4.2 os componentes individuais da arquitetura PDCCM utilizaram poucos recursos disponíveis na plataforma, ou seja, tanto os elementos lógicos quanto os registradores do FPGA foram utilizados menos de 1% e os pinos variaram entre 5% a 43%.

Já a arquitetura completa do PDCCM utilizou apenas 9% dos elementos lógicos, 3% dos registradores e 56% dos pinos do FPGA, ressaltando que na arquitetura implementada, foi simulada no código fonte um trecho da memória RAM e um trecho da *I-cache*, isto é, o tamanho real da arquitetura PDCCM (em elementos lógicos, registradores, pinos e outros) é ainda menor do que o apresentado na tabela de estatísticas de desempenho.

Na Tabela 4.3 é possível verificar que o tempo no pior caso para todos os componentes individuais da arquitetura PDCCM variaram entre 7.797 ns até 10.055 ns, a freqüência do *clock* em MHz e a freqüência do *clock* em tempo (nanosegundos) esteve basicamente próxima para todos os componentes individuais da arquitetura. Porém, a arquitetura geral obteve 9.065 ns para o tempo no pior caso, freqüência do *clock* de 40.05 MHz e a freqüência do *clock* em tempo foi de 24.967 ns para a execução da simulação.

4.4. Variação da Arquitetura PDCCM - *Huffman*

De acordo com os trabalhos correlatos descritos no Capítulo 2 desta dissertação, o algoritmo de *Huffman* foi utilizado em várias pesquisas de compressão/descompressão de código de instrução. Então, fundamentado nestas pesquisas foi implementado em VHDL uma variação da arquitetura PDCCM que tem por método o algoritmo de *Huffman* como base para compressão/descompressão das instruções, e assim, foi possível realizar análises comparativas no desempenho do Método *MIC* e do Método de *Huffman*, afim, de relatar qual o melhor método de compressão/descompressão para a arquitetura PDCCM desenvolvida.

Conforme HUFFMAN [18], este algoritmo (*Huffman*) usa a probabilidade de ocorrências dos símbolos (código das instruções) no conjunto das instruções contidas na *I-cache* com intuito de determinar novos códigos para as instruções que mais se repetem. Agrupando-as e substituindo-as por apenas uma única instrução com um novo valor de código gerado por através da árvore de *Huffman*.

Para o funcionamento correto da arquitetura utilizando o método de *Huffman*, foi necessário fazer algumas mudanças nos componentes da arquitetura PDCCM sendo uma delas a substituição do componente ST pelo HT (Tabela de *Huffman*). A Figura 4.12 mostra o novo modelo (variação da arquitetura) da arquitetura PDCCM usando o método de *Huffman* para realizar a compressão/descompressão dos códigos de instruções. No Apêndice A desta dissertação encontra-se o detalhamento dos componentes modificados na arquitetura (LAT, Compressor e Descompressor), a inclusão da HT na arquitetura PDCCM e o exemplo do formato das instruções na *I-cache* usando o método de *Huffman*.

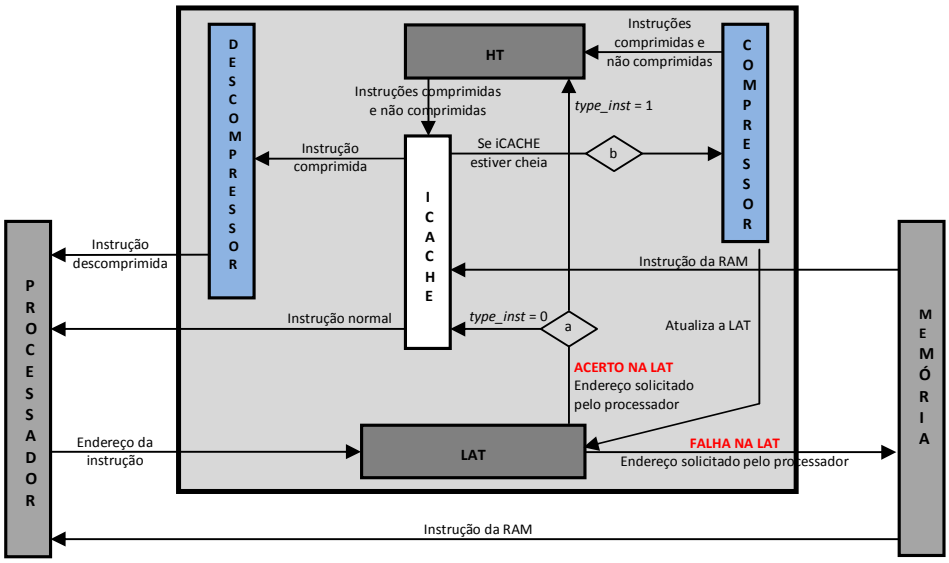

Figura 4.12 – Arquitetura PDCCM para usar o método de *Huffman*

De forma geral, a arquitetura mostrada na Figura 4.12 funciona da seguinte maneira: o processador solicita uma instrução para a *I-cache* passando um endereço de memória. Primeiramente será pesquisado na LAT se existe ou não esse endereço de instrução na *I-cache*. Caso a pesquisa na LAT retorne um ACERTO, isso indica que a instrução encontra-se na *I-cache*; então a LAT informa através do pino `type_inst` se a instrução pesquisada e localizada na *I-cache* está ou não comprimida. Se o pino `type_inst` estiver com o valor setado em 0 (nível baixo), significa que a instrução está normal (não comprimida) e a mesma é repassada da *I-cache* para o processador; agora, se o pino `type_inst` estiver com o valor setado em 1 (nível alto), então significa que a instrução está comprimida, a mesma será pesquisada na HT para verificar o seu valor correspondente na árvore de *Huffman* e repassado para o descompressor que fará todo o processo de descompressão e repassará a instrução descomprimida para o processador.

Se na pesquisa da LAT for retornada uma FALHA, então a instrução é buscada na memória RAM. Uma cópia desta instrução é passada para o processador e outra cópia é salva na *I-cache*.

Toda vez que uma instrução for salva na *I-cache* é feito um teste para verificar se a *I-cache* já está cheia ou não, se o teste for positivo (*I-cache* cheia), o compressor tenta

comprimir e salvar as instruções contidas na *I-cache* usando o método de *Huffman* e em seguida atualiza a LAT. Mas se o teste for negativo (*I-cache* não cheia), a instrução só é salva na *I-cache* e a LAT é atualizada.

4.5. Descrição do Método de *Huffman*

Este método de compressão permite a representação em binário das instruções da *I-cache* a partir de sua probabilidade de ocorrência. Esta representação é gerada por um sistema de compressão usando árvore binária, o que impede a ambigüidade na análise do código das instruções comprimidas, conforme [18].

A ambigüidade, neste caso, refere-se a uma decodificação que permite a confusão com outras instruções também comprimidas. Por exemplo, determinada instrução *ADD* tem o código binário 01 e outra instrução *MOV* tem o código binário 0100, isto implica que, ao verificarmos a seqüência binária para a instrução *MOV* poderemos estar interpretando como *ADD*, ao serem lidos apenas os *bits* 01.

Por isso, a codificação de *Huffman* utiliza a árvore binária para representar as instruções comprimidas, de forma que permitam uma decodificação única para cada instrução. A compressão usando o método de *Huffman* necessita de que cada instrução na *I-cache* tenha um valor de probabilidade de ocorrência (freqüência).

Portanto, a árvore binária gerada para a arquitetura PDCCM tem tamanho fixo, ou seja, cada código da árvore é formado por *32 bits*. Isto é necessário devido a implementação em hardware obrigar que todas as variáveis tenham seu tamanho definido de forma estática.

4.6. Resultados das Simulações usando o método de *Huffman*

Para realizar as simulações, foi descrito no código fonte em VHDL da variação da arquitetura PDCCM um trecho de memória RAM contendo os endereços e seus respectivos valores de instruções exemplos.

Para melhorar a didática e facilitar a exemplificação foram substituídas as instruções binárias (32 *bits*) por instruções de nível mais alto. A Figura 4.13 mostra as instruções e os seus correspondentes códigos em binário e a Figura 4.14 exemplifica um trecho de memória RAM usando o método de *Huffman*.

Instrução	Código da instrução em binário
ADD	00000000000000001010101010101010
MOV	10101010101010101111111111111111
SUB	00000000000000000000000000000011
ADD	00000000000000001010101010101010
SUB	00000000000000000000000000000011
SW	10000011011111001111111100000000
ADD	00000000000000001010101010101010
LOAD	00001111111100000000111111110000
LOAD	00001111111100000000111111110000
MOV	10101010101010101111111111111111
MULT	10101010000011111111000001010101
ADD	00000000000000001010101010101010
...	...

Figura 4.13 – Instruções em alto nível e seus correspondentes em binário

addr_RAM	instrução
0x000fc4	ADD
0x000fc3	MOV
0x000fc2	SUB
0x000fc1	ADD
0x000fc0	SUB
0x000fbf	SW
0x000fbe	ADD
0x000fbd	LOAD
0x000fbc	LOAD
0x000fbb	MOV
0x000fba	MULT
0x000fb9	ADD
...	...

Figura 4.14 – Exemplo de trecho de memória RAM usando o método de *Huffman*

Além disso, para facilitar as simulações de compressão e descompressão usando o método de *Huffman* também foi descrito um trecho da memória *I-cache* já contendo algumas

linhas preenchidas com instruções comprimidas e instruções não comprimidas. A mesma está exemplificada na Figura 4.15.

addr_I-cache	Código da instrução em binário
0x000fc4	11111111111111111111111111111111
0x000fc3	01111111111111111111111111111111
0x000fc2	00111111111111111111111111111111
0x000fc1	00011111111111111111111111111111
0x000fc0	10000011011111001111111100000000
0x000fbf3	10101010000011111111000001010101
...	...

Figura 4.15 – Exemplo de trecho da *I-cache* usando o método de *Huffman*

Para que a arquitetura realize a descompressão em nível de simulação, é necessário que a LAT também já esteja previamente preenchida com os endereços das instruções e com o `type_inst` para identificar instruções comprimidas das não comprimidas. A Figura 4.16 mostra um trecho da LAT preenchida usando o método de *Huffman*.

addr_inst	*type_inst*
0x000fc4	1
0x000fc3	1
0x000fc2	1
0x000fc1	1
0x000fc0	0
0x000fbf3	0
...	...

Figura 4.16 – Exemplo de trecho da LAT usando o método de *Huffman*

Código da instrução	Freqüência	Código de *Huffman*
ADD	4	11111111111111111111111111111111
MOV	2	01111111111111111111111111111111
SUB	2	00111111111111111111111111111111
LOAD	2	00011111111111111111111111111111
SW	1	10000011011111001111111100000000
MULT	1	10101010000011111111000001010101
...		...

Figura 4.17 – Exemplo de trecho da HT usando o método de *Huffman*

Para simulação, a HT também necessita estar preenchida para ocorrer a descompressão. Então, a Figura 4.17 mostra exemplo de um trecho da HT preenchida usando o método de *Huffman*.

Mediante as informações das Figuras 4.14, 4.15, 4.16 e 4.17 é possível realizar uma exemplificação mais clara e concisa do funcionamento da arquitetura PDCCM usando o método de *Huffman*. A Figura 4.18 mostra de forma geral as ondas da simulação de todos os pinos da arquitetura PDCCM. As simulações foram obtidas usando a ferramenta Quartus II Web Edition da empresa ALTERA®.

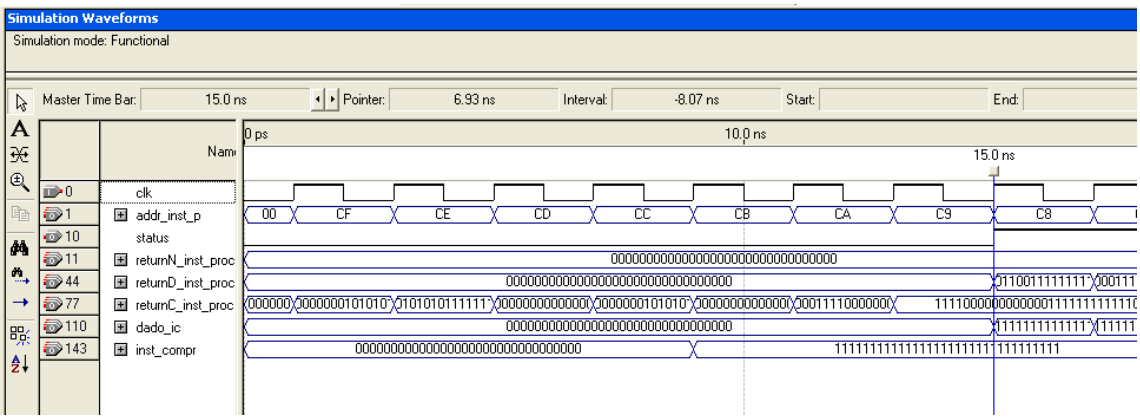

Figura 4.18 – Simulação em forma de ondas da arquitetura PDCCM usando o método de *Huffman*

A arquitetura simulada tem como pino de entrada o **clock** (`clk`) com formação de ondas com o período de 1 ns. Os pinos de saída são:

- **addr_inst_p**: esse conjunto de pinos mostra os valores dos endereços da memória RAM;
- **status**: esse pino indica quando está havendo uma compressão e uma descompressão. O pino `status` estando setado com o valor 0, ou seja, nível baixo, indica que está havendo uma compressão já o pino `status` estando setado com o valor 1, ou seja, nível alto, indica que está havendo o processo inverso (uma descompressão);
- **returnN_inst_proc**: mostra a instrução normal (não comprimida) que foi buscada da *I-cache* e retornada para o processador;

- **returnD_inst_proc**: mostra a instrução descomprimida que foi buscada na *I-cache* passada pelo descompressor e retornada para o processador;
- **returnC_inst_proc**: mostra a instrução que foi buscada na memória RAM repassada uma cópia para o processador e outra cópia para ser salva na *I-cache*;
- **dado_ic**: esse pino mostra o código da instrução na *I-cache* que foi solicitado pelo processador através da variável `end_inst_proc` (PC atual);
- **inst_compr**: esse pino mostra o código da instrução comprimida ou não comprimida que foi buscada na memória RAM, salva na *I-cache* e passada pelo compressor.

Nas Figuras 4.19 e 4.20 são mostradas as simulações das operações de descompressão e compressão passo a passo.

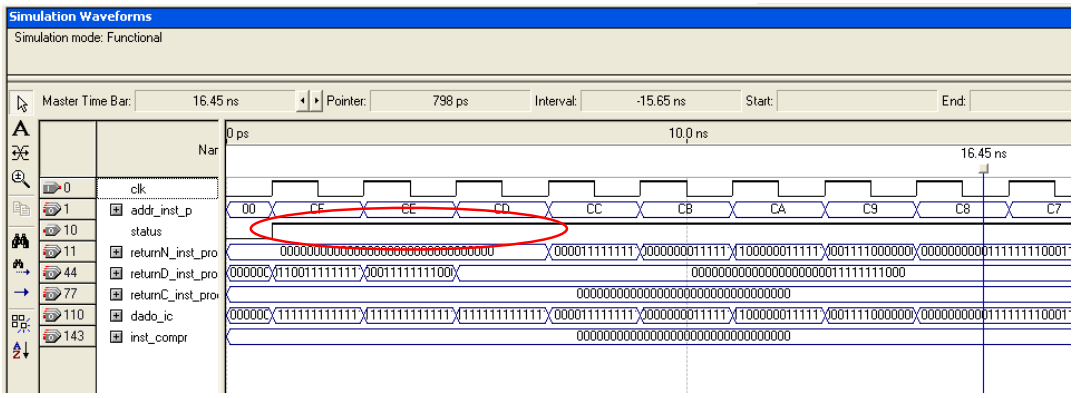

Figura 4.19 – Exemplo 1 da simulação em forma de ondas - Descompressão usando o método de *Huffman*

Como pode-se observar na Figura 4.19 o círculo em vermelho no pino `status` indica o ponto da simulação onde ocorreu a descompressão de instruções. Através do pino `returnN_inst_proc` destaca-se que as instruções que foram descomprimidas estão contidas nos endereços CF_h, CE_h e CD_h, já no pino `returnD_inst_proc` é possível verificar o código das instruções descomprimidas.

Neste exemplo, o processador solicitou uma instrução e passou o endereço CF_h. A LAT foi pesquisada e retornou um ACERTO indicando que a instrução solicitada encontra-se comprimida na *I-cache*. Então o pino `type_inst` da LAT retornou o valor 1, indicando que a

instrução solicitada está comprimida na *I-cache*. O pino `dado_ic` informa o código das instruções comprimidas e não comprimidas (instruções normais), ou seja, é o conteúdo da *I-cache*.

A fórmula para o cálculo do tempo de descompressão de cada instrução está descrita a seguir:

Tdescomp = número de *clocks*

Em nosso exemplo, 1 *clock* é igual a 2 ns. Então, **Tdescomp = 2 ns**

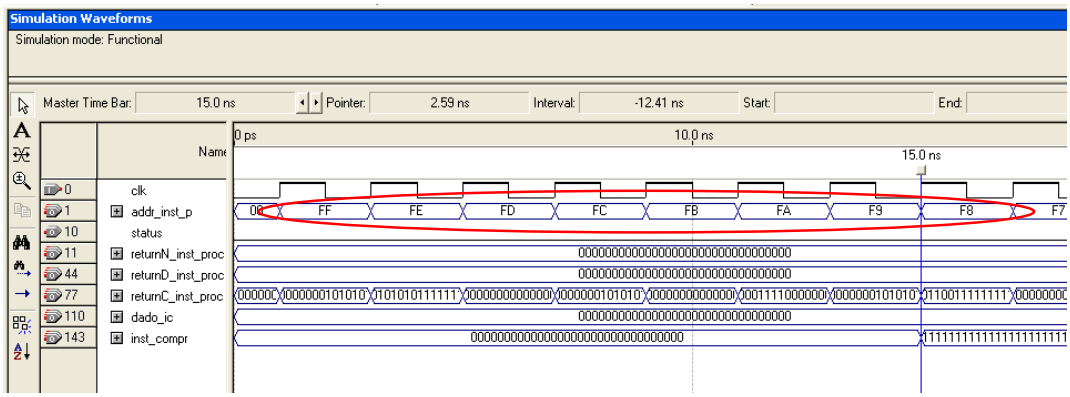

Figura 4.20 – Exemplo 2 da simulação em forma de ondas - Compressão usando o método de *Huffman*

Observa-se na Figura 4.20, que o círculo em vermelho no pino `status` indica o ponto da simulação onde ocorreu a compressão de instruções. Através do pino `addr_inst_p` destacamos que as instruções, que estão na memória RAM nos endereços FF_h até $F8_h$, foram repassadas para o compressor.

Neste exemplo, o processador solicitou uma instrução e passou o endereço FF_h. A LAT foi pesquisada e retornou uma FALHA indicando que a instrução solicitada não encontra-se comprimida na *I-cache*. Então, a LAT repassou para a memória RAM o endereço solicitado pelo processador e agora uma nova busca da instrução foi realizada na memória RAM. Achando a instrução, foi repassada uma cópia para o processador onde podemos ver através do pino `returnC_inst_proc` e outra cópia da instrução para o compressor.

O compressor realiza todo o processo de compressão e salva as instruções comprimidas e não comprimidas na *I-cache*, e, em seguida atualiza a LAT. Então, através do pino `inst_compr` é possível ver o código da instrução comprimida, que para esse exemplo vemos a partir do endereço $F8_h$.

A fórmula para o cálculo do tempo de compressão de cada instrução está descrita a seguir:

Tcomp = número de *clocks*

Em nosso exemplo, 1 *clock* é igual a 2 ns. Então, **Tcomp = 2 ns**

4.6.1. Estatísticas de Recursos utilizados do FPGA no método de *Huffman*

As características do FPGA juntamente com o ambiente de simulação para a compressão/descompressão das instruções usando o método de *Huffman* são as mesmas apresentas na Tabela 4.1.

A Tabela 4.4 mostra as estatísticas de desempenho dos componentes da arquitetura PDCCM e da arquitetura completa usando o método de *Huffman* para realizar a compressão e descompressão das instruções contidas na *I-cache*.

Tabela 4.4 – Estatística de desempenho da arquitetura PDCCM usando o método de *Huffman*

FPGA / Componentes	Elementos Lógicos	Registradores	Pinos
LAT	84 (<1%)	44 (<1%)	19 (6%)
HT	---	---	---
Compressor	70 (<1%)	128 (<1%)	170 (54%)
Descompressor	132 (<1%)	84 (<1%)	170 (54%)
Arquitetura PDCCM método de *Huffman*	**3.568 (19%)**	**266 (1%)**	**170 (54%)**

Na Tabela 4.5 visualizaremos os tempos dos componentes da arquitetura PDCCM juntamente com a arquitetura final usando o método de *Huffman*.

Tabela 4.5 – Temporização da arquitetura PDCCM usando o método de *Huffman*

FPGA / Componentes	Tempo no pior caso	Freqüência do *clock* em MHz	Freqüência do *clock* em Tempo
LAT	10.391 ns	165.02 MHz	6.060 ns
HT	---	---	---
Compressor	11.964 ns	8.71 MHz	114.776 ns
Descompressor	9.140 ns	112.49 MHz	8.890 ns
Arquitetura PDCCM método de *Huffman*	**11.231 ns**	**9.19 MHz**	**108.862 ns**

Como podemos observar na Tabela 4.4 os componentes individuais da arquitetura PDCCM usando o método de *Huffman* utilizaram poucos recursos disponíveis na plataforma, ou seja, foram utilizados menos de 1% dos elementos lógicos e dos registradores do FPGA e a utilização dos pinos variou entre 6% a 54%.

Já a arquitetura completa do PDCCM usando o método de *Huffman* utilizou apenas 19% dos elementos lógicos, 1% dos registradores e 54% dos pinos do FPGA, ressaltando que na arquitetura implementada foi simulado no código fonte um trecho da memória RAM e um trecho da *I-cache*, isto é, o tamanho real da arquitetura PDCCM (em elementos lógicos, registradores, pinos e outros) usando o método de *Huffman* é ainda menor do que o apresentado na Tabela 4.4.

Na Tabela 4.5 é possível verificar que o tempo no pior caso para todos os componentes individuais da arquitetura PDCCM variaram entre 9.140 ns a 10.391 ns, a freqüência do *clock* em MHz varia de 112.49 MHz até 198.37 MHz e a freqüência do *clock* em tempo (nanosegundos) esteve basicamente próxima para todos os componentes individuais da arquitetura, ou seja, de 5.041 ns a 8.890 ns. No entanto, a arquitetura geral usando o método de *Huffman* obteve 11.231 ns para o tempo no pior caso, freqüência do *clock* em MHz de 9.19 MHz e a freqüência do *clock* em tempo foi de 108.862 ns para a execução da simulação.

Portanto, observando os resultados das Tabelas 4.2 e 4.4 que tratam a estatística de desempenho para as arquiteturas gerais usando o método *MIC* e *Huffman* notamos que os elementos lógicos para o método *MIC* apresentaram uma menor taxa de ocupação no FPGA.

O mesmo não ocorre quando observamos os registradores do FPGA onde o método de *Huffman* apresentou utilizar menos desse recurso.

Quando comparamos as Tabelas 4.3 e 4.5, referentes à temporização da arquitetura podemos concluir que o método *MIC* apresentou melhores resultados em todos os aspectos abordados no estudo: tempo no pior caso, na freqüência do *clock* em MHz e na freqüência do *clock* em tempo. O que nos mostra ser um método bem eficiente para a arquitetura PDCCM.

4.7. Resumo do Capítulo

Este capítulo descreveu sobre a arquitetura PDCCM, isto é, seu funcionamento e componentes. Também foi mostrado o código fonte em VHDL dos componentes individuais da arquitetura PDCCM usando o método de compressão/descompressão ***MIC***, sendo os componentes: LAT, ST, Compressor e Descompressor. Para melhor entendimento foram mostradas várias figuras que exemplificam trechos da memória RAM, *I-cache*, LAT e ST facilitando assim a compreensão do leitor sobre o funcionamento da arquitetura PDCCM.

Ainda foi descrito de forma detalhada o "Método *MIC*", criado para realizar a compressão/descompressão das instruções na arquitetura PDCCM, e depois mostrado os resultados obtidos através das simulações realizadas.

Em seguida foram descritas as modificações na arquitetura PDCCM, utilizando o método de *Huffman* como algoritmo de compressão/descompressão das instruções. Também foram mostradas várias figuras para exemplificar o funcionamento do método de *Huffman* na arquitetura PDCCM. Além disso, abordamos os resultados das simulações obtidas através do referido método.

Na conclusão do capítulo, foi mostrada uma análise comparativa das simulações realizadas com os métodos de compressão/descompressão ***MIC*** e ***Huffman*** usando a arquitetura PDCCM, sendo que o método ***MIC*** apresentou melhor desempenho computacional para esta arquitetura desenvolvida.

Portanto, o próximo capítulo tratará aspectos de simulações reais através de alguns programas do *benchmark MiBench* para medições de desempenhos no qual será visto um comparativo entre os métodos *MIC* e *Huffman*.

CAPÍTULO 5

5. SIMULAÇÕES COM *BENCHMARK MIBENCH*

Neste capítulo apresentamos as simulações realizadas com alguns trechos de programas do *benchmark MiBench*. Os resultados enfatizam a utilização da arquitetura PDCCM que usa os métodos de compressão e descompressão *MIC* e *Huffman*. Após simulações usando a ferramenta Quartus II da ALTERA®, os resultados obtidos são dados como as estatísticas de recursos utilizados no FPGA, assim como a respectiva temporização e finalizamos com a taxa de compressão.

As aplicações do pacote de *benchmark MiBench* usadas nas simulações foram escolhidas de forma que representasse a maioria das categorias definidas pelo *MiBench* (ver Capítulo 3, Seção 3.3).

