Resumo

As empresas estão cada vez mais procurando um diferencial em seus negócios criando novos modelos de gestão e novos serviços para seus clientes, com isso ficou imprescindível o emprego de novas tecnologias tais como: sistemas embarcados, celulares e notebooks. Com o advento dessas novas demandas, as empresas de tecnologia foram obrigadas a desenvolver novos componentes eletrônicos, com maior capacidade de armazenamento, alto desempenho, baixo consumo de energia, comunicação com a internet e ainda reduzir o tempo para produção de suas novas tecnologias. No desenvolvimento de uma arquitetura de uma nova tecnologia a fase de projeto é fundamental, pois levantar todos os requisitos necessários, design e mapeamento dos componentes, analisar o desempenho da arquitetura e, ainda relacionar o custo/benefício para produção ideal é uma corrida contra o tempo. Hoje, existem ferramentas para projetar, testar e analisar o resultado das arquiteturas sem que para isso seja necessário fabrica-las. Isso é muito importante, pois produzir uma única unidade de arquitetura é um custo relativamente alto. No desenvolvimento dessas novas arquiteturas a memória cache se tornou essencial para garantir o alto desempenho do processador. A cache funciona como um pequeno repositório para o processador fornecendo dados e novas instruções de forma rápida e precisa sem usar o barramento do sistema, diminuindo o fluxo de informações no barramento. Dessa forma, o processador trabalha mais e passa menos tempo ocioso. Este trabalho apresenta uma descrição de uma implementação de cache usando uma linguagem para descrever *hardware*. Este estudo mostra a relação existente entre diferentes tamanhos de cache, comparando estatisticamente a área ocupada da implementação final em uma arquitetura reprogramável.

Abstract

The companies are looking for ways to improve their business, creating new management models and new services to their clients, that is why the use of new technologies such as: embedded systems, mobile phones and notebook, became essential. Because of the growing of new demands, technology companies were forced to reduce the production time of their new technologies, and develop new electronic components with larger storage capacity, higher performance, lower energy consumption and internet connection. The project phase is an essential step on the development of a new architectures, to evaluate all the necessary requirements, design and components mapping, to analyze the architecture's performance and to evaluate the production's cost/benefit is a race against the time. Nowadays, we have tools to project, test and analyze the architecture's results without manufacturing it. This is very important, because the production of one single unit can be expensive. The cache memory became essential to assure the high performance of processors on the development of these architectures. The cache memory works like a little repository to the processor, supplying data and new instructions in a fast and precise way, without using the system's bus, decreasing the information' stream on the bus. With this, the processor works more often and spend less in idle. This work describe a cache implementation using a *hardware* language description. This study shows the relation between different sizes of cache, comparing statically the used area of the final implementation in a reprogramable architecture.

Sumário

Índice de Figuras

Índice de Tabelas

Tabela de Símbolos e Siglas

(Dispostos por ordem de aparição no texto)

ASIC – *Application Specific Integrated Circuit*
FPGA – *Field Programmable Gate Arrays*
VHDL – *Very High Speed Integrated Circuit Hardware Description Language*
CAD – *Computer Aided Design*
DRAM – *Dynamic Random Access Memory*
SRAM – *Synchronous Random Access Memory*
LRU – *Least Recently used*
FIFO – *First-In-First-Out*
LFU – *Least Frequently used*
ISA - *Instruction Set Architecture*
CLB – *Configuration Logical Blocks* (Blocos de Configuração Lógica)
IOB – *Input Output Blocks* (Blocos de Entrada/Saída)
HDL – *Hardware Description Language* (Linguagem de Descrição de *Hardware*)
RWS – *Read-Write Signal* (Sinal de controle que determina a operação na cache)
ADL – Architecture Design Language

Dedico esse trabalho a todos os profissionais que contribuem para o surgimento e desenvolvimento de novas tecnologias que auxiliam no progresso humano.

Agradecimentos

- Gostaria de agradecer a minha família pela dedicação e apoio para que eu me tornasse um profissional de engenharia, em especial a meu pai Nelson Prado e minha mãe Inês Rejane, que foram principais incentivadores e aos meus irmãos que compartilharam dessa minha trilha de algumas tristezas e muitas alegrias;

- Ao professor Abel Guilhermino pela sua dedicação, paciência e sua orientação nos momentos das minhas dificuldades, e hoje, o tenho como amigo;

- Dedico este trabalho aos meus amigos de infância e companheiros de aventuras: Fabrício Fernandes, André Luiz, Diego Lucena, Lucas Coutinho, Ricardo Fernandes e Carlo Moreno;

- Aos amigos que se formaram e que hoje tenho como companheiros de profissão e amigos para toda a vida: Fernando Antônio, Allan Bruno, Adilson Arcoverde, Cláudio Cavalcanti, Gabriel Alves, Gabriel da Luz, Lívia Brito, Laureano Montarroyos, Alcides Bezerra, Diogo Cavalcanti, Tiago Lima, Rafael Bandeira, Rodrigo Cursino, Rodrigo Brayner, Reinaldo Melo, Adélia Carolina, Carolina Baldisseroto, Carolina Matos, Maíra Pachoalino, Bruna Bunzen, Polyana Olegário, Juliana Lima, César Augusto, Pedro Neto, Adriano Nântua, Hilton e Túlio Campos;

- Aos professores do curso de Engenharia da Computação que me ensinaram muitas das coisas que aprendi e por me ensinar a ser um profissional digno e honesto. Agradecimento em especial aos professores Carlos Alexandre, Fernando Buarque, Renato Corrêa, Ricardo Massa, Adriano Lorena e Edson Lisboa.

Capítulo 1

Introdução

Neste capítulo, serão abordadas as motivações de fazer este trabalho e também discutir os problemas deste trabalho. Alguns tópicos descritos são relevantes para compreensão deste estudo e serão discutidos no andamento do trabalho.

1.1 Evolução Computacional

As empresas estão cada vez mais procurando soluções seguras e rápidas (muitas vezes em tempo real) para os seus empreendimentos e fazem uso das tecnologias emergentes para tal finalidade. A informática e a engenharia têm proporcionado inovações nos modelos de negócios e agregado novos serviços a muitos setores de empresas e em muitos ramos da ciência. A evolução tecnológica, por exemplo, tem influenciado em avanços na medicina como mapeamento genético, processamento de DNA, processamento de imagens raios-X, etc.

Nos últimos anos tem-se observado um aumento gradativo no processamento de dados em todos os campos da ciência. Muitos desses processamentos são bastante complexos e exigem uma grande demanda de recursos, tais como: processador, grande capacidade de armazenamento, transmissão interna e externa de dados, processamento de voz e imagens, segurança de dados, sistemas embutidos e indústrias com segmentos robotizados.

O avanço das arquiteturas de *hardware* gerou uma rápida evolução nos computadores. Segundo Gordon Moore, um dos fundadores da Intel, a capacidade dos processadores dobra a cada 18 meses enquanto os custos dos *chips* permanecem constantes [6]. Os componentes eletrônicos estão cada vez menores e mais baratos, oferecendo máquinas multiprocessadas mais acessíveis e menores, diferentes dos primeiros computadores e *mainframes*, que eram máquinas gigantescas e com elevados custos.

A crescente necessidade de poder de processamento das aplicações criou a exigência de novos conceitos de tecnologia, novas metodologias, novas implementações e inovações nos modelos de arquitetura existentes para que essa demanda de processamento fosse atendida.

1.2 Estratégias de Processamento

Desde o ENIAC [1], primeiro computador digital, muitas mudanças aconteceram nas arquiteturas de computadores garantindo um aumento de desempenho. Mudanças simples como uma nova

abordagem no mapeamento dos componentes, implementações diferentes para os barramentos do sistema, acréscimo de registradores, uso de memórias programáveis, o uso de memórias diferentes para instruções e dados, e, mudanças mais complexas como o uso de *pipelines* dentro dos processadores, o uso de memórias caches, uso de co-processadores para cálculos auxiliares e processamento gráfico, têm sido fundamental para evolução dos computadores.

O processador foi um componente que sofreu grande evolução desde a sua primeira concepção, com transistores cada vez menores os chips de computador diminuíram de tamanho e ficaram ainda mais potentes. Mas este alto desempenho do processador não foi acompanhado pelos outros componentes do computador como memórias e dispositivos de entrada/saída, que ficaram mais lentos em relação ao processador no decorrer do tempo [13]. Isso gerou um grande problema, pois o processador com seu alto desempenho passou a ficar ocioso aguardando a comunicação dos outros componentes da arquitetura, ou seja, trabalhava muito rápido, mas tinha ajudantes muito lentos. Na Figura 1 [19] pode se observar a disparidade (segundo a lei de Moore) entre a evolução do processador e a memória, enquanto o processador crescia em média 60% por ano as memórias evoluíam lentamente, em média 7% por ano.

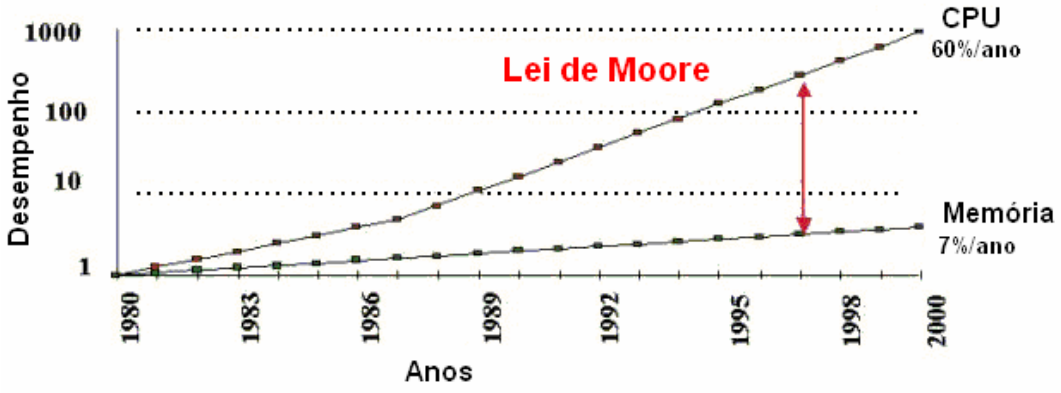

Figura 1. Evolução das arquiteturas de processador e memória.

Muitas das estratégias usadas para melhorar o desempenho do computador foram devidos a esse problema do processador ocioso. Os processadores estavam cada vez mais robustos e rápidos, mas ficavam penalizados devido à ausência de implementações de arquitetura mais eficientes. Para suprir essa necessidade, projetistas de *hardware* criaram novos componentes auxiliares e de controle, e, mudaram muitos conceitos anteriormente implementados. Exemplos disso são as memórias que ficaram cada vez mais rápidas, com blocos diferentes para instruções e dados e novas implementações que permitiram múltiplos acessos às memórias.

Uma das estratégias que teve um grande sucesso na época dos primeiros computadores foi a memória cache, ela aumentava consideravelmente o desempenho do processador. No final década de 80, por exemplo, computadores como o 80486 da Intel trabalhava com um *clock* de 25 MHz e ciclos de 80 ns, mas as memórias mais rápidas da época eram as de 100 ns de atraso, portanto era um tempo de acesso não muito satisfatório [13]. A solução da Intel foi acrescentar ao núcleo do processador uma memória que tivesse um tamanho menor de armazenamento que as memórias existentes (para não aumentar demais a área ocupada pelo processador) e que fosse mais rápida que as memórias convencionais para obter respostas (novas instruções e dados).

O crescimento exponencial da quantidade de transistores para uma mesma unidade de área provocou uma motivação em desenvolver circuitos mais complexos, com mais funcionalidade e integrados em um único chip. Mas a construção de arquiteturas mais complexas ocasionou um aumento no tempo de desenvolvimento dos projetos das empresas. Com isso, ferramentas de desenvolvimento de trabalho são necessárias para conseguir atingir a janela de mercado em

tempo hábil. Desenvolver projetos de maneira rápida passa a ser um desafio para muitas empresas.

Projetar arquiteturas em *hardware* normalmente demora mais tempo para ser concluído quando comparado com projetos de arquiteturas em software. Normalmente projetos desenvolvidos em *hardware* usam linguagens de descrição mais próximas da máquina (ex: VHDL), exigindo muitas vezes experiência e conhecimento dos projetistas. Por outro lado, projetos desenvolvidos em software, normalmente usam linguagens de alto nível (Ex: C e Java) na qual exige, muitas vezes, pouca experiência do projetista.

Uma estratégia para tentar reduzir o tempo de desenvolvimento de projeto é a utilização de ferramentas como as linguagens de descrição de arquitetura (ADL) baseadas em simuladores executáveis, na qual consegue-se emular o comportamento de uma arquitetura que se deseja projetar antes da descrição do *hardware* propriamente dita. Linguagens como ArchC [16], podem servir como suporte para prover ao projetista análises e resultados que façam-no decidir qual a arquitetura adequada para um certo tipo de aplicação.

Existe, hoje no mercado, diferentes abordagens para desenvolvimento de projetos em *hardware*. Dentre essas abordagens, três ganharam muito destaque: ASICs (*Application Specific Integrated Circuit*), Microcontroladores e FPGAs (*Field Programmable Gate Arrays*).

ASICs são circuitos integrados que são fabricados para executar uma tarefa especifica. ASICs são utilizados para maximizar o desempenho de uma aplicação ou serviço. As desvantagens dos projetos com ASICs são os elevados custos para fabricação desse componente e a falta de flexibilidade do circuito projetado, uma vez que o ASIC é fabricado, sua lógica interna não pode ser modificada.

Microcontroladores são *chips* têm memórias e interface de entrada/saída integrados e que podem ser programados mais de uma vez. Os microcontroladores são muito utilizados em equipamentos eletrônicos para executar tarefas de controle. A grande vantagem de utilizar microcontroladores no desenvolvimento de uma arquitetura é baixo custo da fabricação em relação aos projetos que contém ASICs.

A terceira abordagem de desenvolvimento são os FPGAs, que são semicondutores digitais que são usados frequentemente para prototipação. Os FPGAs foram concebidos com o intuito de absorver as vantagens existentes nos ASICs e microcontroladores, ou seja, a vantagem de se programar *hardware*, dos ASICs, e a possibilidade da reprogramação existente nos microcontroladores. A grande diferença (e também a grande vantagem) que os FPGAs apresentam em relação as outras duas abordagens vistas, é, a possibilidade de reconfigurabilidade parcial existente nos FPGAs (os detalhes sobre FPGAs serão vistos no capitulo 3), que foi uma das motivações do uso de FPGAs neste trabalho.

A intenção deste trabalho foi acompanhar a evolução das arquiteturas de computadores, principalmente aquelas que se beneficiaram com o uso das memórias caches, e, fazer um estudo aprofundado sobre as caches e suas implementações.

1.3 Objetivos Gerais do Trabalho

A idéia do trabalho foi desenvolver uma arquitetura de cache, partindo do conjunto processador/cache/memória, que fornecesse informações suficientes para estudos em arquiteturas reconfiguráveis. Seria desenvolvida uma única arquitetura de cache e a partir desta, variar a capacidade de armazenamento desde modelo para analisar o comportamento da cache em uma arquitetura de FPGA relacionando a área ocupada.

O objetivo deste trabalho é mapear os diferentes tamanhos de cache implementados em uma determinada família de FPGA's com o intuito de encontrar uma relação existente entre a implementação e quantidade de unidades lógicas programáveis (CLB's) dentro de um FPGA.

A conclusão deste trabalho nos motiva a entender que o resultado obtido da área ocupada pelo modelo de cache possa gerar uma equação em termos de unidades lógicas que possa ajudar o projetista a escolher a família de FPGA ideal para o desenvolvimento do projeto.

1.4 Objetivos Específicos do Trabalho

A partir da idéia geral, foram especificados os passos seqüenciais para o desenvolvimento do trabalho. A seguir serão descritos os objetivos específicos.

- ✓ Fazer um estudo bibliográfico das arquiteturas de caches existentes no mercado, estudar os métodos de implementação, *design*, comportamento, algoritmos utilizados e benefício (desempenho) que uma cache proporciona em uma arquitetura do computador.

- ✓ Estudar o núcleo do MIPS para compreender a comunicação deste processador com a memória cache e a memória principal. Levantar os dados, as instruções e os sinais de controle que são trocados entre esses componentes.

- ✓ Definir qual a arquitetura de cache a ser implementada em VHDL e determinar, também, a política (tipo de acesso, controle de acesso, algoritmo de atualização) que vai descrever o comportamento da cache.

- ✓ Definir quais os componentes que vão fazer parte da implementação do projeto. Saber a quantidade precisa de componentes para o funcionamento da cache escolhida e determinar a quantidade de sinais para comunicação desses componentes.

- ✓ Implementar em VHDL o modelo específico de cache escolhido com as políticas, sinais de controle e componentes pré-definidos. A partir desse modelo codificado haverá diferentes variações de tamanho na cache para estudos estatísticos da área ocupada.

- ✓ Compilar e simular a implementação da cache através da ferramenta *ISE Foundation* da Xilinx.

- ✓ Analisar os resultados e estudar o impacto da área de FPGA ocupada devido a variação do parâmetro tamanho da cache.

Capítulo 2

Memória Cache

Este capítulo aborda os conceitos necessários para compreensão de memórias de computadores. Apresenta uma introdução sobre todos os aspectos de memórias, hierarquias e em específico sobre memórias cache.

2.1 Introdução a Hierarquia de Memória

Os sistemas operacionais e programas de computadores estão cada vez maiores e vêem exigindo capacidades ainda maiores de memória e mais rápidas. A memória passou por muitas evoluções desde sua idealização por Von Neumann, onde ele teve a brilhante idéia de unir em um único dispositivo de armazenamento instruções e dados de um programa.

A partir da idéia de Von Neumann, surgiu uma variedade muito grande de memórias e diferentes abordagens de implementação na arquitetura. As diferentes implementações tratavam uma forma de deixar a arquitetura mais eficiente, fazer com que os programas fossem executados mais rápidos. No decorrer do tempo, as memórias ganharam um tempo de acesso quase que instantâneo e ainda pode se observar que o custo por *bit* ficou mais barato.

Existem hoje, algumas tecnologias tradicionais de memórias: as SRAM's (memórias de acesso estático), as DRAM's (memórias de acesso dinâmico) e os discos rígidos. As SRAM's são as memórias de menor capacidade e que tem tempo de acesso mais rápido, mas são relativamente mais caras; as DRAM's têm maior capacidade de armazenamento e mais baratas que as SRAM's, mas em compensação são bem mais lentas; e os discos rígidos são dispositivos de grande armazenamento, são mais lentos que as outras memórias e devido a isso tem o custo mais reduzido. A Figura 2 (modificada do livro de Stallings [15]) mostra uma hierarquia de memória relacionando a velocidade, a capacidade de armazenamento e custo por *bit*. Na base estão as memórias com maior capacidade, com menor custo e mais lentas, à medida que avança para o pico da pirâmide as memórias vão ficando mais rápidas, menores e preço por bit vai aumentando.

O ideal da arquitetura é fazer com que as memórias menor de capacidade e mais rápidas trabalhem mais perto do processador, enquanto as memórias de maior capacidade e mais lentas trabalhem indiretamente mais longe do processador [15]. Isto esta associado a um conceito determinado de "princípio da localidade". Imagine que um funcionário esteja trabalhando em uma mecânica e que para executar consertos em um carro precise de uma certa quantidade de ferramentas. A oficina dispõe de todas as ferramentas para conserto de um carro em um único

armário, mas o mecânico não consegue carregar todas as ferramentas ao mesmo tempo e também não quer se dirigir ao armário todas as vezes que precisar de uma ferramenta. Então, o mecânico carrega consigo a quantidade de ferramentas que ele sabe que é necessário para trocar um pneu ou consertar o motor, por exemplo, e dá prioridade as ferramentas que ele sabe que precisará usar mais de uma vez.

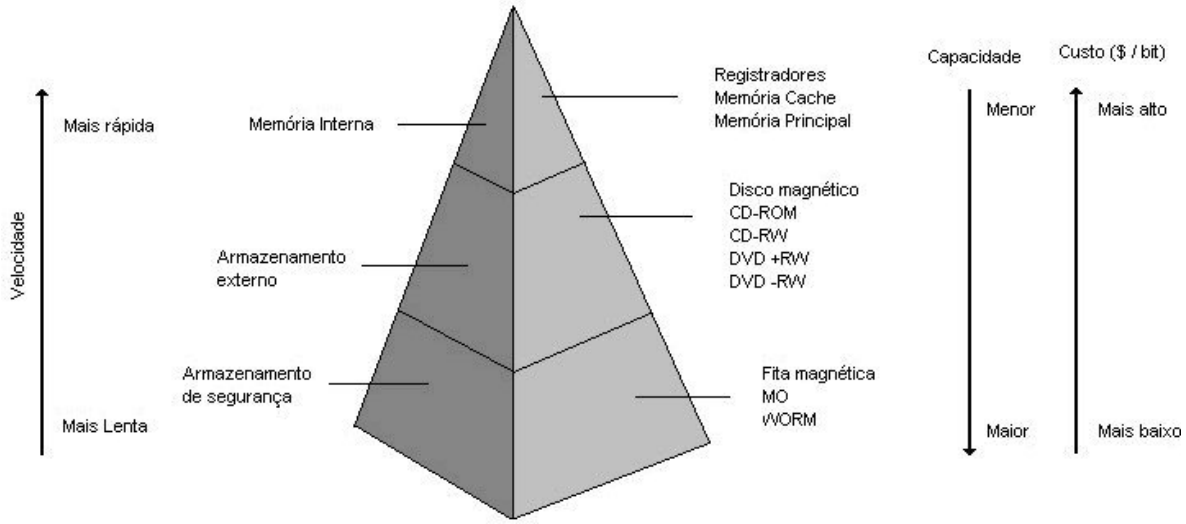

Figura 2. Hierarquia de memórias

O princípio da localidade está por trás deste exemplo, ele estabelece que os programas acessem uma parte relativamente pequena do seu espaço de armazenamento em um instante qualquer [13][17], assim como o mecânico que usa uma parte das ferramentas para executar um conserto específico no carro. Existem dois tipos de localidade:

- Localidade temporal: Se um item é acessado, ele tende a ser acessado novamente em um espaço de tempo curto. Se o mecânico do carro usa uma chave de roda para desparafusar uma roda e trocar pelo estepe, existe uma grande chance do mecânico usar a mesma chave de roda para apertar os parafusos após o conserto.

- Localidade espacial: Se um determinado item é acessado, os itens próximos ao item acessado tendem a ser acessados também. Quando o mecânico vai trocar o pneu do carro, ele vai ao armário de ferramentas e pega a chave de roda, mas ele sabe que vai precisar também do braço mecânico (macaco) para levantar o carro e vai usar o mesmo para baixar o carro após trocar o pneu.

As arquiteturas de computadores se utilizam bastante do princípio da localidade, pois como a execução de um programa é seqüencial, existe uma grande probabilidade de que a próxima instrução a ser executada seja a instrução subseqüente. Muitos destes programas executam funções em *loop*s fazendo com que a mesma instrução seja executada mais de uma vez, logo, seria extremamente necessário que esta instrução ficasse facilmente disponível.

Nas arquiteturas modernas, as memórias são organizadas hierarquicamente de forma que as memórias mais rápidas fiquem próximas ao processador e contenham informações específicas de um programa, para que estas possam fornecer informações com mais eficiência. As memórias

com maior capacidade ficam mais distantes do processador, atualizando apenas as memórias adjacentes com novos blocos de informação.

No projeto de uma nova arquitetura, o projetista procura relacionar a mais eficiente (mais rápida e precisa) hierarquia de memória com o menor custo de produção, ou seja, o custo/benefício da implementação. As caches são memórias que têm muita influência na hora da implementação de uma arquitetura, elas têm menor capacidade de armazenamento que as memórias principais, são acessadas mais rapidamente, mas em compensação tem o custo muito elevado, por este motivo as arquiteturas não tem somente memórias caches.

A seguir serão discutidos os elementos de projeto que são levados em consideração na construção de uma arquitetura de cache, como: os tipos de mapeamento, algoritmos de substituição, níveis de cache, políticas de escrita, bem como as dificuldades da implementação e os benefícios que ela proporciona à arquitetura de computadores.

