UNIVERSIDADE ESTADUAL DE MARINGÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RENATA CRISTINA LOPES DA SILVA

D-Power: ferramenta VHDL para estimar o consumo de potência dinâmica em arquiteturas superescalares

Maringá
2010

RENATA CRISTINA LOPES DA SILVA

D-Power: ferramenta VHDL para estimar o consumo de potência dinâmica em arquiteturas superescalares

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Departamento de Informática, Centro de Tecnologia da Universidade Estadual de Maringá, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.
Área de Concentração: Ciência da Computação.

Orientador: Prof. Dr. Ronaldo Augusto de Lara Gonçalves

Maringá
2010

"Dados Internacionais de Catalogação-na-Publicação (CIP)"
(Biblioteca Setorial - UEM. Nupélia, Maringá, PR, Brasil)

S586d

Silva, Renata Cristina Lopes da, 1984-
D-Power: ferramenta VHDL para estimar o consumo de potência dinâmica e arquiteturas superescalares / Renata Cristina Lopes da Silva. -- Maringá, 2010.
119 f. : il. (algumas color.).

Dissertação (mestrado em Ciência da Computação)--Universidade Estadual de Maringá, Centro de Tecnologia, Dep. de Informática, 2010.
Orientador: Prof. Dr. Ronaldo Augusto de Lara Gonçalves.

1. Computadores - Arquiteturas superescalares - Potência dinâmica - *D-Power* (Ferramenta VHDL). 2. *D-Power* (Ferramenta VHDL) - Computadores - Arquiteturas superescalares - Estimativa de consumo. I. Universidade Estadual de Maringá. Departamento de Informática. Programa de Pós-Graduação em "Ciência da Computação".

CDD 22. ed. -004.2565
NBR/CIP - 12899 AACR/2

Maria Salete Ribelatto Arita CRB 9/858
João Fábio Hildebrandt CRB 9/1140

FOLHA DE APROVAÇÃO

RENATA CRISTINA LOPES DA SILVA

D-Power: ferramenta VHDL para estimar o consumo de potência dinâmica em arquiteturas superescalares

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Departamento de Informática, Centro de Tecnologia da Universidade Estadual de Maringá, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

COMISSÃO JULGADORA

Prof. Dr. Ronaldo Augusto de Lara Gonçalves (Presidente)
Universidade Estadual de Maringá – DIN/UEM

Prof. Dr. João Angelo Martini
Universidade Estadual de Maringá – DIN/UEM

Prof. Dr. Marcos Antonio Cavenaghi
Universidade Estadual Paulista Júlio de Mesquita Filho – DCo/UNESP-Bauru

Aprovado em: 01 de março de 2010.
Local da defesa: Bloco C56, sala 118, *campus* da Universidade Estadual de Maringá.

AGRADECIMENTOS

Aos meus pais pelo apoio incondicional e incentivo nos momentos difíceis. Aos meus tios, que me acolheram e estiveram ao meu lado nessa trajetória. Ao Renato por sempre estar ao meu lado. À Maria Vithória, por me fazer rir mesmo em momentos de trabalho árduo.

Ao professor Ronaldo Augusto de Lara Gonçalves pela orientação no trabalho, incentivo, amizade, paciência e cobranças necessárias para que o trabalho se concretizasse. Ao professor Joao Angelo Martini pela disposição em colaborar com o projeto.

Aos colegas do Programa de Pós-Graduação em Ciência da Computação, em especial Everton de Melo, Maurílio Hirata e Rodrigo Pagno pelos momentos de descontração e conversas produtivas no decorrer do projeto. Aos colegas de laboratório pelas ideias e companhia. Ao amigo Nelson pelo incentivo, apoio e ajuda durante o período do mestrado. À Maria Inês Davanço pela amizade, carinho e dedicação a todos os alunos do Programa.

A todos que, de alguma forma, contribuíram para a realização deste trabalho.
À Deus, por permitir que tudo isso fosse possível.