5.1. Descrição das Simulações

A Tabela 5.1 mostra um resumo dos programas do *benchmark MiBench* usados neste capítulo, os quais estão em código Assembly do processador ARM9, conforme encontrados em [48, 49, 50, 51].

Tabela 5.1 – Aplicação do pacote *MiBench* usado nas simulações

Categorias	*MiBench*
Telecommunications	CRC32
Consumer Devices	JPEG
Automotive and Industruial Control	QuickSort
Security	SHA

Para realização das simulações de compressão e descompressão foram escolhidos os seguintes *benchmark MiBench*:

- **CRC32**: é um algoritmo utilizado para detectar erros na transmissão de dados e a redundância do CRC em um arquivo;
- **JPEG**: é um compressor e descompressor de imagens com o formato JPEG;
- **QuickSort**: é um algoritmo que faz ordenação de dados;
- **SHA**: é um algoritmo que gera chaves de criptografia para troca segura de dados e assinaturas digitais.

Foi utilizado o processador embarcado ARM para simular o funcionamento da compressão e descompressão dos métodos *MIC* e *Huffman* na arquitetura PDCCM. No entanto, o apêndice B desta dissertação apresenta uma tabela com atribuição de valores binários para o conjunto de instruções do ARM (família ARM9, versão ARM922T, ISA ARMv4T), sendo que cada instrução é formada por 32 *bits*.

Para as simulações de compressão e descompressão dos métodos *MIC* e *Huffman*, foram selecionadas as 256 primeiras instruções de cada *MiBench*, obtidas através do código compilado (Assembly) para o processador embarcado ARM, formando assim o conjunto de seqüências de instruções que foram usadas para carregar um trecho das memórias RAM e a *I-cache*.

O trecho da memória RAM descrita em VHDL foi utilizada em todas as simulações com os *benchmark MiBench* e teve tamanho fixo de 256 linhas de 4 *bytes* cada (modelando uma memória de 1Kbyte), contabilizando assim 8.192 *bits* e a *I-cache* teve o tamanho de 32 linhas de 32 *bits* cada (completando então, um *I-cache* de 1Kbit). Assim, se observa que há uma relação de 8:1 entre os tamanhos da memória RAM e da respectiva *cache* de instruções.

Em relação ao dimensionamento das memórias RAM e *I-cache* para as simulações dos métodos de compressão/descompressão, observamos as seguintes características:

- Para o método *MIC* foi possível armazenar na *I-cache* de 32 linhas, 64 instruções comprimidas, ou seja, 100% a mais de instruções em relação a qualquer outra arquitetura com *I-cache* de 32 *bits* e que não usa nenhum método de compressão;
- Para o método de *Huffman* foi possível armazenar 32 instruções normais e/ou comprimidas. Sendo que, após o processo de compressão uma única instrução comprimida pode representar várias vezes uma mesma instrução normal e repetida na *I-cache*.

5.2. Impacto da Compressão dos Códigos de Instruções

Para a simulação do processo de compressão das instruções, foram utilizadas 256 instruções obtidas através de códigos Assembly (compilado para a plataforma ARM) de cada *benchmark MiBench* (usados para as simulações) e essas instruções foram inseridas no código fonte do trecho da memória RAM simulada.

A Tabela 5.2 mostra as estatísticas de desempenho da arquitetura PDCCM usando os métodos *MIC* e *Huffman* na "compressão das instruções" dos *benchmark MiBench* usados nas simulações.

Tabela 5.2 – Estatísticas de desempenho da arquitetura PDCCM na compressão das instruções dos *benchmark MiBench*

	Método *MIC*			Método de *Huffman*		
	Elementos Lógicos	Registradores	Pinos	Elementos Lógicos	Registradores	Pinos
CRC32	2.461 (13%)	997 (5%)	177 (56%)	2.277 (12%)	362 (2%)	170 (54%)
JPEG	3.077 (16%)	1.110 (6%)	177 (56%)	2.604 (14%)	369 (2%)	170 (54%)
QuickSort	3.013 (16%)	1.076 (6%)	177 (56%)	2.612 (14%)	367 (2%)	170 (54%)
SHA	2.992 (16%)	1.109 (6%)	177 (56%)	2.534 (14%)	367 (2%)	170 (54%)
Médias	**2.886 (15%)**	**1.073 (6%)**	**177 (56%)**	**2.507 (14%)**	**366 (2%)**	**170 (54%)**

Observando a Tabela 5.2 é visível a igualdade nos recursos utilizados do FPGA entre os métodos *MIC* e *Huffman*, sendo que os dois métodos não utilizaram muitos recursos

disponíveis da plataforma. No entanto, o método de *Huffman* apresentou-se um pouco melhor do que o método *MIC* nos recursos utilizados do FPGA.

Na Tabela 5.3 é possível visualizarmos a temporização da arquitetura PDCCM usando os métodos *MIC* e *Huffman* na "compressão das instruções" dos *benchmark MiBench* usados nas simulações.

Tabela 5.3 – Temporização da arquitetura PDCCM na compressão das instruções dos *benchmark MiBench*

	Método *MIC*			**Método de *Huffman***		
	Tempo no pior caso	Freqüência do *clock* em MHz	Freqüência do *clock* em Tempo	Tempo no pior caso	Freqüência do *clock* em MHz	Freqüência do *clock* em Tempo
CRC32	9.012 ns	42.01 MHz	23.805 ns	10.072 ns	13.48 MHz	74.179 ns
JPEG	9.430 ns	30.68 MHz	32.593 ns	9.594 ns	13.08 MHz	76.454 ns
QuickSort	9.600 ns	31.27 MHz	31.980 ns	10.266 ns	13.36 MHz	74.857 ns
SHA	9.209 ns	30.11 MHz	33.217 ns	9.465 ns	12.72 MHz	78.591 ns
Médias	**9.314 ns**	**33.52 MHz**	**30.398 ns**	**9.849 ns**	**13.16 MHz**	**76.020 ns**

Como podemos observar na Tabela 5.3 a média do tempo no pior caso para os métodos *MIC* e *Huffman* foram bem idênticos, agora a freqüência do *clock* em MHz e Tempo apresentou uma diferença notória, sendo que o método *MIC* apresentou melhores desempenhos de temporização do FPGA em relação ao método de *Huffman*, pois sua freqüência de operação é mais alta.

Ainda notamos nas Tabelas 5.2 e 5.3 que na arquitetura PDCCM a compressão das instruções para o *MiBench* CRC32 apresentou ser mais eficiente em comparação com os demais *MiBench* analisados. Pois este *MiBench* (CRC32) tem mais de 50% das operações inteiras na ULA.

Portanto, para o processo de compressão das instruções o método *MIC* mostrou ser o mais eficiente na arquitetura PDCCM, sendo que nas "estatísticas de desempenho" (Tabela 5.2) a quantidade de elementos lógicos, registradores e pinos entre os dois métodos não passou de 4% a favor do método de *Huffman* e na "temporização" (Tabela 5.3) o método *MIC* apresentou-se melhor para todos os *benchmark MiBench* analisados. Porém, é observada uma diferença com mais de 60% na média da freqüência do *clock* em MHz dos *MiBench*. Então,

destacamos que o método criado para esta dissertação (*MIC*), proporcionou bons resultados nos recursos do FPGA requeridos na compressão dos códigos das instruções de 32 *bits* na arquitetura PDCCM comparando com o método de compressão tradicional no mundo científico (*Huffman*).

5.3. Impacto da Descompressão dos Códigos de Instruções

Para a simulação do processo de descompressão das instruções, foi utilizada uma *I-cache* com 32 linhas contendo instruções comprimidas de alguns programas *benchmark MiBench* obtidas a partir do processo de compressão. As instruções comprimidas foram inseridas no código fonte da *I-cache* para então simular a descompressão das instruções.

A Tabela 5.4 mostra as estatísticas de desempenho da arquitetura PDCCM usando os métodos *MIC* e *Huffman* na "descompressão das instruções" dos *benchmark MiBench* usados nas simulações.

Tabela 5.4 – Estatísticas de desempenho da arquitetura PDCCM na descompressão das instruções dos *benchmark MiBench*

	Método *MIC*			Método de *Huffman*		
	Elementos Lógicos	Registradores	Pinos	Elementos Lógicos	Registradores	Pinos
CRC32	2.982 (16%)	1.287 (7%)	176 (56%)	4.814 (26%)	175 (<1%)	170 (54%)
JPEG	3.667 (20%)	1.439 (8%)	176 (56%)	4.630 (25%)	180 (<1%)	170 (54%)
QuickSort	3.493 (19%)	1.391 (7%)	176 (56%)	5.540 (30%)	134 (<1%)	170 (54%)
SHA	3.639 (19%)	1.438 (8%)	176 (56%)	9.170 (49%)	138 (<1%)	170 (54%)
Médias	**3.445 (19%)**	**1.389 (7%)**	**176 (56%)**	**6.039 (33%)**	**157 (<1%)**	**170 (54%)**

Observa-se na Tabela 5.4 que os recursos utilizados do FPGA nos métodos *MIC* e *Huffman* apresentam diferenças entre os elementos lógicos e os registradores, ambos os métodos não utilizaram muitos recursos disponíveis na plataforma. No entanto, o método *MIC* utilizou menos elementos lógicos e mais registradores do que o método de *Huffman*.

Na Tabela 5.5 é possível visualizarmos a temporização da arquitetura PDCCM usando os métodos *MIC* e *Huffman* na "descompressão das instruções" dos *MiBench* usados

nas simulações.

Tabela 5.5 – Temporização da arquitetura PDCCM na descompressão das instruções dos *benchmark MiBench*

	Método *MIC*			**Método de *Huffman***		
	Tempo no pior caso	Freqüência do *clock* em MHz	Freqüência do *clock* em Tempo	Tempo no pior caso	Freqüência do *clock* em MHz	Freqüência do *clock* em Tempo
CRC32	9.252 ns	35.35 MHz	28.289 ns	10.555 ns	5.69 MHz	175.776 ns
JPEG	9.535 ns	28.93 MHz	34.570 ns	10.693 ns	6.35 MHz	157.459 ns
QuickSort	8.996 ns	30.58 MHz	32.700 ns	11.952 ns	5.75 MHz	173.890 ns
SHA	9.155 ns	28.85 MHz	34.658 ns	10.827 ns	4.32 MHz	231.302 ns
Médias	**9.234 ns**	**30.92 MHz**	**32.554 ns**	**11.006 ns**	**5.52 MHz**	**184.606 ns**

Observando a Tabela 5.5 verificamos que as médias do tempo no pior caso para os métodos *MIC* e *Huffman*, foram bem próximas uma da outro, em contra partida a freqüência do *clock* em MHz e Tempo apresentaram diferenças grandes para os métodos. Contudo, destacamos que o método *MIC* apresentou melhores desempenhos de temporização do FPGA em relação ao método de *Huffman* usando a arquitetura PDCCM, pois sua freqüência de operação é mais alta.

Nas Tabelas 5.4 e 5.5 também é verificado que para o processo de descompressão usando o método *MIC* o *MiBench* CRC32 mostrou-se ter mais desempenho computacional na arquitetura PDCCM, enquanto que no método de *Huffman* o *MiBench* JPEG apresentou melhores desempenhos na arquitetura PDCCM em comparação com os demais *MiBench* analisados.

Portanto, na descompressão dos códigos de instruções o método *MIC* apresentou melhores resultados de desempenhos computacionais para a arquitetura PDCCM, sendo que nas "estatísticas de desempenho" (Tabela 5.4) a quantidade média de registradores no método de *Huffman* foi apenas 6% a menos, mas a média dos elementos lógicos apresentou uma diferença de 14% a mais em comparação com o método *MIC*.

Quanto a "temporização" (Tabela 5.5), outra vez o método *MIC* apresentou-se melhor para todos os *MiBench* analisados, obtendo uma diferença média de mais 82% na freqüência do *clock* em MHz. Portanto, para a descompressão das instruções dos *benchmark*

MiBench (CRC32, JPEG, QuickSort e SHA) na arquitetura PDCCM, o método *MIC* apresenta-se mais viável que o método de *Huffman*.

No apêndice C desta dissertação é mostrado o código em Assembly dos quatro *benchmark MiBench* escolhidos para serem simulados, estando os mesmos compilados para o processador embarcado ARM9.

5.4. Impacto na Taxa de Compressão das Instruções

Conforme visto no Capítulo 4, Seção 4.2 desta dissertação, todas as instruções que são comprimidas pelo método *MIC* sofrem uma redução de 50% no seu tamanho original. Com isso, é possível termos uma melhor performance (desempenho, tempo de execução, capacidade de armazenamento, acerto da busca na *I-cache* e outros) do sistema usando um processador embarcado com algum método de compressão de códigos.

Para as simulações realizadas com alguns programas do *benchmark MiBench* a memória *I-cache* formada por 32 linhas de 32 *bits* cada, após o processo de compressão armazenou 64 instruções de 32 *bits* cada, ou seja, o dobro da sua capacidade na mesma quantidade de linhas.

Em se tratando do método de *Huffman* após o processo de compressão pode ter instruções comprimidas que representam um grande número de instruções repetidas na *I-cache* gerando assim um possível benefício na capacidade de armazenamento, mas em contra partida a árvore de *Huffman* a ser gerada sempre fica limitada ao tamanho da instrução, ou seja, não pode ter valores na árvore de *Huffman* maiores que a quantidade de *bits* de uma instrução da plataforma.

Para as simulações realizadas só foi possível ter 32 códigos na árvore de *Huffman*, sendo que cada código comprimido pode representar várias instruções repetidas na *I-cache*. Então, a *I-cache* para este método foi composta por instruções normais e instruções comprimidas. A Tabela 5.6 mostra um comprativo na taxa de compressão das instruções dos *benchmark MiBench* usando os métodos *MIC* e *Huffman*.

Tabela 5.6 – Comparativo na taxa de compressão das instruções do *benchmark MiBench*

MiBench	256 instruções	
	MIC	*Huffman*
CRC32	128 (50%)	159 (38%)
JPEG	128 (50%)	181 (29%)
QuickSort	128 (50%)	192 (25%)
SHA	128 (50%)	164 (36%)
Médias	**128 (50%)**	**174 (32%)**

Tendo como base as 256 primeiras instruções dos *benchmark MiBench* obtidas através do código assembly compilado para a plataforma ARM, observamos na Tabela 5.6 que o método *MIC* comprimiu em 50% o tamanho das instruções, ou seja, as 256 linhas do trecho da memória RAM usada na simulação, após o processo de compressão passou a ocupar apenas 128 linhas na *I-cache*. Já as instruções comprimidas usando o método de *Huffman* obtiveram uma média geral na compressão de 32% a menos em relação ao tamanho da memória RAM usada na simulação.

Então, constatamos que para a arquitetura PDCCM usando as 256 primeiras instruções dos *benchmark MiBench* (CRC32, JPEG, QuickSort e SHA), o método *MIC* mostrou-se mais eficiente na taxa de compressão, ou seja, um percentual 36% maior se comparado ao método de *Huffman*.

5.5. Resumo do Capítulo

Este capítulo descreveu sobre as simulações realizadas na arquitetura desenvolvida PDCCM usando como parâmetro de medidas os *benchmark MiBench* (CRC32, JPEG, QuickSort e SHA) e tendo como algoritmo de compressão e descompressão os métodos *MIC* e *Huffman*.

O método proposto nesta dissertação (*MIC*) apresentou melhores resultados nas estatísticas de desempenho e temporização do FPGA para a arquitetura PDCCM. Quanto à capacidade de armazenamento, tudo depende do conjunto das instruções geradas pelo compilador, pois o método *MIC* sempre armazenará o dobro do tamanho da *I-cache* da

arquitetura do processador embarcado que está sendo usado e o método de *Huffman* se limita apenas ao tamanho da instrução da *I-cache* em *bits*, podendo gerar a árvore de *Huffman* do tamanho em *bits* de uma instrução.

Portanto, o próximo capítulo tratará aspectos das conclusões e trabalhos futuros para esta dissertação.

CONCLUSÕES E TRABALHOS FUTUROS

O resultado central desta dissertação foi o desenvolvimento de uma arquitetura e um método de compressão/descompressão e respectiva prototipação em hardware (FPGA) capaz de comprimir e descomprimir, em tempo de execução, os códigos de instruções armazenadas na memória de sistemas embarcados e que seja compatível com a arquitetura de processadores RISC.

A arquitetura desenvolvida foi intitulada de PDCCM (***P****rocessor* ***D****ecompressor* ***C****ache* ***C****ompressor* ***M****emory*) na qual identificou-se o posicionamento do hardware compressor e do descompressor, já o novo método de compressão/descompressão recebeu o nome de *MIC* (***M****iddle* ***I****nstruction* ***C****ompression*).

Como foi visto no Capítulo 2 desta dissertação, existem várias técnicas para otimizar a execução e o consumo de energia em sistemas embarcados. Sendo que uma dessas técnicas é a compressão do código de instrução. Mas a maioria das propostas existentes atualmente focaliza apenas na descompressão assumindo que o código da instrução é comprimido em tempo de compilação. Então a técnica, desenvolvida com arquitetura PDCCM e o método *MIC*, tornou-se inédita, até o presente momento, pois a mesma enfatiza tanto a compressão quanto a descompressão dos códigos de instruções em tempo de execução.

Conforme comprovado com as simulações realizadas no Capítulo 5 e os trabalhos correlatos apresentados no Capítulo 2, o uso de técnicas de compressão do código de instrução que utilizam uma implementação em hardware mostrou-se viável para os sistemas embarcados que usam arquitetura RISC. Pois futuramente esta técnica pode tornar-se um

componente necessário em projetos de sistemas embarcados. Com a utilização de técnicas de compressão do código de instrução, as arquiteturas RISC conseguem minimizar um de seus maiores problemas, que é a quantidade de memória para armazenar os programas.

Após análises, destacamos como sendo um dos problemas no método *MIC* o fato de que o mesmo sempre possui uma taxa de compressão de tamanho fixo, isto é, sempre comprime em 50% o tamanho das instruções não podendo sofrer nenhuma outra variação de tamanho.

Já a eficácia na arquitetura PDCCM, fica difícil compará-la com outras arquiteturas uma vez que a mesma é inédita, a única validação que pode ser feita é a análise no método de compressão/descompressão desenvolvido nesta dissertação (*MIC*) para ser utilizado na arquitetura e comparado com o tradicional método de *Huffman*.

Mediante as simulações realizadas com alguns programas do *benchmark MiBench* verificamos que o método *MIC* apresentou as seguintes médias: 26% a menos na utilização dos elementos lógicos do FPGA, uma freqüência (MHz) de operação em aproximadamente 3 vezes maior para os processos de compressão/descompressão dos códigos de instruções e 36% mais eficiente na taxa de compressão dos *MiBench* analisados em relação ao método de *Huffman*, que também foi prototipado em hardware (FPGAs).

Portanto, analisando os dados obtidos através da ferramenta de simulação e prototipação da ALTERA (Quartus II Web Edition), onde foram estimadas as estatísticas de desempenhos e temporização dos recursos do FPGA, conclui-se que o método desenvolvido nesta dissertação, chamado de *MIC* mostrou ser mais eficiente computacionalmente em comparação com o método de *Huffman* implementado em hardware. As simulações utilizaram os programas CRC32, JPEG, QuickSort e SHA do *benchmark MiBench* para obter as medições de desempenho.

Ainda, é importante resaltar que na análises realizadas, utilizou-se uma arquitetura compatível com o processador embarcado ARM9, RISC de 32 *bits*.

6.1. Trabalhos Futuros

Embora todos os objetivos iniciais tenham sido atingidos, alguns melhoramentos podem ser feitos na implementação do método *MIC*, tais como:

- Projetar e implementar um processador de arquitetura RISC que já tenha o hardware descompressor embutido em seu núcleo;
- Obter uma forma melhor de codificação para as instruções comprimidas, gerando assim a possibilidade de excluir da arquitetura PDCCM a tabela de sinais (ST);
- Obter uma forma direta de resolução de endereços que não utiliza uma tabela, podendo proporcionar um melhor desempenho computacional para arquitetura PDCCM;
- Realizar testes de compressão e descompressão usando os métodos *MIC* e *Huffman* com mais programas do *benchmark MiBench* a fim de caracterizar o desempenho da arquitetura projetada;
- Realizar uma validação experimental mais detalhada em uma plataforma real que integre um processador ARM ao compressor/descompressor em FPGAs;
- Chegar a uma implementação em ASIC da arquitetura PDCCM, de modo que o trabalho transcendesse o âmbito acadêmico, servindo como uma contribuição, também, para o meio industrial.

REFERÊNCIAS BIBLIOGRÁFICAS

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

[2] AZEVEDO, R. Uma Arquitetura para Código Comprimido em Sistemas Dedicados. Tese de Doutorado, Instituto de Computação, Universidade Estadual de Campinas, Junho de 2002.

[3] BELL, T.; CLEARY, J.; WITTEN, I. Text Compression. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[4] BENINI, L.; MACII, A.; NANNARELLI, A. Cached-Code Compression for Energy Minimization in Embedded Processor. Proc. of ISPLED'01, pages 322-327, August 2001.

[5] BENINI, L.; MACII, A.; NANNARELLI, A. Code Compression for Cache Energy Minimization in Embedded Systems. IEE Proceedings on Computers and Digital Techniques, 149(4):157-163, July 2002.

[6] BENTLEY, J. L.; MCGEOCH, C. Amortized Analyses of Self-Organizing Sequential Search Heuristcs. Communications of ACM, 28(4), pages 404-411, April 1985.

[7] BURGER, D; AUSTIN, T. M. The Simplescalar Tool Set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.

[8] CARRO, L. Projeto e Prototipação de Sistemas Digitais. – Porto Alegre: Ed. Universidade / UFRGS, 2001. 171p.

[9] COSTA, C. da. Projetando Controladores Digitais com FPGA. – São Paulo: Novatec Editora, 2006, 159p.

[10] DAVIS II, J.; GOEL, M.; HYLANDS, C.; KIENHUIS, B.; LEE, E. A.; LIU, J.; LIU, X.; MULIADI, L.; NEUENDORFFER, S.; REEKIE, J.; SMYTH, N.; TSAY, J.; XIONG, Y. Overview of the Ptolemy Project, ERL Technical Memorandum UCB/ERL Tech. Report Nº M-99/37, Dept. EECS, University of California, Berkeley, July 1999.

[11] D'AMORE, R. VHDL - Descrição e Síntese de Circuitos Digitais. – Rio de Janeiro: LTC, 2005, 259p.

[12] ELIAS, P. Universal Codeword Sets and Representation of Integers. IEEE Transaction on Information Theory, 21(2), pages 194-203, March 1975.

[13] ERCEGOVAC, M.; LANG, T.; MORENO, J. H. Tradução José Carlos Barbosa dos Santos. Introdução aos Sistemas Digitais. – Porto Alegre: Bookman, 2000. 453p.

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

[15] FANO, R. M. Transmission of Information. M.I.T. Press, 1949.

[16] GUTHAUS, M.; RINGENBERG, J.; ERNST, D.; AUSTIN, T.; MUDGE, T.; BROWN, R. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In Proc. of the IEEE 4th Annual Workshop on Workload Characterization, pages 3-14, December 2001.

[17] HENNESSY, J. L.; PATTERSON, D. A. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, San Francisco, 3rd edition, 2003.

[18] HUFFMAN, D. A. A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the IRE, 40(9):1098-1101, September 1952.

[19] KEMP, T.; MONTOYE, R.; HARPER, J.; PALMER, J.; AUERBACH, D. A Decompression Core for PowerPC. IBM Journal of Research and Development, 42(6):807-812, September 1998.

[20] KILLIAN, E.; WARTHMAN, F. Xtensa Instruction Set Architecture Reference Manual. Tensilica, 2001.

[21] LEFURGY, C.; BIRD, P.; CHEN, I-C.; MUDGE, T. Improving Code Density Using Compression Techniques. In Proc. Int'l Symposium on Microarchitecture, pages 194-203, December 1997.

[22] LEKATSAS, H.; HENKEL, J.; JAKKULA, V. Design of One-Cycle Decompression Hardware for Performance Increase in Embedded Systems. In Proc. ACM/IEEE Design Automation Conference, pages 34-39, June 2002.

[23] LEKATSAS, H.; HENKEL, J.; WOLF, W. Code Compression for Low Power Embedded System Design. In Proc. ACM/IEEE Design Automation Conference, pages 294-299, 2000.

[24] LEKATSAS, H.; HENKEL, J.; WOLF, W. Design and Simulation of a *pipelined* Decompression Architecture for Embedded Systems. In Proc. ACM/IEEE Int'l Symposium on System Synthesis, pages 63-68, October 2001.

[25] LEKATSAS, H.; WOLF, W. Code Compression for Embedded Systems. In Proc. ACM/IEEE Design Automation Conference, pages 516-521, June 1998.

[26] MARTINS, C. A. P. da S.; ORDOÑEZ, E. D. M.; CÔRREA, J. B. T.; CARVALHO, M. B. Computação Reconfigurável: Conceitos, Tendências e Aplicações. In: XXIII Congresso da Sociedade Brasileira de Computação, Campinas, SP, 2 a 8 de Agosto. Anais. Campinas, SBC/JAI/UNICAMP, 2003. v. II, p. 339-388.

[27] NETTO, E. B. W. Compressão de Código Baseada em Multi-Profile. Tese de Doutorado, Instituto de Computação, Universidade Estadual de Campinas, Maio de 2004.

[28] NETTO, E. B. W.; AZEVEDO, R.; CENTODUCATTE, P.; ARAÚJO, G. Mixed Static/Dynamic Profiling for Dictionary Based Code Compression. The Proc. of the International System-on-Chip Symposium, Finland, pages 159-163, November 2003.

[29] NETTO, E. B. W.; AZEVEDO, R.; CENTODUCATTE, P.; ARAUJO, G. Multi-Profile Based Code Compression. In Proc. ACM/IEEE Design Automation Conference, pages 244-249, June 2004.