2.2 Memória Cache

Memórias cache são dispositivos de armazenamento que visam obter uma velocidade mais rápida que as memórias principais e discos rígidos de grande capacidade de armazenamento, mas que possa disponibilizar ainda uma capacidade de armazenamento de dados maior que os bancos de registradores [15].

A rapidez de processamento de dados da cpu é muito alta se comparado com a rapidez da memória principal [13][15]. O processador opera em uma freqüência de ciclos muita alta e necessita de novos dados e novas instruções em um período de tempo muito curto no qual a memória principal não tem condições de acompanhar, pois o tempo de acesso de um dado em uma memória é muito lento. Devido a esse atraso, o processador fica penalizado (ocioso) pela velocidade de fornecimento de informações da memória.

A idéia da memória cache é trabalhar juntamente com a memória principal fornecendo ao processador instruções e dados para execução. A cache contém copias de partes da memória principal e quando o processador requer um novo dado ou uma nova instrução, ao invés de ele solicitar a memória principal, que é relativamente lenta em relação ao processador, ele busca esses novos dados na cache, que é muito mais rápida que a memória principal [13].

O processador não faz somente busca de instruções e dados na cache, ele também envia dados que são resultados da execução das instruções no processador para que estes possam ser armazenados ou atualizados na memória principal. A Figura 3 [15] mostra um desenho esquemático da cache em uma arquitetura de computador.

2.2.1 Estrutura Básica de uma Cache

A idéia do funcionamento da cache é extremamente simples. Quando um processador vai executar um programa, este movido da memória principal e armazenado total ou parcialmente na cache dependendo do tamanho do programa ou do espaço ainda vazio na cache. Durante a execução de uma instrução do programa, o processador envia para o controlador da cache o endereço referente ou à próxima instrução do programa ou endereço de um dado que seja necessário para alguma operação do processador. Além do endereço, o processador envia ainda sinais de controle que indicam se o endereço passado é para fazer uma busca ou uma escrita na cache (ver Figura 3).

Considerando o processador MIPS, o endereço enviado do processador para a cache é uma palavra de 32 *bits* composta por três campos: rótulo (ou *tag*), *index* e *offset*. O campo *index* é uma seqüência de *bits* que seleciona precisamente qual a linha da cache que se deseja buscar,

neste caso, poderia ser uma nova instrução ou um novo dado. O *offset* funciona como um deslocamento de linha. A cache é composta de várias linhas e cada linha desta pode ter um ou mais blocos de informações (instruções ou dados), o *offset* aponta qual dos blocos, em uma linha, será usado para leitura ou escrita (depende dos sinais de controle).

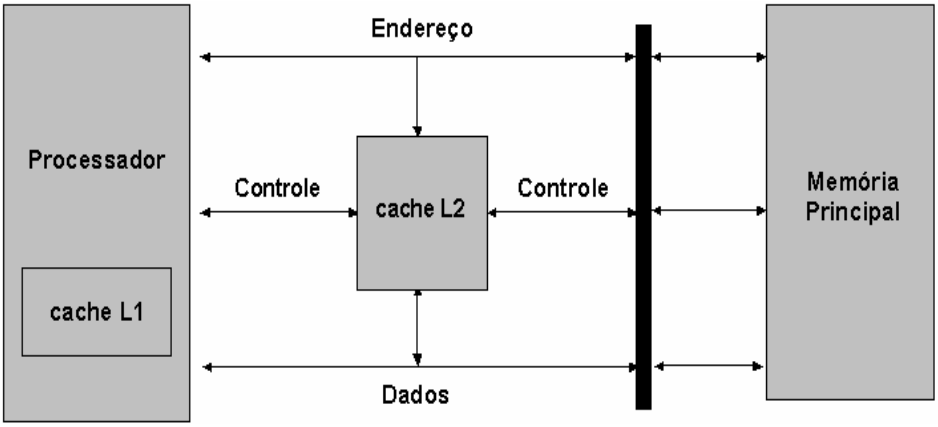

Figura 3. Diagrama de comunicação entre processador e memória.

Durante o processo de execução de um programa, o processador às vezes executa uma instrução condicional e salta para um outro ponto do programa ao invés de executar a instrução subseqüente, esse desvio condicional do programa traz uma situação inesperada para a cache, pois é muito provável que ela não tenha essa próxima instrução proveniente do desvio e tenha que atualizar um dos seus blocos armazenados por um novo bloco da memória principal.

Na execução de um programa, diferentes partes do programa estão dentro da cache para serem utilizados pelo processador. Quando o processador busca/escreve um novo dado, a cache precisa saber se o dado buscado (ou dado a ser atualizado) ainda esta em um dos blocos da cache. A cache faz uma simples comparação de rótulos para saber se o dado está na sua memória. O rótulo do endereço é uma seqüência de bits que é responsável por essa comparação e que será usado para comparar com o rótulo de uma determinada linha da cache para verificar se a informação buscada (ou que se deseja escrever) está na cache ou não.

A cache é constituída de muitos componentes internos e a sua estrutura básica é composta pelos seguintes componentes:

- Memórias SRAM (RAM Sincronizada): São usadas na cache para armazenar os dados ou as instruções de um programa e também utilizadas para armazenar os rótulos que serão comparados com o rótulo do endereço passado pelo processador.
- Bit de paridade: Cada linha da cache tem um *bit* que funciona como um sinalizador para o controlador, ele serve para indicar a consistência da informação, ou seja, se aquela linha da cache está igual a da memória principal caso contrario o processador saberá que precisa atualizar aquela linha na memória principal.
- Comparadores: São componentes que testam se o rótulo do endereço é igual a algum dos rótulos da cache. Esse teste serve verificar se a informação buscada pelo processador está na cache ou não. O resultado da comparação é um sinal enviado para o controlador indicando que a informação está ou não na cache.
- Multiplexadores: São componentes que recebem diferentes entradas e seleciona uma dessas entradas em uma única saída. Quem indica qual entrada deve ser "jogada" na saída é um sinal de seleção que escolhe uma das entradas. O multiplexador na cache recebe como sinal de seleção o *offset* e como entrada

recebe os blocos de uma determinada linha da cache. O *offset* selecionará qual o bloco será enviado para a saída ou qual o bloco vai ser atualizado no caso de uma escrita de dados.

- *Buffers*: São unidades de armazenamento temporário. O *buffer* é bastante utilizado para armazenar temporariamente o endereço passado para cache ou armazenar o dado que se quer ler ou escrever na cache.
- Controlador de cache: É o principal componente da cache. Ele é responsável por todo fluxo da informação e tomada de decisão dentro da cache. O controlador recebe do processador sinais de controle indicando se a operação é de leitura ou escrita e um sinal que habilita/desabilita (ativação) a cache.

A figura abaixo (modificada do livro de Stallings [15]) ilustra uma cache simplificada com memórias SRAM, para armazenar rótulos, dados e o *bit* de paridade, um comparador, um multiplexador, um *buffer* de armazenamento temporário e o controlador da cache.

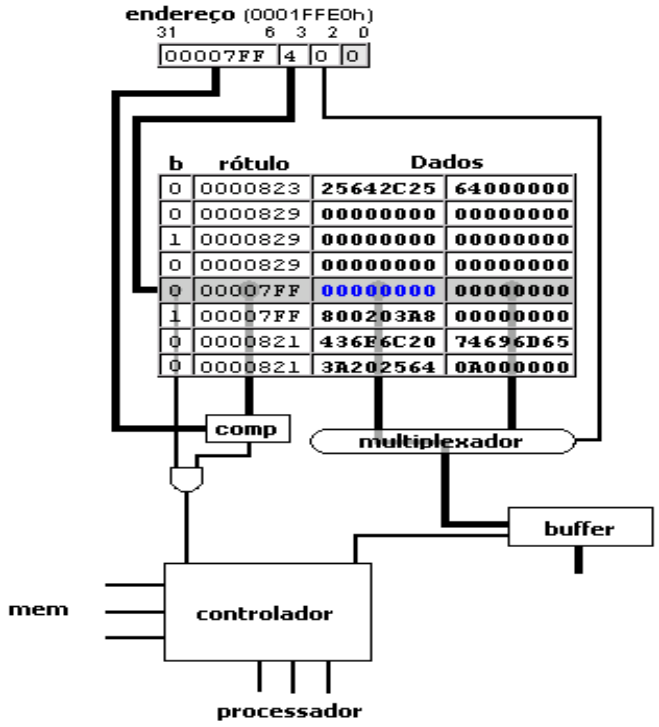

Figura 4. Componentes físicos da cache.

Na Figura 4 pode se observar que o endereço enviado pelo processador é dividido em três partes, a primeira parte (rótulo) vai para o comparador para testar se o rótulo do endereço e o rótulo da linha da cache são equivalentes. A segunda é o *index*, que é responsável pela seleção da linha da cache que o processador referenciou. A terceira parte é o *offset* que seleciona quais dos blocos da linha selecionada vai ser enviado para o processador.

2.2.2 Fluxo de Informação na Cache

Durante a execução do processador, dois possíveis fluxos de dados podem ocorrer na cache. Um dos fluxos existente é o de leitura. Durante a execução de um programa o processador faz varias requisições a memória cache atrás de novas instruções ou de novos dados (para terminar uma operação). A cache recebe do processador um endereço de memória que indica qual palavra da cache ele está precisando no momento, caso esta palavra esteja na cache, ela envia esse novo dado para o processador. Pode acontecer do dado desejado não estar presente na cache e o processador precise esperar uma atualização da cache antes de receber o novo dado.

O outro fluxo possível é o de atualização dos dados, o processador após executar uma instrução pode querer gravar os resultados do processamento na memória, neste exemplo o processador além de enviar o endereço de memória, ele também envia o dado a ser armazenado na cache.

Para esses dois fluxos de informações, o que determina se a cache vai executar uma leitura ou uma escrita em suas memórias são os sinais de controle enviados do processador para a cache, estes sinais são emitidos tanto na leitura como na escrita de dados. Nas implementações mais simples de cache, os sinais de controle trocados são três:

- Sinal de Habilitação *(Enable – EN)*: Este sinal (um *bit*) habilita a cache. Quando o processador executa uma instrução que não faz uso das memórias, este sinal é setado para ‘0’ (zero). Quando o processador faz uma leitura ou escrita na cache este sinal é setado para o nível lógico ‘1’ (um) sinalizando a cache que uma operação esta para acontecer.
- Sinal de escrita (*Write Signal* – WS): Este sinal indica a cache que o processador deseja escrever um dado na memória. O valor que habilita a escrita é ‘1’ (um). Quando o processador deseja atualizar um dado na cache, além de ele enviar um o sinal de habilitação setado para ‘1’ (um), ele envia também o sinal de escrita setado para ‘1’ (um).
- Sinal de leitura (*Read Signal* – RS): Este sinal indica a cache que o processador deseja fazer uma leitura de um dado armazenado na memória. O valor que habilita a leitura é ‘1’ (um). Quando o processador deseja ler um dado da cache ele envia o sinal de habilitação setado para ‘1’ (um) e o sinal de leitura setado para ‘1’ (um) também.

Em muitas implementações, estes sinais de controle são ativados em nível lógico ‘0’ (zero), isso depende do requisito do projeto. Na implementação desenvolvida nesta monografia, os módulos da cache são ativados quando estes sinais estão em nível lógico alto (um), como descritos acima.

Quando o processador vai fazer uma escrita na cache, ele habilita a cache (*enable* igual a um), seta o sinal de escrita para um, conseqüentemente o sinal de leitura para zero e envia o dado e o endereço onde deve ser armazenada a informação. Quando a requisição é uma leitura, o sinal de escrita vai para zero e o sinal leitura para um, o sinal de habilitação continua em nível alto e o processador envia o endereço referente à palavra a ser buscada.

2.2.3 Fluxo de Leitura

Na seção anterior foi visto quais palavras e quais os sinais que são trocados entre o processador e a memória cache. Durante a execução da CPU, muitas instruções são efetuadas no processamento

e muitos dados são solicitados e atualizados nas memórias. O ciclo de execução de uma leitura de uma instrução é diferente da escrita de dados na cache.

Após o processador enviar o endereço do bloco onde contém a nova instrução, o controlador da cache compara se o rótulo do endereço é igual ao rótulo que está no bloco referenciado, isto é feito porque o processador precisa saber se o endereço que ele queria está na cache ou se o controlador já trocou por outro bloco. Caso os rótulos sejam iguais o controlador da cache acessa o bloco endereçado e envia a informação para o processador. Quando há um acerto na informação buscada se diz que houve um *cache-hit* (acerto).

Quando os rótulos diferem na comparação, o processador aguarda para que o controlador da cache faça uma busca na memória principal, com o mesmo endereço que o processador enviou para a cache, e atualize uma linha da cache (através de escolha) com um novo bloco da memória principal. Essa atualização é feita através de um algoritmo implementado na cache, na seção de elementos de projeto serão apresentados com mais detalhes alguns algoritmos de substituição.

Após o bloco que veio da memória ser atualizado na linha da cache reservada, o controlador da cache envia para o processador a instrução buscada inicialmente. Quando a informação pesquisada pelo processador não está na cache diz-se que houve um *cache-miss* (erro de procura).

O fluxo de execução acima pode ser visto na Figura 5 [15].

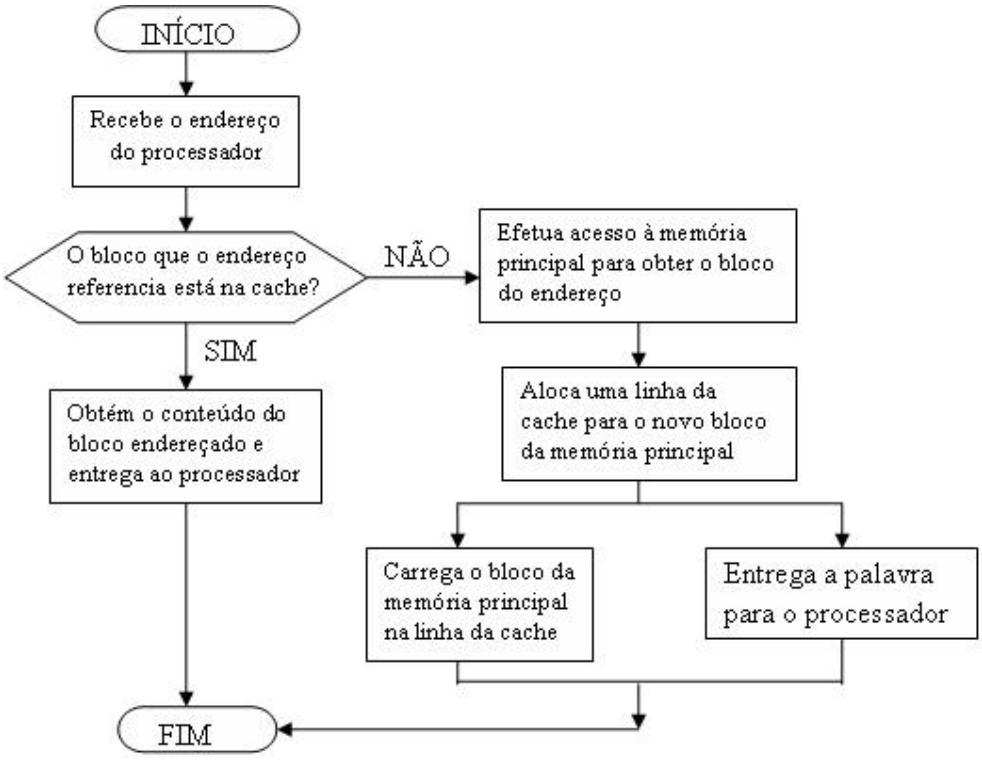

Figura 5. Diagrama de processo de leitura na cache.

2.2.4 Fluxo de Escrita

O ciclo de execução da escrita na cache é um pouco mais complexo que o ciclo de leitura. Quando o processador vai atualizar um novo dado, ele precisa saber se o dado desatualizado ainda está armazenado na cache só para depois sobrepor o dado antigo.

O processador envia para a cache além do endereço e sinais de controle, envia também o dado a ser armazenado na cache. O primeiro passo do controlador da cache é verificar se os rótulos do endereço e da linha referenciada pelo processador são iguais, se afirmativo, o controlador atualiza a cache sobrepondo o dado armazenado pelo dado enviado pelo processador. Diferente da leitura a escrita tem uma preocupação a mais, o controlador tem que atualizar a memória principal também, pois como o processador sobrepôs um novo dado, o dado da memória principal está desatualizado causando inconsistência dos dados. Existem diferentes algoritmos que atualizam a memória depois que a cache é atualizada, esses algoritmos serão vistos com mais detalhes na seção de elementos de projeto.

Caso haja um erro na comparação dos rótulos, dois procedimentos podem acontecer: no primeiro o controlador pode reservar um espaço na cache para ser armazenado o dado enviado pelo processador e só depois atualizar a memória. O segundo caso, o controlador da cache pode preferir atualizar a memória principal primeiro e só depois atualizar a cache fazendo uma busca na memória principal e trazendo o bloco novo para cache.

No caso da escrita, acontecendo um *cache-hit* ou *cache-miss* o controlador terá que se preocupar em atualizar os dados da memória principal. Essa atualização depende do algoritmo de substituição dos blocos implementado na cache.

O diagrama descrito na Figura 6 representa um ciclo de escrita na cache, onde pode ser visto o que acontece se ocorrer um *cache-hit* ou *cache-miss*.

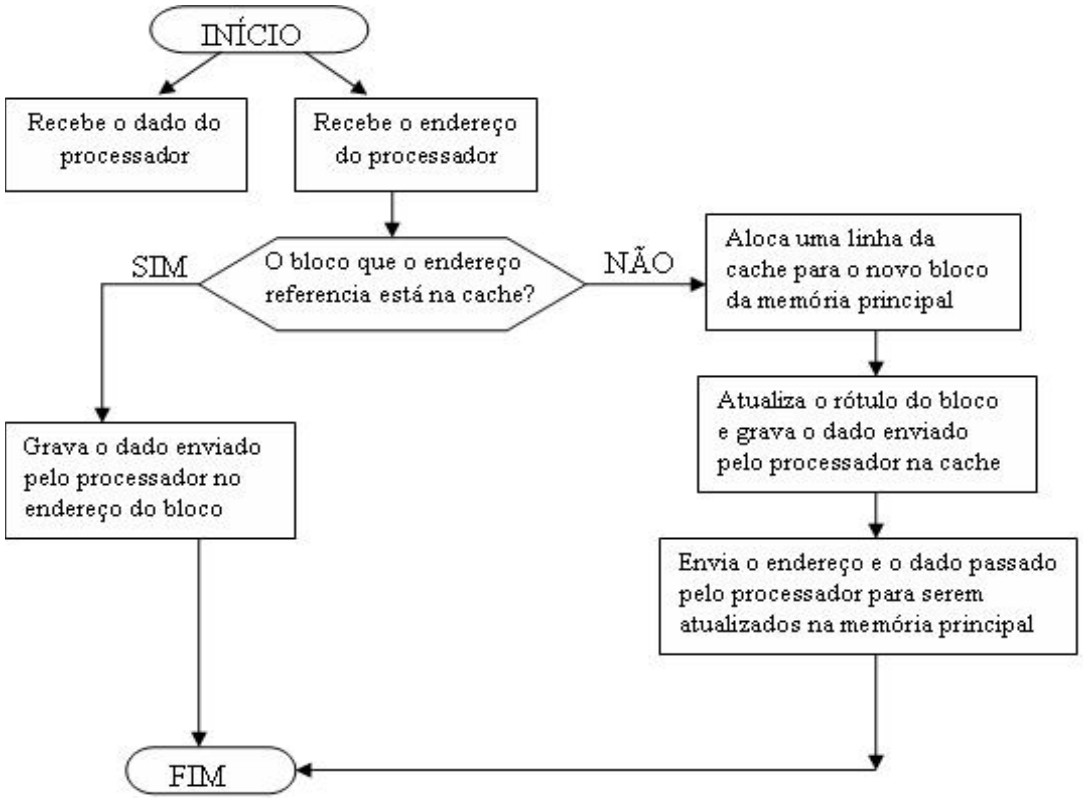

Figura 6. Diagrama de processo de escrita na cache.

O sucesso da busca na cache é fundamental para um bom desempenho do sistema, uma busca com acerto, leva não mais que dois ciclos de clock para retornar o resultado para o processador, mas quando há um *cache-miss*, uma nova linha da cache tem que ser atualizada com um novo bloco da memória principal, esse processo leva em média de 48 ciclos de clock e

penaliza o *pipeline* em quatro estágios [12], podendo piorar quando a operação é de escrita na cache. Para contornar essa perda de eficiência no sistema, existem vários algoritmos e técnicas de implementação de cache, que serão vistas com mais detalhes na seção a seguir.

2.2.5 Elementos de um Projeto de Cache

Hoje em dia, existem diferentes tipos de tecnologia de cache no mercado, estas caches têm implementações diferentes uma das outras e dependem da necessidade do sistema. Quando uma arquitetura vai ser descrita os projetistas fazem a opção de usar hierarquias de memórias e o uso de cache. Na descrição da cache, alguns elementos de projeto podem ser levados em consideração. A Tabela 1 relaciona elementos básicos de projeto de cache.

Tabela1. Elementos de projeto de uma memória cache.

Tamanho da memória cache	**Algoritmos de substituição**
Função de mapeamento	Menos recentemente usado (LRU)
Direto	Primeiro a chegar primeiro a sair (FIFO)
Associativo	Menos freqüentemente usado (LFU)
Associativo por conjuntos	**Política de escrita**
Número de memórias cache	Escrita direta (write-throgh)
Um ou dois níveis	Escrita de volta (write-back)

Os elementos mostrados na tabela dizem respeito a:

- Tamanho da cache: Diz respeito à capacidade total de armazenamento da cache. O tamanho da cache deve ser relativamente pequeno, pois o custo por bit da memória cache é muito alto, mas o tamanho deve ser suficiente para que o tempo de acesso seja muito rápido e o numero *cache miss* seja o menor possível.
- Função de mapeamento: Esta função determina como a cache é organizada. Como o tamanho da cache é menor que a memória principal, é necessária uma função que faça mapeamento dos blocos da memória principal na memória cache e determine quais os blocos da memória principal estão armazenados na cache. As função de mapeamento direto, associativo e associativo por conjuntos vão ser descritos também na seção 2.2.6, 2.2.7 e 2.2.8 respectivamente.
- Algoritmo de substituição: Quando um bloco de cache vai ser atualizado deve existir um algoritmo que determine qual o bloco da cache deve ser retirado para comportar o novo bloco copiado da memória principal. Este algoritmo deve ser implementado em *hardware* para que a velocidade de acesso seja alta. Existem três algoritmos bem difundidos e bastante utilizados, que são:
  - Menos recentemente usado (LRU) – Este algoritmo substitui o bloco que não é usado há mais tempo no conjunto da cache.
  - Primeiro a Chegar Primeiro a Sair (FIFO) – Este algoritmo substitui o bloco que está há mais tempo armazenado na cache.
  - Menos freqüentemente usado (LFU) – Este algoritmo substitui o bloco da cache que foi menos utilizado.
- Políticas de atualização: são políticas de controle que agem de forma a manter a consistência dos dados. Antes de um bloco da cache ser substituído é preciso verificar se

houve alguma alteração e se esta alteração foi propagada até a memória principal. Se a alteração do bloco foi atualizada na memória principal, este bloco é descartado. Caso a alteração do bloco não tenha sido repassada para a memória, a cache tem que atualizar a memória, só para depois descartar o bloco. Existem três políticas de atualização de cache:

- Escrita direta (*write-through*) – é a técnica mais simples. Toda vez que a cache é atualizada, essa atualização é automaticamente repassada para memória principal. Essa técnica tem um problema crítico, como ela faz atualizações sucessivas, ela pode criar um gargalo no barramento de comunicação do sistema, pois como muitas atualizações estão sendo enviadas para memória principal, elas podem deixar o barramento muito ocupado e fazer com que periféricos que compartilhem o barramento esperem desnecessariamente. Uma solução para esse problema é o uso de *buffers* auxiliares para armazenar temporariamente os dados que têm que ser atualizados na memória principal e quando houver uma oportunidade repassar os dados do *buffers* para a memória principal.
- Escrita de volta (write-back) – Essa técnica visa diminuir as atualizações sucessivas da cache. Quando a cache é alterada, um bit de *flag* é setado para '1' (um) indicando que aquela linha da cache sofreu alteração, quando o bloco vai ser descartado a cache verifica as linhas que têm o *bit* indicativo igual a '1' (um) e atualiza estas linhas na memória principal. A desvantagem dessa técnica é que os dados na memória podem ficar desatualizados por muito tempo, pois as linhas da cache só serão atualizadas na memória principal quando houver descarte de algum bloco devido a alguma substituição.