D-Power: ferramenta VHDL para estimar o consumo de potência dinâmica em arquiteturas superescalares

RESUMO

Inovações tecnológicas surgem constantemente em sistemas de computadores, tendo como principal foco a melhoria no desempenho, uma vez que as aplicações estão cada vez mais complexas. Uma forma de melhorar o desempenho é aumentar o hardware. Este é o caso dos processadores superescalares, que possuem diversas filas e unidades funcionais capazes de proporcionar a execução simultânea de várias instruções. Porém, tal aumento de hardware tem suas implicações, como uma maior área ocupada no chip e um consequente aumento no consumo de potência. Esse acréscimo no consumo de potência aumenta a dissipação de calor, dificulta o resfriamento e a expansão do circuito, diminui a autonomia dos dispositivos móveis, dificulta a proximidade de componentes, entre outros fatores. Devido a esses problemas, o consumo de potência é alvo de diversas pesquisas que procuram estimá-lo e encontrar alternativas para reduzi-lo antes da concepção do sistema em chip. Nesse contexto, o presente trabalho apresenta uma alternativa para analisar e estimar o consumo de potência em componentes de processadores superescalares baseada na linguagem de descrição de hardware VHDL (*Very High-Speed Integrated Circuit Hardware Description Language*). O trabalho apresenta a ferramenta D-Power, uma ferramenta para estimar o consumo de potência dinâmica nos componentes da fase de busca de processadores superescalares. Com base nos parâmetros da ferramenta é possível verificar, dentre diversos modelos de componentes, quais são os mais vantajosos em relação ao consumo de potência e desempenho.

Palavras-chave: Arquiteturas superescalares. Consumo de potência

.D-Power: VHDL tool for dynamic power consumption estimation in superscalar architectures

ABSTRACT

Technological innovations constantly emerge in computer systems. Its primary focus is on the improve of performance, since applications are increasingly complex. One way to improve performance is to increase the hardware. This is the case of superscalar processors, which have several queues and functional units capable of providing simultaneous execution of multiple instructions. However, this increase in hardware has its implications, like a larger area required on the chip and a consequent increase in power consumption. This boost in power consumption raises heat dissipation, difficult cooling and circuit expansion and reduces the autonomy of mobile devices, among other factors. Because of these problems, the power consumption is target of several studies which try to estimate it and find alternatives to reduce it before the design of the chip. In this context, this paper presents an alternative to analyze and estimate power consumption in components of superscalar processors based on VHDL (Very High-Speed Integrated Circuit Hardware Description Language). The paper presents the D-Power tool, a tool designed to estimate dynamic power consumption in components of the fetch stage in superscalar processors. Based on the entries parameters, the tool is able to verify, among several models of components, which one are more advantageous in relation to power consumption and performance.

Keywords: Superscalar architectures. Power consumption.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BT/FNT	*Backward Taken/Forward Not Taken*
BTB	*Branch Target Buffer*
CMOS	*Complementary Metal Oxide Semiconductor*
DRAM	*Dynamic Random Access Memory*
DVS	*Dynamic Voltage Scaling*
FIFO	*First In First Out*
FPGA	*Field Programmable Gate Array*
GHR	*Global History Register*
HDL	*Hardware Description Language*
IPC	*Instructions per Cicle*
L1	*Level 1*
L2	*Level 2*
LFU	*Least Frequently Used*
LHT	*Local History Table*
LRU	*Least Recently Used*
MOS	*Metal Oxide Semiconductor*
nMOS	*Negative Complementary Metal Oxide Semiconductor*
PC	*Program Counter*
PDA	*Personal Digit Assistant*
PHT	*Pattern History Table*
PLB	*Pipeline Balancing*
pMOS	*Positive Complementary Metal Oxide Semiconductor*
RAW	*Read after Write*
RC	*Circuits Resistor-Capacitor*
RISC	*Reduction Instruction Set Computer*
RTL	*Register Transfer Level*
RUU	*Register Update Unit*
SAMQ	*Statically Allocated Multi-Queue*
SRAM	*Static Random Access Memory*
VHDL	*Very High-Speed Integrated Circuits Hardware Description Language*
WAR	*Write after Read*
WAW	*Write after Write*

LISTA DE SÍMBOLOS

KB	Kilobyte – Unidade de medida para armazenamento de informação digital
F	Farads – Unidade de medida para capacitância
Hz	Hertz – Unidade de medida para frequência
J	Joule – Unidade de medida para energia elétrica
MHz	MegaHertz – Unidade de medida para frequência
mW	MiliWatts – Unidade de medida para potência
V	Volts – Unidade de medida para tensão
μm	Micrômetro – Unidade de medida para tamanho

SUMÁRIO

CAPÍTULO

1

Introdução

1.1 Considerações Iniciais

A busca pela melhora no desempenho em sistemas de computadores é cada vez mais evidenciada, uma vez que há um constante crescimento na complexidade das aplicações. Procurando atender as exigências dessas aplicações, inovações arquiteturais são frequentemente investigadas e propostas para melhorar o desempenho dos sistemas. Uma das maiores inovações arquiteturais propostas para otimizar o desempenho dos sistemas foram as arquiteturas superescalares. Utilizadas até hoje, essas arquiteturas são caracterizadas pela presença de várias unidades funcionais e estruturas que paralelizam a execução de instruções.