[30] NETTO, E. B. W.; OLIVEIRA, R. S. de; AZEVEDO, R.; CENTODUCATTE, P. Compressão de Código em Sistemas Embarcados. HOLOS CEFET-RN. Natal, Ano 19, páginas 23-28, Dezembro, 2003. 94p.

[31] OLIVEIRA, A. S. de; ANDRADE, F. S. de. Sistemas Embarcados - Hardware e Firmware na Prática. – São Paulo: Editora Érica, 2006, 316p.

[32] ORDOÑEZ, E. D. M. Caches Remotos e *Prefetching* em Sistemas Multiprocessadores de Alto Desempenho - Considerações Arquiteturais. Tese de Doutorado, Departamento de Engenharia Eletrônica, Escola Politécnica da Universidade de São Paulo, Agosto de 1998.

[33] ORDOÑEZ, E. D. M.; PEREIRA, F. D.; PENTEADO, C. G.; PERICINI, R. de A. Projeto, Desempenho e Aplicações de Sistemas Digitais em Circuitos Programáveis (FPGAs). – Pompéia: Bless, 2003, 240p.

[34] PowerPC. MPC5200B Data Sheet. Freescale Semiconductor, January 2006.

[35] RAMALHO, D. G. Desenvolvimento de uma Plataforma de Prototipação Rápida usando um Sistema Multi-FPGAs. Trabalho de Conclusão de Curso, Centro de Informática, Universidade Federal de Pernambuco, Novembro de 2001.

[36] SHANNON, C. E. A Mathematical Theory of Communication. The Bell System Techinical Journal, July 1948.

[37] SLOSS, A. N.; SYMES, D.; WRIGHT, C. ARM System Developer's Guide - Designing and Optimizing System Software. – San Francisco: Morgam Kaufmann, 2004, 689p.

[38] WILNER, W. Burroughs B1700 Memory Utilization. In Fall Joint Computer Conference, pages 579-586, 1972.

[39] WOLFE, A.; CHANIN, A. Executing Compressed Programs on an Embedded RISC Architecture. Proc. of International Symposium on Microarchitecture, pages 81-91, December 1992.

[40] XScale. Intel XScale Core Developer's Manual. Intel, January 2004.

[41] ZIV, J.; LEMPEL; A. A Universal Algorithm for Sequential Data Compression. IEEE Transaction on Information Theory, 23(3), pages 337-343, May 1977.

SITES

[42] ACTEL© Corporation. Disponível em: http://www.actel.com. Acessado em 10 de julho de 2008.

[43] ALTERA® Corporation. Disponível em: http://www.altera.com. Acessado em 09 de julho de 2008.

[44] ARM© Corporation Ltd. Disponível em http://www.arm.com. Acessado em 25 de junho de 2008.

[45] SIMPLESCALAR© LLC. Disponível em http://www.simplescalar.com. Acessado em 22 de março de 2008.

[46] SPEC© Corporation. Disponível em http://www.spec.org. Acessado em 03 de abril de 2008.

[47] XILINX©, Inc. Disponível em: http://www.xilinx.com. Acessado em 09 de julho de 2008.

CÓDIGOS EM ASSEMBLY

[48] Código Assembly compilado para o ARM9 do *MiBench* CRC32. Disponível em: http://www.efn.org/~rick/work/. Acessado em 17 de fevereiro de 2009.

[49] Código Assembly compilado para o ARM9 do *MiBench* JPEG. Disponível em: http://www.zophar.net/roms/files/gba/supersnake.zip. Acessado em 17 de fevereiro de 2009.

[50] Código Assembly compilado para o ARM9 do *MiBench* QuickSort. Disponível em: http://ww.shruta.net/download/archives/project/report/5/5.2/ARM9. Acessado em 17 de fevereiro de 2009.

[51] Código Assembly compilado para o ARM9 do *MiBench* SHA1. Disponível em: http://www.openssl.org/. Acessado em 17 de fevereiro de 2009.

APÊNDICE A

A.1. Descrição dos Componentes da Arquitetura com Variação para o método de *Huffman*

Para a implementação da arquitetura PDCCM usando o método de *Huffman* na compressão/descompressão das instruções foram criados os seguintes componentes em VHDL:

- **LAT** (Tabela da Linha de Endereços - *Line Address Table*): é uma tabela que tem por função fazer o mapeamento dos endereços das instruções na *I-cache* e informar se as instruções estão ou não comprimidas. Essa tabela contém um *bit* chamado de *type_inst* que sinaliza se a instrução contida na linha da *I-cache* está ou não comprimida. A LAT deve ter o mesmo tamanho da *I-cache*. A Figura A.1 mostra um trecho da LAT usando o método de *Huffman*, onde se observa que no campo *addr_inst* a primeira instrução encontra-se armazenada no endereço $000fff_h$. No campo *type_inst* observa se que para a primeira instrução o *bit flag* foi setado em 1 (um) significando que está instrução encontra-se salva de forma comprimida na *I-cache*.

addr_old	*type_inst*
0x000fff	1
0x000ffe	1
0x000ffd	0
0x000ffc	0
...	...

Figura A.1 – Exemplo de um trecho da LAT usando o método de *Huffman*

O Código Fonte A.1 apresenta a descrição da LAT em VHDL, onde pode-se notar os tipos de dados juntamente com os seus respectivos tamanhos. Os tipos de dados são: “natural” e “bit” e as variáveis de mais interesse neste componente são: `addr_inst` e `type_inst`.

Código Fonte A.1 - Descrição da LAT usando o método de *Huffman* - VHDL

```
type reg_LAT is record
      indice_LAT  : natural range (SIZE_CACHE - 1) downto 0;
      addr_inst   : natural range SIZE_ADDR downto 0;
      type_inst   : bit;
end record;
```

```
type vetor_LAT is array (0 to (SIZE_CACHE - 1)) of reg_LAT;
```

- **HT** (Tabela de *Huffman* - *Huffman Table*): é uma tabela que contém informações das instruções, tais como: código da instrução não comprimida, freqüência de repetições da instrução na *I-cache* e código de *Huffman* da instrução comprimida. O descompressor utiliza a HT para fazer todo o processo de descompressão das instruções comprimidas. A HT deve ter o mesmo tamanho da *I-cache*. A Figura A.2 mostra um trecho da HT usando o método de *Huffman*;

codInst_HT	*freq_HT*	*codHuff_HT*
00000000000000001010101010101010	5	11111111111111111111111111111111
00000000000000000000000000000011	3	01111111111111111111111111111111
10101010101010101111111111111111	1	00111111111111111111111111111111
10000011011111001111111100000000	1	00011111111111111111111111111111
...	...	...

Figura A.2 – Exemplo de um trecho da HT usando o método de *Huffman*

O Código Fonte A.2 apresenta a descrição da HT em VHDL, onde pode-se notar os tipos de dados juntamente com os seus respectivos tamanhos. Os tipos de dados para a HT são: "natural" e "wordT", sendo que o tipo wordT é um subtipo do tipo "bit_vector", ou seja, vetor de *bits*. As variáveis de mais interesse neste componente são: `codInst_HT`, `freq_HT` e `codHuff_HT`.

Código Fonte A.2 - Descrição da HT usando o método de *Huffman* - VHDL

```
type reg_HT is record
      indice_HT   : natural range (SIZE_CACHE - 1) downto 0;
      codInst_HT  : wordT;
      freq_HT     : natural;
      codHuff_HT  : wordT;
end record;

type vetor_HT is array (0 to (SIZE_CACHE - 1)) of reg_HT;
```

- **Compressor**: tem por função fazer a compressão de todos os códigos das instruções que possuem freqüência > 1, ou seja, códigos de instruções que se

repetem várias vezes na *I-cache*. O compressor é acionado toda vez que a *I-cache* não tiver mais linhas vagas (*I-cache* cheia).

O processo de compressão é realizado da seguinte forma: primeiramente é verificado código a código de instrução na linha da *I-cache* se o mesmo já está ou não contido na HT, caso a linha da *I-cache* verificada já esteja contida na HT, então o contador de freqüência da instrução (*freq_HT*) é incrementado em 1, se não uma variável interna da função de compressão (*line_instruction*) recebe o valor (do código da instrução) da linha da *I-cache* que está sendo analisada. Então, todas as instruções da *I-cache* são verificadas e comparadas com a variável *line_instruction* para ver se existe ou não mais instruções repetidas na *I-cache*. A variável *line_instruction* é salva na HT juntamente com a sua quantidade de repetições na *I-cache*.

Depois de serem pesquisadas todas as instruções da *I-cache* é realizada uma ordenação na HT, posicionando de forma decrescente as instruções que mais se repetem, ou seja, que possuem uma freqüência maior na *I-cache*.

Por fim, são salvos na *I-cache* os novos valores correspondentes da árvore de *Huffman* para cada instrução que possui freqüência maior que 1 (*freq_HT* > 1) e a LAT é atualizada.

O Código Fonte A.3 apresenta a função de compressão descrita em VHDL que faz todo o processo de compressão das instruções através do método de *Huffman*. Os tipos de dados para a função de compressão são: “natural”, “bit” e os subtipos “vetor_iCACHE”, “vetor_LAT”, “vetor_HT”, “reg_HT” e “wordT”. A variável de mais interesse nesta função é a `line_instruction` (contém a instrução que está sendo analisada e a sua freqüência de repetição na *I-cache*).

Código Fonte A.3 - Descrição da Função de Compressão usando o método de *Huffman* VHDL

```
function CompressorHuff(iCACHE      : vetor_iCACHE;
                        LAT         : vetor_LAT;
                        HT          : vetor_HT;
                        ind1        : natural)
return vetor_iCACHE is
```

```
variable cach           : vetor_iCACHE;
variable tb_lat         : vetor_LAT;
variable tb_h           : vetor_HT;
variable aux_tb_h       : reg_HT;
variable line_instruction : wordT;
variable cont           : natural := 0;
variable id1            : natural := 0;
variable ok             : bit;
variable comp           : bit := '0';

begin
   cach     := iCACHE;
   tb_lat   := LAT;
   tb_h     := HT;
   id1      := ind1;

   for i in 0 to (SIZE_CACHE - 1) loop
     if (tb_lat(i).type_inst /= '1') then
       ok := '0';
       for k in 0 to (SIZE_CACHE - 1) loop
            if (iCACHE(i).dado_memoria_icache= tb_h(k).codInst_HT) then
              ok := '1';
              if (tb_lat(i).type_inst = '1') then
                tb_h(cont).freq_HT := tb_h(cont).freq_HT + 1;
              end if;
            end if;
       end loop;

       if (ok = '0') then
            line_instruction := iCACHE(i).dado_memoria_icache;
            for x in 0 to (SIZE_CACHE - 1) loop
              if (line_instruction = iCACHE(x).dado_memoria_icache) then
                   tb_h(cont).codInst_HT := line_instruction;
                   tb_h(cont).freq_HT := tb_h(cont).freq_HT + 1;
              end if;
            end loop;
            cont := cont + 1;
       end if;
     end if;
   end loop;

   for i in 1 to (SIZE_CACHE - 1) loop
     if (tb_h(i).freq_HT > tb_h(i-1).freq_HT) then
       aux_tb_h.codInst_HT := tb_h(i-1).codInst_HT;
       aux_tb_h.freq_HT    := tb_h(i-1).freq_HT;

       tb_h(i-1).codInst_HT := tb_h(i).codInst_HT;
       tb_h(i-1).freq_HT    := tb_h(i).freq_HT;

       tb_h(i).codInst_HT := aux_tb_h.codInst_HT;
       tb_h(i).freq_HT    := aux_tb_h.freq_HT;
     end if;
   end loop;

     -- valores da árvore de Huffman
     tb_h(0).codHuff_HT  := "11111111111111111111111111111111";
     tb_h(1).codHuff_HT  := "01111111111111111111111111111111";
     tb_h(2).codHuff_HT  := "00111111111111111111111111111111";
     tb_h(3).codHuff_HT  := "00011111111111111111111111111111";
     tb_h(4).codHuff_HT  := "00001111111111111111111111111111";
     tb_h(5).codHuff_HT  := "00000111111111111111111111111111";
```

```
    tb_h(6).codHuff_HT  := "00000011111111111111111111111111";
    tb_h(7).codHuff_HT  := "00000001111111111111111111111111";
    tb_h(8).codHuff_HT  := "00000000111111111111111111111111";
    tb_h(9).codHuff_HT  := "00000000011111111111111111111111";
    tb_h(10).codHuff_HT := "00000000001111111111111111111111";
    tb_h(11).codHuff_HT := "00000000000111111111111111111111";
    tb_h(12).codHuff_HT := "00000000000011111111111111111111";
    tb_h(13).codHuff_HT := "00000000000001111111111111111111";
    tb_h(14).codHuff_HT := "00000000000000111111111111111111";
    tb_h(15).codHuff_HT := "00000000000000011111111111111111";
    tb_h(16).codHuff_HT := "00000000000000001111111111111111";
    tb_h(17).codHuff_HT := "00000000000000000111111111111111";
    tb_h(18).codHuff_HT := "00000000000000000011111111111111";
    tb_h(19).codHuff_HT := "00000000000000000001111111111111";
    tb_h(20).codHuff_HT := "00000000000000000000111111111111";
    tb_h(21).codHuff_HT := "00000000000000000000011111111111";
    tb_h(22).codHuff_HT := "00000000000000000000001111111111";
    tb_h(23).codHuff_HT := "00000000000000000000000111111111";
    tb_h(24).codHuff_HT := "00000000000000000000000011111111";
    tb_h(25).codHuff_HT := "00000000000000000000000001111111";
    tb_h(26).codHuff_HT := "00000000000000000000000000111111";
    tb_h(27).codHuff_HT := "00000000000000000000000000011111";
    tb_h(28).codHuff_HT := "00000000000000000000000000001111";
    tb_h(29).codHuff_HT := "00000000000000000000000000000111";
    tb_h(30).codHuff_HT := "00000000000000000000000000000011";
    tb_h(31).codHuff_HT := "00000000000000000000000000000001";

    for i in 0 to (SIZE_CACHE - 1) loop
      if (tb_h(i).freq_HT > 1) then
        cach(i).dado_memoria_icache := tb_h(i).codHuff_HT;
        tb_lat(i).type_inst := '1';
      else
        cach(i).dado_memoria_icache := tb_h(i).codInst_HT;
        tb_lat(i).type_inst := '0';
      end if;
      tb_lat(i).addr_inst := cach(i).addr_memoria_icache;
    end loop;

    id1 := cont - 1;

    return cach;
end function;
```

- **Descompressor**: tem por função fazer a descompressão de todas as instruções comprimidas que estão armazenadas na *I-cache* e serão repassadas ao processador. O descompressor é acionado toda vez que a LAT for consultada e retornar um ACERTO e o pino *type_inst* da LAT esteja setado com o valor 1 (nível alto).

O processo de descompressão é realizado da seguinte forma: a LAT indica qual o endereço da instrução na *I-cache* e se o pino *type_inst* está setado com o valor 0 (instrução

normal) ou 1 (instrução comprimida). Se *type_inst* = 1, então verifica-se na *I-cache* qual o valor da instrução correspondente ao endereço passado pelo processador. Logo em seguida, é chamada a função de descompressão passando como parâmetro a HT e a instrução comprimida. Uma variável (*index*) interna da função de descompressão é criada e atribuído um valor obtido através da consulta feita com a instrução comprimida para saber o seu posicionamento na árvore de *Huffman*. Então, uma nova variável (*instruction*) também interna da função de descompressão é criada e atribuído o valor do código da instrução da HT no posicionamento *index* (HT(*index*).*codInst_HT*). Assim, o descompressor descomprime a instrução e a passa de forma descomprimida para o processador.

O Código Fonte A.4 apresenta a função de descompressão descrita em VHDL que faz todo o processo de descompressão das instruções. Os tipos de dados para a função de descompressão são: "natural" e os subtipos "vetor_HT" e "wordT". A variável de mais interesse nesta função é a `instruction` (forma a instrução descomprimida, ou seja, instrução com o código original).

Código Fonte A.4 - Descrição da Função de Descompressão usando o método de *Huffman* VHDL

```
function DescompressorHuff(dado_icache          : wordT;
                           HT                   : vetor_HT)
return wordT is

variable tb_h           : vetor_HT;
variable instruction    : wordT;
variable index          : natural;

begin
      tb_h := HT;

      if (dado_icache = "11111111111111111111111111111111") then
            index := 0;

      elsif (dado_icache = "01111111111111111111111111111111") then
            index := 1;

      elsif (dado_icache = "00111111111111111111111111111111") then
            index := 2;

      elsif (dado_icache = "00011111111111111111111111111111") then
            index := 3;

      elsif (dado_icache = "00001111111111111111111111111111") then
            index := 4;

      elsif (dado_icache = "00000111111111111111111111111111") then
```

```
      index := 5;

elsif (dado_icache = "00000011111111111111111111111111") then
      index := 6;

elsif (dado_icache = "00000001111111111111111111111111") then
      index := 7;

elsif (dado_icache = "00000000111111111111111111111111") then
      index := 8;

elsif (dado_icache = "00000000011111111111111111111111") then
      index := 9;

elsif (dado_icache = "00000000001111111111111111111111") then
      index := 10;

elsif (dado_icache = "00000000000111111111111111111111") then
      index := 11;

elsif (dado_icache = "00000000000011111111111111111111") then
      index := 12;

elsif (dado_icache = "00000000000001111111111111111111") then
      index := 13;

elsif (dado_icache = "00000000000000111111111111111111") then
      index := 14;

elsif (dado_icache = "00000000000000011111111111111111") then
      index := 15;

elsif (dado_icache = "00000000000000001111111111111111") then
      index := 16;

elsif (dado_icache = "00000000000000000111111111111111") then
      index := 17;

elsif (dado_icache = "00000000000000000011111111111111") then
      index := 18;

elsif (dado_icache = "00000000000000000001111111111111") then
      index := 19;

elsif (dado_icache = "00000000000000000000111111111111") then
      index := 20;

elsif (dado_icache = "00000000000000000000011111111111") then
      index := 21;

elsif (dado_icache = "00000000000000000000001111111111") then
      index := 22;

elsif (dado_icache = "00000000000000000000000111111111") then
      index := 23;

elsif (dado_icache = "00000000000000000000000011111111") then
      index := 24;

elsif (dado_icache = "00000000000000000000000001111111") then
      index := 25;
```

```
      elsif (dado_icache = "00000000000000000000000000111111") then
            index := 26;

      elsif (dado_icache = "00000000000000000000000000011111") then
            index := 27;

      elsif (dado_icache = "00000000000000000000000000001111") then
            index := 28;

      elsif (dado_icache = "00000000000000000000000000000111") then
            index := 29;

      elsif (dado_icache = "00000000000000000000000000000011") then
            index := 30;

      elsif (dado_icache = "00000000000000000000000000000001") then
            index := 31;
      end if;

      instruction := tb_h(index).codInst_HT;

      return instruction;
end function;
```

A.2. Instruções na *I-cache* usando o método de *Huffman*

Todas as instruções que são salvas na *I-cache* podem ou não estar comprimidas, porém, o tamanho de cada instrução na *I-cache* é sempre 32 *bits*. Verificando as linhas da *I-cache* não é possível afirmar se uma determinada instrução está ou não comprimida, então um *bit* na LAT (`type_inst`) informará qual o status da instrução na linha da *I-cache*. A Figura A.3 mostra um trecho da *I-cache* com instruções não comprimidas e a Figura A.4 mostra um trecho da *I-cache* com instruções comprimidas e não comprimidas.

address I-cache	Instruções Normais (32 *bits*)
0x003c	11111111111111111111111111111111
0x003b	01111111111111111111111111111111
0x003a	00111111111111111111111111111111
0x0039	00011111111111111111111111111111
0x0038	10101010101010101111111111111111
0x0037	00000000000000001111111010101010
...	...

Figura A.3 – *I-cache* não comprimida usando o método de *Huffman*

address I-cache	Instruções Comprimidas e Normais (32 *bits*)
0x003c	11111111111111111111111111111111
0x003b	01111111111111111111111111111111
0x003a	00111111111111111111111111111111
0x0039	00011111111111111111111111111111
0x0038	10101010101010101111111111111111
0x0037	00000000000000001111111010101010
...	...

Figura A.4 – *I-cache* comprimida usando o método de *Huffman*

Observa-se nas Figuras A.3 e A.4 que os códigos das instruções são iguais; na Figura A.3 contém apenas instruções normais (não comprimidas) e na Figura A.4 as instruções das linhas da *I-cache* $003c_h$, $003b_h$, $003a_h$ e 0039_h são instruções comprimidas usando o método de *Huffman*. O que definirá se uma instrução na linha da *I-cache* está ou não comprimida é o sinal do pino `type_inst` da LAT.

Uma das vantagens de aplicar esse tipo de compressão é justamente a possibilidade de verificar as instruções que mais se repetem na *I-cache*, agrupá-las em uma única linha na *I-cache* atribuindo um novo código para esse conjunto de instruções idênticas (repetidas). Assim é possível aumentar a quantidade de instruções na *I-cache*, lembrando que em sistemas embarcados esse tipo de recurso é muito limitado.

APÊNDICE B

Neste apêndice temos uma tabela que contém o conjunto de instruções para o processador embarcado ARM9 [37] no qual foram atribuídos valores distintos em binários (32 *bits*) para cada instrução.

Características do processador embarcado

Fabricante: ARM

Família: ARM9

Versão: ARM922T

ISA: ARMv4T

Operação	Instrução	Descrição da Instrução	Valor em Binário da Instrução
Aritméticos	**ADD**	Soma	00000000000000001010101010101010
	ADC	Soma com *carry*	00000000000000001110101010111011
	SUB	Subtração	00000000000000000000000000000011
	SBC	Subtração com *carry*	00000000000000000000000000110011
	RSB	Subtração reversa	00000000000000000000001100110011
	RSC	Subtração reversa com *carry*	00000000000000000000001111000011
	MUL	Multiplica	10101010000011111111000001010101
	MLA	Multiplica com acumulação	10101010000011111111000001111111
	UMULL	Multiplicação (32 *bits* não sinalizados)	10101010000011111111001001110111
	UMLAL	Multiplicação com acumulação (32 *bits* não sinalizados)	10101010111111111111000001010101
	SMULL	Multiplicação (32 *bits* sinalizados)	10101010111100001111000001010101
	SMLAL	Multiplicação com acumulação (32 *bits* sinalizados)	10101010000000001111000001010111
	SUBS		11000000000000000000000000011111
	ADDS		10000000000001010101010101010101
Mover	**MOV**	Mover	10101010101010101111111111111111
	MVN	Mover, efetuando um complemento	10101010101010101111111111111000
	MRS	Mover SPSR ou CPSR para o registrador	10101010101010100001111111111111
	MSR	Mover registrador para SPSR ou CPSR para o registrador. Mover *flags* de SPSR ou CPSR	10101010101010100001111111111000
	MOVS		11101010101010101111111111111100
	MOVEQ		11111010101010101111111111111110
Lógicos	**CMP**	Comparação	00000000000000000000011111111000
	CMN	Comparação negativa	00000000001111111100011111111000
	TST	Teste	11000110111000000111111100111000
	TEQ	Teste equivalente	11000111111000000111111111111000
	AND	Lógica E	11100011100000000111111100111000
	EOR	Lógica OU-exclusivo	11110000000000001111111111110000
	ORR	Lógica OU	01110000000000000011100001111110
	BIC	Limpa *bit*	00011000110000111111111000111001
	ANDS		11110011110000000111111100111111
	CMPS		00000000000000000111111111111000
Desvio (*Branch*)	**B**	Desvia	00000000000011111111111111110000
	BL	Desvia com link	00000000000000111111111111000000
	BX	Desvia para o endereço	00000000000011001111111100110000
	BNE	Desvia se Z=0	10000000000000111111111111100000

	BEQ	Desvia se Z=1	00000000000000001111111111110011
	BLT	Desvia se N=1 e V=0 ou se N=0 e v=1	00000000000001111111111100110011
	BGT	Desvia se Z=0 e N=1 ou V=1, ou se Z=0 e N=0 ou V=0	11110000000000001111111100110011
	BXNE		11000000000011001111111100111100
Carga de Constante	**LDR**	Carrega constante	11111000001100111111111000001100
	LDRB*	Carrega um byte (não sinalizado)	11111000001100111110000000001111
	LDRH*	Carrega metade de uma palavra (não sinalizado)	00111110001100001111111000001100
	LDRSB*	Carrega um byte (sinalizado)	00001000001100111111111110001110
	LDRSH*	Carrega metade de uma palavra (sinalizado)	10110110001100111111111000011110
Bloco múltiplo de operações com dados	**LDM**	Carrega dados	11110000000000000000110000000011
	LDMIA	Carrega múltiplos endereços	10110000000000000011110000000011
	LDMFD		00110000000000000001110000000111
	LDMEA		01110000000000000000110000001111
Acumulação	**STR**	Armazena dado	00001111111111110000000000001111
	STRB*	Armazena *byte*	00001111000011110000000000001111
	STRH*	Armazena palavra	00001111111111100000000000111100
	STM	Armazena (requer parâmetro adicional)	00000011110000110000000000000011
	STMIA		00000011111111110000000000000111
	STMFD		11000011110000111100000000110011
	STMDB		00000011110001110000000000111011
Inversão	**SWP**	Inversão	11110000000011110000000011110000
	SWPB*	Inversão de *byte*	11000000000000111100000011110000
Co-processador	**CDP**	Operação de dados	01111111111110000001100000001111
	MRC	Mover registrador ARM para co-processador	01111111111000000001100000000111
	MCR	Mover co-processador para registrador ARM	00011111110000000001100000001111
	LDC	Carrega	01011001111110000001100000001101
	STC	Acumula	01111111111100000001111000001011
Interrupção por SW	**SWI**	Interrupção por software	11000000000000000000111111000111
Thumb	**STFD**		11111100000011111111111111111111
	DVFD		11111100000011111111111111110000
	LDFD		00001100000011111111111111111111
	LFTD		00111100000011111111111111110000

APÊNDICE C

Neste apêndice temos os códigos em Assembly compilado para o ARM9 dos *benchmark MiBench* usados nas simulações de compressão e descompressão dos métodos *MIC* e *Huffman* para a arquitetura PDCCM.