- Número de memórias cache: Com o avanço das tecnologias de cache, novas implementações e protótipos surgiram aumentando o desempenho da cache. Uma dessas novas técnicas foi a possibilidade de inclusão da cache na pastilha do processador, antes externa. Com a cache agora interna (chamada de L1 – *Level* 1), o processador não precisa mais acessar o barramento do sistema para fazer uma consulta na cache, diminuindo o gargalo do sistema e melhorando o tempo de resposta. A desvantagem desse avanço é o fato de que a cache ainda acessa o barramento do sistema para fazer a substituição dos blocos.

  Uma nova abordagem foi adicionar uma cache externa (L2 – *Level* 2) ao sistema. Esta cache externa faz a comunicação com a memória principal e com a cache interna. Existem duas vantagens nesta abordagem: uma é que a cache interna não mais acessa o barramento do sistema, diminuindo o gargalo e a outra vantagem é que o numero de acertos aumentou consideravelmente, pois quando os dados não forem encontrados na cache L1 é bastante provável que eles já estejam na cache L2 [15].

O uso de hierarquia de memórias foi uma evolução nas arquiteturas de *hardware*, ela proveu um melhor desempenho aos computadores que antes sofriam com o lento tempo de resposta, o processador passou a ficar menos tempo ocioso aguardando novas instruções.

As diferentes implementações de hierarquia de memórias e uso de cache dependem da necessidade da arquitetura. Os computadores pessoais, por exemplo, usam caches de tamanho pequeno, pois o fluxo de informação não é tão grande e constante. Já máquinas que são usadas como servidores em empresas, são máquinas robustas e têm fluxo (quantidade) de informações muito grande e constante, precisando de um tempo de resposta eficiente, por isso as arquiteturas dessas máquinas são compostas de processadores poderosos e caches de tamanho grande, elevando o custo do computador.

Toda nova arquitetura que vai ser desenvolvida passa por uma fase de projeto demorada e minuciosa, pois os projetistas tentam ajustar os componentes de acordo com a finalidade da

implementação. Os projetistas precisam testar a placa antes do processo de fabricação, porque fabricar uma única peça tem um custo muito elevado e não justifica os gastos.

2.2.6 Função de Mapeamento Direto

Nesta seção será descrita a função de mapeamento direto. Esta explanação será muito útil brevemente, pois a implementação desenvolvida neste trabalho usa mapeamento direto como função de preenchimento da cache.

A função de mapeamento direto é simples e tem baixo custo de implementação [15]. Nessa técnica, cada bloco da memória principal é mapeado em uma única linha da cache. O endereço enviado pelo processador é usado como referência nesta técnica.

A memória cache tem um tamanho muito menor (em *kbytes*) que a memória principal e precisa de uma função para mapear diferentes blocos da memória em linhas da cache. A idéia do mapeamento direto é associar blocos da memória a posições fixas na cache usando uma equação (1). A função módulo é expressa segundo:

"i = j módulo m" (1)

onde 'i' é igual ao número da linha da cache (onde o bloco vai ser armazenado), 'j' é igual ao número do bloco da memória principal e 'm' é a quantidade de linhas da cache. Por exemplo, digamos que haja uma memória principal com duzentos blocos e uma cache com dez linhas para armazenamento, e, que se queira mapear os blocos desta memória principal nas dez linhas da cache. O bloco 0 (zero) ficaria mapeado (segundo a equação) da seguinte maneira, i = 0 mod 10, que é igual a zero, logo o bloco 0 da memória principal ficaria na primeira linha da cache (linha zero), o bloco 56 seria i = 56 mod 10, que é igual a 6, logo o bloco seria mapeado na linha seis da cache (sétima posição) e assim sucessivamente.

A Tabela 2 demonstra o mapeamento completo deste exemplo (m=10 e j=200). A primeira coluna da tabela mostra as linhas da cache, a segunda coluna mostra os blocos mapeados por linha seguindo a equação vista anteriormente e a terceira mostra os diferentes blocos mapeados por linha se baseando na equação. É importante observar que uma mesma linha da cache fica reservada a diferentes blocos, veja que a primeira linha esta reservada para receber os blocos 0, 10, 20, 30,..., 70,..., 120,..., 190, 200, ou seja, todos os blocos que calculados pela equação tem como resultado (resto) zero.

Tabela 2. Mapeamento de blocos em linhas da cache.

Linha da memória cache		Blocos mapeados por linha	Blocos de exemplo
0	0	0, m, 2m, 3m,...	0, 10, 20, 30,... , 190, 200.
1	1	1, m+1, 2m+1, 3m+1,...	1, 11, 21, 31,... , 181, 191.
2	2	2, m+2, 2m+2, 3m+2,...	2, 12, 22, 32,... , 182, 192.
	. . .	. . .	. . .
8	m-2	m-2, 2m-2, 3m-2, 4m-2,...	8, 18, 28, 38,... , 188, 198.
9	m-1	m-1, 2m-1, 3m-1, 4m-1,...	9, 19, 29, 39,... , 189, 199.

O mapeamento direto tem a vantagem de ser uma implementação simples e de custo de projeto baixo. A principal desvantagem deste tipo de mapeamento é que diferentes linhas da memória principal estão mapeadas na mesma linha da cache e isso pode trazer um problema grave. Dessa maneira, se o processador estiver executando instruções distintas, mas que estão

mapeadas na mesma linha da cache (ex. instruções 10, 20, 30, etc), o controlador da cache terá que trocar a mesma linha a cada nova instrução causando um grande atraso devido às atualizações dos blocos.

2.2.7 Função do Mapeamento Associativo

A idéia mapeamento associativo é evitar a desvantagem do mapeamento direto, que mapeia cada bloco da memória principal em uma única linha da cache. A vantagem do mapeamento associativo é que qualquer bloco memória principal pode ser mapeado em qualquer linha da cache, desse modo, o endereço enviado do processador é reconhecido como um endereço contendo apenas o rótulo (para comparação) e o dado (informação).

O rótulo é armazenado juntamente com o dado na mesma linha da cache. Quando o processador envia um novo endereço, o controlador compara (ao mesmo tempo) todos os rótulos presentes na cache com o rótulo do endereço enviado, se a informação desejada estiver na cache o controlador envia o dado para o processador, caso contrário um novo bloco da memória principal deverá substituir um dos blocos da cache.

A desvantagem desse mapeamento está justamente no módulo comparador, pois o conjunto de circuitos que são necessários para comparar todos os rótulos em paralelo são muito complexos.

A grande vantagem desse mapeamento é a flexibilidade de carregar um bloco da memória em qualquer posição da cache, dessa maneira, pode-se utilizar um bom algoritmo de substituição para maximizar a quantidade de acertos na cache.

2.2.8 Função do Mapeamento Associativo por conjuntos

O mapeamento associativo por conjuntos aproveita as vantagens do mapeamento direto e associativo e atenua as desvantagens. A idéia desse mapeamento é dividir a cache em diversos conjunto, e cada conjunto desse com uma certa quantidade de linhas.

O controlador da cache, identifica o endereço enviado pelo processador como constituído de três campos: rótulo, conjunto e palavra. O campo rótulo, como nos outros mapeamentos, é utilizado para verificar se a informação desejada pelo processador está na cache ou não, através de uma comparação. Os *bits* que compõe o campo conjunto são utilizados para selecionar um dos conjuntos existentes na cache. Diferente do mapeamento completamente associativo, que compara o rótulo do endereço com todos os rótulos existentes na cache, o mapeamento associativo por conjuntos faz a comparação somente com os rótulos existentes em um único conjunto, que foi selecionado pelo campo conjunto do endereço enviado pelo processador.

A idéia do mapeamento associativo por conjuntos é unir diferentes blocos da memória principal em conjuntos dentro cache. A lógica de controlador é quando receber um novo endereço do processador, selecionar primeiro um dos conjuntos da cache, comparar os rótulos existentes dentro desse conjunto com o rótulo do endereço para obter um dos blocos e selecionar a linha que deve ser retornada para o processador. A taxa de acerto do mapeamento associativo por conjuntos é significativamente maior que os mapeamentos direto e completamente associativo [15].

O projeto desenvolvido nesta monografia usa mapeamento direto porque é mais simples e mais rápido de codificar do que os outros tipos de mapeamento (associativo e associativo por conjuntos). A implementação deste trabalho não visa alcançar um ótimo desempenho nem avaliar os mapeamento e algoritmos existentes, o propósito desta monografia é avaliar o impacto de área, dentro de um FPGA, causado por diferentes tamanhos da cache.

A seguir será introduzido os conceitos básicos e usos mais comuns de arquiteturas reconfiguráveis.

Capítulo 3

Arquiteturas Reconfiguráveis

3.1 Introdução

A evolução dos computadores proporcionou um beneficio muito grande para o homem, as novas tecnologias inventadas são usadas de formas diferentes em muitos setores da sociedade, gerando novas oportunidades e melhorando a qualidade de vida. Os computadores atuais são frutos de muito estudo e esforço de cientistas de todo o mundo, muitas dessas novas arquiteturas são só possíveis devido a anos de sacrifício em pesquisas exploratórias.

No desenvolvimento de uma nova arquitetura de computador existem muitas fases de projeto como: pesquisas, levantamento de requisitos, estimativas de custos e tempo, codificação, síntese, simulação, testes e só depois os projetistas passam para a fase final de fabricação.

Os projetos de arquitetura, hoje em dia, visam diminuir o tempo de resposta do sistema, melhorar o desempenho, diminuir a dissipação de potência, agregar novas tecnologias a arquiteturas antigas e ainda estimar o custo/benefício de cada máquina. Para isto, novas abordagens de processamento estão sendo estudadas e projetadas a exemplo do processamento paralelo (tanto em nível de *software* como em nível de *hardware*), *grids*, *clusters*, FPGA's, processadores quânticos e nanotecnologias.

Muitas das soluções podem ser concebidas tão somente por *software* quanto por *hardware* ou ainda apresentar um híbrido dessas abordagens. Um dos grandes problemas na construção de sistemas é justamente saber qual paradigma usar para o seu desenvolvimento. As implementações por *software* ou *hardware* apresentam algumas vantagens e desvantagens relacionadas ao desempenho, flexibilidade e principalmente ao custo.

As implementações por software têm a vantagem de ser bastante flexíveis, ou seja, no decorrer do desenvolvimento do sistema podem ocorrer mudanças no projeto que alterem algum requisito ou codificação do *software*; e ainda apresentam a vantagem de ser soluções mais baratas do que as soluções em *hardware*. A principal desvantagem das implementações em *software* é o fato destas apresentarem um desempenho, muitas vezes, insatisfatório nas suas aplicações.

A implementação por *hardware* tem a vantagem do alto poder de processamento, pois suas funções são implementadas em baixo nível e suas instruções estão bem formatadas para processamento. Mas desenvolver soluções em *hardware* são caras (alto custo na produção), demandam muito tempo de simulação e uma vez produzidas não podem mais ser alteradas. Muitas das soluções em *hardware* são para um propósito especifico e não podem ser utilizadas

para outras aplicações, sub-utilizando esta implementação. As construções baseadas em *hardware* são muito indicadas para processamento especifico ou implementação de sistemas que exijam alto processamento, respostas precisas e rápidas.

As desvantagens desses paradigmas apresentados (*software* e *hardware*) causaram uma necessidade de desenvolver novos modelos e implementações computacionais, desse modo, originou-se um novo modelo computacional chamado de computação reconfigurável. Esses sistemas tentam implementar soluções intermediárias entre os modelos de *hardware* e *software.*

Diferente dos microprocessadores que executam programas de forma seqüencial, as arquiteturas reconfiguráveis têm a vantagem de executar programas paralelamente, pois os chips são fabricados com várias unidades programáveis que se comunicam através de vias; cada uma destas unidades pode implementar diferentes tipos de circuitos integrados e podem ser usados independentemente um dos outros, muitas vezes aumentando o poder de processamento.

A base das arquiteturas reconfiguráveis é componente eletrônico reprogramável chamado FGPA, que é na verdade um conjunto de blocos lógicos interligados por vias e unidades de roteamento. FPGA será discutido com mais detalhes na próxima seção.

3.2 FPGA's

A computação reconfigurável tem o intuito de diminuir o espaço existente entre os paradigmas de *hardware* e *software*, possibilitando aos projetistas de computadores uma nova visão de desenvolvimento[8] . Apesar da computação reconfigurável ser bastante recente e seus conceitos ainda não estarem firmados [9], essa tecnologia permitiu a implementação em nível de *hardware*, mantendo o alto desempenho das implementações, e agora também com uma flexibilidade que antes não existia.

Arquitetura de FPGA's permitem que estruturas internas programáveis sofram reconfigurações diferentes da original, durante a execução (ou após o termino de uma tarefa, ou ainda parte dela) de um programa de acordo com a necessidade do usuário [9]. É importante frisar que isto só é possível quando se tem em mãos uma arquitetura de FPGA's que suporte reconfiguração dinâmica parcial. Com bons olhos nessa característica de reconfiguração e partindo da idéia que o intervalo de aplicações também aumenta quando se usa uma arquitetura destas, vamos nos ater a explorar a arquitetura de FPGA's do fabricante Xilinx [18] que possui esta característica.

Os chips reprogramáveis (FPGA) da Xilinx são constituídos de três elementos de configuração: blocos lógicos programáveis (*CLB*), circuitos de interfaceamento e vias de interconexões, esses elementos podem ser melhores ilustrados na Figura 10.

Os CLB's são circuitos constituídos de flip-flops e lógica combinacional, que podem trabalhar isoladamente ou em conjunto, através de interconexões. Um único FPGA contém vários CLB's idênticos que podem ser programados de forma diferente, tanto pelo projetista como pelo usuário, para ser adequar a necessidade do programa. A programação do FPGA e a configuração das vias podem ser feitas estática ou dinamicamente, ou seja, a programação dos componentes internos do *chip* podem ser feitas antes da execução das tarefas de um programa ou podem acontecer durante a execução das tarefas. O projetista (ou usuário) pode ainda reconfigurar (de forma estática ou dinâmica) uma conexão existente entre CLB's e fazer com que certas unidades não se comuniquem entre si e passem a se comunicar com outras unidades com as quais não se comunicavam.

Os circuitos de interfaceamento, também chamados de IOB's (*Input Output Blocks*) são buffers bidirecionais com saídas em alta impedância, que fazem a comunicação do FPGA com os demais componentes do computador. Esses pinos podem ser configurados para funcionar

somente como entrada do FPGA quando outros componentes da placa enviam dados para serem processados internamente, e, podem ser programados somente como pinos de saída quando o FPGA envia dados para outros componentes ou ainda podem ser configurados como bidirecionais quando o FPGA está se comunicando nas duas direções ao mesmo tempo.

Existem ainda no FGPA as vias de interconexões, que são grupos de trilhas que fazem as conexões entre as CLB's, comunicando suas entradas e saídas e também fazem a comunicação com os circuitos de interfaceamento. A programação destas interconexões é chamada de roteamento e geralmente são programadas internamente por células que determinam as funções de roteamento, indicando quais CLB's estão se comunicando e através de quais trilhas se comunicam com IOB's. Todas as vias internas são constituídas de segmentos metálicos que suportam chaveamento, desta maneira pode-se programar o destino desejado da comunicação.

A Figura 7 [9] mostra um diagrama representativo de um FPGA da Xilinx, as interconexões e a configuração dos pinos do FPGA podem ser alteradas durante a execução dos programas, bem como a lógica interna de cada CLB.

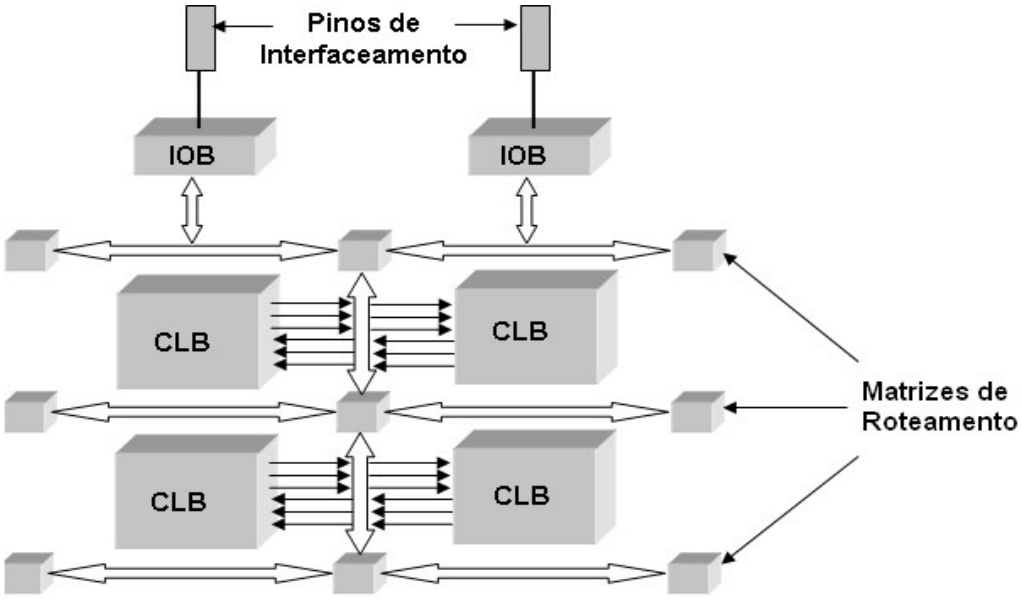

Figura 7. Diagrama de comunicação interna de um FPGA.

As primeiras arquiteturas eram projetadas e fabricadas com núcleos independentes e com circuitaria fixa. Cada um dos componentes eram programados e fabricados antes de serem agregados com a arquitetura final, esses componentes podiam ser programados por empresas diferentes de *hardware*, dessa forma o fabricante da arquitetura se abstraía da lógica interna de cada componente. A desvantagem desse tipo de desenvolvimento é que se um componente tivesse que ser alterado, por necessidade de requisito, ele tinha que ser produzido novamente agregando novos custos ao projeto. Através da Figura 8 pode se visualizar o método de desenvolvimento das primeiras arquiteturas, onde os chips são produzidos separadamente e soldados na arquitetura final.

A programação dos componentes internos do FPGA como memórias, registradores, microcontrolador, cache, *buffers* e toda configuração de CLB's, IOB's e matrizes de roteamento é feito de forma diferente das arquiteturas fixas. Ao invés dos componentes serem pré-fabricados independentemente para serem colocados na placa, eles são desenvolvidos, estruturados e são sintetizados em string de bits para serem carregados dentro do FPGA. As implementações dos

componentes podem ser feitas com ferramentas de descrição de *hardware* como o *Foundation*, da Xilinx, ou o Max Plus II da Altera.

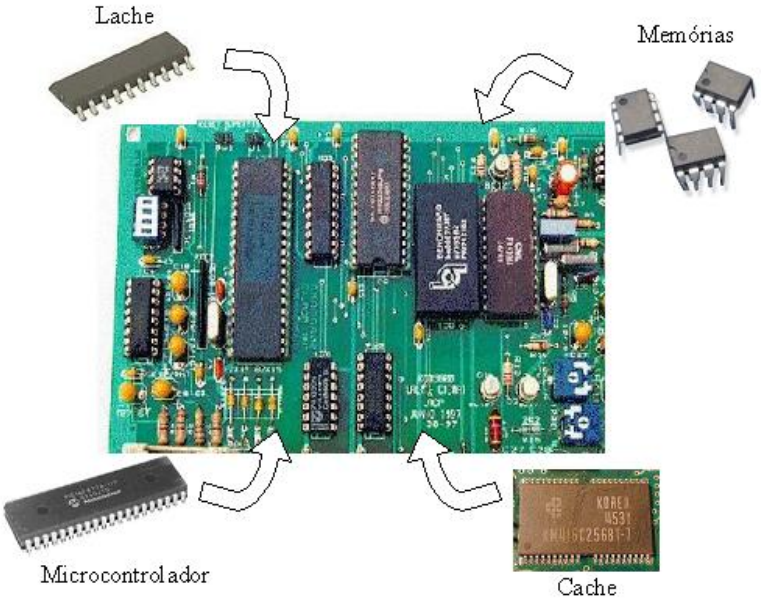

Figura 8. Visualização de um exemplo de placa.

A Figura 9 mostra um exemplo de desenvolvimento de uma arquitetura usando FPGA, desde a sua concepção até a programação dos CLB's, IOB's e matrizes de roteamento. A primeira etapa é implementar os arquivos dos componentes e compilá-los sintaticamente, a segunda fase é a de sintetização dos arquivos gerados em *string* de *bits* para armazenar no FPGA e a fase final é programar as unidades do FPGA para receber os arquivos sintetizados.

As arquiteturas reprogramáveis apresentam algumas vantagens sobre as arquiteturas fixas com relação ao custo e reutilização. Construir arquiteturas usando FGPA tem um custo relativamente mais barato, pois muitos dos componentes necessários para a arquitetura não precisam ser fabricados porque eles podem ser codificados e sintetizados dentro do FPGA, o único custo empregado seria pagar pelo desenvolvimento dos *cores* (circuito lógico) do componente que serão utilizados dentro do FPGA [3]. Uma outra vantagem é a reprogramação, dinâmica ou estática, do FPGA. O programador ou usuário pode reutilizar o FPGA para outras aplicações, precisando somente baixar novos *cores* para o FPGA e programar novamente as unidades de roteamento e os pinos de interfaceamento.

As únicas desvantagens são o tempo de resposta e a complexidade de programação do FPGA. O desempenho do FPGA é mais lento do que nas arquiteturas fixas porque existem muitos circuitos lógicos se interligando dentro do FPGA, por isso os atrasos dos pulsos elétricos são maiores que nos componentes dedicados das arquiteturas fixas. Como o FPGA suporta uma grande quantidade de circuitos internos, a programação das matrizes de roteamento e pinos de entrada/saída se tornam muito complexas, pois os blocos lógicos precisam ser interligados internamente e os componentes do FGPA precisam se comunicar com os dispositivos externos.

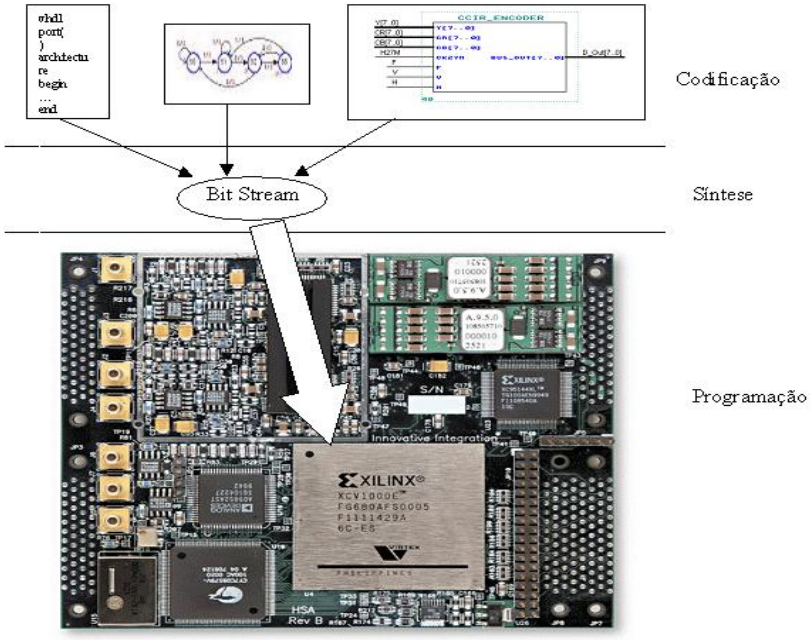

Figura 9. Ciclo básico de desenvolvimento.