Porém, a adição de novos hardwares acaba elevando o consumo de potência nos sistemas, o que aumenta a dissipação de calor no processador, diminui a vida útil dos componentes e eleva os custos com energia. O alto consumo também acaba influenciando nos preços dos sistemas, pois mais dispositivos são necessários para reduzir a temperatura interna dos componentes, tais como *coolers* e dissipadores de calor. Com a dissipação de calor cada vez maior, a temperatura dos ambientes também acaba aumentando, sendo necessária a utilização de equipamentos adicionais, como condicionadores de ar.

Assim, a redução no consumo de potência vem se tornando um fator essencial no projeto de sistemas de computadores, principalmente em sistemas móveis alimentados por baterias, nos quais um baixo consumo aumenta o tempo de vida da bateria entre as cargas

além de ajudar a diminuir o peso e o tamanho de tais dispositivos. Porém, essa redução deve ser alcançada sem comprometer o desempenho do sistema.

Nesse sentido, é importante estimar o consumo de potência de um circuito durante seu projeto, pois, em computação de alto desempenho, o consumo de potência tem influência direta no número de transistores que podem ser integrados em uma mesma pastilha de silício além de limitar a frequência de *clock* de operação do sistema.

Diversas ferramentas são propostas para analisar o consumo em diferentes níveis de projeto. Muitas delas operam em níveis de porta lógica e transistores, o que requer a prévia realização da síntese do circuito, o que nem sempre é possível em níveis iniciais do projeto. Então, alternativas para a estimativa do consumo em níveis comportamentais são necessárias para que projetistas possam tratar as restrições do consumo logo no início do projeto.

Nesse sentido é apresentada a ferramenta D-Power, uma ferramenta descrita em VHDL (*Very High-Speed Integrated Circuits Hardware Description Language*) que implementa uma arquitetura MIPS configurável com capacidade para estimar o consumo de potência dinâmica de um processador. A estimativa é realizada por meio do monitoramento de atividades de chaveamento no sistema. Uma atividade de chaveamento ocorre quando o valor de um sinal do circuito é alterado. Essa é principal fonte de dissipação de potência em circuitos CMOS (*Complementary Metal Oxide Semiconductor*). A análise é realizada nos sistemas de *caches* e fase de busca da arquitetura implementada.

A ferramenta apresenta diversos parâmetros arquiteturais configuráveis e o comportamento do consumo com a variação desses parâmetros pode ser verificado por meio da execução de simulações. Isso possibilita uma análise entre o consumo e o desempenho, fornecendo meios dos projetistas escolherem as melhores configurações para atender os requisitos dos seus projetos.

1.2 Objetivos

O principal objetivo do trabalho é proporcionar meios para realizar a estimativa do consumo de potência em fases iniciais do projeto. Nesse sentido, a ferramenta D-Power é apresentada, fornecendo formas de estimar o consumo de potência dinâmica sob diferentes configurações na fase inicial do *pipeline* de uma arquitetura superescalar.

1.3 Organização

O presente trabalho está organizado da seguinte forma: no Capítulo 2 são apresentados os principais conceitos referentes às arquiteturas superescalares, suas organizações e principais estruturas. O Capítulo 3 apresenta fundamentos de consumo de potência, introduzindo conceitos relacionados ao consumo, técnicas de redução e ferramentas para a estimativa do consumo de potência. Já o Capítulo 4 introduz a ferramenta D-Power, utilizada para a estimativa do consumo de potência dinâmica na fase de busca de um processador superescalar. Nesse capítulo são apresentados detalhes de implementação e da estrutura utilizada pelo D-Power. O Capítulo 5 apresenta alguns experimentos realizados com a ferramenta desenvolvida, mostrando seu potencial na análise do consumo de potência e do desempenho sob diferentes configurações arquiteturais. Por fim, o Capítulo 6 apresenta as conclusões e trabalhos futuros.

CAPÍTULO

2

Arquiteturas Superescalares

A constante busca por melhores desempenhos em sistemas de computadores é a principal meta dos projetistas, uma vez que as aplicações exigem cada vez mais dos processadores. As inovações arquiteturais que surgem tentam aumentar o desempenho da máquina e proporcionar um menor tempo de resposta ao usuário. É o caso dos processadores superescalares, que utilizam execução fora de ordem e especulativa para melhorar o número de instruções executadas por ciclo de *clock* (IPC). Surgindo como soluções mais agressivas para aumentar o desempenho de sistemas computacionais, essas arquiteturas começaram a ser implantadas em processadores comerciais a partir de 1988, com o Intel i960 (Garbus, 1992). Esses processadores dedicam uma grande quantidade da área do chip para estruturas que buscam otimizar o desempenho, como filas para armazenar instruções nos estágios do *pipeline*, tabela de renomeação de registradores e unidade de previsão de desvios. O presente capítulo apresenta a arquitetura de tais processadores, introduzindo os principais conceitos e técnicas empregadas visando à melhora no desempenho.