C.1. Código Assembly da execução do *MiBench* CRC32

```
/*  linux/arch/arm/lib/crc32.S
 *  Copyright (C) 2005 Rick Bronson
 *  ASM optimised crc32 function */

#include <linux/linkage.h>
#include <asm/assembler.h>

      .extern crc_table

            .text

#define ENTER  \
            mov     ip, sp  ;\
            stmfd   sp!, {r4-r9, sl, fp, ip, lr, pc}       ;\
            sub     fp, ip, #4

#define EXIT   \
            LOADREGS(ea, fp, {r4 - r9, sl, fp, sp, pc})

/* Prototype: ulong crc32 (ulong crc, const unsigned char *, uint
size); registers: pc = r15, lr = r14, sp = r13, ip = r12, fp = r11, sl =
r10 */
ENTRY(crc32asm)
      ENTER
      ldr     r3, crctab            /*  let r3 point to the crc table */

odd:    ands    r4, r1, #3         /*  on an odd boundry? */
      beq     even
      subs    r2, r2, #1
      blt     out
      ldrb    r4, [r1], #1          /*   do up to 3 bytes here */
      eor     r4, r0, r4
      and     r4, r4, #255          /*  0xff */
      ldr     r4, [r3, r4, lsl #2]
      eor     r0, r4, r0, lsr #8
      b       odd

evenlp: ldmia   r1!,{r4 - r9, sl, ip}   /*  grab 8 longs, 32 bytes */
/*  crc = crc_table[((int)crc ^ (*buf++)) & 0xff] ^ (crc >> 8) */

      eor     lr, r0 ,r4            /*  do 1st byte of r4 */
      and     lr, lr, #255          /*  0xff */
      ldr     lr, [r3, lr, lsl #2]
      eor     r0, lr, r0, lsr #8

      eor     lr, r0 ,r4, lsr #8     /*  do 2nd byte */
      and     lr, lr, #255          /*  0xff */
      ldr     lr, [r3, lr, lsl #2]
      eor     r0, lr, r0, lsr #8

      eor     lr, r0 ,r4, lsr #16    /*  do 3rd byte */
      and     lr, lr, #255           /*  0xff */
      ldr     lr, [r3, lr, lsl #2]
      eor     r0, lr, r0, lsr #8

      eor     lr, r0 ,r4, lsr #24    /*  do 4th byte */
      and     lr, lr, #255           /*  0xff */
      ldr     lr, [r3, lr, lsl #2]
      eor     r0, lr, r0, lsr #8

      eor     lr, r0 ,r5            /*  do 1st byte of r5 */
      and     lr, lr, #255          /*  0xff */
      ldr     lr, [r3, lr, lsl #2]
      eor     r0, lr, r0, lsr #8

      eor     lr, r0 ,r5, lsr #8     /*  do 2nd byte */
      and     lr, lr, #255          /*  0xff */
      ldr     lr, [r3, lr, lsl #2]
      eor     r0, lr, r0, lsr #8

      eor     lr, r0 ,r5, lsr #16    /*  do 3rd byte */
      and     lr, lr, #255          /*  0xff */
      ldr     lr, [r3, lr, lsl #2]
      eor     r0, lr, r0, lsr #8

      eor     lr, r0 ,r5, lsr #24    /*  do 4th byte */
      and     lr, lr, #255          /*  0xff */
      ldr     lr, [r3, lr, lsl #2]
      eor     r0, lr, r0, lsr #8

      eor     lr, r0 ,r6            /*  do 1st byte of r6 */
      and     lr, lr, #255          /*  0xff */
      ldr     lr, [r3, lr, lsl #2]
      eor     r0, lr, r0, lsr #8

      eor     lr, r0 ,r6, lsr #8     /*  do 2nd byte */
      and     lr, lr, #255          /*  0xff */
      ldr     lr, [r3, lr, lsl #2]
      eor     r0, lr, r0, lsr #8

      eor     lr, r0 ,r6, lsr #16    /*  do 3rd byte */
      and     lr, lr, #255          /*  0xff */
      ldr     lr, [r3, lr, lsl #2]
      eor     r0, lr, r0, lsr #8

      eor     lr, r0 ,r6, lsr #24    /*  do 4th byte */
      and     lr, lr, #255          /*  0xff */
      ldr     lr, [r3, lr, lsl #2]
      eor     r0, lr, r0, lsr #8

      eor     lr, r0 ,r7            /*  do 1st byte of r7 */
      and     lr, lr, #255          /*  0xff */
      ldr     lr, [r3, lr, lsl #2]
      eor     r0, lr, r0, lsr #8

      eor     lr, r0 ,r7, lsr #8     /*  do 2nd byte */
      and     lr, lr, #255          /*  0xff */
      ldr     lr, [r3, lr, lsl #2]
      eor     r0, lr, r0, lsr #8

      eor     lr, r0 ,r7, lsr #16    /*  do 3rd byte */
      and     lr, lr, #255          /*  0xff */
      ldr     lr, [r3, lr, lsl #2]
      eor     r0, lr, r0, lsr #8

      eor     lr, r0 ,r7, lsr #24    /*  do 4th byte */
      and     lr, lr, #255          /*  0xff */
      ldr     lr, [r3, lr, lsl #2]
```

```
    eor    r0, lr, r0, lsr #8
    eor    lr, r0 ,r8          /*  do 1st byte of r8 */
    and    lr, lr, #255        /*  0xff */
    ldr    lr, [r3, lr, lsl #2]
    eor    r0, lr, r0, lsr #8

    eor    lr, r0 ,r8, lsr #8    /*  do 2nd byte */
    and    lr, lr, #255         /*  0xff */
    ldr    lr, [r3, lr, lsl #2]
    eor    r0, lr, r0, lsr #8

    eor    lr, r0 ,r8, lsr #16   /*  do 3rd byte */
    and    lr, lr, #255         /*  0xff */
    ldr    lr, [r3, lr, lsl #2]
    eor    r0, lr, r0, lsr #8

    eor    lr, r0 ,r8, lsr #24   /*  do 4th byte */
    and    lr, lr, #255         /*  0xff */
    ldr    lr, [r3, lr, lsl #2]
    eor    r0, lr, r0, lsr #8

    eor    lr, r0 ,r9          /*  do 1st byte of r9 */
    and    lr, lr, #255        /*  0xff */
    ldr    lr, [r3, lr, lsl #2]
    eor    r0, lr, r0, lsr #8

    eor    lr, r0 ,r9, lsr #8    /*  do 2nd byte */
    and    lr, lr, #255         /*  0xff */
    ldr    lr, [r3, lr, lsl #2]
    eor    r0, lr, r0, lsr #8

    eor    lr, r0 ,r9, lsr #16   /*  do 3rd byte */
    and    lr, lr, #255         /*  0xff */
    ldr    lr, [r3, lr, lsl #2]
    eor    r0, lr, r0, lsr #8

    eor    lr, r0 ,r9, lsr #24   /*  do 4th byte */
    and    lr, lr, #255         /*  0xff */
    ldr    lr, [r3, lr, lsl #2]
    eor    r0, lr, r0, lsr #8

    eor    lr, r0, sl          /*  do 1st byte of sl */
    and    lr, lr, #255        /*  0xff */
    ldr    lr, [r3, lr, lsl #2]
    eor    r0, lr, r0, lsr #8

    eor    lr, r0, sl, lsr #8    /*  do 2nd byte */
    and    lr, lr, #255         /*  0xff */
    ldr    lr, [r3, lr, lsl #2]
    eor    r0, lr, r0, lsr #8

    eor    lr, r0, sl, lsr #16   /*  do 3rd byte */
    and    lr, lr, #255         /*  0xff */
    ldr    lr, [r3, lr, lsl #2]
    eor    r0, lr, r0, lsr #8

    eor    lr, r0, sl, lsr #24   /*  do 4th byte */
    and    lr, lr, #255         /*  0xff */
    ldr    lr, [r3, lr, lsl #2]
    eor    r0, lr, r0, lsr #8

    eor    lr, r0, ip          /*  do 1st byte of ip */
    and    lr, lr, #255        /*  0xff */
    ldr    lr, [r3, lr, lsl #2]
    eor    r0, lr, r0, lsr #8

    eor    lr, r0, ip, lsr #8    /*  do 2nd byte */
    and    lr, lr, #255         /*  0xff */
    ldr    lr, [r3, lr, lsl #2]
    eor    r0, lr, r0, lsr #8

    eor    lr, r0, ip, lsr #16   /*  do 3rd byte */
    and    lr, lr, #255         /*  0xff */
    ldr    lr, [r3, lr, lsl #2]
    eor    r0, lr, r0, lsr #8

    eor    lr, r0, ip, lsr #24   /*  do 4th byte */
    and    lr, lr, #255         /*  0xff */
    ldr    lr, [r3, lr, lsl #2]
    eor    r0, lr, r0, lsr #8

even:  subs   r2, r2, #32    /*  can we do 32 bytes at a time? */
    bge    evenlp          /*  keep going if we have 32 */

    adds   r2, r2, #32    /*  add back 32 to get remainder */
    ble    out             /*  any left? */

          /* ; do one byte at a time */
one:   ldrb   r4, [r1], #1         /*   */
    eor    r4, r0, r4
    and    r4, r4, #255         /*  0xff */
    ldr    r4, [r3, r4, lsl #2]
    eor    r0, r4, r0, lsr #8
    subs   r2, r2, #1     /*  0x1 */
    bne    one
out:   EXIT

crctab: .long  SYMBOL_NAME(crc_table)

          .align
```

C.2. Código Assembly da execução do *MiBench* JPEG

```
@ jpeg.s
@
@ Support GAS assembly code for jpeg.c and possibly later
@ versions. v1.0318 - Original version

    .ARM
    .ALIGN
    .GLOBL  JPGGetByte

@ Entry: -
@ Exit: r0 = 8bit value

JPGGetByte:
    ldr    r2,_JPGfindex
    ldr    r1,[r2]
    ldrb   r0,[r1],#1
    str    r1,[r2]
    bx     lr

    .GLOBL  JPGGetWord

@ Entry: -
@ Exit: r0 = 16bit value
```

```
JPGGetWord:
    ldr    r2,_JPGfindex
    ldr    r0,[r2]
    add    r1,r0,#2
    str    r1,[r2]
    ldrb   r1,[r0],#1
    ldrb   r0,[r0]
    orr    r0,r0,r1,lsl #8
    bx     lr

_JPGfindex:
    .word  JPGfindex

    .GLOBL JPGNextBit

@ Entry: -
@ Exit: r0 = 1bit value (0 or 1)

JPGNextBit:
    ldr    r1,0f
    ldrb   r0,[r1]
    add    r2,r0,r0
    strb   r2,[r1]          @ curByte <<= 1
    mov    r0,r0,lsr #7   @ return = CurByte >> 7;
    ldr    r1,1f
    ldrb   r2,[r1]
    movs   r2,r2,lsr #1
    strb   r2,[r1]       @ curBits >>= 1
    bxne   lr            @ curBits !=0, so exit

    stmfd  sp!,{r0}
    mov    r2,#128
    strb   r2,[r1]       @ curBits = 128

@ Get a byte
    ldr    r2,_JPGfindex
    ldr    r1,[r2]
    ldrb   r0,[r1],#1
    str    r1,[r2]       @ r0 = next byte

    ldr    r1,0f
    strb   r0,[r1]       @ curByte = JPGGetByte();
    cmps   r0,#255
    bne    9f
@ curByte == 255

@ Get a byte
    ldr    r1,[r2]
    ldrb   r0,[r1],#1
    str    r1,[r2]       @ r0 = next byte

    cmps   r0,#0xd9
    bne    9f

    ldr    r1,_JPGEOI
    mov    r0,#1
    strb   r0,[r1]       @ EOI = 1

    ldmfd  sp!,{r0}
    mov    r0,#0
    bx     lr

9:
    ldmfd  sp!,{r0}
    bx     lr

0:    .word  JPGcurByte
1:    .word  JPGcurBits
_JPGEOI:
    .word  JPGeoi

    .GLOBL JPGClear128Bytes

@ Entry: r0 = array address
@ Exit: -

JPGClear128Bytes:
    stmfd  sp!,{r4}
    mov    r1,#0
    mov    r2,#0
    mov    r3,#0
    mov    r4,#0
    stmia  r0!,{r1-r4}
    stmia  r0!,{r1-r4}
    stmia  r0!,{r1-r4}
    stmia  r0!,{r1-r4}
    stmia  r0!,{r1-r4}
    stmia  r0!,{r1-r4}
    stmia  r0!,{r1-r4}
    stmia  r0!,{r1-r4}
    ldmfd  sp!,{r4}
    bx     lr

    .GLOBL JPGDecode

@ r0 = &inArray
@ r1 = temp
@ r2 = &EOI
@ r3 = index & curVal
@ r4 = cnt up 256
@ r5 = temp
@ r6 = cnt down 256
@ r7 = L
@ r8 = r0 save
@ r9 = MatchFound

@ Entry: r0 = Huffman table address
@ Exit:  r0 = coefficient

JPGDecode:
    stmfd  sp!,{r3-r9,r14}

    mov    r5,r0
    ldr    r3,_JPGfindex
    ldr    r4,[r3]

@ Get a byte
    mov    r1,r4
    ldrb   r0,[r1],#1
    str    r1,[r3]

    cmps   r0,#255
    strne  r4,[r3]          @ findex -= 1
    bne    _JD9

    bl     JPGGetByte
    cmps   r0,#0xd0
    strcc  r4,[r3]          @ findex -= 2
    bcc    _JD9

    cmps   r0,#0xd8
    strcs  r4,[r3]          @ findex -= 2
    bcs    _JD9

    ldr    r1,_HM11
    ldrb   r0,[r1]
    sub    r0,r0,#1         @ r0 = curBits - 1;

    ldr    r1,_HM12
    ldrb   r2,[r1]
    and    r1,r2,r0         @ r1 = curByte & n2
    cmps   r1,r0
```

```
    bne    _JD9

    ldr    r1,_JPGEOI
    mov    r0,#1
    strb   r0,[r1]          @ EOI = 1

    b      _HM8

_JD9:
    mov    r0,r5

    mov    r9,#-1         @ MatchFound = -1
    mov    r8,r0

    mov    r3,#0
    mov    r7,#1
_HM0:
    bl     JPGNextBit
    add    r3,r0,r3,lsl #1              @ CurVal = (CurVal << 1) +
                                        @ JPGNextBit()

    mov    r0,r8
    ldr    r2,_JPGEOI

    ldrb   r5,[r2]          @ r8 = EOI
    cmps   r5,#0
    bne    _HM8

    mov    r6,#256
    mov    r4,#0
_HM1:
    ldrh   r5,[r0,#4]        @ r5 = inArray[i].Length
    cmps   r7, r5
    bcc    _HM6
    bne    _HM3

    ldrh   r5,[r0]          @ r5 = inArray[i].Index
    cmps   r3,r5

    moveq  r9,r4
    beq    _HM6
_HM3:
    add    r0,r0,#6 @size of packed structure element is 6 bytes
    add    r4,r4,#1
    subs   r6,r6,#1
    bne    _HM1
_HM6:
    adds   r5,r9,#1
    cmps   r5,#0
    bne    _HM7

    add    r7,r7,#1
    cmps   r7,#17
    bne    _HM0

_HM7:
    mov    r2,r8
    mov    r0,r9
    ldmfd  sp!,{r3-r9,r14}
                        @ rtn(1)
    adds   r1,r0,#1
    bxeq   lr           @ if (MatchFound == -1) return(-1);

    add    r1,r0,r0,lsl #1
    add    r1,r1,r1
    add    r0,r1,#2      @ r1 = 6 * MatchFound + .Code
    ldrh   r0,[r2,r0]        @ return(inArray[MatchFound].Code)
    bx     lr

_HM8:
    ldmfd  sp!,{r3-r9,r14}    @ return(0)
    mov    r0,#0
    bx     lr

_HM11:  .word  JPGcurBits
_HM12:  .word  JPGcurByte

    .GLOBL  JPGReceiveBits

@ r0 = cat
@ r1 = power2(cat)
@ r2 = destroyed by JPGNextBit
@ r3 = temp0
@ r4 = cat
@ r5 = temp0 << 1
@ r6 = cat

@ Entry: r0 = bits
@ Exit:  r0 = bitVal

JPGReceiveBits:
    cmps   r0,#0
    bxeq   lr

    stmfd  sp!,{r1-r5,r14}
    mov    r4,r0
    mov    r5,r0
    mov    r3,#0
1:
    bl     JPGNextBit
    add    r3,r0,r3,lsl #1
    subs   r5,r5,#1
    bne    1b

    mov    r5,r3,lsl #1       @ r5 = temp0 << 1
    mov    r1,#1
    mov    r1,r1,lsl r4       @ r1 = 2 ^ (cat)

    mov    r0,r3
    cmps   r5,r1

    subcc  r0,r3,r1
    addcc  r0,r0,#1    @ return = -(JPGpower2(cat) - 1) + temp0

    ldmfd  sp!,{r1-r5,r14}
    bx     lr


    .GLOBL  JPGGetBlockBits

@ r0 = temp
@ r1 = destroyed by JPGDecode
@ r2 = destroyed by JPGDecode
@ r3 = zeros
@ r4 = bits
@ r5 = temp
@ r6 = HuffACNum
@ r7 = JPGZig1
@ r8 = JPGZig2
@ r9 = ZigIndex
@ r10 = &array2
@ r11 = ACCount

@ Entry: r0 = HuffACNum, r1 = &array2[]
@ Exit:  -

JPGGetBlockBits:
    stmfd  sp!,{r4-r11,r14}

    mov    r2,#11
    mov    r6,r0
    ldr    r7,_GBB3
```

```
    ldr    r8,_GBB4
    mov    r9,#1
    mov    r10,r1
    mov    r11,#1
1:
    cmps   r6,#0       @ if (HuffACNum)
    ldrne  r0,_GBB2 @   d = JPGDecode(HuffmanAC1)
                       @ else
    ldreq  r0,_GBB1 @   d = JPGDecode(HuffmanAC0)
    bl     JPGDecode

    mov    r3,r0,asr #4      @ zeros = d >> 4
    and    r4,r0,#0xf        @ bits = d & 15

    mov    r0,r4
    bl     JPGReceiveBits

    cmps   r4,#0
    beq    2f

    add    r9,r9,r3          @ ZigIndex += zeros
    add    r11,r11,r3        @ ACCount += zeros
    cmps   r11,#64
    bcs    4f

    ldrb   r5,[r7,r9]
    ldrb   r4,[r8,r9]
    add    r5,r4,r5,lsl #3
    add    r5,r5,r5
    strh   r0,[r10,r5]           @ array2[(JPGZig1[ZigIndex]<<3) +
                                 @ JPGZig2[ZigIndex]] = bitVal
    add    r9,r9,#1          @ ZigIndex++
    add    r11,r11,#1        @ ACCount++
    b      3f
2:
    cmps   r3,#15            @ if (zeros != 15) break
    bne    4f
    add    r9,r9,#15         @ ZigIndex += 15
    add    r11,r11,#16       @ ACCount += 16
3:
    cmps   r11,#64
    bcc    1b                @ while (ACCount < 64)
4:
    ldmfd  sp!,{r4-r11,r14}
    bx     lr

_GBB1:  .word  JPGHuffmanAC0
_GBB2:  .word  JPGHuffmanAC1
_GBB3:  .word  JPGZig1
_GBB4:  .word  JPGZig2

    .GLOBL  JPGDraw8x8

@ r0 = temp
@ r1 = temp
@ r2 = temp
@ r3 = vram address
@ r4 = &Vector[]
@ r5 = i offset
@ r6 = j offset
@ r7 = i count
@ r8 = j count
@ r9 = temp
@ r10 = temp
@ r11 = temp
@ r12 = temp

@ Entry: r0 = &YVector[], r1 = i, r2 = j, r3 = vram address
@ Exit: -

JPGDraw8x8:
    stmfd  sp!,{r4-r12}

    mov    r4,r0
    mov    r5,r1
    mov    r6,r2
    mov    r7,#0
    mov    r8,#0
1:
    add    r0,r8,r7,lsl #3
    add    r0,r0,r0
    ldrsh  r0,[r4,r0]        @ r0 = YVector[(i<<3)+j]

    add    r1,r5,r7
    mov    r1,r1,lsr #1      @ r1 = (i+io) >> 1
    add    r2,r6,r8
    mov    r2,r2,lsr #1      @ r2 = (j+jo) >> 1

    add    r10,r2,r1,lsl #3
    add    r10,r10,r10
    ldr    r12,_D88b
    ldrsh  r9,[r12,r10]      @ r9 = CbVector[(i2<<3)+j2]

    ldr    r12,_D88r
    ldrsh  r10,[r12,r10]     @ r10 = CrVector[(i2<<3)+j2]

    sub    r1,r10,#128
    mov    r11,#45
    mul    r2,r1,r11
    add    r11,r0,r2,asr #5@r11 = red = y + (((cr-128) * 45) >> 5)

@ Range limit red to 0-255

    cmps   r11,#0x80000000
    movcs  r11,#0
    cmps   r11,#0x100
    movcs  r11,#0xff

@ r12 = blue = y + (((cb-128) * 57) >> 5)

    sub    r1,r9,#128
    mov    r12,#57
    mul    r2,r1,r12
    add    r12,r0,r2,asr #5

@ Range limit blue to 0-255

    cmps   r12,#0x80000000
    movcs  r12,#0
    cmps   r12,#0x100
    movcs  r12,#0xff

@ r10 = green = y - (((cb-128) * 11) >> 5) - (((cr-128) * 23) >> 5)

    mov    r2,#11
    mul    r2,r1,r2
    sub    r9,r0,r2,asr #5     @ r9 = ((cb0))

    sub    r1,r10,#128
    mov    r2,#23
    mul    r2,r1,r2

    sub    r10,r9,r2,asr #5

@ range limit to 0-255

    cmps   r10,#0x80000000
    movcs  r10,#0
    cmps   r10,#0x100
    movcs  r10,#0xff

@ Convert RGB (24bit) to BGR (16bit)
```

```
@  r2 = ((((B) >> 3) << 10)+(((G) >> 3) << 5)+((R) >> 3))

    mov    r2,r11,lsr #3
    mov    r1,r10,lsr #3
    add    r2,r2,r1,lsl #5
    mov    r1,r12,lsr #3
    add    r2,r2,r1,lsl #10    @ r2 = 16bit color

@ Prepare x & y coordinates

    ldr    r12,_D88xi
    ldrh   r10,[r12]
    add    r0,r10,r8          @ xj
    add    r0,r0,r6

    ldr    r12,_D88yi
    ldrh   r11,[r12]
    add    r1,r11,r7          @ yi
    add    r1,r1,r5

@ Plot pixel at r0,r1 using color r2 at screen address r3

    mov    r10,#480
    mul    r11,r10,r1
    add    r11,r11,r0,lsl #1
    add    r10,r11,r3
    strh   r2,[r10]

    add    r8,r8,#1
    cmps   r8,#8
    bne    1b

    mov    r8,#0

    add    r7,r7,#1
    cmps   r7,#8
    bne    1b

    ldmfd  sp!,{r4-r12}
    bx     lr

_D88xi: .word  JPGXOrigin
_D88yi: .word  JPGYOrigin
_D88b:  .word  JPGCbVector
_D88r:  .word  JPGCrVector
```