Para facilitar o projeto de desenvolvimento de novas placas e arquiteturas são utilizadas ferramentas CAD (*Computer Aided Design*), que são ferramentas que ajudam no desenvolvimento do projeto, pois elas disponibilizam toda a estrutura necessária para implementação de uma arquitetura como: design da arquitetura, ambiente de codificação, compilação da sintaxe, simulação dos componentes (isolados ou projeto final), geram relatórios de dissipação de energia e desempenho, fazem mapeamento dos circuitos lógicos implementados dentro do FGPA e ainda fazem programação automática das matrizes de roteamento.

Capítulo 4

Descrição da Cache em VHDL

Nos capítulos anteriores tentou-se expor o estado da arte dos temas abordados neste trabalho, bem como explicar os detalhes necessários para compreensão do mesmo. Com a base teórica findada, espera-se agora demonstrar os aspectos práticos deste trabalho e que as informações obtidas com os resultados alcançados, sirva para ajudar no desenvolvimento de projetos novas arquiteturas de *hardware.*

A apresentação da metodologia objetiva demonstrar os aspectos práticos implementados e as simulações executadas neste trabalho

4.1 Idéia e dificuldade inicial

No capitulo 2, foram discutidos os componentes da cache, suas implementações e os aspectos de projeto que influenciam no desenvolvimento. Neste capítulo, será discutida a implementação da cache feita pelo autor, bem como as dificuldades do desenvolvimento e a escolha da cache a ser desenvolvida.

A modelo de cache implementado foi baseado em estudos sobre arquiteturas de cache existentes [15]. O intuito deste trabalho não foi desenvolver novos tipos de mapeamento e algoritmos de substituição da cache, a idéia foi codificar um modelo de cache existente usando a linguagem de descrição VHDL.

A idéia inicial era fazer uma cache parametrizável onde um usuário pudesse configurar a cache em termos de capacidade de armazenamento, tipo de mapeamento (direto, associativo, associativo por conjunto) e o algoritmo de substituição.

Uma cache totalmente parametrizável era um projeto que gastaria muito tempo de implementação e não caberia no escopo deste trabalho. Foi preciso então definir um escopo de projeto menor de cache para desenvolver.

4.2 Descrição e modelagem da cache

A arquitetura de cache a ser modelada foi selecionada aleatoriamente baseada nos estudos e implementações iniciais. As implementações iniciais sofrearam bastantes modificações devido à dificuldade, induzindo a uma escolha de projeto com uma complexidade menor.

O projeto da cache produzido é baseado na Figura 4 vista anteriormente. Neste modelo pode ser visto a memória que armazena as informações (bit de paridade, dados e rótulos), o módulo comparador que sinaliza se a palavra esta no banco de informações, o multiplexador que seleciona o bloco da linha que será enviado (ou atualizado) para o processador, o *buffer* de dados que armazena temporariamente o dado da cache e o módulo controlador que é responsável por todo o fluxo dentro da cache.

A arquitetura de cache definida e implementada é muito parecida com a descrita anteriormente, com algumas particularidades. Por exemplo, a memória foi separada em memória de dados e memória de rótulos para facilitar a comparação, foi implementado um módulo decodificador, foi acrescido um *buffer* para armazenamento temporário do endereço da instrução e sinais de controle. Os detalhes e a definição dos componentes serão vistos na próxima seção.

4.2.1 Detalhes da implementação da cache

O desenho esquemático representado na Figura 10 foi essencial para compreensão da implementação. Os estudos relacionados a arquiteturas de cache foram muito trabalhosos, porque muitas das literaturas sobre organização de computadores abordam sobre o funcionamento da cache, seus algoritmos, desempenho, custos, estatísticas, detalhes de projeto, mas não explica a implementação física dos componentes em um nível lógico mais baixo que seria de muita utilidade para o projeto deste trabalho.

O artigo de Mamidipaka e Dutt [7], foi um dos poucos lidos que descrevia em nível de detalhes os principais componentes de uma arquitetura de cache. Este artigo foi fundamental para o entendimento do projeto, para esclarecer algumas idéias e ajustar a descrição final do trabalho antes de partir para implementação.

O projeto final da arquitetura da cache ficou definido da seguinte maneira (ver Figura 10): um módulo decodificador, um comparador de *bits*, dois *buffers* de armazenamento temporário (um *buffer* para armazenar dados e outro para armazenar um endereço), um banco de armazenamento para guardar dados e instruções, outro banco para armazenar rótulos identificadores de linha e o módulo cérebro da cache, o controlador.

O decodificador é responsável por receber o endereço que vem do processador e definir qual a linha da cache vai ser endereçada na operação de escrita/leitura da cache. Quando o processador deseja executar uma operação na cache, ele envia um endereço que referencia uma única linha na cache e envia também sinais de controle para o controlador da cache. Perceba que o endereço vai direto para o decodificador e não para o controlador, isto é feito para economizar ciclos de máquina, pois se o endereço fosse para controlador primeiro, para que este ficasse responsável em mandar para o decodificador, haveria uma perda desnecessária de *clocks*. É mais vantajoso o endereço ir direto para o decodificador, enquanto, o controlador já vai decidindo o que fazer firmado nos sinais de controle (enviados também do processador). A responsabilidade do decodificador é unicamente identificar qual a linha da cache vai ser usada na operação.

O comparador é um modulo que compara duas palavras (*string* de *bits*) *bit*-a-*bit* para verificar se seus valores são iguais. O comparador, dentro da cache, tem a função de sinalizar ao controlador se a instrução (ou dado) requerida pelo processador esta ou não na cache. Por exemplo, durante a execução de um programa o processador busca o que seria a próxima instrução, mas a cache precisa saber se a instrução desejada esta ou não na linha referenciada pelo endereço do processador. Contudo, pode acontecer da instrução não estar presente em um bloco vindo da memória principal ou até mesmo já ter sido substituído por um novo bloco. O comparador é responsável justamente por identificar a existência da informação dentro da cache através de comparações de identificadores.

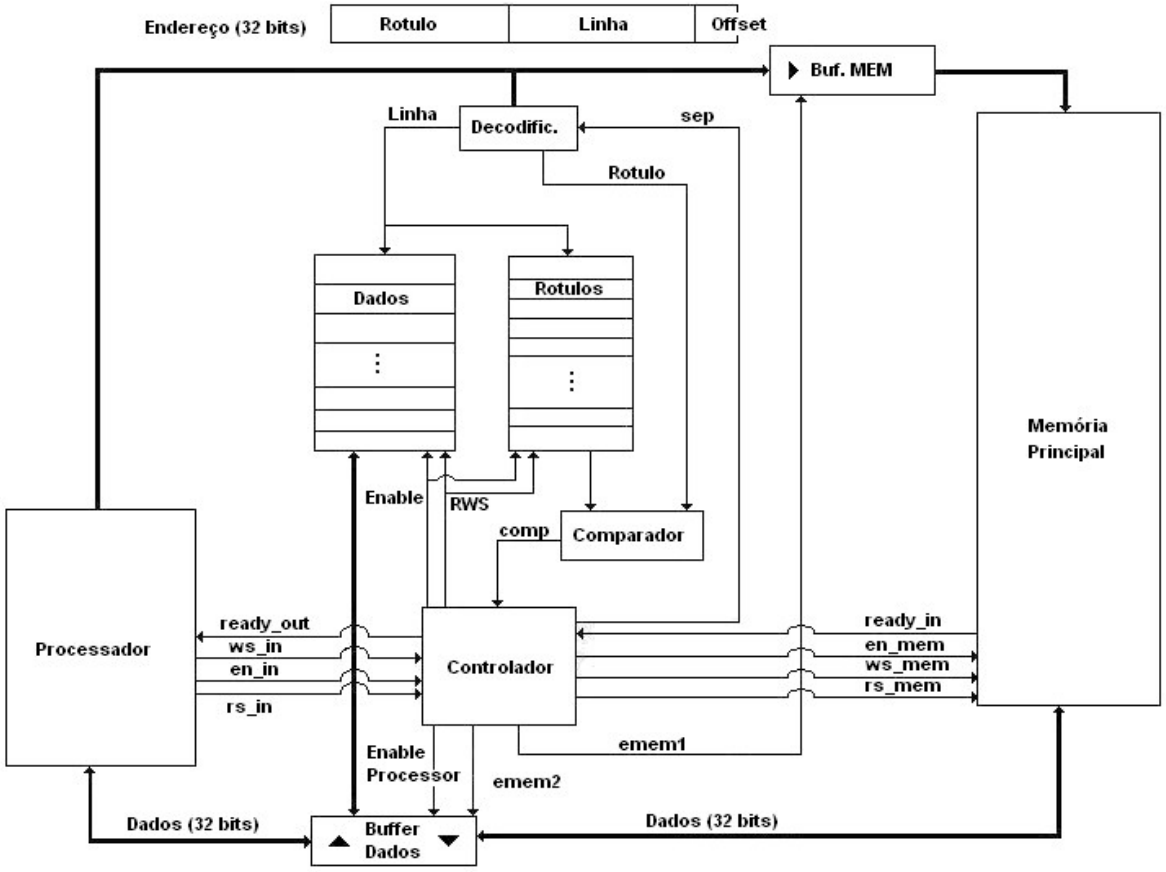

Figura 10. Desenho esquemático da cache.

O *buffer* desempenha um papel importante da arquitetura de cache. As primeiras implementações da cache possuíam um problema grave de desempenho, pois toda vez que uma linha da cache sofria uma atualização, essa alteração tinha que ser propagada para a memória principal. Como a cache passa por muitas atualizações na execução de um único programa, a cache acessava direto o barramento do sistema para enviar novos dados (ou buscar novos dados) para a memória, mas esse constante acesso deixava muito o barramento principal muito ocupado com as transações da cache. O *buffer* foi inserido na arquitetura de cache para tentar amenizar o tráfego no barramento do sistema com solicitações da cache.

Toda vez que novas linhas da cache precisavam ser atualizadas na memória principal, essas novas linhas eram armazenadas em um conjunto de *buffers* dentro da arquitetura da cache, e eram atualizadas todas de uma só vez sempre que era preciso (tempos em tempos). A quantidade de *buffers* varia dependendo da implementação e do algoritmo de atualização da cache. O algoritmo de atualização "Escrita de volta" (descrito na seção 2.2.5) já implementa uma técnica que visa diminuir os acessos da cache ao barramento do sistema.

No desenho esquemático da cache na Figura 10 só contém dois *buffers*, um para armazenar dados e outro para endereço. Eles foram dispostos na arquitetura para agilizar a execução da cache. Quando o processador busca uma nova instrução, ele envia um endereço, que é decodificado, indicando uma linha da cache onde possivelmente tem a instrução desejada, caso essa instrução não seja a esperada pelo processador, esta linha precisa ser atualizada na cache antes de enviada como resposta para o processador. Quando isto acontece o controlador da cache precisa enviar o mesmo endereço para a memória, pois este endereço localiza exatamente a

instrução almejada pelo processador. É por este motivo que o controlador necessita guardar o endereço corrente, pois se a cache desejar se comunicar com a memória ela usa o mesmo endereço da execução. Se o *buffer* de endereço não existisse, quando a cache decodificasse o endereço e percebesse que a instrução desejada não esta presente, o controlador teria que solicitar ao processador novamente o endereço para poder ir buscar um novo conjunto de blocos na memória principal, gastando mais ciclos de máquina. Com o *buffer* é diferente, quando há um erro na busca (ou necessidade de atualização) o controlador já dispõe do endereço (armazenado), ele só precisa liberar (sinalizar) a informação armazenada no *buffer* de endereço. Esta simples implementação já tem uma ganho de desempenho bastante considerável [15].

O *buffer* de dados foi introduzido na arquitetura de cache com o mesmo intuito do *buffer* de endereço que é proporcionar um melhor tempo de resposta da cache, mas o *buffer* de dados trabalha de forma diferente (do *buffer* de endereço) para propiciar um bom desempenho.

Quando o processador realiza uma leitura na cache, duas operações são executas em paralelo para que o resultado da leitura seja retornado mais rapidamente. Uma das operações é selecionar a linha da cache (a informação) que será retornada, e a outra é saber (comparando rótulos) se esta linha selecionada é a esperada pelo processador. A Figura 11 expõe o fluxo destas operações durante a leitura da cache.

Quando o endereço chega a cache, o decodificador seleciona a linha da cache que será retornada, é a partir desde ponto que a cache executa as operações em paralelo. O dado selecionado pelo endereço é armazenado no *buffer* de dados (ver fluxo em vermelho na Figura 11), e, aguarda que o controlador sinalize o resultado da comparação entre o rótulo contido no endereço (enviado pelo processador) e o rótulo da linha selecionada (ver fluxo em azul na Figura 11).

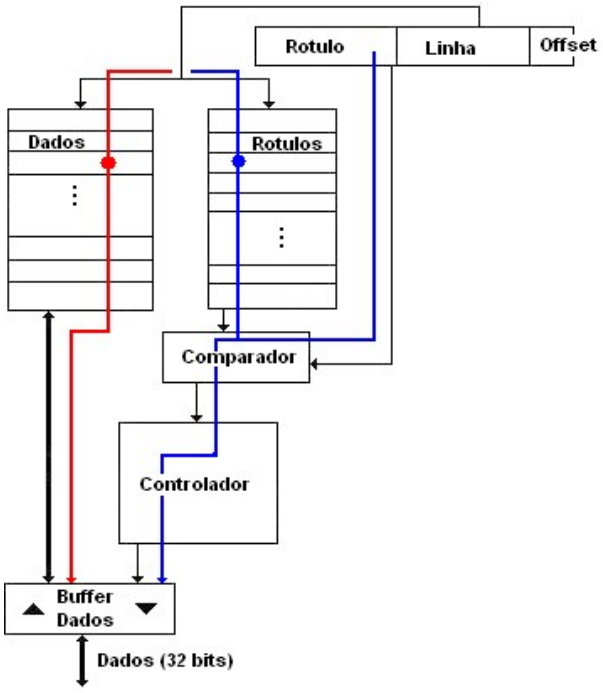

Figura 11. Paralelismo de execução – Operação de leitura.

A compreensão desta abordagem é muito simples, o dado selecionado é armazenado no *buffer* independentemente do resultado da comparação (no comparador), se houver um acerto (*cache-hit*) na comparação, ou seja, o dado selecionado é o esperado pelo processador, o

controlador libera a informação que esta armazenada no *buffer* de dados, caso o resultado da comparação seja negativa (*cache-miss*) o controlador apenas descarta a informação que esta armazenada no *buffer* e parte para a atualização da linha da cache.

Na operação de escrita de um novo dado na cache, o *buffer* de dados apenas armazena o dado (temporariamente), enquanto o decodificador seleciona a linha em que será subscrita pelo novo dado.As arquiteturas mais recentes de cache usam uma abordagem dedicada para armazenar dados e instrução. Essa realização implementa uma cache para somente conter dados (chamada de cache de dados) e uma cache somente para instrução (cache de instrução). O processador faz mais buscas por instruções do que dados na cache na hora que esta executando um programa[13], por isso esse tipo de arquitetura dedicada tem um alto desempenho, pois para esse tipo de implementação pode-se usar memórias de múltiplos acessos para cache de instrução e cache de acesso único para cache de dados. A Figura 12 exemplifica uma arquitetura dedicada de cache.

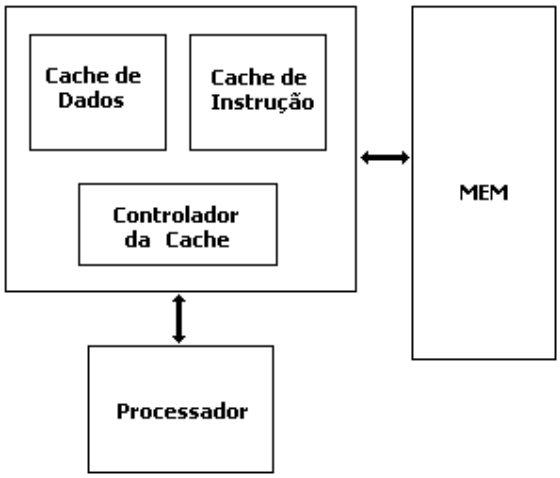

Figura 12. Arquitetura dedicada de cache.

A implementação de cache dedicada é muito difícil porque nesta abordagem o controlador se torna muito complexo, pois gerenciar duas caches não é tarefa fácil. O projeto implementado não usou cache dedicada para dados e instruções, mas foi aproveitada a idéia de usar memórias dedicadas.

Na arquitetura codificada foram usados dois bancos de armazenamento. Um dos *arrays* foi implementado para comportar os dados e as instruções necessárias para execução do processador, e o outro para conter somente os rótulos identificadores das linhas. O primeiro *array* suporta uma quantidade de linhas variável dependendo do tamanho da cache escolhida (16kb, 32kb, 64kb, 128kb ou 256 kbytes), mas a linha é fixada em 32 *bits* por linha, que foi adotado de acordo com o padrão das arquiteturas RISC [14] (também baseado no processador MIPS).

O segundo *array* é exclusivo para armazenar os rótulos que identificam as informações em cada linha. Este tem a quantidade de linhas variando também de acordo com o tamanho definido, mas diferente do *array* de dados, ele varia também a quantidade de bits por linha. Isto acontece porque tirando o *index* e o *offset* o que sobra é usado como rótulo para identificar a informação armazenada. Por exemplo, considerando uma cache de 16kb e que a são 32 *bits* por linha, tem-se uma cache com 4096 linhas ((16384 * 8) / 32). No endereço tem 32 bits, geralmente dois são do *offset*, e para endereçar 4096 linhas são necessários 12 bits (2 ^ 12 = 4096), o que daria 14 *bits* (junto com o *offset*) e o resto seria usado como rótulo para identificar as diferentes linhas (os 18 *bits* restantes). Aumentando o tamanho da cache para 64kb (com 32 bits fixos por linha), tem-se que a cache aumenta para 16384 linhas ((65536 * 8) / 32), novamente fazendo os

cálculos: 2 *bits* de *offset* + 14 *bits* para endereçar 16384 linhas, que dá um total de 16 *bits*, restando 16 bits para o rótulo. Vale salientar que não necessáriamente são usados todos os *bits* restantes, no primeiro exemplo pode ser visto que como existe 4096 linhas, seria preciso somente 4096 rótulos para identificar cada linha como única, ou seja, dos 18 *bits* restantes só precisariam ser usados 12 *bits* para gerar todas as combinações de rótulos necessárias e o restante das combinações seriam descartadas (*don't care*).

Observe nos exemplos acima que quando a quantidade de linhas aumenta, o tamanho do *index* também aumenta (para poder endereçar todas as linhas), e, a quantidade de *bits* que são usados como rótulos diminui. É por este motivo que o *array* de rótulos varia tanto no tamanho (quantidade de linhas) quanto na quantidade de *bits* por linha.

O componente mais importante da arquitetura de cache sem dúvidas é controlador, ele é quem gerencia toda comunicação entre os outros módulos, recebendo sinais, tratando-os e decidindo o que fazer. A cada novo ciclo de máquina, o controlador da cache adota um comportamento diferente, direcionando o fluxo dentro da arquitetura da cache através de sinais de controle.

Uma arquitetura de cache bem implementada e bem integrada com o processador e memória tem uma resposta rápida e precisa. O primeiro processador Pentium, da Intel, tinha uma arquitetura de cache com 16 Kb (dedicada com 8 Kb de dados e 8 Kb de instrução) de capacidade, mapeamento associativo por conjunto e algoritmo de atualização *write back*. Esta arquitetura da Intel gastava não mais que dois ciclos de máquina para realizar uma leitura (com sucesso) na cache e cinco ciclos para escrita (com sucesso) [12].

A quantidades de ciclos que uma instrução gasta para ser completada depende da técnica usada para sincronizar os sinais internos do sistema. Na arquitetura do Pentium (e nas mais recentes), todos os componentes trabalhavam de acordo com um *clock* externo do sistema, dessa maneira, todos os componentes podiam trabalhar de forma sincronizada.

O controlador da cache implementada nesta monografia não se preocupa com os atrasos dos sinais dos componentes, pois este não é um dos objetivos do trabalho, mas para garantir a validação da saída (resposta da cache) foi utilizada uma máquina de estados para compor o núcleo do controlador.

Máquinas de estados são modelos para representação de sistemas seqüenciais bastante poderosas, pois tem grande poder de decisão e quando bem modeladas não geram estados desconhecidos (*deadlock*). Máquinas de estados são compostas de estados, eventos (entradas) e ações (saídas). Os resultados das saídas, gerados em cada estado, dependem das entradas recebidas em um determinado estado. Maiores detalhes sobre máquinas de estados, diferentes abordagens e ferramentas podem ser encontradas em [11].

O controlador da cache codificada neste trabalho foi desenvolvido baseado em máquinas de estados. A escolha desta abordagem possibilitou algumas vantagens na implementação, uma delas foi a capacidade de representação gráfica do controlador em um modo compacto, facilitando a compreensão do controlador. Outra vantagem foi a possibilidade de suprir a falta de sincronização entre os módulos; com máquinas de estados pode-se permanecer (recursivamente) em um dos estados até que uma determinada entrada seja recebida. Dessa maneira, o controlador poderia continuar em um estado definido até as tarefas de estados anteriores serem totalmente compridas, por exemplo, tarefa de comparação ou decodificação do endereço. A principal desvantagem desta técnica de modelagem é o fato do controlador poder se conservar em um dos estados por muitos ciclos de máquina devido aos atrasos dos circuitos lógicos codificados.

O projeto desenvolvido utilizou uma única máquina de estado para modelar quatro possíveis fluxos de informações dentro da cache, que são: uma leitura com acerto, uma leitura com erro, uma escrita com acerto e uma escrita com erro. A Figura 13 representa o diagrama da máquina de estado utilizada.

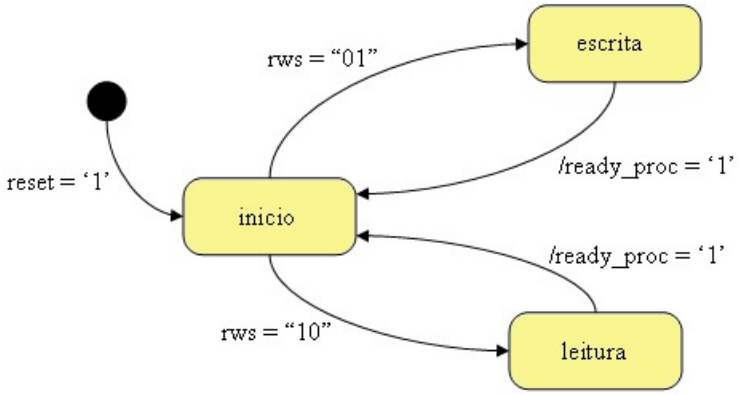

Figura 13. Diagrama da máquina de estado do controlador.

A máquina de estado do controlador é iniciada quando o sistema é iniciado. O sinal de controle '*reset*' inicializa todos os componentes do sistema, quando setado para nível lógico alto (um) o controlador vai para o estado inicial, as memórias (dados e rótulos) e os *buffers* (dados e endereço) são apagados, se ele estiver em nível lógico baixo (zero) o sistema permanece o mesmo (sem alterações).

No primeiro estágio da máquina de estado, as variáveis e sinais de controle são inicializados e o controlador aguarda o sinal de controle que habilita a operação, este sinal indica se o processador deseja executar uma escrita ou uma leitura na cache. Se o sinal de controle '*rws*' estiver em nível lógico alto, o controlador executa uma operação de escrita, caso contrário efetua uma operação de leitura (nível lógico baixo). Depois da execução de todo o fluxo dentro da cache, o controlador volta para o estado inicial (ver Figura 14).

A Figura 14 é um diagrama minimizado da máquina de estado, somente para ilustrar o fluxo interno do controlador e abstrair o leitor das minuciosidades. A seguir será visto este modelo com mais detalhes, abrangendo os fluxos de leitura e escrita separadamente.

O estado inicial (da máquina de estados) do controlador é comum tanto ao fluxo de leitura como ao fluxo de escrita vistos na Figura 14. No estado inicial o controlador decide se vai executar uma escrita ou leitura baseado num sinal de controle chamado de '*rws*' (*read-write signal*), este sinal de controle é composto de dois *bits* que dependendo da combinação determina a operação a ser feita na cache. Por exemplo, se o sinal for '*rws = 01*', indica que o processador deseja escrever um dado na cache, se o sinal for '*rws = 10*' sinaliza uma operação de leitura na cache, as outras duas combinações ('00' e '11') são situações que não representam uma operação, mas são necessárias para execução. Estas outras duas combinações servem para quando o controlador não esta executando nenhuma escrita ou leitura interna no banco de informações. Quando o controlador está no estágio de comparação de rótulos, por exemplo, ele não esta fazendo uso dos *arrays* de armazenamento, então o sinal '*rws*' precisa assumir um valor diferente do parâmetro que representa a escrita ou a leitura, se este tivesse apenas um *bit* não poderia fazer isso, pois ele só poderia assumir 'zero' ou 'um'. Por este motivo é que o sinal '*rws*' tem dois *bits*, pois uma combinação representa a escrita, outra representa a leitura e as outras duas combinações são usadas quando o controlador não deseja usar os *arrays* de armazenamento.

A máquina de estados que representa o fluxo da leitura na cache é simples de ser compreendida, ela é composta de apenas quatro estados como pode ser visto na Figura 14. No estado inicial o controlador determina qual a operação o processador esta executando (se é leitura ou escrita). No estado inicial também são executados a decodificação do endereço, a seleção da linha desejada e o armazenamento da informação referenciada pelo endereço. Após a execução das ações do estado inicial, se a operação for de leitura a máquina de estados vai para o estado de

comparação, caso contrario executa a operação de escrita (que será vista em mais detalhes adiante).

O próximo passo do controlador é saber se a informação desejada esta (ou não) na cache. O estado "compara", do diagrama de leitura, é responsável pela comparação dos rótulos, neste estágio o controlador a aguarda o resultado do módulo comparador, se o dado estiver na cache (acerto - *hit*) a máquina de estado pula para o próximo passo que é somente liberar o dado para o processador. Para entender melhor este processo, ver a figura abaixo.

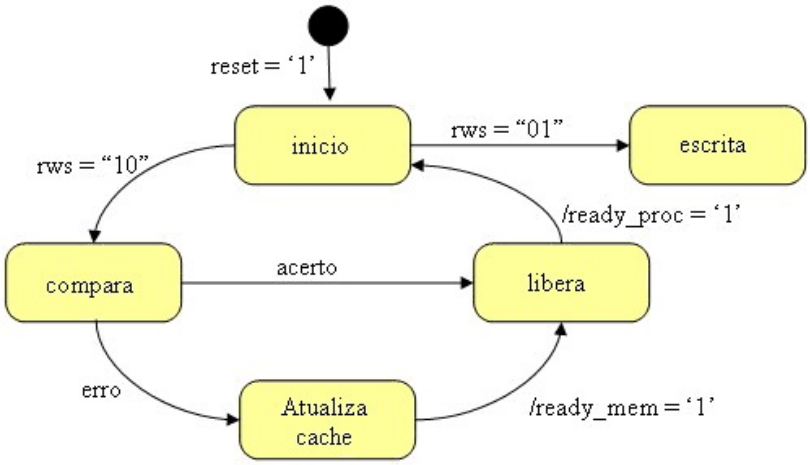

Figura 14. Diagrama da máquina de estados – Operação de leitura.

Caso haja um erro na leitura da cache (*miss*), o controlador precisa atualizar a linha da cache com um novo bloco da memória principal antes de liberar a informação para o processador. No estado de atualização da cache (ver Figura 14), o controlador envia o endereço do processador para memória principal, solicitando o novo bloco para troca na cache. Neste estado o controlador fica aguardando que a memória principal forneça o novo bloco e sinalize (*ready_mem = '1'*) que o novo dado já esta disponível para atualização, enquanto isso não ocorrer o controlador permanece no mesmo estado até a nova entrada ser recebida.

O último estado do controlador é de retorno da informação buscada pelo processador. Neste estágio o controlador apenas libera a informação que esta armazenada no *buffer* dados para o processador e sinaliza o término da operação (*ready_proc = '1'*). Todos os sinais de controle citados até agora, serão descritos com mais ênfase na seção 4.3 que trata das simulações da cache.

O diagrama de estados que representa a operação de escrita visualmente é mais simples que o diagrama da leitura, mas durante a execução o fluxo de escrita é mais complexo, pois mais informações são envolvidas além do endereço da memória e sinais de controle (presente no fluxo de leitura), o controlador se preocupa com o dado que vem do processador para ser escrito na cache, e, conseqüentemente atualizado na memória principal.

O diagrama de estados da operação de leitura tem três estados, além do estado inicial descrito anteriormente. Estes estados são: atualização da cache, atualização da memória e finalização (ver Figura 16).

No estado de atualização de cache, o controlador sobrescreve a linha referenciada pelo endereço (decodificada no estado inicial) com o dado que é enviado pelo processador. Quando toda a operação de escrita na cache é terminada, o controlador fica incumbido de atualizar a memória principal, que é o próximo estado (atualização da memória) após a atualização da cache.

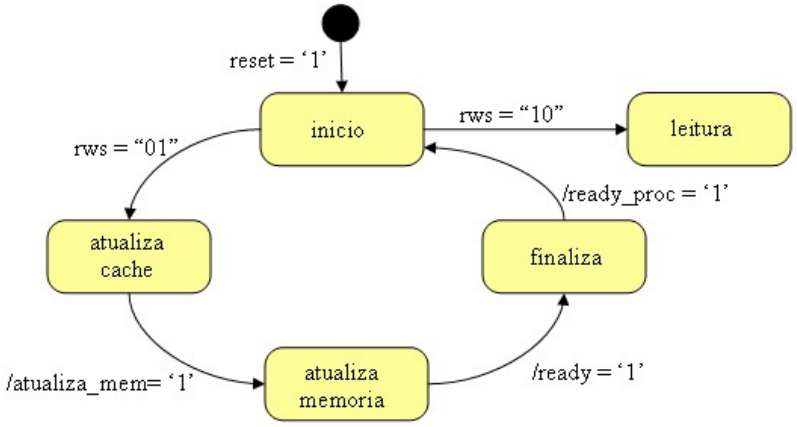

Figura 15. Diagrama da máquina de estados – Operação de escrita.

No estado de atualização da memória principal, o controlador envia o endereço para a memória (endereço armazenado previamente no *buffer*) e envia os sinais de controle que habilita a memória principal e sinaliza a escrita. Neste estado o controlador aguarda uma resposta da memória principal, sinalizando que a escrita foi efetuada com sucesso (*ready_mem = '1'*), caso isso não aconteça o controlador permanece neste estado durante o próximo ciclo.

O último estado é o de finalização da operação de escrita. Neste estado o controlador zera todos os sinais internos da cache, sinaliza (*ready_proc = '1'*) para o processador o término da operação de escrita e vai para o estado inicial da máquina de estados.

Um detalhe deve ser posto em evidência neste diagrama de operação da Figura 15, note que não há o estado de comparação neste diagrama, como existe no diagrama de leitura, e a execução da comparação não faz parte internamente de nenhum dos estados do diagrama. Na implementação desenvolvida, o fluxo de escrita não executa nenhuma comparação, quando o processador envia um dado para ser escrito na cache, o controlador apenas sobrescreve o dado sem fazer nenhuma comparação, isto se deve ao fato do algoritmo de escrita direta (write-trough). Quando um dado é escrito na cache, tão brevemente ele é atualizado na memória principal (devido ao algoritmo), logo, a memória principal dificilmente terá um dado desatualizado, por este motivo não há necessidade de enviar a linha que será sobrescrita para memória principal antes de atualizar a cache. É devido ao tipo de implementação desenvolvida, que não se faz necessário à comparação entre rótulos antes da escrita (atualização) na cache.

Depois de ter apresentado toda a modelagem da arquitetura da cache implementada, será visto na próxima seção a estrutura interna dos componentes e as simulações do projeto.

4.3 Simulações da implementação

Nesta seção serão apresentadas as simulações executadas de cada módulo individualmente e a simulação final com os componentes todos integrados.

O projeto desta monografia estava sendo desenvolvida inicialmente com a ferramenta *Max Plus II* da Altera [3]. Esta ferramenta apresenta uma versão para avaliação destinada a estudantes que pode ser usada para fins acadêmicos com intuito de demonstrar como funcionam as ferramentas que auxiliam no desenvolvimento de projetos de *hardware*. Pode ser demonstrado, por exemplo, a fase de desenvolvimento, integração de módulos, testes através de simulações e a síntese do projeto para fabricação de *cores*.

Apesar da versão estudante ser gratuita, ela apresenta uma limitação muito grande de utilização, pois seu conjunto de funções e bibliotecas são bastante reduzidas. Nas primeiras implementações da cache, foram utilizados funções e chamadas a bibliotecas que eram suportados pela linguagem VHDL, mas que não foram reconhecidas como válidas pelo *Max Plus II* devido à versão reduzida para estudantes. Por exemplo, dentro da codificação estava sendo aplicada uma função para converter uma seqüência de *bits* em inteiro, que é uma função implementada por uma biblioteca bastante usada no desenvolvimento de componentes em VHDL, a ferramenta simplesmente não reconhecia a biblioteca e, além disso, apontava a chamada à função como sendo um erro de sintaxe durante a compilação do código. Um outro problema é que a ferramenta da *Altera* não suporta reconfiguração parcial, que é imprescindível para projetos de caches reconfiguráveis. Este foi o principal motivo que levou ao uso de uma ferramenta mais completa.

A solução buscada, foi tentar desenvolver o trabalho com uma ferramenta da empresa Xilinx, o *ISE Foundation* [18]. Esta ferramenta tem uma versão gratuita para uso acadêmico mais completa e com mais funcionalidades. Uma desvantagem desta ferramenta é que ela não apresenta um simulador integrado, feito a ferramenta da *Altera*, foi preciso instalar um *software* adicional somente para gerar as simulações. O *ModelSim* é um simulador para desenvolvimento de projetos eletrônicos de automação, ele é de propriedade da empresa *Mentor Graphics* [10]. Quando instalado no computador ele se integra com o *Foundation* da Xilinx e fica responsável pelas simulações dos projetos desenvolvidos.

A seguir serão apresentadas as simulações dos componentes da implementação. Como foram modelados cinco tamanhos diferentes de cache, houve muitas simulações de componentes que podem ser compreendidos (e visualizados) com uma única implementação da cache. Serão mostradas as simulações da cache de 32kbytes.

4.3.1 Módulo Decodificador

Este componente é responsável por decodificar o endereço enviado pelo processador e indicar precisamente a linha da cache que está sendo referenciada. Este módulo tem como entrada um endereço de 32 *bits* e um sinal de controle de um *bit*, e, tem duas saídas que serão utilizadas para propósitos diferentes.

O sinal de controle de um bit é utilizado para sinalizar ao decodificador quando seus dados (decodificados) devem ser colocados na saída. Ele é gerado pelo controlador da cache. Toda vez que há um novo ciclo de *clock*, o controlador envia um sinal para o decodificador indicando se as palavras decodificadas devem ser enviadas para os bancos de *arrays* de dados e de rótulos. Este sinal de controle é necessário para evitar que o decodificador envie dados repetitivos para os bancos de *arrays*. Isto se deve ao fato do controlador só precisar do decodificador nos primeiros ciclos máquina, quando é necessário decodificar o endereço enviado pelo processador, nos demais ciclos o controlador desabilita o uso deste componente. É claro que se a busca (ou atualização) do processador falhar (*cache miss*), o controlador vai precisar do mesmo endereço para atualizar a cache, ou seja, ele precisará habilitar o decodificador novamente.

A Figura 16 ilustra a simulação do decodificador habilitado pelo controlador. A primeira linha da simulação (envolto pelo quadrado verde) representa o sinal de controle do decodificador, responsável por habilitar ou desabilitar o decodificador. Nesta primeira simulação o sinal de controle (definido como ‘a’) esta em nível lógico ‘1’, habilitando as saídas (definidas como ‘linha’ e ‘rotulo’) do decodificador. A segunda linha da simulação (envolta pelo quadrado azul) representa endereço enviado pelo processador. O sinal ‘endereco’ é uma entrada de 32 *bits* no decodificador.

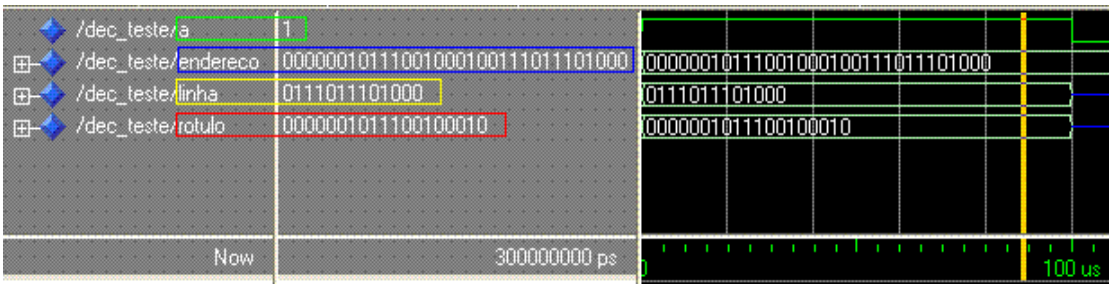

Figura 16. Simulação do decodificador habilitado.

A terceira linha da simulação, envolta pelo quadrado amarelo, representa uma das saídas do decodificador. O sinal 'linha' contém o endereço real que referencia a linha da cache que será utilizada na operação de busca ou atualização. Na simulação da Figura 16, o sinal 'linha' é uma palavra de 13 bits que irá selecionar uma das 8192 linhas da cache de 32kbytes (32 bits x 8192 linhas). O decodificador enviará este sinal para os bancos de *arrays*, tanto dados como rótulos. No *array* de dados o sinal 'linha' selecionará a palavra que retornará para o processador, caso a busca seja feita com sucesso. No *array* de rótulos o sinal 'linha' selecionará a linha que vai ser comparada com o rótulo presente no endereço do processador.

Na Figura 16, a quarta linha da simulação (envolta pelo quadrado vermelho) representa a outra saída do decodificador. Este sinal (identificado como 'rotulo') é enviado do decodificador para o componente comparador para ser comparado com a linha selecionada no array de rótulos. Os detalhes deste processo podem ser visualizados e entendidos na explicação da Figura 11.

A Figura 17, expõe o resultado de uma simulação em que o decodificador estaria desabilitado em um determinado momento da execução da cache.

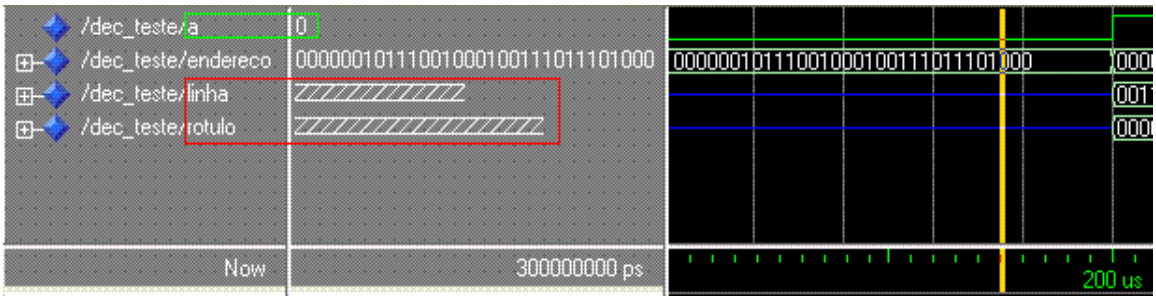

Figura 17. Simulação do decodificador desabilitado.

Pode-se observar (na Figura 17) que o sinal de controle 'a' esta em nível lógico baixo (zero), fazendo com que o endereço enviado pelo processador não seja "jogado" na saída do decodificador e as saídas 'linha' e 'rotulo' fiquem em alta impedância, ou seja, que as variáveis fiquem desconhecidas no determinado momento. A implementação em VHDL deste componente pode ser visto no anexo 1.a.

4.3.2 Módulo Comparador

Este componente é responsável pela comparação de rótulos existente na cache. Quando o processador busca uma nova linha da cache, o controlador precisa saber se a linha que o processador deseja esta ou não na cache, para isto o controlador compara se o rótulo enviado no endereço do processador equivale ao rótulo armazenado na cache. O componente comparador é justamente responsável por esta comparação.

Na Figura 18, esta o resultado de uma simulação em que a comparação entre os rótulos seria falsa, ou seja, que o dado buscado pelo processador não estaria armazenado na cache e precisaria ser atualizado.

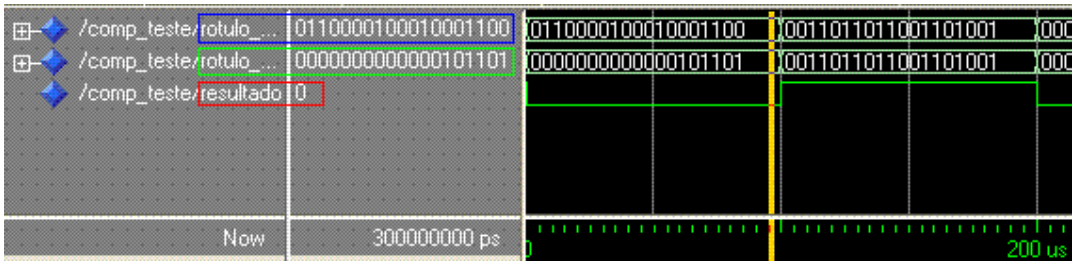

Figura 18. Simulação de um erro na comparação.

As duas primeiras linhas da simulação são entradas do comparador e a terceira linha é a única saída do comparador. O primeiro rótulo da simulação (ver figura acima), envolto pelo quadrado azul, é o rotulo que esta armazenado na cache, no *array* de rótulos mais precisamente. O segundo rótulo (envolto pelo quadrado verde) é o rótulo que veio no endereço do processador. Pode-se ver nitidamente que os rótulos comparados não são iguais, levando a saída comparador ao resultado zero (envolto pelo quadrado vermelho).

A seguir, pode-se visualizar uma simulação em que a comparação entre os rótulos é verdadeira, ou seja, a palavra buscada pelo processador esta armazenada na cache.

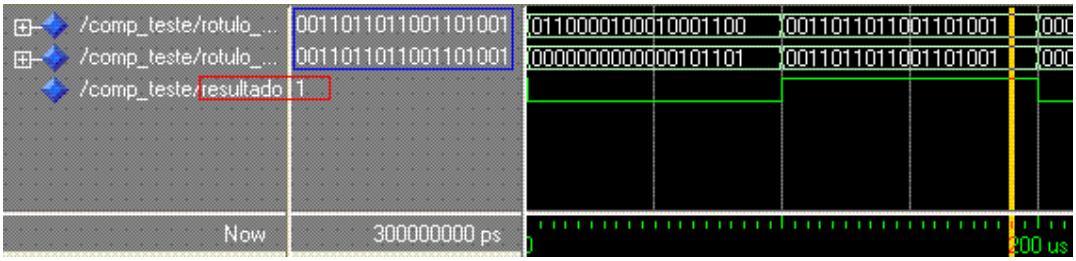

Figura 19. Simulação de um acerto na comparação.

Na Figura 19 pode se ver que os rótulos (envoltos pelo quadrado azul) comparados são idênticos, resultando em um nível lógico ‘1’ (quadrado vermelho). O resultado da comparação feita é enviado para o controlador saber se o dado desejado esta ou não na cache. Nesta ultima simulação o resultado seria verdadeiro, ou seja, o dado desejado pelo processador estaria na cache. A implementação em VHDL deste componente pode ser visto no anexo 1.b.

4.3.3 Módulo *Buffer* de Endereço

Este componente é responsável pelo o armazenamento temporário do endereço enviado pelo processador. Este *buffer* é utilizado toda vez que há um erro na busca (*cache miss*) efetuada pelo processador. Toda vez que a cache precisa ser atualizada com um novo bloco de dados, o endereço armazenado no *buffer* de memória é propagado para memória principal. Os detalhes deste processo estão descritos na seção 4.2.

Na Figura 20, esta exposto o resultado da simulação do *buffer* de memória habilitado. Este *buffer* é composto por três entradas: um *clock* do sistema, um sinal para habilitar o *buffer* (*enable*) e o endereço enviado pelo processador (identificado como ‘entrada’). A única saída é uma palavra que representa o endereço armazenado no estado anterior (representado por ‘saida’).

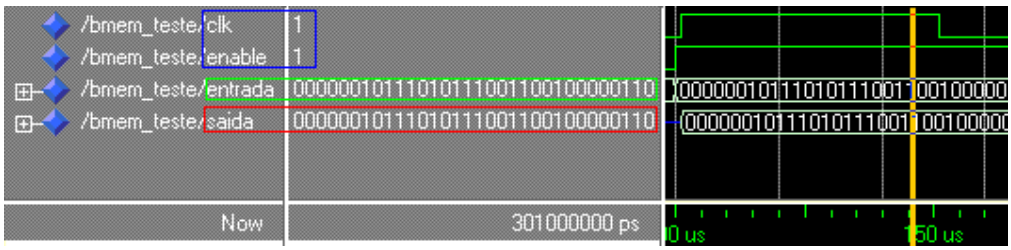

Figura 20. Simulação do *buffer* de memória habilitado.

Como pode ser visto, para habilitar o uso deste *buffer* é necessário que o clock do sistema esteja em nível lógico alto (um) e que o sinal '*enable*' esteja setado para '1' também. Quando este cenário acontece, o valor que está armazenado (anteriormente) no *buffer* é colocado na saída co componente.

A simulação a seguir (Figura 21) representa um cenário em que o *buffer* de memória não estaria sendo utilizado nem para armazenar um novo endereço nem para propagar o endereço armazenado. Note que os sinais '*clk*' e '*enable*' estão desabilitados (nível lógico baixo), fazendo com que a saída do *buffer* fique em alta impedância. É importante salientar que é necessário somente um deles esta em nível lógico baixo para o *buffer* estar desabilitado; mesmo que o *clock* do sistema esteja setado para '1', se o *enable* estiver em '0' a saída fica desconhecida e vice-versa.

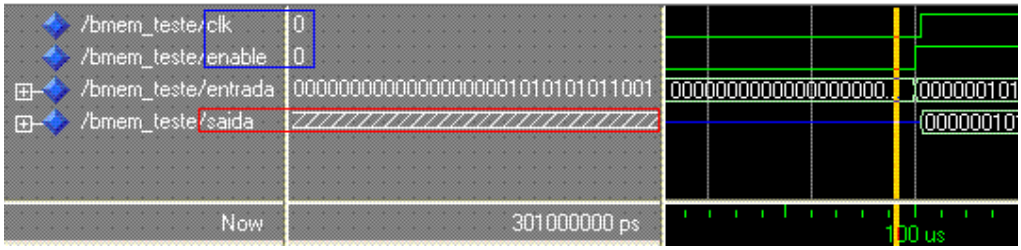

Figura 21. Simulação do *buffer* de memória desabilitado.

Este *buffer* de memória só é utilizado em duas situações, a primeira delas é quando a cache necessita de um novo bloco da memória principal, nessa situação o *buffer* propaga o endereço enviado pelo processador para memória principal. A outra situação é quando o novo bloco vem da memória principal para substituir um bloco da cache, nessa situação o controlador da cache utiliza o endereço armazenado neste *buffer* para saber qual linha da cache dever ser substituída com o novo bloco. A implementação em VHDL deste componente pode ser visto no anexo 1.c.

4.3.4 Módulo *Buffer* de Dados

O *buffer* de dados foi um dos componentes mais difíceis durante a implementação, devido ao gerenciamento de três portas de entrada/saída. Além destas três portas, este módulo é composto por quatro sinais de entrada que são usados para controle do módulo.

Para entender o fluxo mais facilmente é mostrado abaixo uma tabela que representa os possíveis fluxos deste componente. Este módulo funciona quando há um sinal de *clock* ativo (sinal do sistema) e quando não há um sinal de *clock* o *buffer* de dados não faz nada (fluxo 1).

O fluxo 2 é um exemplo de quando o processador envia um dado para ser atualizado na cache. Este dado é armazenado primeiramente no *buffer* e depois (em um outro ciclo de máquina) armazenado no *array* de dados. O fluxo 3 é um exemplo de quando a cache envia um dado para o

processador (*cache-hit*), esse dado é armazenado no *buffer* primeiramente e depois enviado para o processador. O fluxo 4 é um exemplo em que a memória principal envia um dado para ser armazenado na cache (atualização de uma linha da cache), ele inicialmente é armazenado no *buffer* e depois enviado para o *array* de dados.