C.3. Código Assembly da execução do *MiBench* QuickSort

```
LC0:
    .ascii  "cpu_time_used = %f\n\000"
    .text
    .align  2
    .global main
    .type   main, %function
main:
    @ args = 0, pretend = 0, frame = 20
    @ frame_needed = 1, uses_anonymous_args = 0
    mov    ip, sp
    stmfd  sp!, {fp, ip, lr, pc}
    sub    fp, ip, #4
    sub    sp, sp, #20
    bl     clock
    mov    r3, r0
    str    r3, [fp, #-16]
    ldr    r0, .L2+8
    mov    r1, #4992
    add    r1, r1, #8
    bl     quickSort
    bl     clock
    mov    r3, r0
    str    r3, [fp, #-20]
    ldr    r2, [fp, #-20]
    ldr    r3, [fp, #-16]
    rsb    r3, r3, r2
    fltd   f1, r3
    ldfd   f0, .L2
    dvfd   f0, f1, f0
    stfd   f0, [fp, #-28]
    ldr    r0, .L2+12
    sub    r1, fp, #28
    ldmia  r1, {r1-r2}
    bl     printf
    mov    r3, #0
    mov    r0, r3
    ldmea  fp, {fp, sp, pc}
.L3:
    .align  2
.L2:
    .word  1093567616
    .word  0
    .word  numbers
    .word  .LC0
    .size  main, .-main
    .align  2
    .global quickSort
    .type   quickSort, %function
quickSort:
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 1, uses_anonymous_args = 0
    mov    ip, sp
    stmfd  sp!, {fp, ip, lr, pc}
    sub    fp, ip, #4
    sub    sp, sp, #8
    str    r0, [fp, #-16]
    str    r1, [fp, #-20]
    ldr    r3, [fp, #-20]
    sub    r3, r3, #1
    ldr    r0, [fp, #-16]
    mov    r1, #0
    mov    r2, r3
    bl     q_sort
    ldmea  fp, {fp, sp, pc}
    .size  quickSort, .-quickSort
    .align  2
    .global q_sort
    .type   q_sort, %function
q_sort:
    @ args = 0, pretend = 0, frame = 24
    @ frame_needed = 1, uses_anonymous_args = 0
    mov    ip, sp
    stmfd  sp!, {fp, ip, lr, pc}
    sub    fp, ip, #4
    sub    sp, sp, #24
    str    r0, [fp, #-16]
    str    r1, [fp, #-20]
    str    r2, [fp, #-24]
```

```
    ldr    r3, [fp, #-20]
    str    r3, [fp, #-32]
    ldr    r3, [fp, #-24]
    str    r3, [fp, #-36]
    ldr    r3, [fp, #-20]
    mov    r2, r3, asl #2
    ldr    r3, [fp, #-16]
    add    r3, r2, r3
    ldr    r3, [r3, #0]
    str    r3, [fp, #-28]
.L6:
    ldr    r2, [fp, #-20]
    ldr    r3, [fp, #-24]
    cmp    r2, r3
    blt    .L9
    b      .L7
.L9:
    ldr    r3, [fp, #-24]
    mov    r2, r3, asl #2
    ldr    r3, [fp, #-16]
    add    r3, r2, r3
    ldr    r2, [r3, #0]
    ldr    r3, [fp, #-28]
    cmp    r2, r3
    blt    .L10
    ldr    r2, [fp, #-20]
    ldr    r3, [fp, #-24]
    cmp    r2, r3
    blt    .L11
    b      .L10
.L11:
    ldr    r3, [fp, #-24]
    sub    r3, r3, #1
    str    r3, [fp, #-24]
    b      .L9
.L10:
    ldr    r2, [fp, #-20]
    ldr    r3, [fp, #-24]
    cmp    r2, r3
    beq    .L13
    ldr    r3, [fp, #-20]
    mov    r2, r3, asl #2
    ldr    r3, [fp, #-16]
    add    r1, r2, r3
    ldr    r3, [fp, #-24]
    mov    r2, r3, asl #2
    ldr    r3, [fp, #-16]
    add    r3, r2, r3
    ldr    r3, [r3, #0]
    str    r3, [r1, #0]
    ldr    r3, [fp, #-20]
    add    r3, r3, #1
    str    r3, [fp, #-20]
.L13:
    mov    r0, r0  @ nop
.L14:
    ldr    r3, [fp, #-20]
    mov    r2, r3, asl #2
    ldr    r3, [fp, #-16]
    add    r3, r2, r3
    ldr    r2, [r3, #0]
    ldr    r3, [fp, #-28]
    cmp    r2, r3
    bgt    .L15
    ldr    r2, [fp, #-20]
    ldr    r3, [fp, #-24]
    cmp    r2, r3
    blt    .L16
    b      .L15
.L16:
    ldr    r3, [fp, #-20]
    add    r3, r3, #1
    str    r3, [fp, #-20]
    b      .L14
.L15:
    ldr    r2, [fp, #-20]
    ldr    r3, [fp, #-24]
    cmp    r2, r3
    beq    .L6
    ldr    r3, [fp, #-24]
    mov    r2, r3, asl #2
    ldr    r3, [fp, #-16]
    add    r1, r2, r3
    ldr    r3, [fp, #-20]
    mov    r2, r3, asl #2
    ldr    r3, [fp, #-16]
    add    r3, r2, r3
    ldr    r3, [r3, #0]
    str    r3, [r1, #0]
    ldr    r3, [fp, #-24]
    sub    r3, r3, #1
    str    r3, [fp, #-24]
    b      .L6
.L7:
    ldr    r3, [fp, #-20]
    mov    r2, r3, asl #2
    ldr    r3, [fp, #-16]
    add    r2, r2, r3
    ldr    r3, [fp, #-28]
    str    r3, [r2, #0]
    ldr    r3, [fp, #-20]
    str    r3, [fp, #-28]
    ldr    r3, [fp, #-32]
    str    r3, [fp, #-20]
    ldr    r3, [fp, #-36]
    str    r3, [fp, #-24]
    ldr    r2, [fp, #-20]
    ldr    r3, [fp, #-28]
    cmp    r2, r3
    bge    .L19
    ldr    r3, [fp, #-28]
    sub    r3, r3, #1
    ldr    r0, [fp, #-16]
    ldr    r1, [fp, #-20]
    mov    r2, r3
    bl     q_sort
.L19:
    ldr    r2, [fp, #-24]
    ldr    r3, [fp, #-28]
    cmp    r2, r3
    ble    .L5
    ldr    r3, [fp, #-28]
    add    r3, r3, #1
    ldr    r0, [fp, #-16]
    mov    r1, r3
    ldr    r2, [fp, #-24]
    bl     q_sort
.L5:
    ldmea   fp, {fp, sp, pc}
    .size   q_sort, .-q_sort
    .ident  "GCC: (GNU) 3.3.4"
```

C.4. Código Assembly da execução do *MiBench* SHA

```
.text

.global sha1_block_data_order
.type   sha1_block_data_order,%function

.align  2
sha1_block_data_order:
        stmdb   sp!,{r4-r12,lr}
        add     r2,r1,r2,lsl#6  @ r2 to point at the end of r1
        ldmia   r0,{r3,r4,r5,r6,r7}
.Lloop:
        ldr     r8,.LK_00_19
        mov     r14,sp
        sub     sp,sp,#15*4
        mov     r5,r5,ror#30
        mov     r6,r6,ror#30
        mov     r7,r7,ror#30            @ [6]
.L_00_15:
        ldrb    r10,[r1],#4
        ldrb    r11,[r1,#-3]
        ldrb    r12,[r1,#-2]
        add     r7,r8,r7,ror#2          @ E+=K_00_19
        orr     r10,r11,r10,lsl#8
        ldrb    r11,[r1,#-1]
        orr     r10,r12,r10,lsl#8
        add     r7,r7,r3,ror#27         @ E+=ROR(A,27)
        orr     r10,r11,r10,lsl#8
        add     r7,r7,r10               @ E+=X[i]
        eor     r11,r5,r6               @ F_xx_xx
        str     r10,[r14,#-4]!
        and     r11,r4,r11,ror#2
        eor     r11,r11,r6,ror#2        @ F_00_19(B,C,D)
        add     r7,r7,r11               @ E+=F_00_19(B,C,D)
        ldrb    r10,[r1],#4
        ldrb    r11,[r1,#-3]
        ldrb    r12,[r1,#-2]
        add     r6,r8,r6,ror#2          @ E+=K_00_19
        orr     r10,r11,r10,lsl#8
        ldrb    r11,[r1,#-1]
        orr     r10,r12,r10,lsl#8
        add     r6,r6,r7,ror#27         @ E+=ROR(A,27)
        orr     r10,r11,r10,lsl#8
        add     r6,r6,r10               @ E+=X[i]
        eor     r11,r4,r5               @ F_xx_xx
        str     r10,[r14,#-4]!
        and     r11,r3,r11,ror#2
        eor     r11,r11,r5,ror#2        @ F_00_19(B,C,D)
        add     r6,r6,r11               @ E+=F_00_19(B,C,D)
        ldrb    r10,[r1],#4
        ldrb    r11,[r1,#-3]
        ldrb    r12,[r1,#-2]
        add     r5,r8,r5,ror#2          @ E+=K_00_19
        orr     r10,r11,r10,lsl#8
        ldrb    r11,[r1,#-1]
        orr     r10,r12,r10,lsl#8
        add     r5,r5,r6,ror#27         @ E+=ROR(A,27)
        orr     r10,r11,r10,lsl#8
        add     r5,r5,r10               @ E+=X[i]
        eor     r11,r3,r4               @ F_xx_xx
        str     r10,[r14,#-4]!
        and     r11,r7,r11,ror#2
        eor     r11,r11,r4,ror#2        @ F_00_19(B,C,D)
        add     r5,r5,r11               @ E+=F_00_19(B,C,D)
        ldrb    r10,[r1],#4
        ldrb    r11,[r1,#-3]
        ldrb    r12,[r1,#-2]
        add     r4,r8,r4,ror#2          @ E+=K_00_19
        orr     r10,r11,r10,lsl#8
        ldrb    r11,[r1,#-1]
        orr     r10,r12,r10,lsl#8
        add     r4,r4,r5,ror#27         @ E+=ROR(A,27)
        orr     r10,r11,r10,lsl#8
        add     r4,r4,r10               @ E+=X[i]
        eor     r11,r7,r3               @ F_xx_xx
        str     r10,[r14,#-4]!
        and     r11,r6,r11,ror#2
        eor     r11,r11,r3,ror#2        @ F_00_19(B,C,D)
        add     r4,r4,r11               @ E+=F_00_19(B,C,D)
        ldrb    r10,[r1],#4
        ldrb    r11,[r1,#-3]
        ldrb    r12,[r1,#-2]
        add     r3,r8,r3,ror#2          @ E+=K_00_19
        orr     r10,r11,r10,lsl#8
        ldrb    r11,[r1,#-1]
        orr     r10,r12,r10,lsl#8
        add     r3,r3,r4,ror#27         @ E+=ROR(A,27)
        orr     r10,r11,r10,lsl#8
        add     r3,r3,r10               @ E+=X[i]
        eor     r11,r6,r7               @ F_xx_xx
        str     r10,[r14,#-4]!
        and     r11,r5,r11,ror#2
        eor     r11,r11,r7,ror#2        @ F_00_19(B,C,D)
        add     r3,r3,r11               @ E+=F_00_19(B,C,D)
        teq     r14,sp
        bne     .L_00_15                @ [((11+4)*5+2)*3]
        ldrb    r10,[r1],#4
        ldrb    r11,[r1,#-3]
        ldrb    r12,[r1,#-2]
        add     r7,r8,r7,ror#2          @ E+=K_00_19
        orr     r10,r11,r10,lsl#8
        ldrb    r11,[r1,#-1]
        orr     r10,r12,r10,lsl#8
        add     r7,r7,r3,ror#27         @ E+=ROR(A,27)
        orr     r10,r11,r10,lsl#8
        add     r7,r7,r10               @ E+=X[i]
        eor     r11,r5,r6               @ F_xx_xx
        str     r10,[r14,#-4]!
        and     r11,r4,r11,ror#2
        eor     r11,r11,r6,ror#2        @ F_00_19(B,C,D)
        add     r7,r7,r11               @ E+=F_00_19(B,C,D)
        ldr     r10,[r14,#15*4]
        ldr     r11,[r14,#13*4]
        ldr     r12,[r14,#7*4]
        add     r6,r8,r6,ror#2          @ E+=K_xx_xx
        eor     r10,r10,r11
        ldr     r11,[r14,#2*4]
        add     r6,r6,r7,ror#27         @ E+=ROR(A,27)
        eor     r10,r10,r12
        eor     r10,r10,r11
        eor     r11,r4,r5               @ F_xx_xx, but not in 40_59
        mov     r10,r10,ror#31
        add     r6,r6,r10               @ E+=X[i]
        str     r10,[r14,#-4]!
        and     r11,r3,r11,ror#2
        eor     r11,r11,r5,ror#2        @ F_00_19(B,C,D)
        add     r6,r6,r11               @ E+=F_00_19(B,C,D)
        ldr     r10,[r14,#15*4]
        ldr     r11,[r14,#13*4]
        ldr     r12,[r14,#7*4]
        add     r5,r8,r5,ror#2          @ E+=K_xx_xx
        eor     r10,r10,r11
        ldr     r11,[r14,#2*4]
        add     r5,r5,r6,ror#27         @ E+=ROR(A,27)
        eor     r10,r10,r12
```

```
	eor	r10,r10,r11
	eor	r11,r3,r4		@ F_xx_xx, but not in 40_59
	mov	r10,r10,ror#31
	add	r5,r5,r10		@ E+=X[i]
	str	r10,[r14,#-4]!
	and	r11,r7,r11,ror#2
	eor	r11,r11,r4,ror#2	@ F_00_19(B,C,D)
	add	r5,r5,r11		@ E+=F_00_19(B,C,D)
	ldr	r10,[r14,#15*4]
	ldr	r11,[r14,#13*4]
	ldr	r12,[r14,#7*4]
	add	r4,r8,r4,ror#2		@ E+=K_xx_xx
	eor	r10,r10,r11
	ldr	r11,[r14,#2*4]
	add	r4,r4,r5,ror#27		@ E+=ROR(A,27)
	eor	r10,r10,r12
	eor	r10,r10,r11
	eor	r11,r7,r3		@ F_xx_xx, but not in 40_59
	mov	r10,r10,ror#31
	add	r4,r4,r10		@ E+=X[i]
	str	r10,[r14,#-4]!
	and	r11,r6,r11,ror#2
	eor	r11,r11,r3,ror#2	@ F_00_19(B,C,D)
	add	r4,r4,r11		@ E+=F_00_19(B,C,D)
	ldr	r10,[r14,#15*4]
	ldr	r11,[r14,#13*4]
	ldr	r12,[r14,#7*4]
	add	r3,r8,r3,ror#2		@ E+=K_xx_xx
	eor	r10,r10,r11
	ldr	r11,[r14,#2*4]
	add	r3,r3,r4,ror#27		@ E+=ROR(A,27)
	eor	r10,r10,r12
	eor	r10,r10,r11
	eor	r11,r6,r7		@ F_xx_xx, but not in 40_59
	mov	r10,r10,ror#31
	add	r3,r3,r10		@ E+=X[i]
	str	r10,[r14,#-4]!
	and	r11,r5,r11,ror#2
	eor	r11,r11,r7,ror#2	@ F_00_19(B,C,D)
	add	r3,r3,r11		@ E+=F_00_19(B,C,D)

	ldr	r8,.LK_20_39		@ [+15+16*4]
	sub	sp,sp,#25*4
	cmn	sp,#0			@ [+3], clear carry to denote 20_39
.L_20_39_or_60_79:
	ldr	r10,[r14,#15*4]
	ldr	r11,[r14,#13*4]
	ldr	r12,[r14,#7*4]
	add	r7,r8,r7,ror#2		@ E+=K_xx_xx
	eor	r10,r10,r11
	ldr	r11,[r14,#2*4]
	add	r7,r7,r3,ror#27		@ E+=ROR(A,27)
	eor	r10,r10,r12
	eor	r10,r10,r11
	eor	r11,r5,r6		@ F_xx_xx, but not in 40_59
	mov	r10,r10,ror#31
	add	r7,r7,r10		@ E+=X[i]
	str	r10,[r14,#-4]!
	eor	r11,r4,r11,ror#2	@ F_20_39(B,C,D)
	add	r7,r7,r11		@ E+=F_20_39(B,C,D)
	ldr	r10,[r14,#15*4]
	ldr	r11,[r14,#13*4]
	ldr	r12,[r14,#7*4]
	add	r6,r8,r6,ror#2		@ E+=K_xx_xx
	eor	r10,r10,r11
	ldr	r11,[r14,#2*4]
	add	r6,r6,r7,ror#27		@ E+=ROR(A,27)
	eor	r10,r10,r12
	eor	r10,r10,r11
	eor	r11,r4,r5		@ F_xx_xx, but not in 40_59
	mov	r10,r10,ror#31
	add	r6,r6,r10		@ E+=X[i]
	str	r10,[r14,#-4]!
	eor	r11,r3,r11,ror#2	@ F_20_39(B,C,D)
	add	r6,r6,r11		@ E+=F_20_39(B,C,D)
	ldr	r10,[r14,#15*4]
	ldr	r11,[r14,#13*4]
	ldr	r12,[r14,#7*4]
	add	r5,r8,r5,ror#2		@ E+=K_xx_xx
	eor	r10,r10,r11
	ldr	r11,[r14,#2*4]
	add	r5,r5,r6,ror#27		@ E+=ROR(A,27)
	eor	r10,r10,r12
	eor	r10,r10,r11
	eor	r11,r3,r4		@ F_xx_xx, but not in 40_59
	mov	r10,r10,ror#31
	add	r5,r5,r10		@ E+=X[i]
	str	r10,[r14,#-4]!
	eor	r11,r7,r11,ror#2	@ F_20_39(B,C,D)
	add	r5,r5,r11		@ E+=F_20_39(B,C,D)
	ldr	r10,[r14,#15*4]
	ldr	r11,[r14,#13*4]
	ldr	r12,[r14,#7*4]
	add	r4,r8,r4,ror#2		@ E+=K_xx_xx
	eor	r10,r10,r11
	ldr	r11,[r14,#2*4]
	add	r4,r4,r5,ror#27		@ E+=ROR(A,27)
	eor	r10,r10,r12
	eor	r10,r10,r11
	eor	r11,r7,r3		@ F_xx_xx, but not in 40_59
	mov	r10,r10,ror#31
	add	r4,r4,r10		@ E+=X[i]
	str	r10,[r14,#-4]!
	eor	r11,r6,r11,ror#2	@ F_20_39(B,C,D)
	add	r4,r4,r11		@ E+=F_20_39(B,C,D)
	ldr	r10,[r14,#15*4]
	ldr	r11,[r14,#13*4]
	ldr	r12,[r14,#7*4]
	add	r3,r8,r3,ror#2		@ E+=K_xx_xx
	eor	r10,r10,r11
	ldr	r11,[r14,#2*4]
	add	r3,r3,r4,ror#27		@ E+=ROR(A,27)
	eor	r10,r10,r12
	eor	r10,r10,r11
	eor	r11,r6,r7		@ F_xx_xx, but not in 40_59
	mov	r10,r10,ror#31
	add	r3,r3,r10		@ E+=X[i]
	str	r10,[r14,#-4]!
	eor	r11,r5,r11,ror#2	@ F_20_39(B,C,D)
	add	r3,r3,r11		@ E+=F_20_39(B,C,D)
	teq	r14,sp			@ preserve carry
	bne	.L_20_39_or_60_79	@ [+((12+3)*5+2)*4]
	bcs	.L_done			@ [+((12+3)*5+2)*4], spare 300
bytes

	ldr	r8,.LK_40_59
	sub	sp,sp,#20*4		@ [+2]
.L_40_59:
	ldr	r10,[r14,#15*4]
	ldr	r11,[r14,#13*4]
	ldr	r12,[r14,#7*4]
	add	r7,r8,r7,ror#2		@ E+=K_xx_xx
	eor	r10,r10,r11
	ldr	r11,[r14,#2*4]
	add	r7,r7,r3,ror#27		@ E+=ROR(A,27)
	eor	r10,r10,r12
	eor	r10,r10,r11
	mov	r10,r10,ror#31
	add	r7,r7,r10		@ E+=X[i]
	str	r10,[r14,#-4]!
	and	r11,r4,r5,ror#2
	orr	r12,r4,r5,ror#2
```

```
	and	r12,r12,r6,ror#2
	orr	r11,r11,r12		@ F_40_59(B,C,D)
	add	r7,r7,r11		@ E+=F_40_59(B,C,D)
	ldr	r10,[r14,#15*4]
	ldr	r11,[r14,#13*4]
	ldr	r12,[r14,#7*4]
	add	r6,r8,r6,ror#2		@ E+=K_xx_xx
	eor	r10,r10,r11
	ldr	r11,[r14,#2*4]
	add	r6,r6,r7,ror#27		@ E+=ROR(A,27)
	eor	r10,r10,r12
	eor	r10,r10,r11
	mov	r10,r10,ror#31
	add	r6,r6,r10		@ E+=X[i]
	str	r10,[r14,#-4]!
	and	r11,r3,r4,ror#2
	orr	r12,r3,r4,ror#2
	and	r12,r12,r5,ror#2
	orr	r11,r11,r12		@ F_40_59(B,C,D)
	add	r6,r6,r11		@ E+=F_40_59(B,C,D)
	ldr	r10,[r14,#15*4]
	ldr	r11,[r14,#13*4]
	ldr	r12,[r14,#7*4]
	add	r5,r8,r5,ror#2		@ E+=K_xx_xx
	eor	r10,r10,r11
	ldr	r11,[r14,#2*4]
	add	r5,r5,r6,ror#27		@ E+=ROR(A,27)
	eor	r10,r10,r12
	eor	r10,r10,r11
	mov	r10,r10,ror#31
	add	r5,r5,r10		@ E+=X[i]
	str	r10,[r14,#-4]!
	and	r11,r7,r3,ror#2
	orr	r12,r7,r3,ror#2
	and	r12,r12,r4,ror#2
	orr	r11,r11,r12		@ F_40_59(B,C,D)
	add	r5,r5,r11		@ E+=F_40_59(B,C,D)
	ldr	r10,[r14,#15*4]
	ldr	r11,[r14,#13*4]
	ldr	r12,[r14,#7*4]
	add	r4,r8,r4,ror#2		@ E+=K_xx_xx
	eor	r10,r10,r11
	ldr	r11,[r14,#2*4]
	add	r4,r4,r5,ror#27		@ E+=ROR(A,27)
	eor	r10,r10,r12
	eor	r10,r10,r11
	mov	r10,r10,ror#31
	add	r4,r4,r10		@ E+=X[i]
	str	r10,[r14,#-4]!
	and	r11,r6,r7,ror#2
	orr	r12,r6,r7,ror#2
	and	r12,r12,r3,ror#2
	orr	r11,r11,r12		@ F_40_59(B,C,D)
	add	r4,r4,r11		@ E+=F_40_59(B,C,D)
	ldr	r10,[r14,#15*4]
	ldr	r11,[r14,#13*4]
	ldr	r12,[r14,#7*4]
	add	r3,r8,r3,ror#2		@ E+=K_xx_xx
	eor	r10,r10,r11
	ldr	r11,[r14,#2*4]
	add	r3,r3,r4,ror#27		@ E+=ROR(A,27)
	eor	r10,r10,r12
	eor	r10,r10,r11
	mov	r10,r10,ror#31
	add	r3,r3,r10		@ E+=X[i]
	str	r10,[r14,#-4]!
	and	r11,r5,r6,ror#2
	orr	r12,r5,r6,ror#2
	and	r12,r12,r7,ror#2
	orr	r11,r11,r12		@ F_40_59(B,C,D)
	add	r3,r3,r11		@ E+=F_40_59(B,C,D)
	teq	r14,sp
	bne	.L_40_59		@ [+((12+5)*5+2)*4]

	ldr	r8,.LK_60_79
	sub	sp,sp,#20*4
	cmp	sp,#0			@ set carry to denote 60_79
	b	.L_20_39_or_60_79	@ [+4], spare 300 bytes
.L_done:
	add	sp,sp,#80*4		@ "deallocate" stack frame
	ldmia	r0,{r8,r10,r11,r12,r14}
	add	r3,r8,r3
	add	r4,r10,r4
	add	r5,r11,r5,ror#2
	add	r6,r12,r6,ror#2
	add	r7,r14,r7,ror#2
	stmia	r0,{r3,r4,r5,r6,r7}
	teq	r1,r2
	bne	.Lloop			@ [+18], total 1307

	ldmia	sp!,{r4-r12,lr}
	tst	lr,#1
	moveq	pc,lr			@ be binary compatible with V4,
yet
	.word	0xe12fff1e		@ interoperable with ARM9 ISA
.align	2
.LK_00_19:	.word	0x5a827999
.LK_20_39:	.word	0x6ed9eba1
.LK_40_59:	.word	0x8f1bbcdc
.LK_60_79:	.word	0xca62c1d6
.size	sha1_block_data_order,.-sha1_block_data_order
.asciz	"SHA1 block transform for ARMv4, CRYPTOGAMS by
<appro@openssl.org>"
```

APÊNDICE D

Neste apêndice temos os códigos fontes em VHDL dos métodos *MIC* (*Middle Instruction Compression*) e *Huffman* para a arquitetura PDCCM.