Os fluxos 5, 6 e 7 são na verdade passos subseqüentes dos fluxos 2, 3 e 4. Por exemplo, quando o processador envia o dado para ser armazenado na cache (fluxo 2), no próximo *clock* do sistema o fluxo ativo seria o 5, indicando que o dado armazenado no *buffer* pelo processador no estado anterior deve ser enviado para o *array* de dados.

Se o processador estiver executando uma leitura, a cache envia para o *buffer* (fluxo 3) a informação necessária para o processador, e no próximo ciclo de *clock*, envia o dado armazenado no *buffer* para o processador (fluxo 7 na tabela 3).

Na tabela 3 pode ser visto ainda o fluxo 6 que seria o envio da informação armazenada *buffer* para a memória principal, neste caso seria um exemplo em que a memória estaria sendo atualizada.

Tabela 3. Bits de controle no *buffer* de dados.

Fluxo	clk	eproc	emem2	emem3	Resultado
1	'0'	*don't care*	*don't care*	*don't care*	Não faz nada.
2	'1'	'1'	'0'	'0'	Dado que vem do processador é armazenado no *buffer.*
3	'1'	'0'	'1'	'0'	Dado que vem da cache é armazenado no *buffer.*
4	'1'	'0'	'0'	'1'	Dado que vem da memória principal é armazenado no *buffer.*
5	'1'	'1'	'1'	'0'	Dado que esta armazenado no *buffer* é enviado para cache.
6	'1'	'1'	'0'	'1'	Dado que esta armazenado no *buffer* é enviado para memória principal.
7	'1'	'0'	'1'	'1'	Dado que esta armazenado no *buffer* é enviado para o processador.
8	'1'	'0'	'0'	'0'	Don't care
9	'1'	'1'	'1'	'1'	Don't care

Na Figura 22, pode ser vista com mais detalhes os fluxos descritos na tabela 3. Estas duas simulações fazem parte da operação de escrita na cache. A parte superior da Figura 22 é o momento em que o processador envia o dado para o *buffer*, note que o *clock* do sistema e o sinal 'eproc' estão em níveis lógicos altos, indicando que a informação que esta sendo armazenada no *buffer* vem do processador. O quadrado vermelho representa o dado enviado pelo processador para ser armazenado na cache. É importante observar que as portas de entrada/saída nunca vão

estar ativas ao mesmo tempo, somente uma delas estará ativa (as outras estarão em alto impedância) e funcionando como entrada ou como saída.

A parte inferior da Figura 22 demonstra o momento em que o dado armazenado no *buffer* é enviado para a cache. Além dos sinais 'clk' e 'eproc', o sinal 'emem2' está em nível lógico alto, indicando que o *buffer* irá enviar o dado armazenado para a cache. Note que o *buffer* agora tem uma porta de entrada/saída sendo usada como saída do módulo (quadrado vermelho na parte inferior da Figura 22). É importante salientar que estas duas simulações seriam ciclos consecutivos dentro do *buffer* de dados.

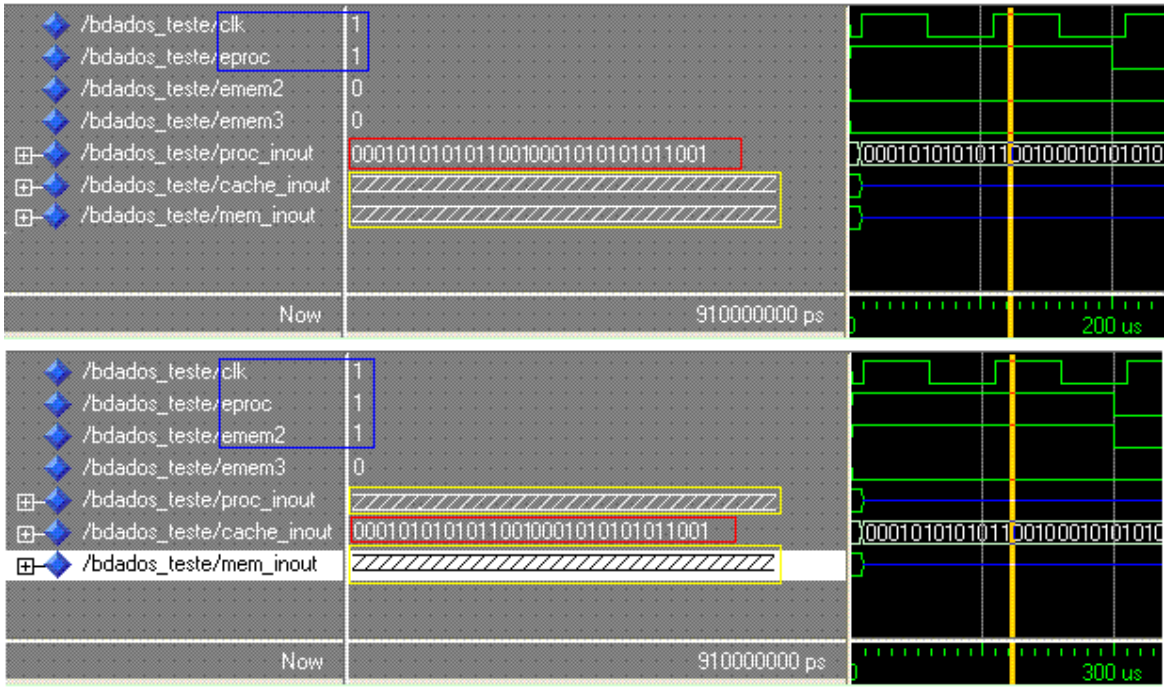

Figura 22. Simulações do fluxo *buffer* de dados e cache.

As simulações presentes na Figura 23 são resultados de uma atualização da memória principal devido à atualização da cache. A parte superior da figura seria o momento em que o processador envia o novo dado para o *buffer*. Note que o *clock* e o sinal 'eproc' estão em níveis lógicos altos, indicando que uma informação do processador esta sendo armazenada no *buffer* de dados. Nesta simulação a porta de entrada/saída 'proc_inout' esta sendo utilizada como entrada do *buffer*.

Na parte da inferior da Figura 23 está representado o momento em que o *buffer* envia o dado para a memória principal. Neste caso, além dos sinais 'clk' e 'eproc', o sinal 'emem3' esta em nível lógico alto, indicando que o dado armazenado no *buffer* será enviado para a memória principal através da porta de entrada/saída 'mem_inout', que neste exemplo esta sendo usada como porta de saída do componente.

A outra situação existente é quando a informação vem da memória principal para a cache. Este tipo de situação acontece toda vez que a cache necessita de uma nova informação da memória principal.

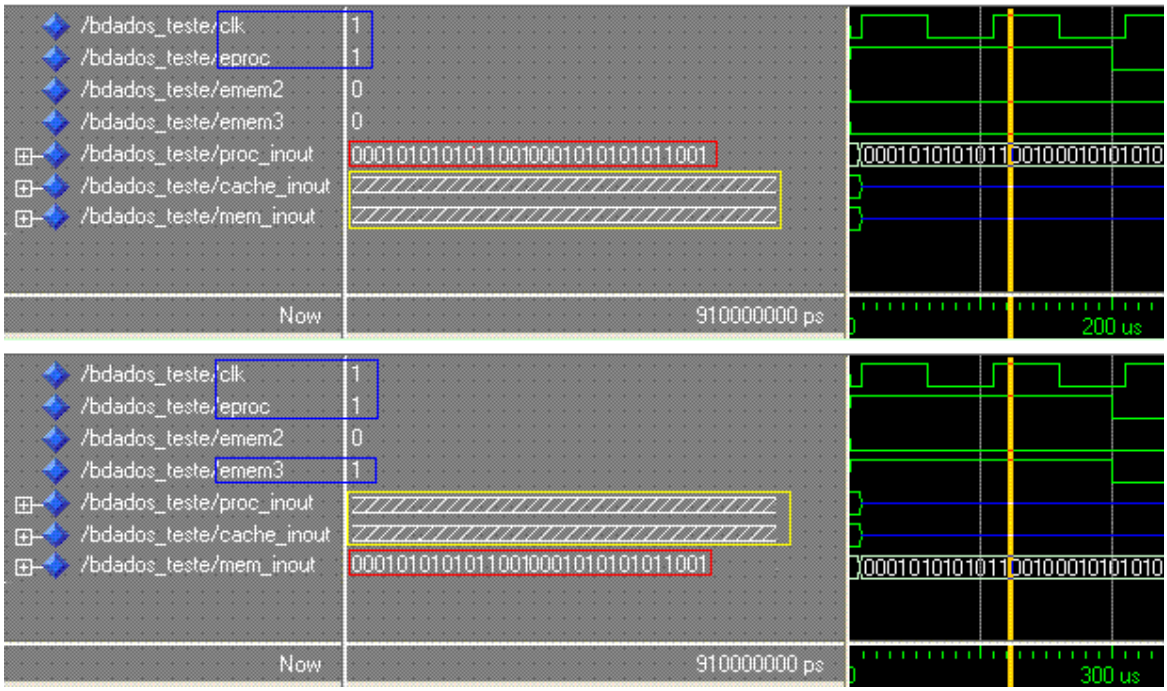

Figura 23. Simulações do fluxo *buffer* de dados e memória principal.

Esta simulação é parecida com as simulações vistas anteriormente, onde a informação é armazenada primeiramente no *buffer* e depois enviada para a cache. Na parte superior da Figura 24, está a simulação de um envio de uma informação da memória principal para o *buffer* de dados. Note que os sinais 'clk' e 'emem3' estão em níveis lógicos altos (ver quadrados azuis na Figura 24), indicando que o *buffer* vai receber uma informação na porta de entrada/saída 'mem_inout'.

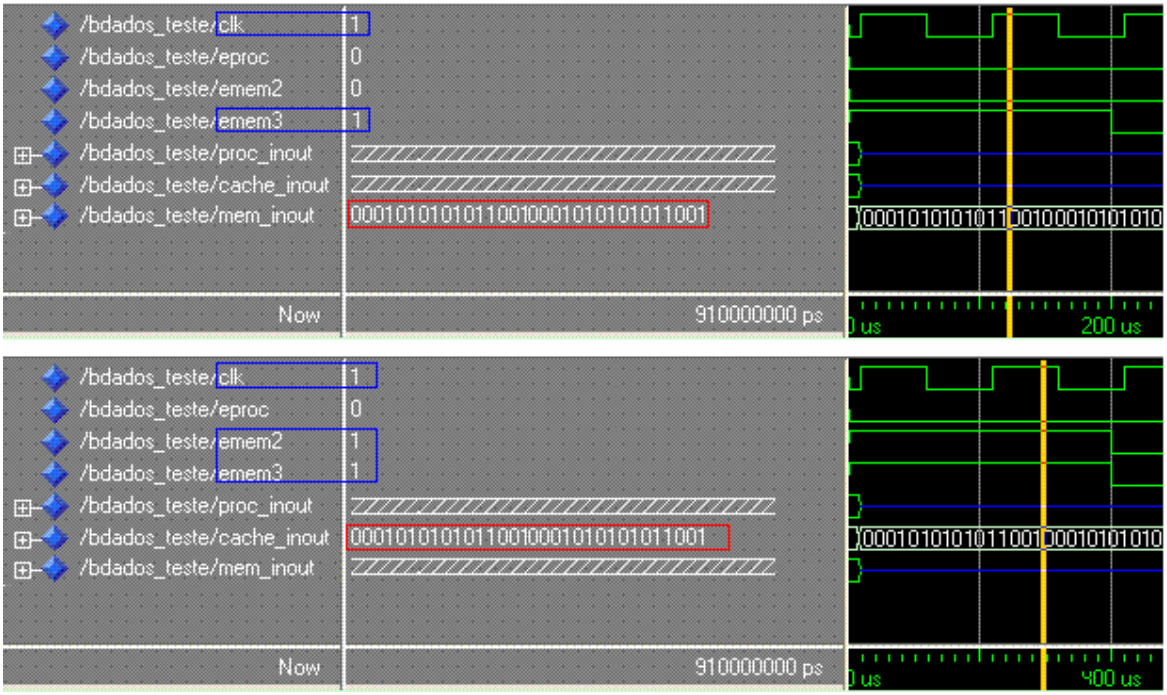

Figura 24. Simulação do fluxo da memória principal e *buffer* de dados.

Os quadrados vermelhos na Figura 24 representam o dado que é manipulado no fluxo entre a memória principal e a cache.

Na parte inferior da Figura 24, está o resultado da simulação do envio da informação armazenada no *buffer* para a cache. Como pode ser visto envolto pelos quadrados azuis, além dos sinais 'clk' e 'emem3' estarem setados para '1', o sinal 'emem2' esta em nível lógico alto, indicando que o *buffer* vai trabalhar com uma informação de saída, ou seja, o dado armazenado será enviado para o *array* de dados da cache. É importante observar que as outras duas portas de entrada/saída ficam em alto impedância durante o fluxo de envio do *buffer* para o *array*. A implementação em VHDL deste componente pode ser visto no anexo 1.d.

4.3.5 Módulo *Array* de Dados

Este módulo é responsável por armazenar dados. Quando a memória principal envia um bloco de dados para ser substituído na cache, ele é armazenado neste *array*. O mesmo acontece quando o processador termina uma operação e deseja atualizar uma nova informação na cache.

Este módulo é composto por cinco portas de entradas e uma porta de entrada/saída. Das cinco entradas quatro são sinais de controle do componente, são eles: o *clock* do sistema (clk), o sinal que habilita a escrita (ws), o sinal que habilita a leitura (rs) e o sinal que habilita o uso da matriz de dados ('habilita'). A outra entrada deste módulo é o endereço que referencia a linha da cache, este endereço é o sinal de treze bits (devido ao endereçamento da cache de 32 kbytes) enviado pelo decodificador.

A porta de entrada/saída (identificado como 'dados') é um pino que funciona tanto como entrada como saída do módulo. Em determinados momentos esta porta pode estar funcionando como entrada, ou seja, recebendo dados da memória principal ou do processador (no caso de atualização da cache) e em outros momentos funcionando como saída do módulo fornecendo dados e instruções ao processador (no caso de busca na cache).

Na Figura 25 pode se ver a simulação de uma escrita na cache. Quando se deseja utilizar o *array* de dados, o *clock* do sistema e o sinal 'habilita' devem estar em nível lógico '1' (ver os quadrados em azul na Figura 25), do contrário o componente fica desabilitado. Dois sinais de entrada decidem qual a operação está sendo executada na matriz (ver quadrado verde). Na simulação da Figura 25 o sinal 'ws' (*write signal*) está setado para '1' e 'rs' (*read signal*) está setado para '0' indicando uma escrita no *array* de dado.

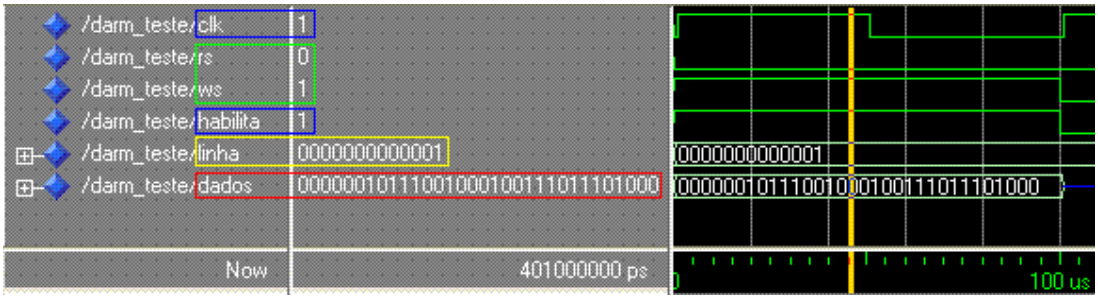

Figura 25. Simulação de uma escrita no *array* de dados.

O quadrado amarelo na figura 25 indica em qual posição do banco vai ser armazenado o novo dado (neste exemplo a linha um). A última linha da simulação (envolto pela quadrado vermelho) representa a informação propriamente dita. Na simulação da Figura 25, a porta 'dados' de entrada/saída esta funcionando como entrada do módulo. Na próxima simulação será mostrado um exemplo em que esta mesma porta funciona como porta de saída.

A Figura 26, ilustra a simulação da busca de uma informação na cache. Os sinais de ‘clk’ e ‘habilita’ permanecem os mesmos (nível lógico alto), o que modifica nesta simulação são os sinais de escrita e leitura. Observe na Figura 26 que o sinal ‘rs’ está agora setado para ‘1’, enquanto o sinal ‘ws’ está setado para ‘0’. Os sinais de escrita e leitura nunca vão estar no mesmo nível lógico ao mesmo tempo, por exemplo, em nível alto ou os dois em nível baixo, se isto acontecer (situação hipotética) nada vai acontecer no *array* de dados.

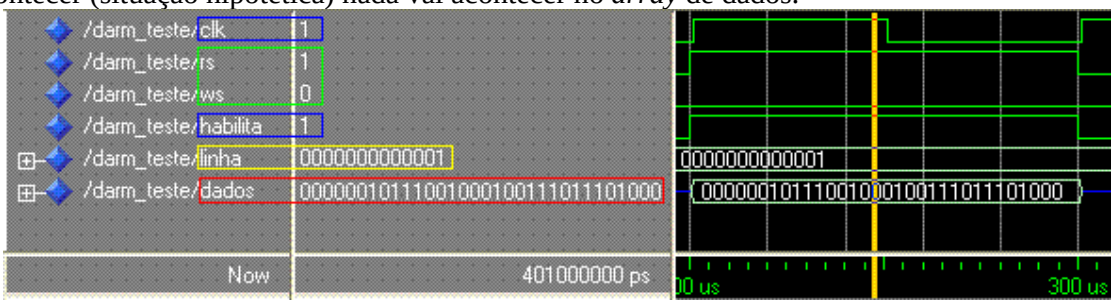

Figura 26. Simulação de uma leitura no *array* de dados.

Na simulação da Figura 26, a informação retornada pela busca é a mesma armazenada anteriormente na simulação da Figura 25, ou seja, na linha um do *array* de dados. Na simulação da Figura 26 a porta ‘dados’ de entrada/saída funciona como sinal de saída e fornece o dado desejado. A implementação em VHDL deste componente pode ser visto no anexo 1.e.

4.3.6 Módulo *Array* de Rótulos

Este módulo é responsável por armazenar os rótulos identificadores das linhas da cache. Quando uma nova instrução ou dado é armazenado na cache, essa nova informação tem que ter um identificador para que o controlador da cache saiba que linha da memória principal está sendo mapeada, pois, diferentes linhas da memória principal podem ser mapeadas na mesma linha da cache. Este identificador é armazenado no *array* de rótulos.

Este módulo é composto por seis sinais de entrada e apenas um de saída. Dos seis sinais de entrada, quatro são *bits* de controle do *array* que são: o *clock* do sistema, o sinal que habilita o *array*, os sinais de escrita (ws) e leitura (rs) que definem a operação no *array*, o sinal (linha) que referencia a linha que será utilizada na operação e um sinal (novo) que é utilizado para receber novas informações. Este sinal ‘novo’ é uma entrada que é utilizada quando a cache tem a necessidade de substituir um bloco da cache por um novo da memória principal, os rótulos que identificam os blocos são recebidos por esta porta.

O módulo comparador recebe dois rótulos para comparação, um deles está presente no endereço enviado pelo processador, o outro rótulo é justamente a única saída (dados) deste *array* de rótulos. Na Figura 27 pode ser visto uma simulação em que um rótulo que está armazenado na cache está sendo fornecido para comparação.

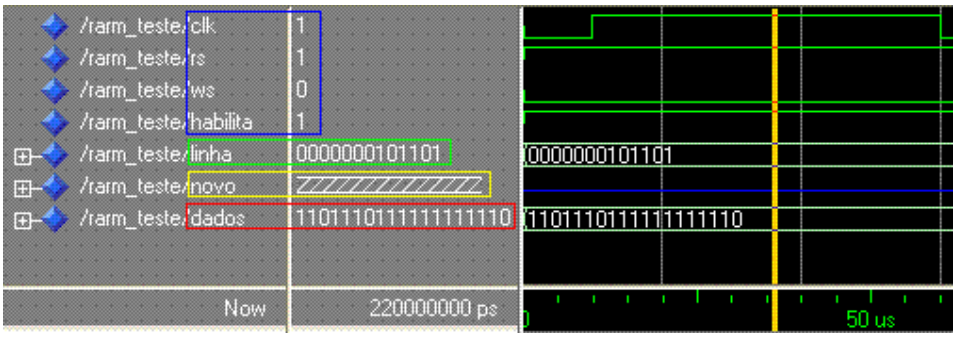

Figura 27. Simulação de saída de um rótulo.

Observe na simulação (Figura 27) que os sinais ‘clk’, ‘habilita’ e ‘rs’ estão em nível lógico alto, indicando que o *array* de rótulos esta habilitado para leitura, ou seja, um rótulo que esta armazenado neste banco será colocado na saída. O sinal ‘linha’ (envolta pelo quadrado verde) representa o endereço que seleciona a linha que contém o rótulo a ser comparado. O sinal ‘dados’ (envolto pelo quadrado vermelho) contém o rótulo selecionado pelo endereço que veio do decodificador. A penúltima linha da simulação, quadrado amarelo, seria um novo rótulo a ser armazenado no *array*, mas como a operação é de busca, este sinal (identificado como ‘novo’) fica em um estado desconhecido.

Na próxima simulação será mostrado o resultado de uma escrita de um novo rótulo dentro do *array*. Observe que os sinais de escrita e leitura estão diferentes da simulação anterior, indicando que a operação agora é uma escrita no *array* de rótulos.

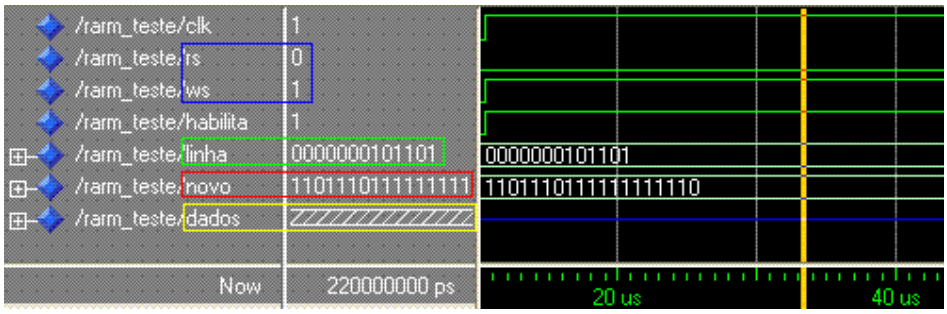

Figura 28. Simulação de uma escrita no *array* de rótulos.

Como a simulação agora é de escrita de um novo rótulo, o sinal ‘novo’ (envolto pelo quadrado vermelho na Figura 28) recebe o novo identificador do bloco que vai ser armazenado na cache e o sinal ‘dados’ (quadrado amarelo) permanece em alta impedância. A implementação em VHDL deste componente pode ser visto no anexo 1.f.

4.3.7 Módulo Controlador

O controlador da cache é o componente mais complexo da implementação, ele é responsável por todo o gerenciamento dos sinais que envolvem a cache. Este módulo é composto por sete sinais de entrada e doze sinais de saída, todos eles de um *bit*. O controlador faz todo o tratamento destes sinais através de uma máquina de estado.

A máquina de estado implementada dentro do controlador possui oito estados possíveis que são descritos em detalhes na seção 4.2 (detalhes na Figura 14 e 15). Para compreender a simulação do controlador é importante conhecer os sinais envolvidos na simulação.

Os sinais envolvidos pelo quadrado azul (ver Figura 29) são os sinais de controle do sistema, que são: o *clock* e o ‘*reset*’, este responsável por iniciar o sistema (zerar variáveis, apagar os *buffers* e inicializar a máquina de estados). Os três sinais envolvidos pelo quadrado (‘en_in’, ‘ws_in’ e ‘rs_in’) verde são sinais oriundos do processador que servem para habilitar e informar qual a operação vai ser executa na cache. O sinal ‘comp’ (envolvido pelo quadrado amarelo) é o sinal de entrada que informa ao controlador o resultado da comparação entre os rótulos. O ultimo sinal de entrada, o ‘*ready_in*’, é um sinal que vem da memória principal informando que o dado buscado pela cache (em uma situação de *cache-miss*) já está disponível no *buffer* de dados.

O sinal ‘*ready_out*’ é enviado da cache para o processador informa que a operação foi concluída na cache. O sinal de saída ‘sep’ é enviado para habilitar o decodificador, de modo que este decodifique o endereço recebido pelo processador.

Os quatro sinais envolvidos pelo quadrado laranja (‘eproc’, ‘emem1’, ‘emem2’ e ‘emem3’) são sinais que habilitam os *buffers* implementados na arquitetura. O ‘eproc’ habilita o *buffer* de dados para que este envie a palavra armazenada (pelo processador no estado anterior) para o *array* de dados da cache. O sinal ‘emem1’ habilita o *buffer* de endereço fazendo que o endereço armazenado seja propagado para memória principal. O sinal ‘emem2’ habilita o *buffer* de dados de modo que o *buffer* envie o dado armazenado para a memória principal, ele é utilizado quando há a necessidade de atualizar a memória principal. E o sinal ‘emem3’ habilita o *buffer* de dados de modo que o *buffer* envie a informação armazenada para o *array* de dados da cache.

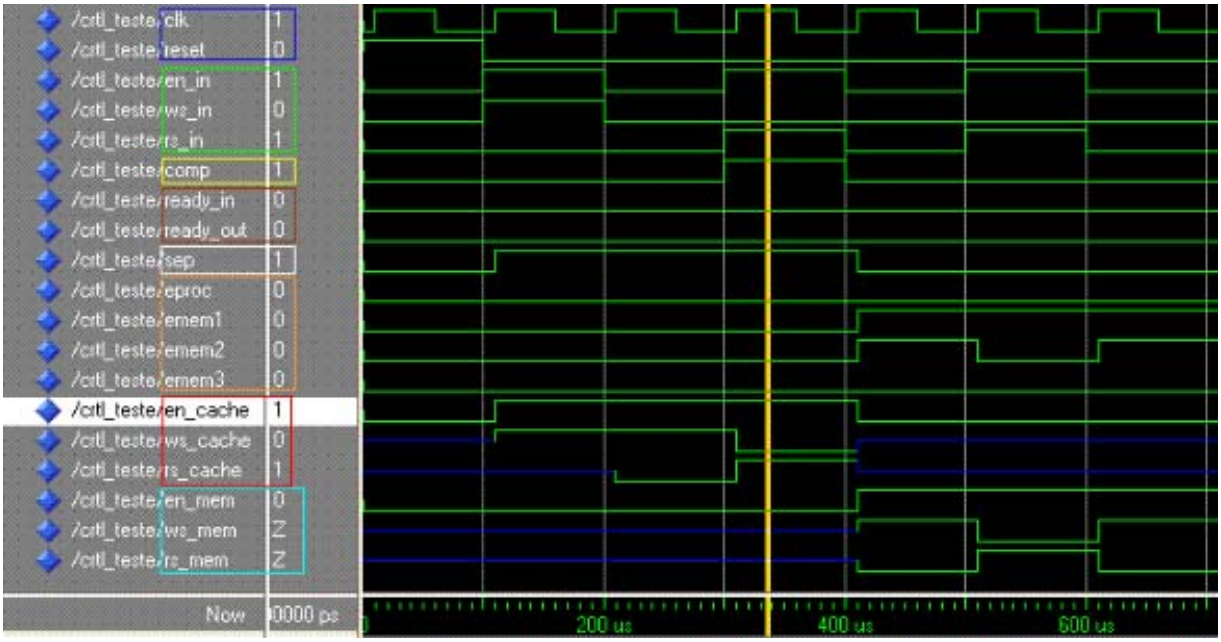

Figura 29. Simulação de uma leitura com sucesso na cache.

Os três sinais envolvidos pelo quadrado vermelho (‘en_cache’, ‘ws_cache’ e ‘rs_cache’) são os sinais que habilitam os *arrays* de dados e de rótulos da cache. Quando uma operação vai ser executada na cache, o controlador sempre envia estes sinais para os bancos de *arrays* sinalizando se estes devem estar ativos e como eles devem se comportar (se leitura ou escrita).

Os últimos três sinais da simulação (‘en_mem’, ‘ws_mem’ e ‘rs_mem’), envoltos pelo quadrado azul claro, são sinais que são enviados do controlador para a memória principal. Estes sinais somente são utilizados quando a cache pretende fazer uma comunicação com a memória principal, seja para atualizar a memória ou para buscar um novo bloco.

Na Figura 29, está representado o resultado da simulação de uma leitura na cache com sucesso (*cache-hit*). O *clock* do sistema está em nível lógico alto e o ‘*reset*’ em nível baixo; os sinais que vem do processador indicam uma leitura na cache (en_in = ‘1’ e rs = ‘1’); o resultado da comparação entre os rótulos foi verdadeira (comp = ‘1’), ou seja, o dado esta na cache; o controlador sinaliza para os bancos de *arrays* o que deve ser feito (uma leitura, en_cache = ‘1’ e rs = ‘1’) e sinaliza também para memória informando que naquele determinado momento nenhuma comunicação será feita com a memória principal (en_mem = ‘0’). A implementação em VHDL deste componente pode ser visto no anexo 1.g.

4.4 Estrutura da arquitetura

Um dos objetivos desta monografia é desenvolver uma estrutura básica de uma memória cache com esquema de escrita mapeado diretamente, política de substituição *write-through* e tamanho da palavra fixada em 32 bits. A abordagem implementada foi especificada com base em arquiteturas de memórias caches reconfiguráveis, onde a memória cache ou partes dela pode ser reconfigurada em tempo de execução. Dispositivos como FPGAs são fortemente recomendados para este tipo de abordagem.

Na estrutura interna das cinco configurações da cache (16kb, 32kb, 64kb, 128kb e 256kb) há um detalhe muito importante em relação a reconfigurabilidade. Todas as implementações apresentam uma parte fixa na arquitetura, ou seja, que é inerente a cada um dos cinco modelos, e existe também, uma parte que pode ser reconfigurada (na arquitetura) dentro FPGA.

Os elementos que fazem parte do conjunto reconfigurável são os componentes que variam consideravelmente quanto a sua estrutura física. A Figura 30 mostra um exemplo geral das implementações da cache dentro de um FGPA. No lado esquerdo estão os *arrays* de dados e rótulos que crescem de tamanho de acordo com a variação das quantidades de linhas da memória cache e são eles que aumentam a região ocupada dentro do FGPA. No lado direito da figura estão os componentes que tem a implementação fixa e que não se alteram entre os projetos, ou seja, na cache 32 *kbytes* a quantidade de unidades lógicas utilizadas para implementar o controlador, comparador, decodificador e *buffers*, é a mesma de uma cache de 256 *kbytes*.

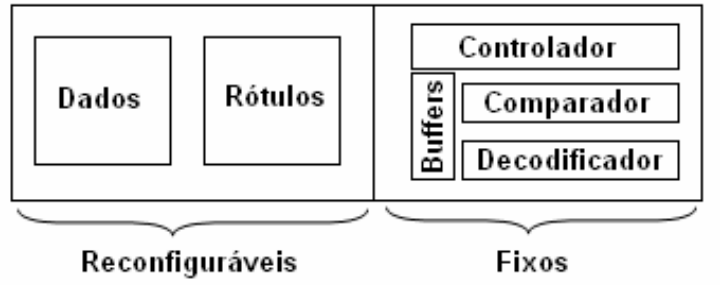

Figura 30. Elementos reconfiguráveis da arquitetura de cache.

Entre os componentes da parte fixa na verdade existem dois componentes que sofrem pequenas mudanças no código: o decodificador e comparador. O decodificador recebe sempre o mesmo endereço de 32 *bits*, mas a quantidade de *bits* (saídas do módulo) que são designados para endereçar a linha e o rótulo (para comparação) são diferentes de implementação para implementação (já explicado nos detalhes do projeto na subseção 4.2.1). Já o módulo comparador tem sempre a mesma saída, que é um *bit* que sinaliza o resultado da comparação, mas em compensação tem sempre as entradas diferenciadas entre os projetos, pois os tamanhos das palavras (rótulos) comparadas dependem de qual organização de cache está utilizada (ver subseção 2.4.1).

As variações ocorridas por esses componentes são tão pequenas que não causam alterações na quantidade de blocos lógicos que os implementam no FPGA. No capítulo de estudo de casos serão mostrados dados em tabelas que comprovam esta afirmação.

A compreensão desta seção é fundamental para entender o próximo capítulo de estudo de casos, pois todas as análises feitas e os resultados encontrados são todos baseados nesta idéia de componentes fixos e passíveis de reconfiguração.

Capítulo 5

Estudo de Casos e Resultados

Nos capítulos anteriores foram apresentados os conceitos básicos necessários para compreensão deste trabalho, e os detalhes sobre a arquitetura de cache desenvolvida. Neste ponto do trabalho espera-se que a utilização da ferramenta *ISE Foundation* disponibilize informações relevantes para analise de impacto de área em arquiteturas reconfiguráveis.

A apresentação deste capítulo visa mostrar os resultados do mapeamento da arquitetura sobre determinadas famílias de FPGA.

5.1 Simulação e mapeamento dos módulos no FPGA

Nesta etapa dos estudos preocupou-se em analisar os efeitos da variação da quantidade de linhas da memória cache em termos de área ocupada num dispositivo reconfigurável (FPGA) para vários dispositivos da família Virtex II da Xilinx.

A ferramenta utilizada para validar a cache implementada foi o *ISE Foundation*, da Xilinx. É uma ferramenta que apesar de apresentar uma certa dificuldade de manuseio, ela dispõe de muitas funcionalidades e dá suporte a todas as fases do projeto de *hardware*. Além disso, permiti desenvolvimento de projetos, simulações, geração gráfica dos componentes automaticamente, ele disponibiliza também para o usuário uma série de relatórios sobre a implementação, tais como: relatório de mapeamento, relatório de pinagem do FPGA, relatório sobre os atrasos das portas lógicas, relatório sobre o roteamento adotado pela arquitetura, etc.

Após executar todas as simulações necessárias, o próximo passo foi gerar os relatórios individuais da área ocupada por cada componente, e depois da integração, gerar um relatório final do mapeamento da arquitetura dentro de um FGPA.

5.2 Análise da implementação no FPGA Virtex II

Nesta seção será mostrado o resultado da implementação da cache num FGPA e esclarecer algumas informações retiradas durante a análise do projeto. Para demonstrar como as análises foram organizadas e para otimizar os resultados das simulações, será mostrada apenas a informação de uma cache. O modelo escolhido foi a cache de 256*kbytes*, pois as simulações e

mapeamentos com este tamanho de cache, apresentou todas as situações necessárias para este estudo de caso.

A tabela 4 exibe área ocupada, em termos percentuais, pelos componentes da cache de 256kb, a família de FGPA usada para mapear estes módulos foi a da família Virtex 2. Dentre algumas vantagens de escolha de tal família de FPGA estão: preços ainda competitivos no mercado quando comparados com a família Virtex 4, assim como o potencial de reconfigurabilidade parcial suportado pelos FPGA's da Xilinx, essencial para projetos envolvendo caches reconfiguráveis.

Na primeira coluna da tabela pode ser visto os componentes que fazem parte da arquitetura implementada, na coluna seguinte está a quantidade de LUTs *(LookUp Table*) que foram necessários para comportar o determinado componente, por exemplo, para implementar a memória de rótulos foram precisos 4.313 LUTs.

Tabela 4. Área percentual dos componentes individuais da cache de 256kb.

Componentes	Nº de LUTS	xc2v500	xc2v1000	xc2v1500	xc2v2000	xc2v3000	xc2v4000	xc2v6000
Mem. de Dados	7961	129% (overmap)	77%	51%	37%	27%	17%	11%
Mem. de Rótulos	4313	70%	42%	28%	20%	15%	9%	6%
Buffer de Dados	71	1%	1%	1%	1%	1%	1%	1%
Buffer de Endereço	0	0%	0%	0%	0%	0%	0%	0%
Comparador	10	1%	1%	1%	1%	1%	1%	1%
Controlador	49	1%	1%	1%	1%	1%	1%	1%
Decodificador	0	0%	0%	0%	0%	0%	0%	0%
Total	12404	201% (overmap)	121% (overmap)	81%	57%	43%	26%	18%

As demais colunas da tabela indicam a área ocupada pelos componentes em valores percentuais, com base nos diferentes dispositivos (*device*) existentes na família Virtex II. Um FPGA pode ter muitos dispositivos, o que muda de um dispositivo para outro (em uma mesma família) é a quantidade de blocos lógicos que compõem o FPGA e as operações internas de cada bloco lógico. O FPGA da Virtex II, por exemplo, pode implementar nove operações lógicas ('e', 'ou', 'ou exclusivo', 'não e', etc.) dentro de cada bloco lógico do FPGA.

Atrelado a este gráfico outra informação é necessária para decidir sobre qual dispositivo FPGA deve ser usado dado uma configuração de cache. A Tabela 5 mostra a quantidade máxima de LUTs permitido para cada dispositivo. Após toda a implementação e mapeamento feito pelo *ISE Foundation*, é possível saber, através dos relatórios da ferramenta, a quantidade de unidades lógicas que foram necessárias para comportar a arquitetura. Através desta quantidade de LUTs que é determinado, com o auxílio da Tabela 5, qual o dispositivo ideal para a implementação.

Tabela 5. Quantidade de LUTs por dispositivo da família Virtex II da Xilinx.

Device	LUTs
xc2v500	6144
xc2v1000	10240
xc2v1500	15360
xc2v2000	21504
xc2v3000	28672
xc2v4000	46080
xc2v6000	67584

Na tabela 4, pode se ver que alguns componentes não afetam muito no mapeamento da arquitetura a exemplo do comparador, *buffer* de dados e controlador, que ocupam no máximo 1%

(cada um) no FGPA com qualquer um dos dispositivos da Virtex II. As células em laranja mostram que os módulos de *buffer* de endereço e decodificador são tão pequenos que não ocupam nenhuma LUT, porque dependendo da ação que o componente executa e o tamanho, quando eles vão ser mapeados no FPGA, eles podem ser implementados no circuito interno de um pino 'A', por exemplo, e ser jogado diretamente na saída de um pino 'B', deste modo, eles não ocupariam nenhuma LUT do FPGA.

Visualizando as células em amarelo (ver tabela 4) pode-se extrair duas informações importantes. A primeira, é que a implementação da memória de dados excedeu (*overmap* de 129%) a quantidade de LUTs do FPGA com os dispositivos "xc2v500", isto conclui que este dispositivo não suporta esta implementação da cache e devem ser trocados por outros.

O outro detalhe é que a memória de dados ocupa 77% do FPGA usando o dispositivo "xc2v1000", esta arquitetura reconfigurável poderia ser considerada ideal se ela somente implementasse a memória de dados, mas como a cache integrada contém outros módulos, e a arquitetura é composta por outros componentes como processador e memória principal, este FPGA com certeza excederia o tamanho do dispositivo e não comportaria a implementação final, obrigando uma redefinição do dispositivo do FPGA.

A implementação de cada componente da cache de 256kb causou o uso de 12404 unidades lógicas (LUTs). Com o auxílio de Tabela 5, pode-se verificar que o dispositivo "xc2v1500" seria suficiente para comportar a cache.

5.3 Resultados

Como foi visto anteriormente, foram geradas cinco implementações de cache variando somente os *arrays* de armazenamento e mantendo os outros componentes fixos. Cada uma das implementações de cache gerou um conjunto de informações que foram organizadas em tabelas separadamente, como pode ser visto no exemplo da seção 5.2.

Depois de coletar a variação de espaço dentro do FPGA para cada um dos cinco tamanhos de cache, foi gerada uma tabela reunindo todas as informações relevantes. A Tabela 6 mostra relação entre o tamanho da cache e quantidade de unidades lógicas (LUTs) que foram necessárias para implementá-las, levando em consideração a arquitetura final da cache com os componentes, as matrizes de roteamento e os circuitos de interfaceamento.

É possível verificar ainda na tabela, a área ocupada (em valores percentuais) em cada uma das implementações da cache. Pode-se observar que as caches de 128 e 256 kb ocuparam 100% do FPGA (*overmap*) em alguns dispositivos da família Virtex II (células amarelas na tabela), fazendo com que estes dispositivos não fossem suficientes para acomodar tais implementações.

Os dispositivos "xc2v4000" e "xc2v6000" têm uma quantidade de blocos lógicos muito grande, fazendo com que todas as diferentes implementações de cache (desta monografia) ocupassem menos de 30% destes dispositivos. Se o escopo desta monografia estivesse analisando uma arquitetura com outros componentes, por exemplo, microprocessador, memória principal, displays, com certeza os melhores (ideais) dispositivos da família Virtex II seriam os *devices*: "xc2v1500", "xc2v2000" e "xc2v3000".

Tabela 6. Quantidade de LUTs por tamanho de cache no FPGA da família Virtex II.

CACHE	Nº de LUTs	xc2v500	xc2v1000	xc2v1500	xc2v2000	xc2v3000	xc2v4000	xc2v6000
16kb	741	12%	7%	5%	3%	2%	1%	1%
32kb	1545	25%	15%	10%	7%	5%	3%	2%
64kb	3084	50%	30%	21%	14%	10%	6%	4%
128kb	6184	101%	60%	40%	28%	21%	13%	9%
256kb	12404	201%	121%	81%	57%	43%	26%	18%

A partir da Tabela 6, foi possível representar graficamente a relação entre o tamanho de cache e a quantidade de LUTs utilizadas em cada implementação. O gráfico ficou simples, pois se for selecionado um mesmo tamanho de cache, a quantidade de LUTs necessária para implementá-la é a mesma, independente do dispositivo para uma mesma família de FPGA.

Esse gráfico é importante para visualizar os pontos que serão usados para encontrar a equação que determinará qual o FPGA ideal para um projeto de hardware desenvolvido por um projetista. A Figura 31 mostra em uma escala real a variação linear da quantidade de LUTs para caches de tamanhos 16k, 32k, 64k, 128k e 256kbytes (*Este gráfico foi traçado usando os valores das duas primeiras colunas da Tabela 6*).

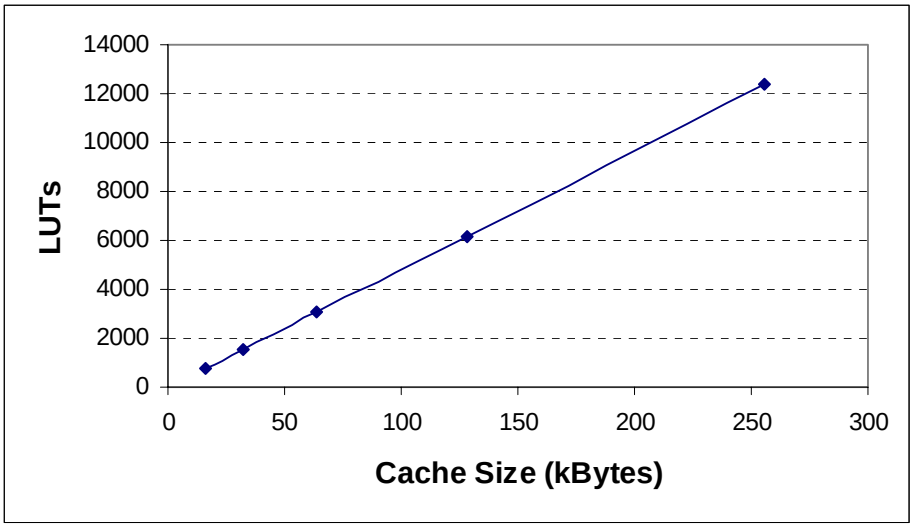

Figura 31. Relação tamanho da cache x LUTs.

Visando obter uma equação que representasse a área do FPGA em termos de LUTs, foi realizada uma regressão linear, com o auxilio da ferramenta Matlab, sobre os pontos obtidos na Figura 31 e obteve-se como resultado a seguinte equação linear:

$$QLUT = 48{,}53 * SIZE - 22{,}08 \qquad \text{(}\textit{Equação 1}\text{)}$$

onde a variável "*SIZE*" é o tamanho da cache em kbytes e "QLUT" é a quantidade de LUTs necessária para implementar um dado tamanho de cache.

A equação acima (equação 1), ajuda o projetista a determinar a quantidade de unidades lógicas que são necessárias para uma implementação de cache, dentro de uma determinada arquitetura de hardware.

Em uma implementação real, outros componentes devem ser considerados tais como processador e memória principal. Com isso, para exemplificar a abordagem proposta, considere um sistema composto de processador Leon2, memória cache e memória principal.

Nessa abordagem, consideramos que o FPGA está dividido em duas partes: uma parte fixa (não reconfigurável) e uma parte reconfigurável como ilustrado na Figura 32. A parte fixa é composta de alguns componentes fixos da memória cache (controlador, decodificador, comparador e buffers), memória principal, processador e módulo ICAP, perfazendo um total de aproximadamente 5000 LUTs. Na parte reconfigurável encontra-se apenas o *array* de dados e *tag* da cache que varia de acordo com sua configuração.

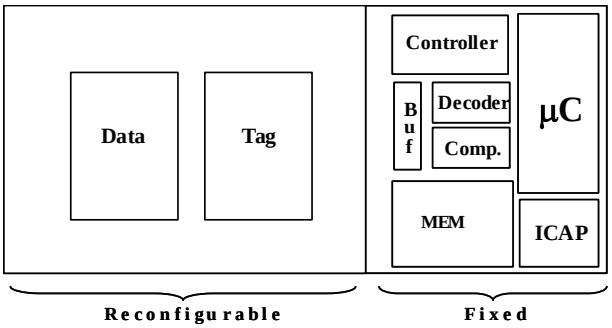

Figura 32. Visão dos componentes dentro de um FPGA.

Nesta abordagem considerou-se o ICAP (*Internal Configuration Access Port*) como recurso de auxílio a reconfiguração interna do FPGA que pode ser instanciado e está disponível para ser usado como auxilio a reconfiguração parcial do dispositivo.

A principal vantagem da utilização do ICAP para aplicações envolvendo caches reconfiguráveis é o poder de auto-reconfiguração do FPGA, possibilitando, por exemplo, que a própria aplicação se ajuste a uma nova configuração de cache.

Exemplo: Considere que um projetista queira saber qual FPGA deve ser usado para comportar a arquitetura proposta acima, utilizando uma cache de instrução (CI) de 32kbytes e cache de dados (CD) de 128kbytes.

$$QLUT(CI) = 48,53 * (32) - 22,08 = 1530,88$$
$$QLUT(CD) = 48,53 * (128) - 22,08 = 6189,76$$

$$TOTAL = (5000\ LUTs)_{fixa} + 1530,88 + 6189,76$$
$$TOTAL = 12720,64\ LUTs$$

Neste caso, obtivemos um total de 7665,16 LUTs necessárias para implementação da cache. Esta quantidade acrescida de 5000 LUTs da parte fixa resulta em um total de 12665,16 LUTs. Com o auxílio Tabela 5, verificamos que o dispositivo “xc2v1500” se enquadra nas condições de tal aplicação. Possivelmente dispositivos que contém quantidades máximas de LUTs acima do escolhido acarretaria um aumento relevante no custo do projeto.

A Equação 1 ajuda ao projetista, antes da etapa de implementação, decidir qual dispositivo FPGA da família Virtex II deve ser usado no desenvolvimento de um projeto contendo hierarquia de memória cache e processador, a partir do tamanho da cache em kbytes.

Capítulo 6

Conclusão e Trabalhos Futuros

6.1 Conclusão

Com o avanço da tecnologia foi possível desenvolver novas soluções em hardware de alto desempenho, mas essas novas arquiteturas agregaram uma maior complexidade computacional. Ficou explícito que as empresas de hardware precisavam desenvolver componentes mais sofisticados para ganhar espaço no mercado.

Essa necessidade obrigou que as empresas investissem mais na fase de projeto com o intuito de criar metodologias, padrões, ferramentas, que conseguissem responder mais rapidamente as necessidades de desenvolvimento, ou seja, diminuir o *gap* existente entre a concepção da idéia e projeto final.