D.1. Código Fonte do método *MIC*

```
-------------------------------------------------------------------------------
-- Title                : Compressor/Descompressor MIC
-- Project              : Arquitetura PDCCM em Hardware para Compressão/Descompressão de
--                        Instruções em Sistemas Embarcados
-------------------------------------------------------------------------------
-- Architecture         : Processador Descompressor iCache Compressor Memória
-------------------------------------------------------------------------------
-- Método               : Middle Instruction Compression
-------------------------------------------------------------------------------
-- File                 : MIC.vhd
-- Author               : Wanderson Roger Azevedo Dias
-- e-mail               : roger@dcc.ufam.edu.br
-- Organization         : Universidade Federal do Amazonas - UFAM
-- Created              : 20/11/2008
-- Last update          : 16/04/2009
-- Plataform            : Cyclone-II
-- Device               : EP2C20F484C7
-- Fabricator           : Altera
-- Simulators           : Quartus-II Web Edition
-- Synthesizers         : Quartus-II Web Edition
-- Version              : 8.0
-- Language             : VHDL
-------------------------------------------------------------------------------
-- Description          : Entidade responsável pela compressão/descompressão dos códigos
--                        de instruções (32 bits) usando o método MIC e na arquitetura
--                        PDCCM
-------------------------------------------------------------------------------
-- Copyright (c) notice
--
--                      Universidade Federal do Amazonas - UFAM
--                      Instituto de Ciências Exatas - ICE
--                      Departamento de Ciência da Computação - DCC
--                      Programa de Pós-Graduação em Informática - PPGI
--
--                      Developed by Wanderson Roger Azevedo Dias
--                      This code may be used for educational and non-educational purposes as
--                      long as its copyright notice remains unchanged.
-------------------------------------------------------------------------------
-- Revisions            : 1
-- Revision Number      : 1.2
-- Version              : 2.1.2
-- Date                 : 16/04/2009
-- Description          : Atualização dos comentários
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--- BIBLIOTECAS DO PROJETO
-------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

-------------------------------------------------------------------------------------------
--- ENTIDADE DO MÉTODO MIC
-------------------------------------------------------------------------------------------
entity MIC is
        generic (
                -- Memórias (RAM e iCACHE)
                SIZE_RAM        : natural := 128;               -- define o tamanho da memória RAM
```

```
                SIZE_CACHE          : natural := 16;                    -- define o tamanho da memória iCACHE
                SIZE_ADDR_R         : natural := ((2**8) - 1);          -- define o tamanho máximo do endereço da RAM
                SIZE_ADDR_C         : natural := ((2**8) - 1);          -- define o tamanho máximo do endereço da iCACHE
                -- Palavra
                SIZE_WORD           : natural := 32                     -- define o tamanho da palavra (32 bits ou 4 bytes);
        );

    port (
                clk                 : in  std_logic;                                    -- sinal de clock
                addr_inst_p         : out natural range SIZE_ADDR_R downto 0;   -- valor dos endereços da memória RAM
                new_address         : out natural range SIZE_ADDR_C downto 0;   -- novo posicionamento da instr. na iCACHE e ST
                dupla_bytes         : out bit;              -- mostra em qual dupla de bytes (primeira ou segunda) está a instrução
                status              : out bit;              -- mostra se está havendo uma compressão ou descompressão
                i1_iCACHE_ST        : out natural;          -- mostra o índice do posicionamento da instrução na iCACHE e ST
                i2_LAT              : out natural;          -- mostra o índice do posicionamento da instrução na LAT

                returnC_icache      : out bit_vector(32 downto 1);  -- mostra a instrução que foi comprimida e será salva na iCACHE
                returnC_inst_proc   : out bit_vector(32 downto 1);  -- retorno da instrução para o processador
                returnD_inst_proc   : out bit_vector(32 downto 1)   -- retorno da instrução para o processador
        );
end entity MIC;

-------------------------------------------------------------------------------------------
--- ARQUITETURA DO MÉTODO MIC
-------------------------------------------------------------------------------------------
architecture MIC_arch of MIC is

    subtype wordT is bit_vector(32 downto 1);


--- 0. Endereços do Processador para simulações
-------------------------------------------------------------------------------------------
--- Tabela que contém os endereços solicitados pelo processador para a execução da simulação
type reg_inst_proc is record
    addr_inst_proc : natural range SIZE_ADDR_R downto 0;        -- campo dos endereços solicitados pelo processador
end record;

type vetor_instP is array (0 to (SIZE_RAM - 1)) of reg_inst_proc;


--- 1. Memória RAM
-------------------------------------------------------------------------------------------
--- Tabela que simula uma memória RAM (Address e Valores)
type reg_memoria_ram is record
    addr_memoria_ram : natural range SIZE_ADDR_R downto 0;      -- campo do endereço da memória RAM
    dado_memoria_ram : wordT;                                   -- campo com o código da instrução
end record;

type vetor_mRAM     is array (0 to (SIZE_RAM - 1)) of reg_memoria_ram;


--- 2. Memória iCACHE
-------------------------------------------------------------------------------------------
--- Tabela que simula uma iCACHE (Índice, Address e Valores)
--- Obs.: Todas as instruções contidas na iCACHE estão comprimido
type reg_memoria_icache is record
    indice_icache       : natural range (SIZE_CACHE - 1) downto 0;      -- campo do índice da memória iCACHE
    addr_memoria_icache : natural range SIZE_ADDR_C downto 0;           -- campo do endereço da memória iCACHE
    dado_memoria_icache : wordT;                        -- campo com código de duas instrução comprimidas na iCACHE
end record;

type vetor_iCACHE is array (0 to (SIZE_CACHE - 1)) of reg_memoria_icache;


--- 3. Tabela da Linha de Endereços - LAT (Line Address Table)
-------------------------------------------------------------------------------------------
--- Tabela que conterá o mapeamento das novas posições das instruções comprimidas na iCACHE
type reg_LAT is record
    indice_LAT  : natural range ((2 * SIZE_CACHE) - 1) downto 0;    -- campo do índice da LAT
    addr_old    : natural range SIZE_ADDR_R downto 0;               -- campo do endereço velho da instrução na RAM
    addr_new    : natural range SIZE_ADDR_C downto 0;               -- campo do endereço novo da instrução na iCACHE
```

```
        pair_bytes : bit;                                       -- campo para indicar em qual dupla de byte está a instrução
end record;                                                     -- (1ª = 0, 2ª = 1)

type vetor_LAT is array (0 to ((2 * SIZE_CACHE) - 1)) of reg_LAT;


--- 4. Tabela de Sinais - ST (Sign Table)
-------------------------------------------------------------------------------------------
--- Tabela que conterá o conjuto de sinais das instruções comprimidas
type reg_ST is record
        indice_ST  : natural range (SIZE_CACHE - 1) downto 0;       -- campo do índice da ST - será do tamanho da iCACHE
        addr_ST    : natural range SIZE_ADDR_C downto 0;            -- campo do endereço de memória da instrução na iCACHE
        sinal_ST   : wordT;                                 -- campo com o conjunto de sinais gerados na compressão da instrução
end record;

type vetor_ST is array (0 to (SIZE_CACHE - 1)) of reg_ST;

-------------------------------------------------------------------------------------------
--- FUNÇÃO DO COMPRESSOR  MIC
-------------------------------------------------------------------------------------------
---
--- Esta função é responsável pela compressão das instruções que será gravada na iCACHE
--- A função é acionada toda vez que a iCACHE solicitar uma instrução para a RAM
---
-------------------------------------------------------------------------------------------
function Compressor(iCACHE      : vetor_iCACHE;
                    ST          : vetor_ST;
                    dado_ram    : wordT;
                    ind1        : natural;
                    incr_ind    : natural)
return vetor_iCACHE is

variable cach   : vetor_iCACHE;                 -- receberá o conteúdo da iCACHE e realizará as inserções
variable tb_s   : vetor_ST;                     -- receberá o conteúdo da Tabela de Sinais e realizará as inserções
variable id1    : natural range 0 to (SIZE_CACHE - 1);  -- será usada para indexar a LAT
variable dupla : bit_vector(2 downto 1);        -- formará as duplas para inserir na iCACHE
variable cont   : natural := 1;                 -- servirá como índice para o dado_ram
variable c_ic   : natural;                      -- servirá como índice para a Linha de iCACHE e ST (qual dupla de byte?)

begin
        cach   := iCACHE;
        tb_s   := ST;
        id1    := ind1;

        -- Se não existir mais espaço na iCACHE, zerar o índice, ou seja, começar a sobre-escrever na iCACHE
        if (id1 > SIZE_CACHE) then
                id1 := 0;
        end if;

        -- Atribuição do valor inicial para a variavel que indexará a Linha de iCACHE e ST
        -- e saberá em qual dupla de bytes deverá ser gravado
        if ((incr_ind mod 2) = 0) then                  -- se o índice for par ou zero o c_ic iniciará em 1
                c_ic := 1;
                dupla_bytes <= '0';
        else                                            -- senão o c_ic iniciará em 17
                c_ic := 17;
                dupla_bytes <= '1';
        end if;

        for i in (SIZE_WORD/2) downto 1 loop            -- repte 16 vezes esse laço para formar as 16 duplas
                for x in 2 downto 1 loop
                        dupla(x) := dado_ram(cont);
                        cont := cont + 1;
                end loop;

                if (dupla = "00") then
                        cach(id1).dado_memoria_icache(c_ic) := '0';
                        tb_s(id1).sinal_ST(c_ic) := '0';  -- - = 0

                elsif (dupla = "11") then
                        cach(id1).dado_memoria_icache(c_ic) := '0';
```

```
                        tb_s(id1).sinal_ST(c_ic) := '1';  -- + = 1

                elsif (dupla = "01") then
                        cach(id1).dado_memoria_icache(c_ic) := '1';
                        tb_s(id1).sinal_ST(c_ic) := '0';  -- - = 0

                else -- (dupla = "10")
                        cach(id1).dado_memoria_icache(c_ic) := '1';
                        tb_s(id1).sinal_ST(c_ic) := '1';  -- + = 1
                end if;
                c_ic := c_ic + 1;
        end loop;

        i1_iCACHE_ST <= id1;
        new_address <= cach(id1).addr_memoria_icache;
        returnC_icache <= cach(id1).dado_memoria_icache;

        return cach;
end function;

-------------------------------------------------------------------------------------------
--- FUNÇÃO DO DESCOMPRESSOR  MIC
-------------------------------------------------------------------------------------------
---
--- Esta função é responsável pela descompressão das instruções que serão repassadas ao processador
--- A função é acionada toda vez que a LAT tiver um acerto
---
-------------------------------------------------------------------------------------------
function Descompressor(iCACHE        : vetor_iCACHE;
                       ST            : vetor_ST;
                       new_addr      : natural;
                       parte_dupla : bit)
return bit_vector is

variable cach    : vetor_iCACHE;     -- receberá o conteúdo da CACHE e realizará as inserções
variable tb_s    : vetor_ST;         -- receberá o conteúdo da Tabela de Sinais e realizará as inserções
variable cont    : natural := 1;     -- servirá como índice para a instrução que está sendo descomprimida
variable acerto : bit := '0';        -- indica o acerto da instrução na iCACHE, devido ter 2 instruções em cada linha com o mesmo new_addr
variable instruction : wordT;        -- receberá a instrução descomprimida

begin
        cach   := iCACHE;
        tb_s   := ST;

        for i in 0 to (SIZE_CACHE - 1) loop
                if (cach(i).addr_memoria_icache = new_addr) and (acerto = '0') then
                        acerto := '1';
                        i1_iCACHE_ST <= cach(i).indice_icache;
                        new_address <= cach(i).addr_memoria_icache;
                        if (parte_dupla = '0') then
                                for k in 1 to 16 loop
                                        if ((cach(i).dado_memoria_icache(k) = '0')
                                        and (tb_s(i).sinal_ST(k) = '0') and (cont < 33)) then
                                                instruction(cont) := '0';
                                                cont := cont + 1;
                                                instruction(cont) := '0';
                                                cont := cont + 1;

                                        elsif ((cach(i).dado_memoria_icache(k) = '0')
                                        and (tb_s(i).sinal_ST(k) = '1') and (cont < 33)) then
                                                instruction(cont) := '1';
                                                cont := cont + 1;
                                                instruction(cont) := '1';
                                                cont := cont + 1;

                                        elsif ((cach(i).dado_memoria_icache(k) = '1')
                                        and (tb_s(i).sinal_ST(k) = '0') and (cont < 33)) then
                                                instruction(cont) := '0';
                                                cont := cont + 1;
                                                instruction(cont) := '1';
                                                cont := cont + 1;
```

```
                                        else
                                                instruction(cont) := '1';
                                                cont := cont + 1;
                                                instruction(cont) := '0';
                                                cont := cont + 1;
                                        end if;
                                end loop;
                        else
                                for k in 17 to 32 loop
                                        if ((cach(i).dado_memoria_icache(k) = '0')
                                        and (tb_s(i).sinal_ST(k) = '0') and (cont < 33)) then
                                                instruction(cont) := '0';
                                                cont := cont + 1;
                                                instruction(cont) := '0';
                                                cont := cont + 1;

                                        elsif ((cach(i).dado_memoria_icache(k) = '0')
                                        and (tb_s(i).sinal_ST(k) = '1') and (cont < 33)) then
                                                instruction(cont) := '1';
                                                cont := cont + 1;
                                                instruction(cont) := '1';
                                                cont := cont + 1;

                                        elsif ((cach(i).dado_memoria_icache(k) = '1')
                                        and (tb_s(i).sinal_ST(k) = '0') and (cont < 33)) then
                                                instruction(cont) := '0';
                                                cont := cont + 1;
                                                instruction(cont) := '1';
                                                cont := cont + 1;

                                        else
                                                instruction(cont) := '1';
                                                cont := cont + 1;
                                                instruction(cont) := '0';
                                                cont := cont + 1;
                                        end if;
                                end loop;
                        end if;
                end if;
        end loop;
        return instruction;
end function;

-------------------------------------------------------------------------------------
--- PROGRAMA PRINCIPAL
-------------------------------------------------------------------------------------
begin
process (clk)
        variable RAM            : vetor_mRAM;
        variable iCACHE         : vetor_iCACHE;
        variable LAT            : vetor_LAT;
        variable ST             : vetor_ST;
        variable end_inst_proc : vetor_instP;

        variable new_addr       : natural;
        variable parte_dupla    : bit;
        variable flag           : bit;                          -- servirá como flag para saber se houve ACERTO ou FALHA na LAT

        variable dado_ram       : wordT;                                        -- conterá a instrução da linha da RAM
        variable addr_ram       : natural range SIZE_ADDR_R downto 0;           -- conterá o endereço da linha da RAM

        variable ind1           : natural range 6 to (SIZE_CACHE - 1);          -- usado para indexar a iCACHE e a TBS
        variable ind2           : natural range 12 to ((2 * SIZE_CACHE) - 1);   -- usado para indexar a LAT
        variable incr_ind       : natural := 0;                                 -- usado para incrementar os índices
        variable nextpc         : natural := 0;         -- usado para incrementar o índice do endereços solicitados pelo processador

        begin
                -- Carrega os endereços e valores da memória RAM simulada

                -- benchmark MiBench: CRC32
                -- Processador: ARM
```

```
-- Família: ARM9
-- versão: ARM922T
-- ISA: ARMv4T

-- Código em Assembly compilado para ARM

RAM(0).addr_memoria_ram  := 16#7f#;
RAM(0).dado_memoria_ram  := "10101010101010101111111111111111";          -- MOV
RAM(1).addr_memoria_ram  := 16#7e#;
RAM(1).dado_memoria_ram  := "11000011110000111100000000110011";          -- STDMF
RAM(2).addr_memoria_ram  := 16#7d#;
RAM(2).dado_memoria_ram  := "00000000000000000000000000000011";          -- SUB
RAM(3).addr_memoria_ram  := 16#7c#;
RAM(3).dado_memoria_ram  := "11111000001100111111111000001100";          -- LDR
RAM(4).addr_memoria_ram  := 16#7b#;
RAM(4).dado_memoria_ram  := "11110011110000000111111100111111";          -- ANDS
RAM(5).addr_memoria_ram  := 16#7a#;
RAM(5).dado_memoria_ram  := "00000000000000001111111111110011";          -- BEQ
RAM(6).addr_memoria_ram  := 16#79#;
RAM(6).dado_memoria_ram  := "11000000000000000000000000011111";          -- SUBS
RAM(7).addr_memoria_ram  := 16#78#;
RAM(7).dado_memoria_ram  := "00000000000001111111111100110011";          -- BLT
RAM(8).addr_memoria_ram  := 16#77#;
RAM(8).dado_memoria_ram  := "11111000001100111110000000001111";          -- LDRB*
RAM(9).addr_memoria_ram  := 16#76#;
RAM(9).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(10).addr_memoria_ram  := 16#75#;
RAM(10).dado_memoria_ram  := "11100011100000000111111100111000";         -- AND
RAM(11).addr_memoria_ram  := 16#74#;
RAM(11).dado_memoria_ram  := "11111000001100111111111000001100";         -- LDR
RAM(12).addr_memoria_ram  := 16#73#;
RAM(12).dado_memoria_ram  := "11110000000000001111111111110000";         -- EOR
RAM(13).addr_memoria_ram  := 16#72#;
RAM(13).dado_memoria_ram  := "00000000000011111111111111110000";         -- B
RAM(14).addr_memoria_ram  := 16#71#;
RAM(14).dado_memoria_ram  := "10110000000000000011110000000011";         -- LDMIA
RAM(15).addr_memoria_ram  := 16#70#;
RAM(15).dado_memoria_ram  := "11110000000000001111111111110000";         -- EOR
RAM(16).addr_memoria_ram  := 16#6f#;
RAM(16).dado_memoria_ram  := "11100011100000000111111100111000";         -- AND
RAM(17).addr_memoria_ram  := 16#6e#;
RAM(17).dado_memoria_ram  := "11111000001100111111111000001100";         -- LDR
RAM(18).addr_memoria_ram  := 16#6d#;
RAM(18).dado_memoria_ram  := "11110000000000001111111111110000";         -- EOR
RAM(19).addr_memoria_ram  := 16#6c#;
RAM(19).dado_memoria_ram  := "11110000000000001111111111110000";         -- EOR
RAM(20).addr_memoria_ram  := 16#6b#;
RAM(20).dado_memoria_ram  := "11100011100000000111111100111000";         -- AND
RAM(21).addr_memoria_ram  := 16#6a#;
RAM(21).dado_memoria_ram  := "11111000001100111111111000001100";         -- LDR
RAM(22).addr_memoria_ram  := 16#69#;
RAM(22).dado_memoria_ram  := "11110000000000001111111111110000";         -- EOR
RAM(23).addr_memoria_ram  := 16#68#;
RAM(23).dado_memoria_ram  := "11110000000000001111111111110000";         -- EOR
RAM(24).addr_memoria_ram  := 16#67#;
RAM(24).dado_memoria_ram  := "11100011100000000111111100111000";         -- AND
RAM(25).addr_memoria_ram  := 16#66#;
RAM(25).dado_memoria_ram  := "11111000001100111111111000001100";         -- LDR
RAM(26).addr_memoria_ram  := 16#65#;
RAM(26).dado_memoria_ram  := "11110000000000001111111111110000";         -- EOR
RAM(27).addr_memoria_ram  := 16#64#;
RAM(27).dado_memoria_ram  := "11110000000000001111111111110000";         -- EOR
RAM(28).addr_memoria_ram  := 16#63#;
RAM(28).dado_memoria_ram  := "11100011100000000111111100111000";         -- AND
RAM(29).addr_memoria_ram  := 16#62#;
RAM(29).dado_memoria_ram  := "11111000001100111111111000001100";         -- LDR
RAM(30).addr_memoria_ram  := 16#61#;
RAM(30).dado_memoria_ram  := "11110000000000001111111111110000";         -- EOR
RAM(31).addr_memoria_ram  := 16#60#;
RAM(31).dado_memoria_ram  := "11110000000000001111111111110000";         -- EOR
RAM(32).addr_memoria_ram  := 16#5f#;
```

```
RAM(32).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(33).addr_memoria_ram  := 16#5e#;
RAM(33).dado_memoria_ram  := "11111000001100111111111000001100";          -- LDR
RAM(34).addr_memoria_ram  := 16#5d#;
RAM(34).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(35).addr_memoria_ram  := 16#5c#;
RAM(35).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(36).addr_memoria_ram  := 16#5b#;
RAM(36).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(37).addr_memoria_ram  := 16#5a#;
RAM(37).dado_memoria_ram  := "11111000001100111111111000001100";          -- LDR
RAM(38).addr_memoria_ram  := 16#59#;
RAM(38).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(39).addr_memoria_ram  := 16#58#;
RAM(39).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(40).addr_memoria_ram  := 16#57#;
RAM(40).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(41).addr_memoria_ram  := 16#56#;
RAM(41).dado_memoria_ram  := "11111000001100111111111000001100";          -- LDR
RAM(42).addr_memoria_ram  := 16#55#;
RAM(42).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(43).addr_memoria_ram  := 16#54#;
RAM(43).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(44).addr_memoria_ram  := 16#53#;
RAM(44).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(45).addr_memoria_ram  := 16#52#;
RAM(45).dado_memoria_ram  := "00000000000011001111111100110000";          -- LDR
RAM(46).addr_memoria_ram  := 16#51#;
RAM(46).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(47).addr_memoria_ram  := 16#50#;
RAM(47).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(48).addr_memoria_ram  := 16#4f#;
RAM(48).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(49).addr_memoria_ram  := 16#4e#;
RAM(49).dado_memoria_ram  := "00000000000011001111111100110000";          -- LDR
RAM(50).addr_memoria_ram  := 16#4d#;
RAM(50).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(51).addr_memoria_ram  := 16#4c#;
RAM(51).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(52).addr_memoria_ram  := 16#4b#;
RAM(52).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(53).addr_memoria_ram  := 16#4a#;
RAM(53).dado_memoria_ram  := "00000000000011001111111100110000";          -- LDR
RAM(54).addr_memoria_ram  := 16#49#;
RAM(54).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(55).addr_memoria_ram  := 16#48#;
RAM(55).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(56).addr_memoria_ram  := 16#47#;
RAM(56).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(57).addr_memoria_ram  := 16#46#;
RAM(57).dado_memoria_ram  := "00000000000011001111111100110000";          -- LDR
RAM(58).addr_memoria_ram  := 16#45#;
RAM(58).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(59).addr_memoria_ram  := 16#44#;
RAM(59).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(60).addr_memoria_ram  := 16#43#;
RAM(60).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(61).addr_memoria_ram  := 16#42#;
RAM(61).dado_memoria_ram  := "00000000000011001111111100110000";          -- LDR
RAM(62).addr_memoria_ram  := 16#41#;
RAM(62).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(63).addr_memoria_ram  := 16#40#;
RAM(63).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(64).addr_memoria_ram  := 16#3f#;
RAM(64).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(65).addr_memoria_ram  := 16#3e#;
RAM(65).dado_memoria_ram  := "00000000000011001111111100110000";          -- LDR
RAM(66).addr_memoria_ram  := 16#3d#;
RAM(66).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(67).addr_memoria_ram  := 16#3c#;
RAM(67).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
```

```
RAM(68).addr_memoria_ram  := 16#3b#;
RAM(68).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(69).addr_memoria_ram  := 16#3a#;
RAM(69).dado_memoria_ram  := "00000000000011001111111100110000";          -- LDR
RAM(70).addr_memoria_ram  := 16#39#;
RAM(70).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(71).addr_memoria_ram  := 16#38#;
RAM(71).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(72).addr_memoria_ram  := 16#37#;
RAM(72).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(73).addr_memoria_ram  := 16#36#;
RAM(73).dado_memoria_ram  := "00000000000011001111111100110000";          -- LDR
RAM(74).addr_memoria_ram  := 16#35#;
RAM(74).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(75).addr_memoria_ram  := 16#34#;
RAM(75).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(76).addr_memoria_ram  := 16#33#;
RAM(76).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(77).addr_memoria_ram  := 16#32#;
RAM(77).dado_memoria_ram  := "00000000000011001111111100110000";          -- LDR
RAM(78).addr_memoria_ram  := 16#31#;
RAM(78).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(79).addr_memoria_ram  := 16#30#;
RAM(79).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(80).addr_memoria_ram  := 16#2f#;
RAM(80).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(81).addr_memoria_ram  := 16#2e#;
RAM(81).dado_memoria_ram  := "00000000000011001111111100110000";          -- LDR
RAM(82).addr_memoria_ram  := 16#2d#;
RAM(82).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(83).addr_memoria_ram  := 16#2c#;
RAM(83).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(84).addr_memoria_ram  := 16#2b#;
RAM(84).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(85).addr_memoria_ram  := 16#2a#;
RAM(85).dado_memoria_ram  := "00000000000011001111111100110000";          -- LDR
RAM(86).addr_memoria_ram  := 16#29#;
RAM(86).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(87).addr_memoria_ram  := 16#28#;
RAM(87).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(88).addr_memoria_ram  := 16#27#;
RAM(88).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(89).addr_memoria_ram  := 16#26#;
RAM(89).dado_memoria_ram  := "00000000000011001111111100110000";          -- LDR
RAM(90).addr_memoria_ram  := 16#25#;
RAM(90).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(91).addr_memoria_ram  := 16#24#;
RAM(91).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(92).addr_memoria_ram  := 16#23#;
RAM(92).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(93).addr_memoria_ram  := 16#22#;
RAM(93).dado_memoria_ram  := "00000000000011001111111100110000";          -- LDR
RAM(94).addr_memoria_ram  := 16#21#;
RAM(94).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(95).addr_memoria_ram  := 16#20#;
RAM(95).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(96).addr_memoria_ram  := 16#1f#;
RAM(96).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(97).addr_memoria_ram  := 16#1e#;
RAM(97).dado_memoria_ram  := "00000000000011001111111100110000";          -- LDR
RAM(98).addr_memoria_ram  := 16#1d#;
RAM(98).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(99).addr_memoria_ram  := 16#1c#;
RAM(99).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(100).addr_memoria_ram := 16#1b#;
RAM(100).dado_memoria_ram := "11100011100000000111111100111000";          -- AND
RAM(101).addr_memoria_ram := 16#1a#;
RAM(101).dado_memoria_ram := "00000000000011001111111100110000";          -- LDR
RAM(102).addr_memoria_ram := 16#19#;
RAM(102).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(103).addr_memoria_ram := 16#18#;
```

```
RAM(103).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(104).addr_memoria_ram := 16#17#;
RAM(104).dado_memoria_ram := "11100011100000000111111100111000";          -- AND
RAM(105).addr_memoria_ram := 16#16#;
RAM(105).dado_memoria_ram := "00000000000011001111111100110000";          -- LDR
RAM(106).addr_memoria_ram := 16#15#;
RAM(106).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(107).addr_memoria_ram := 16#14#;
RAM(107).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(108).addr_memoria_ram := 16#13#;
RAM(108).dado_memoria_ram := "11100011100000000111111100111000";          -- AND
RAM(109).addr_memoria_ram := 16#12#;
RAM(109).dado_memoria_ram := "00000000000011001111111100110000";          -- LDR
RAM(110).addr_memoria_ram := 16#11#;
RAM(110).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(111).addr_memoria_ram := 16#10#;
RAM(111).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(112).addr_memoria_ram := 16#0f#;
RAM(112).dado_memoria_ram := "11100011100000000111111100111000";          -- AND
RAM(113).addr_memoria_ram := 16#0e#;
RAM(113).dado_memoria_ram := "00000000000011001111111100110000";          -- LDR
RAM(114).addr_memoria_ram := 16#0d#;
RAM(114).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(115).addr_memoria_ram := 16#0c#;
RAM(115).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(116).addr_memoria_ram := 16#0b#;
RAM(116).dado_memoria_ram := "11100011100000000111111100111000";          -- AND
RAM(117).addr_memoria_ram := 16#0a#;
RAM(117).dado_memoria_ram := "00000000000011001111111100110000";          -- LDR
RAM(118).addr_memoria_ram := 16#09#;
RAM(118).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(119).addr_memoria_ram := 16#08#;
RAM(119).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(120).addr_memoria_ram := 16#07#;
RAM(120).dado_memoria_ram := "11100011100000000111111100111000";          -- AND
RAM(121).addr_memoria_ram := 16#06#;
RAM(121).dado_memoria_ram := "00000000000011001111111100110000";          -- LDR
RAM(122).addr_memoria_ram := 16#05#;
RAM(122).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(123).addr_memoria_ram := 16#04#;
RAM(123).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(124).addr_memoria_ram := 16#03#;
RAM(124).dado_memoria_ram := "11100011100000000111111100111000";          -- AND
RAM(125).addr_memoria_ram := 16#02#;
RAM(125).dado_memoria_ram := "00000000000011001111111100110000";          -- LDR
RAM(126).addr_memoria_ram := 16#01#;
RAM(126).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(127).addr_memoria_ram := 16#00#;
RAM(127).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR

-- Carrega os endereços da iCACHE e alguns dados da linhas de iCACHE simulada
iCACHE(0).addr_memoria_icache  := 16#ff#;
iCACHE(1).addr_memoria_icache  := 16#fe#;
iCACHE(2).addr_memoria_icache  := 16#fd#;
iCACHE(3).addr_memoria_icache  := 16#fc#;
iCACHE(4).addr_memoria_icache  := 16#fb#;
iCACHE(5).addr_memoria_icache  := 16#fa#;
iCACHE(6).addr_memoria_icache  := 16#f9#;
iCACHE(7).addr_memoria_icache  := 16#f8#;
iCACHE(8).addr_memoria_icache  := 16#f7#;
iCACHE(9).addr_memoria_icache  := 16#f6#;
iCACHE(10).addr_memoria_icache := 16#f5#;
iCACHE(11).addr_memoria_icache := 16#f4#;
iCACHE(12).addr_memoria_icache := 16#f3#;
iCACHE(13).addr_memoria_icache := 16#f2#;
iCACHE(14).addr_memoria_icache := 16#f1#;
iCACHE(15).addr_memoria_icache := 16#f0#;
iCACHE(0).dado_memoria_icache  := "00000000000000001111111100000000";
iCACHE(1).dado_memoria_icache  := "00100000001000000000000000000000";
iCACHE(2).dado_memoria_icache  := "00000000000000000000000010000000";
iCACHE(3).dado_memoria_icache  := "00000010000000000000000000000100";
```

```
iCACHE(4).dado_memoria_icache  := "00000000000000000010000001000000";
iCACHE(5).dado_memoria_icache  := "00100000000100000100100010000010";

-- Carrega o ST para a simulação
ST(0).addr_ST := 16#ff#;
ST(1).addr_ST := 16#fe#;
ST(2).addr_ST := 16#fd#;
ST(3).addr_ST := 16#fc#;
ST(4).addr_ST := 16#fb#;
ST(5).addr_ST := 16#fa#;
ST(0).sinal_ST := "10011001100001010000000011111111";
ST(1).sinal_ST := "11000101111000100000000000000001";
ST(2).sinal_ST := "00000000111111011101100011110111";
ST(3).sinal_ST := "00000011111101011000000000000111";
ST(4).sinal_ST := "11000000111111001100010110000011";
ST(5).sinal_ST := "11000101111000101001000011110100";

-- Carrega a primeria posição da LAT para a simulação
LAT(0).addr_old := 16#7f#;       LAT(0).addr_new := 16#ff#;       LAT(0).pair_bytes := '0';
LAT(1).addr_old := 16#7e#;       LAT(1).addr_new := 16#ff#;       LAT(1).pair_bytes := '1';
LAT(2).addr_old := 16#7d#;       LAT(2).addr_new := 16#fe#;       LAT(2).pair_bytes := '0';
LAT(3).addr_old := 16#7c#;       LAT(3).addr_new := 16#fe#;       LAT(3).pair_bytes := '1';
LAT(4).addr_old := 16#7b#;       LAT(4).addr_new := 16#fd#;       LAT(4).pair_bytes := '0';
LAT(5).addr_old := 16#7a#;       LAT(5).addr_new := 16#fd#;       LAT(5).pair_bytes := '1';
LAT(6).addr_old := 16#79#;       LAT(6).addr_new := 16#fc#;       LAT(6).pair_bytes := '0';
LAT(7).addr_old := 16#78#;       LAT(7).addr_new := 16#fc#;       LAT(7).pair_bytes := '1';
LAT(8).addr_old := 16#77#;       LAT(8).addr_new := 16#fb#;       LAT(8).pair_bytes := '0';
LAT(9).addr_old := 16#76#;       LAT(9).addr_new := 16#fb#;       LAT(9).pair_bytes := '1';
LAT(10).addr_old := 16#75#;      LAT(10).addr_new := 16#fa#;      LAT(10).pair_bytes := '0';
LAT(11).addr_old := 16#74#;      LAT(11).addr_new := 16#fa#;      LAT(11).pair_bytes := '1';

-- Carrega os endereços solicitados pelo processador para a simulação
end_inst_proc(0).addr_inst_proc   := 16#47#;
end_inst_proc(1).addr_inst_proc   := 16#6a#;
end_inst_proc(2).addr_inst_proc   := 16#7f#;  -- d
end_inst_proc(3).addr_inst_proc   := 16#27#;
end_inst_proc(4).addr_inst_proc   := 16#26#;
end_inst_proc(5).addr_inst_proc   := 16#0b#;
end_inst_proc(6).addr_inst_proc   := 16#0a#;
end_inst_proc(7).addr_inst_proc   := 16#09#;
end_inst_proc(8).addr_inst_proc   := 16#7d#;  -- d
end_inst_proc(9).addr_inst_proc   := 16#7c#;  -- d
end_inst_proc(10).addr_inst_proc := 16#69#;
end_inst_proc(11).addr_inst_proc := 16#48#;
end_inst_proc(12).addr_inst_proc := 16#67#;
end_inst_proc(13).addr_inst_proc := 16#7b#;  -- d
end_inst_proc(14).addr_inst_proc := 16#68#;
end_inst_proc(15).addr_inst_proc := 16#7a#;  -- d
end_inst_proc(16).addr_inst_proc := 16#79#;  -- d
end_inst_proc(17).addr_inst_proc := 16#66#;
end_inst_proc(18).addr_inst_proc := 16#65#;
end_inst_proc(19).addr_inst_proc := 16#64#;
end_inst_proc(20).addr_inst_proc := 16#70#;
end_inst_proc(21).addr_inst_proc := 16#6b#;
end_inst_proc(22).addr_inst_proc := 16#3d#;
end_inst_proc(23).addr_inst_proc := 16#63#;
end_inst_proc(24).addr_inst_proc := 16#62#;
end_inst_proc(25).addr_inst_proc := 16#6d#;
end_inst_proc(26).addr_inst_proc := 16#6c#;
end_inst_proc(27).addr_inst_proc := 16#77#;  -- d
end_inst_proc(28).addr_inst_proc := 16#5d#;
end_inst_proc(29).addr_inst_proc := 16#5c#;
end_inst_proc(30).addr_inst_proc := 16#75#;  -- d
end_inst_proc(31).addr_inst_proc := 16#5b#;

if (clk'event and clk = '1') then
        -- Carrega os índices da iCACHE e ST
        for i in 0 to (SIZE_CACHE - 1) loop
                iCACHE(i).indice_icache := i;
                ST(i).indice_ST := i;
        end loop;
```

```
-- Carrega os índices da LAT
for i in 0 to ((2 * SIZE_CACHE) - 1) loop
        LAT(i).indice_LAT := i;
end loop;

-- O Processador solicita uma instrução e passa um endereço (end_inst_proc) para a LAT
-- É realizado uma pesquisa na LAT para verificar se existe o endereço passado pelo processador
addr_inst_p <= end_inst_proc(nextpc).addr_inst_proc;

-- Pesquisa o endereço na LAT
flag := '0';
for i in 0 to ((2 * SIZE_CACHE) - 1) loop
        if (LAT(i).addr_old = end_inst_proc(nextpc).addr_inst_proc) then
        -- ACERTO na busca da instrução na iCACHE
                flag := '1';
                new_addr := LAT(i).addr_new;
                if (LAT(i).pair_bytes = '0') then          -- a instrução está na 1ª dupla de bytes
                        parte_dupla := '0';
                        dupla_bytes <= parte_dupla;
                else                                       -- a instrução está na 2ª dupla de bytes
                        parte_dupla := '1';
                        dupla_bytes <= parte_dupla;
                end if;
                i2_LAT <= LAT(i).indice_LAT;
        end if;
end loop;

status <= flag;

if (flag = '1') then
        -- Chama a função de Descompressão
        returnD_inst_proc <= Descompressor(iCACHE, ST, new_addr, parte_dupla);
        -- repassando a instrução para o processador
else
        -- FALHA na busca da instrução na iCACHE, buscará na RAM
        -- Chama a função de Compressão

        -- Então vai na RAM, posição endereço de memória passado pelo processador (end_inst_proc)
        -- 1. Pesquisa a RAM
        -- 2. Retorna ao processador o conteúdo da RAM contido na posição do endereço de memória
passado pelo processador (end_inst_proc)
        --    returnC_inst_proc <= RAM(end_inst_proc).dado_memoria_ram;
        -- 3. Realiza a compressão da instrução e salva na iCACHE e ST
        -- 4. Atualiza a LAT

        ------------------------------------------------------------
        -- 1. Pesquisa na RAM
        -- Inicia o processo de busca da instrução na memória RAM
        for i in 0 to (SIZE_RAM - 1) loop
                -- caso tenha a instrução na RAM
                if (RAM(i).addr_memoria_ram = end_inst_proc(nextpc).addr_inst_proc) then
                        -- a instrução será repassada para o processador
                        dado_ram := RAM(i).dado_memoria_ram;
                        addr_ram := RAM(i).addr_memoria_ram;
                        exit;
                else
                -- caso a RAM não tenha a instrução buscar no HD
                        dado_ram := "00000000000000000000000000000000";
                        addr_ram := 16#00#;
                end if;
        end loop;

        -- 2. Retorna para o processador a instrução da posição do endereço pesquisado
        -- returnC_inst_proc <= RAM(end_inst_proc).dado_memoria_ram;
        returnC_inst_proc <= dado_ram;          -- repassando a instrução para o processador

        -- 3. Compressão da instrução
        -- Chama a função para comprimir e salvar a instrução na iCACHE
        iCACHE := Compressor(iCACHE, ST, dado_ram, ind1, incr_ind);

        -- 4. Atualizar a LAT
```

```
                                        -- Se não existir mais lugar na LAT, zerar o índice, ou seja, começar a sobre-escrever na LAT
                                        if (ind2 > (2 * SIZE_CACHE)) then
                                                ind2 := 0;
                                        end if;
                                        LAT(ind2).addr_old := addr_ram;
                                        LAT(ind2).addr_new := iCACHE(ind1).addr_memoria_icache;
                                        -- se o índice 2 (ind2) for par ou zero o parte_byte recebe 0, ou seja, 1ª dupla de bytes
                                        if ((ind2 mod 2) = 0) then
                                                LAT(ind2).pair_bytes := '0';
                                        else                                    -- senão o parte_byte recebe 1, ou seja, 2ª dupla de bytes
                                                LAT(ind2).pair_bytes := '1';
                                        end if;

                                        i2_LAT <= ind2;
                                        ind2 := ind2 + 1;

                                        -- incrementador do índice 1 que será salvo na iCACHE e na ST
                                        if (incr_ind mod 2 = 1) then
                                                ind1 := ind1 + 1;
                                        end if;
                                end if;

                                incr_ind := incr_ind + 1;
                                nextpc := nextpc + 1;

                        end if;
                end process;

end architecture MIC_arch;
```