Tendo em vista essa necessidade, objetivou-se com esse trabalho criar um método que auxiliasse na escolha do FPGA a ser usado num determinado projeto.

Utilizando a ferramenta *ISE Foundation*, foi implementado, em VHDL, uma memória cache simples e o seu funcionamento validado através de simulação. Foi apresentada uma nova equação que permite mensurar a área ocupada por uma memória cache em termos de LUTs. Esta monografia possibilita demonstrar que a família Virtex II, da Xilinx, consegue prover satisfatoriamente, em termos de espaço ainda disponível, sistemas de caches reais. Considerando um SoC composto de processador e hierarquia de memória, e de posse da área ocupada pelo processador, é possível através da abordagem proposta, escolher o dispositivo da família Virtex II adequado para determinada aplicação. Outras relações podem ser obtidas considerando outras famílias de FPGAs que suportem reconfiguração parcial.

6.2 Trabalhos futuros

Os resultados alcançados com a conclusão deste trabalho proporcionaram a conquista dos objetivos iniciais. Este trabalho pode ser o começo de muitos outros projetos, dos quais alguns serão listados abaixo.

Este projeto não cobre a sincronização dos componentes, apesar de todos os componentes funcionarem com um *clock* interno, mas se um dos componentes atrasar a saída a cache precisará

de mais ciclos de máquinas para concluir uma única operação, penalizando a execução do processador (problema com ociosidade e *timeout*). Um projeto vislumbrado seria o de temporizar a cache, para que os atrasos dos componentes se encaixassem no *clock* do sistema perfeitamente. Esta implementação temporizada seria útil para fazer estimativas de desempenho da cache (cycle accurate), pois desta forma poderia se integrar a cache com os *cores* do processador e da memória principal e fazer as análises necessárias.

Um outro projeto mais complexo seria construir uma cache totalmente parametrizável, para que o usuário (ou projetista de *hardware*) tivesse a opção de escolher seu modelo de cache com o tamanho, mapeamento e algoritmo de substituição definido por ele mesmo. Este projeto proporcionaria um resultado mais preciso na hora do mapeamento da cache em FGPA, pois o trabalho deste autor analisou os resultados somente para tamanhos de cache diferentes. Com a sintetização, não só do tamanho, mas também do mapeamento e o algoritmo de substituição, poder-se-ia obter resultados mais expressivos na hora do mapeamento no FGPA.

Um estudo que poderia ser feito juntamente com o projeto da cache parametrizável e análise de desempenho é o da dissipação de potência gerada pela cache. Em uma arquitetura de hardware com processador, co-processador, hierarquia de memórias e outros pequenos componentes; a cache é responsável em media por 40% do consumo total da placa [7]. É interessante fazer um estudo sobre a dissipação de energia para diferentes implementações da cache. Quando o usuário definisse o modelo da cache e fosse analisar o desempenho do modelo, uma estimativa de dissipação de energia poderia ser gerada, para que o usuário possa ter idéia do consumo do seu modelo.

Complementando as idéias anteriores, outro trabalho bastante interessante seria comparar os resultados de uma aplicação sendo executada em FPGAs com os resultados de uma abordagem em um nível maior de abstração em termos de desempenho e energia consumida, como linguagens de descrição de arquitetura (ADL). A idéia inicial seria tentar refinar o modelo de alto nível visando melhorar sua precisão.

Desta forma, concluo este trabalho de monografia.

Bibliografia

[1] ENGINEERING AT PENN. ENIAC Museum Online. Disponível em: <http://www.seas.upenn.edu/~museum/>. Acesso em: 23 de Janeiro de 2005.

[2] ERCEGOVAC, M., LANG, T. e MORENO, J. H.. *Introdução a Sistemas Digitais*, 1. ed. São Paulo. Bookman, 2001.

[3] *FPGA, CPLD & Structured ASIC Devices; Altera, the Leader in Programmable Logic*, Disponível em: <http://www.altera.com/support/software/sof-maxplus2.html>. Acesso em: 07 de Dezembro de 2004.

[4] GRUN, P., HALAMBI, A., KHARE, A., GANESH, V., DUTT, N. e NICOLAU, N. *EXPRESSION - An ADL for system level design exploration*. Technical Report TR 98-29, University Of California, Irvine, 1998.

[5] HOFFMANN, A., KOGEL, T. e NOHL, A. *A Novel Methodology for the Design of Application Specific Instruction Set Processors* (ASIP) Using a Machine Description Language. IEEE Transactions on Computer-Aided Design, 20(11), 2001.

[6] *Intel Corporation*, Disponível em: <http://www.intel.com/performance/index.htm>. Acesso em: 03 de Maio de 2005.

[7] MAMIDIPAKA, M. e DUTT, N.. *eCACTI: An Enhanced Power Estimation Model for On-chip Caches.* Disponível em: <http://www.cecs.uci.edu/technical_report/TR04-28.pdf>. Acesso em: 15 de Fevereiro de 2005.

[8] MARTINS, C.A.P.S., ORDONEZ, E.D.M. e CARVALHO, M.B. *Computação Reconfigurável: conceitos, tendências e aplicações*, Disponível em: <http://ftp.inf.pucpcaldas.br/CDs/SBC2003/pdf/arq0251.pdf>. Acesso em: 07 de dezembro de 2004.

[9] MENDONÇA, A. e ZELENOVSKY. *Monte seu Protótipo ISA Controlado por FPGA*. Rio de Janeiro. MZ Editora, 2001.

[10] *Mentor Graphics – ModelSim Designer Evaluation*. Disponível em: <http://www.model.com/modelsimdesigner/evaluation.asp>. Acesso em: 23 de maio de 2005.

[11] NIST – National Institute of Standards and Technology. *Finite State Machines*. Disponível em: <http://www.nist.gov/dads/HTML/finiteStateMachine.html>. Acesso em: 01 de novembro de 2005.

[12] NOVITSKY, J. AZIMI, M e GHAZNAVI, R. *Optimizing system performance based on Pentium processors*, Proceedings COMPCON, fevereiro, 1993.

[13] PATTERSON, D.A. e HENNESSY J.L. *Organização e projeto de computadores*. 2 ed. Trad. de Nery Machado Filho. Morgan Kauffmann Publishers, 2000.

[14] *Reduced Instruction Set Computer - RISC*, Disponível em: <http://en.wikipedia.org/wiki/RISC, Acesso em: 25 de outubro de 2005.

[15] STALLINGS, W. *Arquitetura e Organização de Computadores: Projeto para o Desempenho*. 5. ed. Trad. de Carlos Camarão de Figueiredo e rev. de Edson Toshimi Midorikawa. São Paulo: Prentice Hall, 2002.

[16] UNIVERSIDADE DE CAMPINAS. Laboratório de Sistemas Computacional. Projeto: *Linguagem para Descrição de Arquitetura ArchC*. Disponível em: <http://www.archc.org>. Acesso em: 08 de dezembro de 2004.

[17] VIANA, P., BARROS, E., RIGO, S., AZEVEDO, R. e ARAÚJO, G. *Modeling and Simulating Memory Hierarchies in a Platform-based Design Methodology*. Feveriro de 2004.

[18] *Xilinx: The Programmable Logic Company*, Disponível em: <http://www.xilinx.com/ise/logic_design_prod/foundation.htm>. Acesso em: 12 de Março de 2005.

[19] ZDRAVKO, K., POPOV, D., FOUTEKOVA, E., DELCHEV, I. e KRIVULEV, I.. *Cache Memory, Implementation and Design Techniques*. Disponivel em: <http://www.faculty.iu-bremen.de/birk/lectures/PC101-2003/07cache/cache%20memory.htm>. Acesso em: 31 de março de 2005.

Anexo 1

Implementações da cache de 32 kbytes em VHDL.

a. Módulo Decodificador

```
library ieee;
use ieee.std_logic_1164.all;

entity separador32k is
      port (
                        a               : in std_logic ;
                        endereco        : in std_logic_vector(31 downto 0);
                        linha           : out std_logic_vector(12 downto 0);
                        rotulo          : out std_logic_vector(18 downto 0)
          );
end separador32k;

architecture estrutura of separador32k is

begin
      process (a, endereco)
      begin
            if(a'event and a ='1') then
                  linha  <= endereco(12 downto 0);
                  rotulo <= endereco(31 downto 13);
            end if;
      end process;
end estrutura;
```

b. Módulo Comparador

```
library ieee;
use ieee.std_logic_1164.all;

entity comparador19b is
      port (
            rotulo_end : in std_logic_vector(18 downto 0);
            rotulo_arm : in std_logic_vector(18 downto 0);
            resultado  : out std_logic
          );
end comparador19b;

architecture estrutura of comparador19b is

begin
      process (rotulo_end, rotulo_arm)
      begin
            if (rotulo_end = rotulo_arm) then
```

```
                resultado <= '1';
        else
                resultado <= '0';
        end if;
    end process;
```

c. Módulo *Buffer* de Endereço

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bufferMEM IS
    PORT
    (
        clk, enable         : IN        STD_LOGIC;
        entrada             : BUFFER    STD_LOGIC_VECTOR(31 downto 0);
        saida               : BUFFER    STD_LOGIC_VECTOR(31 downto 0)
    );
END bufferMEM;

ARCHITECTURE a OF bufferMEM IS
BEGIN

    PROCESS (clk, enable, entrada)
    BEGIN
        IF(clk'event and clk='1') THEN
            IF enable = '0' THEN
                saida <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
            ELSE
                saida <= entrada;
            END IF;
        END IF;
    END PROCESS;
END a;
```

d. Módulo *Buffer* de Dados

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bufferDados IS
    PORT
    (
        clk, eproc          : IN        STD_LOGIC;
        emem2               : IN        STD_LOGIC;
        emem3               : IN        STD_LOGIC;
        proc_inout          : BUFFER    STD_LOGIC_VECTOR(31 downto 0);
        cache_inout         : BUFFER    STD_LOGIC_VECTOR(31 downto 0);
        mem_inout           : BUFFER    STD_LOGIC_VECTOR(31 downto 0)
    );
END bufferDados;

ARCHITECTURE a OF bufferDados IS
BEGIN
        process(clk, eproc, emem2, emem3)
        BEGIN
            if(clk'event and clk='1') THEN
                IF (eproc = '1') THEN
            -- leitura da cache
```

```
                        proc_inout  <= cache_inout;
                        mem_inout       <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                    END IF;

                    IF (emem2 = '1' and emem3 = '0') THEN
                -- a buffer envia um valor para a memoria (atualizacao)
                        mem_inout <= proc_inout;
                        cache_inout     <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                    END IF;

                    IF (emem3 = '1' and emem2 = '0') THEN
                -- a memoria envia um valor para o buffer (atualizacao)
                        cache_inout <= mem_inout;
                        proc_inout      <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                    END IF;
                -- escrita na cache
                    IF      (eproc = '1' and emem3 = '0' and emem2 = '0')       THEN
                        cache_inout <= proc_inout;
                        mem_inout       <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                    END IF;
                END IF;
    END PROCESS;
END a;
```

e. Módulo *Array* de Rótulos

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity arm_rotulos32k is
  port(
       clk, rs,ws, habilita          : in std_logic;
       linha                         : in std_logic_vector(12 downto 0);
       novo                          : in std_logic_vector(18 downto 0);
       dados                         : BUFFER std_logic_vector(18 downto 0)
    );
end arm_rotulos32k;

architecture estrutura of arm_rotulos32k is
-- criando um tipo de array
 type array_rotulos is array(natural range 8191 downto 0) of std_logic_vector(18 downto 0);
signal rotulo : array_rotulos;                    -- definindo um array para armazenar rotulos

begin
   process(clk, habilita,rs,ws)

      variable inadr : integer;

   begin
    if(clk'event and clk = '1') then
      inadr := to_integer(unsigned(linha));

      if (inadr>=rotulo'low) and (inadr<=rotulo'high) then        -- endereco esta fora do array
         if habilita = '1' then                                   -- habilita escrita ou leitura na cache
            if (rs = '1' and ws = '0')  then
                  dados <= rotulo(inadr) ;                        leitura na memoria
              end if;
```

```
            if (rs = '0' and ws = '1') then
                        rotulo(inadr) <= novo;                          -- escrita na memoria
            end if;
          else
            dados <= (others => 'Z');
          end if;
        else
          dados <= (others => 'Z');
        end if;
        end if;
    end process;

end estrutura;
```

f. Módulo *Array* de Dados

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity arm_dados32K is
  port(
        clk, rs, ws, habilita       : in std_logic;
        linha                       : in std_logic_vector(12 downto 0);
        dados                       : BUFFER std_logic_vector(31 downto 0)
      );
end arm_dados32K;

architecture estrutura of arm_dados32K is
-- criando um tipo de array
 type array_dados is array(natural range 8191 downto 0) of std_logic_vector(31 downto 0);      signal palavra :
array_dados;              -- definindo um array para armazenar rotulos

begin
  process(clk, habilita,rs,ws)
      variable inadr : integer;
   begin
     if(clk'event and clk = '1') then
       inadr := to_integer(unsigned(linha));

       if (inadr>=palavra'low) and (inadr<=palavra'high) then                  -- endereco esta fora do array
          if habilita = '1' then                                               -- habilita escrita ou leitura na cache
             if (rs = '1' and ws = '0') then
                -- leitura na memoria
                  dados <= palavra(inadr) ;
                            end if;

             if (rs = '0' and ws = '1') then
                        palavra(inadr) <= dados;                               -- escrita na memoria
             end if;
          else
             dados <= (others => 'Z');
          end if;
       else
          dados <= (others => 'Z');
       end if;
     end if;
   end process;
```

```
end estrutura;
```

g. Módulo Controlador

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--  library UNISIM;
--  use UNISIM.VComponents.all;

entity controlador is
   Port (   -- sinais do sistema
                clk, reset          : in std_logic;
     -- sinais de entrada
                en_in               : in std_logic;
                ws_in               : in std_logic;
               s_in                 : in std_logic;
                comp                : in std_logic;
                ready_in            : in std_logic;
   -- sinais de saida
                ready_out           : out std_logic;
                sep                 : out std_logic;
                eproc               : out std_logic;
                emem1               : out std_logic;
                emem2               : out std_logic;
                emem3               : out std_logic;
                en_cache            : out std_logic;
                ws_cache            : out std_logic;
                rs_cache            : out std_logic;
                en_mem              : out std_logic;
                ws_mem              : out std_logic;
                rs_mem              : out std_logic
      );
end controlador;

architecture Behavioral of controlador is

     type estados_leitura is (inicio, compara, atualiza, liberacao);
     type estados_escrita is (inicio, compara, atualiza_mem, finaliza);
     signal estado1: estados_leitura;
     signal estado2: estados_escrita;

begin
     process(clk, reset, en_in, ws_in, rs_in)                    -- processo de leitura na cache
     begin
               if(reset =  '1') then
                         ready_out <= '0';
                         sep <= '0';
                        eproc <= '0';
                        emem1 <= '0';
                        emem2 <= '0';
                        en_cache <= '0';
                        ws_cache <= 'Z';
                        rs_cache <= 'Z';
                        en_mem <= '0';
                        ws_mem <= 'Z';
```

```
            rs_mem <= 'Z';

             estado1 <= inicio;
             estado2 <= inicio;

elsif (clk'event and clk = '1') then
            if(rs_in = '1' and en_in ='1') then

             case estado1 is
                        when inicio =>
                                    if(rs_in = '1' and ws_in = '0') then
                                                ready_out <= '0';
                                                sep <= '1';
                                                en_cache <= '1';
                                                ws_cache <= '0';
                                                rs_cache<= '1';
                                                eproc <= '0';
                                                emem1 <= '0';
                                                emem2 <= '0';
                                                emem3 <= '0';
                                                en_mem <= '0';
                                                ws_mem <= 'Z';
                                                rs_mem <= 'Z';

                                                estado1 <= compara;
                                    end if;

                        when compara =>
                                    if(comp = '1')      then
                                                sep <= '0';
                                                en_cache <= '0';
                                                ws_cache <= 'Z';
                                                rs_cache <= 'Z';
                                                emem1 <= '0';
                                                emem2 <= '0';
                                                emem3 <= '0';
                                                en_mem <= '0';
                                                ws_mem <= 'Z';
                                                rs_mem <= 'Z';

                                                estado1 <= liberacao;
                                    else
                                                sep <= '0';
                                                en_cache <= '0';
                                                ws_cache <= 'Z';
                                                rs_cache <= 'Z';
                                                emem1 <= '1';
                                                emem2 <= '0';
                                                emem3 <= '0';
                                                en_mem <= '1';
                                                ws_mem <= '0';
                                                rs_mem <= '1';

                                                estado1 <= atualiza;
                                    end if;

                        when atualiza =>
                                    if(ready_in = '1')  then
                                                en_cache <= '1';
```

```
                                        ws_cache <= '1';
                                        rs_cache<= '0';
                                        emem1 <= '0';
                                        emem2 <= '0';
                                        emem3 <= '1';
                                        en_mem <= '0';
                                        ws_mem <= '0';
                                        rs_mem <= '0';

                                        estado1 <= liberacao;
                                end if;

                        when liberacao =>
                                 ready_out <= '1';
                                 sep       <= '0';
                                eproc <= '1';
                                emem1 <= '0';
                                emem2 <= '0';
                                emem3 <= '0';
                                en_cache <= '0';
                                ws_cache <= 'Z';
                                rs_cache <= 'Z';
                                en_mem <= '0';
                                ws_mem <= 'Z';
                                rs_mem <= 'Z';

                                estado1 <= inicio;

                end case;

        else   -- comeco do process de escrita
                        case estado2 is
                                when inicio =>
                                        if(rs_in = '0' and ws_in = '1') then
                                                ready_out <= '0';
                                                sep <= '1';
                                                en_cache <= '1';
                                                ws_cache <= 'Z';
                                                rs_cache<= 'Z';
                                                eproc <= '0';
                                                emem1 <= '0';
                                                emem2 <= '0';
                                                en_mem <= '0';
                                                ws_mem <= 'Z';
                                                rs_mem <= 'Z';

                                                estado2 <= compara;
                                        end if;

                                when compara =>
                                        if(comp = '1' or comp = '0') then
                                                 sep       <= '1';
                                                en_cache <= '1';
                                                ws_cache <= '1';
                                                rs_cache <= '0';
                                                emem1 <= '0';
                                                emem2 <= '0';
                                                emem3 <= '0';
                                                en_mem <= '0';
```

```
                                        ws_mem <= 'Z';
                                        rs_mem <= 'Z';
                                        eproc <= '0';

                                        estado2 <= atualiza_mem;
                                end if;

                        when atualiza_mem =>
                                        sep        <= '0';
                                        en_cache <= '0';
                                        ws_cache <= 'Z';
                                        rs_cache <= 'Z';
                                        emem1 <= '1';
                                        emem2 <= '1';
                                        emem3 <= '0';
                                        en_mem <= '1';
                                        ws_mem <= '1';
                                        rs_mem <= '0';
                                        eproc <= '0';

                                         estado2 <= finaliza;

                        when finaliza =>
                                        ready_out <= '1';
                                        sep        <= '0';
                                         eproc <= '0';
                                        emem1 <= '0';
                                        emem2 <= '0';
                                        emem3 <= '0';
                                        en_cache <= '0';
                                        ws_cache <= 'Z';
                                        rs_cache<= 'Z';
                                        en_mem <= '0';
                                        ws_mem <= 'Z';
                                        rs_mem <= 'Z';

                                estado2 <= inicio;
                end case;

                end if;  -- if da escolha da operacao read ou write
        end if; -- if do reset
    end process;
end Behavioral;
```

h. Integração dos módulos da cache

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--  library UNISIM;
--  use UNISIM.VComponents.all;

entity cache is                         -- declaracao dos pinos da cache
   Port (
                clk, reset              : in std_logic;
```

```
                end_proc        : in std_logic_vector(31 downto 0);
                dadosp  : BUFFER std_logic_vector(31 downto 0);

                dadosm  : BUFFER std_logic_vector(31 downto 0);
                en_in   : in std_logic;
                rs_in   : in std_logic;
                ws_in   : in std_logic;
                ready_in        : in std_logic;
                ready_out       : out std_logic;
                en_mem          : out std_logic;
                rs_mem : out std_logic;
                ws_mem          : out std_logic;
                end_mem         : out std_logic_vector(31 downto 0));
end cache;

architecture Behavioral of cache is
component separador32k                          -- declaracao dos componentes da cache, separador
    port (
                        a                       : in std_logic ;
                        endereco                : in std_logic_vector(31 downto 0);
                        linha           : out std_logic_vector(12 downto 0);
                        rotulo          : out std_logic_vector(18 downto 0)
                );
    end component;

component arm_dados32K is       -- componente que armazena os dados
      port(
                clk, rs, ws, habilita       : in std_logic;
                linha                       : in std_logic_vector(12 downto 0);
                dados                               : BUFFER std_logic_vector(31 downto 0)
        );
    end component;

component arm_rotulos32k is             -- componente que armazena os rotulos
     port(
                clk, rs,ws, habilita        : in std_logic;
                linha                       : in std_logic_vector(12 downto 0);
                novo                        : in std_logic_vector(18 downto 0);
                dados                               : BUFFER std_logic_vector(18 downto 0)
    );
    end component;

component bufferDados IS        -- componente buffer de dados
                PORT
                (
                        eproc                   : IN            STD_LOGIC;
                        emem2                   : IN            STD_LOGIC;
                        emem3                   : IN            STD_LOGIC;
                        proc_inout              : BUFFER        STD_LOGIC_VECTOR(31 downto 0);
                        cache_inout             : BUFFER        STD_LOGIC_VECTOR(31 downto 0);
                        mem_inout               : BUFFER        STD_LOGIC_VECTOR(31 downto 0)
                );
    end component;

component bufferMEM IS                  -- componente buffer de endereco
                PORT
                (
                        enable          : IN            STD_LOGIC;
                        entrada         : IN            STD_LOGIC_VECTOR(31 downto 0);
```

```
                saida                   : OUT    STD_LOGIC_VECTOR(31 downto 0)
        );
    end component;

component comparador19b is
        port
        (       rotulo_end              : in std_logic_vector(18 downto 0);
                rotulo_arm              : in std_logic_vector(18 downto 0);
                resultado               : out std_logic
        );
    end component;

component controlador is
    Port (      -- sinais do sistema
                 clk, reset         : in std_logic;
                -- sinais de entrada
                en_in               : in std_logic;
                ws_in               : in std_logic;
                rs_in               : in std_logic;
                comp                : in std_logic;
                 ready_in           : in std_logic;
                 -- sinais de saida
                 ready_out          : out std_logic;
                 sep                : out std_logic;
                eproc               : out std_logic;
                emem1               : out std_logic;
                 emem2              : out std_logic;
                 emem3              : out std_logic;
                en_cache            : out std_logic;
                ws_cache            : out std_logic;
                rs_cache            : out std_logic;
                en_mem              : out std_logic;
                ws_mem              : out std_logic;
                rs_mem              : out std_logic
        );
    end component;

    SIGNAL comp, sep_sin                    : std_logic;
    SIGNAL linha_sin                         : std_logic_vector(12 downto 0);
    SIGNAL rotulo_sin, rComp                : std_logic_vector(18 downto 0);
    SIGNAL dado_cache                       : std_logic_vector(31 downto 0);
    SIGNAL en_cache, rs_cache, ws_cache : std_logic;
    SIGNAL eproc_sin, emem1_sin, emem2_sin, emem3_sin : std_logic;

begin

c0: separador32k    port map(sep_sin, end_proc, linha_sin, rotulo_sin);
c1: arm_dados32K    port map(clk, rs_cache, ws_cache, en_cache, linha_sin, dado_cache);
c2: arm_rotulos32K  port map(clk, rs_cache, ws_cache, en_cache, linha_sin, rotulo_sin, rComp);
c3: bufferDados     port map(eproc_sin, emem2_sin, emem3_sin, dadosp, dado_cache, dadosm);
c4: bufferMEM               port map(emem1_sin, end_proc, end_mem);
c5: comparador19b   port map(rotulo_sin, rComp, comp);
c6: controlador     port map(clk, reset, en_in, rs_in, ws_in, comp, ready_in, ready_out, sep_sin, eproc_sin,
emem1_sin, emem2_sin, emem3_sin, en_cache, ws_cache, rs_cache, en_mem, rs_mem, ws_mem);


end Behavioral;
```