D.2. Código Fonte do método de *Huffman*

```
-------------------------------------------------------------------------------
-- Title                : Compressor/Descompressor de Huffman
-- Project              : Arquitetura PDCCM em Hardware para Compressão/Descompressão de
--                        Instruções em Sistemas Embarcados
-------------------------------------------------------------------------------
-- Architecture         : Processador Descompressor iCache Compressor Memória
-------------------------------------------------------------------------------
-- Método               : Huffman
-------------------------------------------------------------------------------
-- File                 : huffman.vhd
-- Author               : Wanderson Roger Azevedo Dias
-- e-mail               : roger@dcc.ufam.edu.br
-- Organization         : Universidade Federal do Amazonas - UFAM
-- Created              : 15/10/2008
-- Last update          : 16/04/2009
-- Plataform            : Cyclone-II
-- Device               : EP2C20F484C7
-- Fabricator           : Altera
-- Simulators           : Quartus-II Web Edition
-- Synthesizers         : Quartus-II Web Edition
-- Version              : 8.0
-- Language             : VHDL
-------------------------------------------------------------------------------
-- Description          : Entidade responsável pela compressão/descompressão dos códigos
--                        de instruções (32 bits) usando o método de Huffman na arquitetura
--                        PDCCM
-------------------------------------------------------------------------------
-- Copyright (c) notice
--
--                      Universidade Federal do Amazonas - UFAM
--                      Instituto de Ciências Exatas - ICE
--                      Departamento de Ciência da Computação - DCC
--                      Programa de Pós-Graduação em Informática - PPGI
```

```
--
--                      Developed by Wanderson Roger Azevedo Dias
--                      This code may be used for educational and non-educational purposes as
--                      long as its copyright notice remains unchanged.
-------------------------------------------------------------------------------
-- Revisions            : 1
-- Revision Number      : 1.4
-- Version              : 2.1.4
-- Date                 : 04/04/2009
-- Description          : Atualização dos comentários
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
--- BIBLIOTECAS DO PROJETO
-------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

-------------------------------------------------------------------------------------------------
--- ENTIDADE DO HUFFMAN
-------------------------------------------------------------------------------------------------
entity huffman is
        generic (
                -- Memórias (RAM e iCACHE)
                SIZE_RAM        : natural := 128;               -- define o tamanho da memória RAM
                SIZE_CACHE      : natural := 32;                -- define o tamanho da memória iCACHE
                SIZE_ADDR       : natural := ((2**8) - 1);       -- define o tamanho máximo do endereço da RAM e iCACHE
                -- Palavra
                SIZE_WORD       : natural := 32                 -- define o tamanho da palavra (32 bits ou 4 bytes)
            );

        port (
                clk             : in  std_logic;                -- sinal de clock
                addr_inst_p     : out natural range SIZE_ADDR downto 0; -- mostra o valor dos endereços da memória RAM
                status          : out bit;                      -- mostra se está havendo uma compressão ou descompressão
                dado_ic         : out bit_vector(32 downto 1);  -- mostra o código da instrução localizado na iCACHE
                inst_compr      : out bit_vector(32 downto 1);  -- mostra o código da instrução comprimida

                returnN_inst_proc : out bit_vector(32 downto 1);-- retorno da instrução normal para o processador
                returnD_inst_proc : out bit_vector(32 downto 1);-- retorno da instrução descomprimida para o processador
                returnC_inst_proc : out bit_vector(32 downto 1) -- retorno da instrução normal e a instrução que estava
            );                                                  -- comprimida para o processador
end entity huffman;

-------------------------------------------------------------------------------------------------
--- ARQUITETURA DO HUFFMAN
-------------------------------------------------------------------------------------------------
architecture huffman_arch of huffman is

        subtype wordT is bit_vector(SIZE_WORD downto 1);


--- 0. Endereços do Processador para simulações
-------------------------------------------------------------------------------------------------
--- Tabela que contém os endereços solicitados pelo processador para a execução da simulação
type reg_inst_proc is record
        addr_inst_proc : natural range SIZE_ADDR downto 0;      -- campo dos endereços solicitados pelo processador
end record;

type vetor_instP is array (0 to (SIZE_RAM - 1)) of reg_inst_proc;


--- 1. Memória RAM
-------------------------------------------------------------------------------------------------
--- Tabela que simula uma memória RAM (Address e Códigos)
type reg_memoria_ram is record
        addr_memoria_ram : natural range SIZE_ADDR downto 0;    -- campo do endereço da memória RAM
        dado_memoria_ram : wordT;                               -- campo com o código da instrução
end record;
```

```
type vetor_mRAM     is array (0 to (SIZE_RAM - 1)) of reg_memoria_ram;


--- 2. Memória iCACHE
-------------------------------------------------------------------------------------------
--- Tabela que simula uma iCACHE (Índice, Address e Códigos)
--- Obs.: Todas as instruções contidas na iCACHE estão comprimido
type reg_memoria_icache is record
            indice_icache           : natural range (SIZE_CACHE - 1) downto 0;          -- campo do índice da memória iCACHE
            addr_memoria_icache : natural range SIZE_ADDR downto 0;                 -- campo do endereço da memória iCACHE
            dado_memoria_icache : wordT;                            -- campo com código da instrução normal ou comprimidas na iCACHE
end record;

type vetor_iCACHE is array (0 to (SIZE_CACHE - 1)) of reg_memoria_icache;


--- 3. Tabela da Linha de Endereços - LAT (Line Address Table)
-------------------------------------------------------------------------------------------
--- Tabela que conterá o mapeamento das instruções na iCACHE e seu status (normal ou comprimida)
type reg_LAT is record
            indice_LAT  : natural range (SIZE_CACHE - 1) downto 0;          -- campo do índice da LAT
            addr_inst   : natural range SIZE_ADDR downto 0;                 -- campo com o endereço da instrução na iCACHE
            type_inst   : bit;                                              -- campo que indica se a instrução está normal ou comprimida
end record;                                                                 -- (type_inst = 0 "normal" e type_inst = 1 "comprimida")

type vetor_LAT is array (0 to (SIZE_CACHE - 1)) of reg_LAT;


--- 4. Tabela de Huffman - HT (Huffman Table)
-------------------------------------------------------------------------------------------
--- Tabela que conterá a árvore de Huffman
type reg_HT is record
            indice_HT   : natural range (SIZE_CACHE - 1) downto 0;          -- campo do índice da HT
            codInst_HT  : wordT;                                            -- campo que contém o código da instrução normal
            freq_HT     : natural;                                          -- campo que contém a freqüência de repetição das instruções
            codHuff_HT : wordT;                                             -- campo que contém o novo valor das instruções comprimidas
end record;

type vetor_HT is array (0 to (SIZE_CACHE - 1)) of reg_HT;

-------------------------------------------------------------------------------------------
--- FUNÇÃO DO COMPRESSOR DE HUFFMAN
-------------------------------------------------------------------------------------------
---
--- Esta função é responsável pela compressão das instruções contidas na iCACHE que mais se
--- repetem usando o médoto de Huffman
---
-------------------------------------------------------------------------------------------
function CompressorHuff(iCACHE : vetor_iCACHE;
                        LAT     : vetor_LAT;
                        HT      : vetor_HT;
                        ind1    : natural)
return vetor_iCACHE is

variable cach   : vetor_iCACHE;         -- receberá o conteúdo da iCACHE para inserir as instruções (normais ou comprimidas)
variable tb_lat : vetor_LAT;            -- receberá o conteúdo da LAT para atualiza-lá
variable tb_h   : vetor_HT;             -- receberá o conteúdo da HT para inserir as instruções
variable aux_tb_h : reg_HT;             -- receberá o registro da HT para realizar a ordenação das instruções pela freqüência
variable cont   : natural := 0;         -- será usada para indexar a tb_h
variable id1    : natural := 0;         -- receberá o índice da iCACHE
variable ok     : bit;                  -- serve como um flag para verificar se a instrução já está ou não contida na tb_h
variable comp   : bit := '0';           -- serve como um índice para a tb_h
variable line_instruction : wordT;      -- receberá a instrução da iCACHE se a mesma ainda não estiver contida na HT

begin
            cach  := iCACHE;
            tb_lat := LAT;
            tb_h  := HT;
            id1   := ind1;

            for i in 0 to (SIZE_CACHE - 1) loop
```

```
		-- teste para quando a função for chamada pela segunda vez em diante, analisar
		-- apenas as instruções comprimidas
		if (tb_lat(i).type_inst /= '1') then

			-- pesquisa se a instrução já está ou não na tb_h, ou seja, não permite
			-- que a mesma instrução seja inserida duas ou mais vezes na HT
			ok := '0';
			for k in 0 to (SIZE_CACHE - 1) loop
				if (iCACHE(i).dado_memoria_icache = tb_h(k).codInst_HT) then
					ok := '1';
					-- significa que não é a primeira vez que a instrução é carregada na memória RAM
					if (tb_lat(i).type_inst = '1') then
						tb_h(cont).freq_HT := tb_h(cont).freq_HT + 1;
					end if;
				end if;
			end loop;

			-- se a instrução ainda não esteja inseria na HT, insere a instrução uma vez na HT
			-- e começa a contar quantas vezes essa instrução se repete na iCACHE
			if (ok = '0') then
				line_instruction := iCACHE(i).dado_memoria_icache;
				for x in 0 to (SIZE_CACHE - 1) loop
					if (line_instruction = iCACHE(x).dado_memoria_icache) then
						tb_h(cont).codInst_HT := line_instruction;
						tb_h(cont).freq_HT := tb_h(cont).freq_HT + 1;
					end if;
				end loop;
				cont := cont + 1;
			end if;
		end if;
end loop;

-- ordena a Tabela de Huffman de forma decrescente, pelo campo freqüência do maior para o menor
for i in 1 to (SIZE_CACHE - 1) loop
		-- se a freq(i) for > que a freq(i+1), então troca as posições na HT
		if (tb_h(i).freq_HT > tb_h(i-1).freq_HT) then
			aux_tb_h.codInst_HT := tb_h(i-1).codInst_HT;
			aux_tb_h.freq_HT    := tb_h(i-1).freq_HT;

			tb_h(i-1).codInst_HT := tb_h(i).codInst_HT;
			tb_h(i-1).freq_HT    := tb_h(i).freq_HT;

			tb_h(i).codInst_HT := aux_tb_h.codInst_HT;
			tb_h(i).freq_HT    := aux_tb_h.freq_HT;
		end if;
end loop;

-- valores da árvore de huffman
tb_h(0).codHuff_HT  := "11111111111111111111111111111111";
tb_h(1).codHuff_HT  := "01111111111111111111111111111111";
tb_h(2).codHuff_HT  := "00111111111111111111111111111111";
tb_h(3).codHuff_HT  := "00011111111111111111111111111111";
tb_h(4).codHuff_HT  := "00001111111111111111111111111111";
tb_h(5).codHuff_HT  := "00000111111111111111111111111111";
tb_h(6).codHuff_HT  := "00000011111111111111111111111111";
tb_h(7).codHuff_HT  := "00000001111111111111111111111111";
tb_h(8).codHuff_HT  := "00000000111111111111111111111111";
tb_h(9).codHuff_HT  := "00000000011111111111111111111111";
tb_h(10).codHuff_HT := "00000000001111111111111111111111";
tb_h(11).codHuff_HT := "00000000000111111111111111111111";
tb_h(12).codHuff_HT := "00000000000011111111111111111111";
tb_h(13).codHuff_HT := "00000000000001111111111111111111";
tb_h(14).codHuff_HT := "00000000000000111111111111111111";
tb_h(15).codHuff_HT := "00000000000000011111111111111111";
tb_h(16).codHuff_HT := "00000000000000001111111111111111";
tb_h(17).codHuff_HT := "00000000000000000111111111111111";
tb_h(18).codHuff_HT := "00000000000000000011111111111111";
tb_h(19).codHuff_HT := "00000000000000000001111111111111";
tb_h(20).codHuff_HT := "00000000000000000000111111111111";
tb_h(21).codHuff_HT := "00000000000000000000011111111111";
tb_h(22).codHuff_HT := "00000000000000000000001111111111";
```

```
		tb_h(23).codHuff_HT := "00000000000000000000000111111111";
		tb_h(24).codHuff_HT := "00000000000000000000000011111111";
		tb_h(25).codHuff_HT := "00000000000000000000000001111111";
		tb_h(26).codHuff_HT := "00000000000000000000000000111111";
		tb_h(27).codHuff_HT := "00000000000000000000000000011111";
		tb_h(28).codHuff_HT := "00000000000000000000000000001111";
		tb_h(29).codHuff_HT := "00000000000000000000000000000111";
		tb_h(30).codHuff_HT := "00000000000000000000000000000011";
		tb_h(31).codHuff_HT := "00000000000000000000000000000001";

		-- salva os novos valores na iCACHE se a freqüência de repetição
		-- da instrução for maior que 1 e em seguida atualiza a LAT
		for i in 0 to (SIZE_CACHE - 1) loop
			if (tb_h(i).freq_HT > 1) then	-- a instrução salva foi comprimida (valor de Huffman)
				cach(i).dado_memoria_icache := tb_h(i).codHuff_HT;
				tb_lat(i).type_inst := '1';
			else				-- a instrução salva foi normal, ou seja, não comprimida
				cach(i).dado_memoria_icache := tb_h(i).codInst_HT;
				tb_lat(i).type_inst := '0';
			end if;
			tb_lat(i).addr_inst := cach(i).addr_memoria_icache;
		end loop;

		-- atribui o novo valor para o índice 1
		id1 := cont - 1;

		return cach;
end function;

-------------------------------------------------------------------------------------------
--- FUNÇÃO DO DESCOMPRESSOR DE HUFFMAN
-------------------------------------------------------------------------------------------
---
--- Esta função é responsável pela descompressão das instruções comprimidas que estão na
--- iCACHE e serão repassadas para o processador
---
-------------------------------------------------------------------------------------------
function DescompressorHuff(dado_icache	: wordT;
			   HT		: vetor_HT)
return wordT is

variable tb_h : vetor_HT;		-- receberá o conteúdo da HT para descomprimir a instrução
variable cont : natural;		-- serve como índice para a árvore de Huffman
variable instruction : wordT;	-- receberá o código da instrução descomprimida que será repassado p/ o processador

begin
		tb_h := HT;

		if (dado_icache = "11111111111111111111111111111111") then
			cont := 0;
		elsif (dado_icache = "01111111111111111111111111111111") then
			cont := 1;
		elsif (dado_icache = "00111111111111111111111111111111") then
			cont := 2;
		elsif (dado_icache = "00011111111111111111111111111111") then
			cont := 3;
		elsif (dado_icache = "00001111111111111111111111111111") then
			cont := 4;
		elsif (dado_icache = "00000111111111111111111111111111") then
			cont := 5;
		elsif (dado_icache = "00000011111111111111111111111111") then
			cont := 6;
		elsif (dado_icache = "00000001111111111111111111111111") then
			cont := 7;
		elsif (dado_icache = "00000000111111111111111111111111") then
			cont := 8;
		elsif (dado_icache = "00000000011111111111111111111111") then
			cont := 9;
		elsif (dado_icache = "00000000001111111111111111111111") then
			cont := 10;
		elsif (dado_icache = "00000000000111111111111111111111") then
```

```
                cont := 11;
        elsif (dado_icache = "00000000000011111111111111111111") then
                cont := 12;
        elsif (dado_icache = "00000000000001111111111111111111") then
                cont := 13;
        elsif (dado_icache = "00000000000000111111111111111111") then
                cont := 14;
        elsif (dado_icache = "00000000000000011111111111111111") then
                cont := 15;
        elsif (dado_icache = "00000000000000001111111111111111") then
                cont := 16;
        elsif (dado_icache = "00000000000000000111111111111111") then
                cont := 17;
        elsif (dado_icache = "00000000000000000011111111111111") then
                cont := 18;
        elsif (dado_icache = "00000000000000000001111111111111") then
                cont := 19;
        elsif (dado_icache = "00000000000000000000111111111111") then
                cont := 20;
        elsif (dado_icache = "00000000000000000000011111111111") then
                cont := 21;
        elsif (dado_icache = "00000000000000000000001111111111") then
                cont := 22;
        elsif (dado_icache = "00000000000000000000000111111111") then
                cont := 23;
        elsif (dado_icache = "00000000000000000000000011111111") then
                cont := 24;
        elsif (dado_icache = "00000000000000000000000001111111") then
                cont := 25;
        elsif (dado_icache = "00000000000000000000000000111111") then
                cont := 26;
        elsif (dado_icache = "00000000000000000000000000011111") then
                cont := 27;
        elsif (dado_icache = "00000000000000000000000000001111") then
                cont := 28;
        elsif (dado_icache = "00000000000000000000000000000111") then
                cont := 29;
        elsif (dado_icache = "00000000000000000000000000000011") then
                cont := 30;
        elsif (dado_icache = "00000000000000000000000000000001") then
                cont := 31;
        end if;

        instruction := tb_h(cont).codInst_HT;

        return instruction;
end function;

-------------------------------------------------------------------------------------------
--- PROGRAMA PRINCIPAL
-------------------------------------------------------------------------------------------
begin
        process (clk)
                variable RAM            : vetor_mRAM;
                variable iCACHE         : vetor_iCACHE;
                variable LAT            : vetor_LAT;
                variable HT             : vetor_HT;
                variable end_inst_proc : vetor_instP;

                variable dado_icache   : wordT;                                 -- conterá a instrução buscada na iCACHE
                variable dado_ram      : wordT;                                 -- conterá a instrução buscada na RAM
                variable addr_ram      : natural range SIZE_ADDR downto 0;      -- conterá o endereço da instrução buscada na RAM
                variable flag          : bit := '0';            -- servirá como flag para saber se houve ACERTO ou FALHA na LAT

                variable tipo      : bit;               -- conterá o status da instrução na iCACHE (normal ou comprimida)
                variable ind1      : natural range 0 to (SIZE_CACHE - 1);  -- será usado para indexar a iCACHE e a HT
                variable nextpc    : natural := 0;    -- usado para incrementar o índice do endereços solicitados pelo processador

        begin
                -- Carrega os endereços e valores da memória RAM simulada
```

```
-- benchmark MiBench: CRC32
-- Processador: ARM
-- Família: ARM9
-- versão: ARM922T
-- ISA: ARMv4T

-- Código em Assembly compilado para ARM

RAM(0).addr_memoria_ram    := 16#7f#;
RAM(0).dado_memoria_ram    := "10101010101010101111111111111111";          -- MOV
RAM(1).addr_memoria_ram    := 16#7e#;
RAM(1).dado_memoria_ram    := "11000011110000111100000000110011";          -- STDMF
RAM(2).addr_memoria_ram    := 16#7d#;
RAM(2).dado_memoria_ram    := "00000000000000000000000000000011";          -- SUB
RAM(3).addr_memoria_ram    := 16#7c#;
RAM(3).dado_memoria_ram    := "11111000001100111111111000001100";          -- LDR
RAM(4).addr_memoria_ram    := 16#7b#;
RAM(4).dado_memoria_ram    := "11110011110000000111111100111111";          -- ANDS
RAM(5).addr_memoria_ram    := 16#7a#;
RAM(5).dado_memoria_ram    := "00000000000000001111111111110011";          -- BEQ
RAM(6).addr_memoria_ram    := 16#79#;
RAM(6).dado_memoria_ram    := "11000000000000000000000000011111";          -- SUBS
RAM(7).addr_memoria_ram    := 16#78#;
RAM(7).dado_memoria_ram    := "00000000000001111111111100110011";          -- BLT
RAM(8).addr_memoria_ram    := 16#77#;
RAM(8).dado_memoria_ram    := "11111000001100111110000000001111";          -- LDRB
RAM(9).addr_memoria_ram    := 16#76#;
RAM(9).dado_memoria_ram    := "11110000000000001111111111110000";          -- EOR
RAM(10).addr_memoria_ram   := 16#75#;
RAM(10).dado_memoria_ram   := "11100011100000000111111100111000";          -- AND
RAM(11).addr_memoria_ram   := 16#74#;
RAM(11).dado_memoria_ram   := "11111000001100111111111000001100";          -- LDR
RAM(12).addr_memoria_ram   := 16#73#;
RAM(12).dado_memoria_ram   := "11110000000000001111111111110000";          -- EOR
RAM(13).addr_memoria_ram   := 16#72#;
RAM(13).dado_memoria_ram   := "00000000000011111111111111110000";          -- B
RAM(14).addr_memoria_ram   := 16#71#;
RAM(14).dado_memoria_ram   := "10110000000000000011110000000011";          -- LDMIA
RAM(15).addr_memoria_ram   := 16#70#;
RAM(15).dado_memoria_ram   := "11110000000000001111111111110000";          -- EOR
RAM(16).addr_memoria_ram   := 16#6f#;
RAM(16).dado_memoria_ram   := "11100011100000000111111100111000";          -- AND
RAM(17).addr_memoria_ram   := 16#6e#;
RAM(17).dado_memoria_ram   := "11111000001100111111111000001100";          -- LDR
RAM(18).addr_memoria_ram   := 16#6d#;
RAM(18).dado_memoria_ram   := "11110000000000001111111111110000";          -- EOR
RAM(19).addr_memoria_ram   := 16#6c#;
RAM(19).dado_memoria_ram   := "11110000000000001111111111110000";          -- EOR
RAM(20).addr_memoria_ram   := 16#6b#;
RAM(20).dado_memoria_ram   := "11100011100000000111111100111000";          -- AND
RAM(21).addr_memoria_ram   := 16#6a#;
RAM(21).dado_memoria_ram   := "11111000001100111111111000001100";          -- LDR
RAM(22).addr_memoria_ram   := 16#69#;
RAM(22).dado_memoria_ram   := "11110000000000001111111111110000";          -- EOR
RAM(23).addr_memoria_ram   := 16#68#;
RAM(23).dado_memoria_ram   := "11110000000000001111111111110000";          -- EOR
RAM(24).addr_memoria_ram   := 16#67#;
RAM(24).dado_memoria_ram   := "11100011100000000111111100111000";          -- AND
RAM(25).addr_memoria_ram   := 16#66#;
RAM(25).dado_memoria_ram   := "11111000001100111111111000001100";          -- LDR
RAM(26).addr_memoria_ram   := 16#65#;
RAM(26).dado_memoria_ram   := "11110000000000001111111111110000";          -- EOR
RAM(27).addr_memoria_ram   := 16#64#;
RAM(27).dado_memoria_ram   := "11110000000000001111111111110000";          -- EOR
RAM(28).addr_memoria_ram   := 16#63#;
RAM(28).dado_memoria_ram   := "11100011100000000111111100111000";          -- AND
RAM(29).addr_memoria_ram   := 16#62#;
RAM(29).dado_memoria_ram   := "11111000001100111111111000001100";          -- LDR
RAM(30).addr_memoria_ram   := 16#61#;
RAM(30).dado_memoria_ram   := "11110000000000001111111111110000";          -- EOR
RAM(31).addr_memoria_ram   := 16#60#;
```

```
RAM(31).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(32).addr_memoria_ram  := 16#5f#;
RAM(32).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(33).addr_memoria_ram  := 16#5e#;
RAM(33).dado_memoria_ram  := "11111000001100111111111000001100";          -- LDR
RAM(34).addr_memoria_ram  := 16#5d#;
RAM(34).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(35).addr_memoria_ram  := 16#5c#;
RAM(35).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(36).addr_memoria_ram  := 16#5b#;
RAM(36).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(37).addr_memoria_ram  := 16#5a#;
RAM(37).dado_memoria_ram  := "11111000001100111111111000001100";          -- LDR
RAM(38).addr_memoria_ram  := 16#59#;
RAM(38).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(39).addr_memoria_ram  := 16#58#;
RAM(39).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(40).addr_memoria_ram  := 16#57#;
RAM(40).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(41).addr_memoria_ram  := 16#56#;
RAM(41).dado_memoria_ram  := "11111000001100111111111000001100";          -- LDR
RAM(42).addr_memoria_ram  := 16#55#;
RAM(42).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(43).addr_memoria_ram  := 16#54#;
RAM(43).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(44).addr_memoria_ram  := 16#53#;
RAM(44).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(45).addr_memoria_ram  := 16#52#;
RAM(45).dado_memoria_ram  := "00000000000001100111111100110000";          -- LDR
RAM(46).addr_memoria_ram  := 16#51#;
RAM(46).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(47).addr_memoria_ram  := 16#50#;
RAM(47).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(48).addr_memoria_ram  := 16#4f#;
RAM(48).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(49).addr_memoria_ram  := 16#4e#;
RAM(49).dado_memoria_ram  := "00000000000001100111111100110000";          -- LDR
RAM(50).addr_memoria_ram  := 16#4d#;
RAM(50).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(51).addr_memoria_ram  := 16#4c#;
RAM(51).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(52).addr_memoria_ram  := 16#4b#;
RAM(52).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(53).addr_memoria_ram  := 16#4a#;
RAM(53).dado_memoria_ram  := "00000000000001100111111100110000";          -- LDR
RAM(54).addr_memoria_ram  := 16#49#;
RAM(54).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(55).addr_memoria_ram  := 16#48#;
RAM(55).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(56).addr_memoria_ram  := 16#47#;
RAM(56).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(57).addr_memoria_ram  := 16#46#;
RAM(57).dado_memoria_ram  := "00000000000001100111111100110000";          -- LDR
RAM(58).addr_memoria_ram  := 16#45#;
RAM(58).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(59).addr_memoria_ram  := 16#44#;
RAM(59).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(60).addr_memoria_ram  := 16#43#;
RAM(60).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(61).addr_memoria_ram  := 16#42#;
RAM(61).dado_memoria_ram  := "00000000000001100111111100110000";          -- LDR
RAM(62).addr_memoria_ram  := 16#41#;
RAM(62).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(63).addr_memoria_ram  := 16#40#;
RAM(63).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
RAM(64).addr_memoria_ram  := 16#3f#;
RAM(64).dado_memoria_ram  := "11100011100000000111111100111000";          -- AND
RAM(65).addr_memoria_ram  := 16#3e#;
RAM(65).dado_memoria_ram  := "00000000000001100111111100110000";          -- LDR
RAM(66).addr_memoria_ram  := 16#3d#;
RAM(66).dado_memoria_ram  := "11110000000000001111111111110000";          -- EOR
```

```
RAM(67).addr_memoria_ram  := 16#3c#;
RAM(67).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(68).addr_memoria_ram  := 16#3b#;
RAM(68).dado_memoria_ram  := "11100011100000000111111100111000";              -- AND
RAM(69).addr_memoria_ram  := 16#3a#;
RAM(69).dado_memoria_ram  := "00000000000011001111111100110000";-             - LDR
RAM(70).addr_memoria_ram  := 16#39#;
RAM(70).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(71).addr_memoria_ram  := 16#38#;
RAM(71).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(72).addr_memoria_ram  := 16#37#;
RAM(72).dado_memoria_ram  := "11100011100000000111111100111000";              -- AND
RAM(73).addr_memoria_ram  := 16#36#;
RAM(73).dado_memoria_ram  := "00000000000011001111111100110000";              -- LDR
RAM(74).addr_memoria_ram  := 16#35#;
RAM(74).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(75).addr_memoria_ram  := 16#34#;
RAM(75).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(76).addr_memoria_ram  := 16#33#;
RAM(76).dado_memoria_ram  := "11100011100000000111111100111000";              -- AND
RAM(77).addr_memoria_ram  := 16#32#;
RAM(77).dado_memoria_ram  := "00000000000011001111111100110000";              -- LDR
RAM(78).addr_memoria_ram  := 16#31#;
RAM(78).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(79).addr_memoria_ram  := 16#30#;
RAM(79).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(80).addr_memoria_ram  := 16#2f#;
RAM(80).dado_memoria_ram  := "11100011100000000111111100111000";              -- AND
RAM(81).addr_memoria_ram  := 16#2e#;
RAM(81).dado_memoria_ram  := "00000000000011001111111100110000";              -- LDR
RAM(82).addr_memoria_ram  := 16#2d#;
RAM(82).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(83).addr_memoria_ram  := 16#2c#;
RAM(83).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(84).addr_memoria_ram  := 16#2b#;
RAM(84).dado_memoria_ram  := "11100011100000000111111100111000";              -- AND
RAM(85).addr_memoria_ram  := 16#2a#;
RAM(85).dado_memoria_ram  := "00000000000011001111111100110000";              -- LDR
RAM(86).addr_memoria_ram  := 16#29#;
RAM(86).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(87).addr_memoria_ram  := 16#28#;
RAM(87).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(88).addr_memoria_ram  := 16#27#;
RAM(88).dado_memoria_ram  := "11100011100000000111111100111000";              -- AND
RAM(89).addr_memoria_ram  := 16#26#;
RAM(89).dado_memoria_ram  := "00000000000011001111111100110000";              -- LDR
RAM(90).addr_memoria_ram  := 16#25#;
RAM(90).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(91).addr_memoria_ram  := 16#24#;
RAM(91).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(92).addr_memoria_ram  := 16#23#;
RAM(92).dado_memoria_ram  := "11100011100000000111111100111000";              -- AND
RAM(93).addr_memoria_ram  := 16#22#;
RAM(93).dado_memoria_ram  := "00000000000011001111111100110000";              -- LDR
RAM(94).addr_memoria_ram  := 16#21#;
RAM(94).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(95).addr_memoria_ram  := 16#20#;
RAM(95).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(96).addr_memoria_ram  := 16#1f#;
RAM(96).dado_memoria_ram  := "11100011100000000111111100111000";              -- AND
RAM(97).addr_memoria_ram  := 16#1e#;
RAM(97).dado_memoria_ram  := "00000000000011001111111100110000";              -- LDR
RAM(98).addr_memoria_ram  := 16#1d#;
RAM(98).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(99).addr_memoria_ram  := 16#1c#;
RAM(99).dado_memoria_ram  := "11110000000000001111111111110000";              -- EOR
RAM(100).addr_memoria_ram := 16#1b#;
RAM(100).dado_memoria_ram := "11100011100000000111111100111000";              -- AND
RAM(101).addr_memoria_ram := 16#1a#;
RAM(101).dado_memoria_ram := "00000000000011001111111100110000";              -- LDR
RAM(102).addr_memoria_ram := 16#19#;
```

```
RAM(102).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(103).addr_memoria_ram := 16#18#;
RAM(103).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(104).addr_memoria_ram := 16#17#;
RAM(104).dado_memoria_ram := "11100011100000000111111100111000";          -- AND
RAM(105).addr_memoria_ram := 16#16#;
RAM(105).dado_memoria_ram := "00000000000011001111111100110000";          -- LDR
RAM(106).addr_memoria_ram := 16#15#;
RAM(106).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(107).addr_memoria_ram := 16#14#;
RAM(107).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(108).addr_memoria_ram := 16#13#;
RAM(108).dado_memoria_ram := "11100011100000000111111100111000";          -- AND
RAM(109).addr_memoria_ram := 16#12#;
RAM(109).dado_memoria_ram := "00000000000011001111111100110000";          -- LDR
RAM(110).addr_memoria_ram := 16#11#;
RAM(110).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(111).addr_memoria_ram := 16#10#;
RAM(111).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(112).addr_memoria_ram := 16#0f#;
RAM(112).dado_memoria_ram := "11100011100000000111111100111000";          -- AND
RAM(113).addr_memoria_ram := 16#0e#;
RAM(113).dado_memoria_ram := "00000000000011001111111100110000";          -- LDR
RAM(114).addr_memoria_ram := 16#0d#;
RAM(114).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(115).addr_memoria_ram := 16#0c#;
RAM(115).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(116).addr_memoria_ram := 16#0b#;
RAM(116).dado_memoria_ram := "11100011100000000111111100111000";          -- AND
RAM(117).addr_memoria_ram := 16#0a#;
RAM(117).dado_memoria_ram := "00000000000011001111111100110000";          -- LDR
RAM(118).addr_memoria_ram := 16#09#;
RAM(118).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(119).addr_memoria_ram := 16#08#;
RAM(119).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(120).addr_memoria_ram := 16#07#;
RAM(120).dado_memoria_ram := "11100011100000000111111100111000";          -- AND
RAM(121).addr_memoria_ram := 16#06#;
RAM(121).dado_memoria_ram := "00000000000011001111111100110000";          -- LDR
RAM(122).addr_memoria_ram := 16#05#;
RAM(122).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(123).addr_memoria_ram := 16#04#;
RAM(123).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(124).addr_memoria_ram := 16#03#;
RAM(124).dado_memoria_ram := "11100011100000000111111100111000";          -- AND
RAM(125).addr_memoria_ram := 16#02#;
RAM(125).dado_memoria_ram := "00000000000011001111111100110000";          -- LDR
RAM(126).addr_memoria_ram := 16#01#;
RAM(126).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR
RAM(127).addr_memoria_ram := 16#00#;
RAM(127).dado_memoria_ram := "11110000000000001111111111110000";          -- EOR

-- Carrega a LAT simulada na descompressão
LAT(0).addr_inst  := 16#00#;
LAT(1).addr_inst  := 16#00#;
LAT(2).addr_inst  := 16#00#;
LAT(3).addr_inst  := 16#00#;
LAT(4).addr_inst  := 16#00#;
LAT(5).addr_inst  := 16#00#;
LAT(6).addr_inst  := 16#00#;
LAT(7).addr_inst  := 16#00#;
LAT(8).addr_inst  := 16#00#;
LAT(9).addr_inst  := 16#00#;
LAT(10).addr_inst := 16#00#;
LAT(11).addr_inst := 16#00#;
LAT(12).addr_inst := 16#00#;
LAT(13).addr_inst := 16#00#;
LAT(14).addr_inst := 16#00#;
LAT(15).addr_inst := 16#00#;
```

```
-- Carrega os endereços solicitados pelo processador para a simulação
end_inst_proc(0).addr_inst_proc  := 16#47#;
end_inst_proc(1).addr_inst_proc  := 16#6a#;
end_inst_proc(2).addr_inst_proc  := 16#7f#;
end_inst_proc(3).addr_inst_proc  := 16#27#;
end_inst_proc(4).addr_inst_proc  := 16#26#;
end_inst_proc(5).addr_inst_proc  := 16#0b#;
end_inst_proc(6).addr_inst_proc  := 16#0a#;
end_inst_proc(7).addr_inst_proc  := 16#09#;
end_inst_proc(8).addr_inst_proc  := 16#7d#;
end_inst_proc(9).addr_inst_proc  := 16#7c#;
end_inst_proc(10).addr_inst_proc := 16#69#;
end_inst_proc(11).addr_inst_proc := 16#48#;
end_inst_proc(12).addr_inst_proc := 16#67#;
end_inst_proc(13).addr_inst_proc := 16#7b#;
end_inst_proc(14).addr_inst_proc := 16#68#;
end_inst_proc(15).addr_inst_proc := 16#7a#;
end_inst_proc(16).addr_inst_proc := 16#79#;
end_inst_proc(17).addr_inst_proc := 16#66#;
end_inst_proc(18).addr_inst_proc := 16#65#;
end_inst_proc(19).addr_inst_proc := 16#64#;
end_inst_proc(20).addr_inst_proc := 16#70#;
end_inst_proc(21).addr_inst_proc := 16#6b#;
end_inst_proc(22).addr_inst_proc := 16#3d#;
end_inst_proc(23).addr_inst_proc := 16#63#;
end_inst_proc(24).addr_inst_proc := 16#62#;
end_inst_proc(25).addr_inst_proc := 16#6d#;
end_inst_proc(26).addr_inst_proc := 16#6c#;
end_inst_proc(27).addr_inst_proc := 16#77#;
end_inst_proc(28).addr_inst_proc := 16#5d#;
end_inst_proc(29).addr_inst_proc := 16#5c#;
end_inst_proc(30).addr_inst_proc := 16#75#;
end_inst_proc(31).addr_inst_proc := 16#5b#;

if (clk'event and clk = '1') then

        -- Carrega os índices da iCACHE, LAT e HT
        for i in 0 to (SIZE_CACHE - 1) loop
                iCACHE(i).indice_icache := i;
                LAT(i).indice_LAT := i;
                HT(i).indice_HT := i;
        end loop;

        -- O Processador solicita uma instrução e passa um endereço (end_inst_proc) para a LAT
        -- É realizado uma pesquisa na LAT para verificar se existe o endereço passado pelo processador
        addr_inst_p <= end_inst_proc(nextpc).addr_inst_proc;

        -- Pesquisa o endereço na LAT e retorna para variável flag: 1 = ACETO ou 0 = FALHA (Default)
        for i in 0 to (SIZE_CACHE - 1) loop
                if (LAT(i).addr_inst = end_inst_proc(nextpc).addr_inst_proc) then
                        -- ACERTO na busca da instrução na LAT
                        flag := '1';
                        tipo := LAT(i).type_inst;
                        -- Pesquisa a instrução na iCACHE
                        for x in 0 to (SIZE_CACHE - 1) loop
                                if (iCACHE(x).addr_memoria_icache =
end_inst_proc(nextpc).addr_inst_proc) then
                                        dado_icache := iCACHE(x).dado_memoria_icache;
                                        dado_ic <= dado_icache;
                                        exit;
                                end if;
                        end loop;
                end if;
        end loop;

        if (flag = '1') then
                if (tipo = '0') then      -- a instrução está normal (não comprimida)
                                          -- só retorna a instrução para o processador
                        returnN_inst_proc <= dado_icache;
                else                      -- a instrução está comprimida
                -- envia a instrução comprimida para o descompressor e retorna para o processador
```

```
                                        returnD_inst_proc <= DescompressorHuff(dado_icache, HT);
                                end if;
                                status <= flag;
                        else                    -- FALHA na busca da instrução na LAT
                                -- 1. Pesquisa a RAM
                                -- 2. Retorna ao processador o conteúdo da RAM contido na posição do endereço de memória
passado pelo processador (end_inst_proc)
                                --    returnC_inst_proc <= RAM(end_inst_proc).dado_memoria_ram;
                                -- 3. Salva a instrução na iCACHE
                                -- 4. Realiza a pesquisa para a compressão
                                -- 5. Atualiza a LAT

                                ------------------------------------------------------------
                                -- 1. Pesquisa na RAM
                                -- Inicia o processo de busca da instrução na memória RAM
                                for i in 0 to (SIZE_RAM - 1) loop
                                        -- se a instrução estiver na RAM uma cópia será repassada para o processador
                                        if    (RAM(i).addr_memoria_ram    =    end_inst_proc(nextpc).addr_inst_proc)    then
                                                dado_ram := RAM(i).dado_memoria_ram;
                                                addr_ram := RAM(i).addr_memoria_ram;
                                                exit;
                                        else
                                        -- se a instrução não estiver na RAM, será buscada no HD
                                                dado_ram := "00000000000000000000000000000000";
                                        end if;
                                end loop;

                                ------------------------------------------------------------
                                -- 2. Retorna para o processador a instrução da posição do endereço pesquisado
                                -- returnC_inst_proc <= RAM(end_inst_proc).dado_memoria_ram;
                                returnC_inst_proc <= dado_ram;       -- repassando uma cópia da instrução para o processador

                                ------------------------------------------------------------
                                -- 3. Salvar a instrução na iCACHE
                                iCACHE(ind1).addr_memoria_icache := addr_ram;
                                iCACHE(ind1).dado_memoria_icache := dado_ram;

                                ------------------------------------------------------------
                                -- 4. Realiza a pesquisa para a compressão
                                -- toda vez que a iCACHE estiver cheia
                                if (ind1 = (SIZE_CACHE - 1)) then
                                        iCACHE := CompressorHuff(iCACHE, LAT, HT, ind1);
                                        inst_compr <= iCACHE(0).dado_memoria_icache;
                                end if;

                                -- Se não existir mais espaço na iCACHE, zerar o índice a partir da instrução normal,
                                -- ou seja, começar a sobre-escrever as instruções não comprimidas na iCACHE
                                if (ind1 > SIZE_CACHE) then
                                        for i in 0 to (SIZE_CACHE - 1) loop
                                                if (LAT(i).type_inst /= '1') then
                                                        ind1 := 0;
                                                        exit;
                                                end if;
                                        end loop;
                                else
                                        ind1 := ind1 + 1;
                                end if;
                                status <= flag;
                        end if;

                        nextpc := nextpc + 1;

                end if;
        end process;
end architecture huffman_arch;
```