2.1 Arquitetura de um Processador Superescalar

Os processadores superescalares são caracterizados por explorar o paralelismo em nível de instrução, fazendo com que várias instruções sejam executadas simultaneamente. Para a

execução de instruções em paralelo algumas funcionalidades são necessárias (Smith e Sohi, 1995), como a determinação das dependências entre as instruções, estratégias para determinar quais estão prontas para execução, técnicas de renomeação de registradores e um circuito responsável por escalonar dinamicamente as instruções que podem ser executadas em paralelo pelas diversas unidades funcionais especializadas.

Os estágios e componentes básicos de uma arquitetura superescalar podem ser observados na Figura 2.1. Uma instrução passa pelos seguintes estágios básicos: busca, decodificação, despacho, remessa, execução e conclusão.

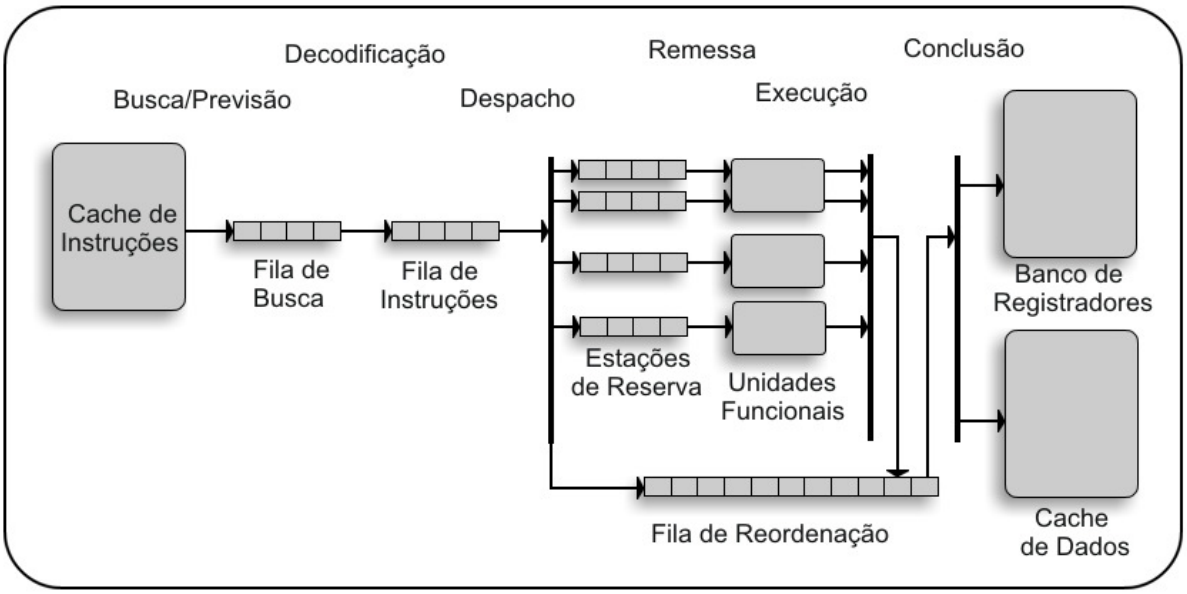


Figura 2.1. Arquitetura Superescalar típica

2.1.1 Busca

O fluxo de execução em um processador superescalar é iniciado na fase de busca das instruções, que é responsável por fornecer instruções para o restante do *pipeline* superescalar. Nessa fase, as instruções são trazidas da *cache* de instruções e inseridas em uma fila de busca. Durante esse estágio, as instruções são pré-analisadas para determinar se a operação é de desvio. Caso seja, a previsão de desvio é realizada de acordo com alguma técnica de previsão existente. Nas arquiteturas superescalares a fase de busca traz diversas instruções em um mesmo ciclo de *clock* e, para suportar essa demanda de instruções, torna-se necessário separar a *cache* de dados da *cache* de instruções.

A *cache* de instruções é organizada em blocos contendo instruções consecutivas. O contador de programa (*Program Counter* - PC) é utilizado para acessar o conteúdo da *cache* e verificar se a informação requerida se encontra nela. Em caso positivo, há um acerto (*hit*) na

cache e a instrução é inserida no *pipeline*. Caso contrário ocorre uma falta (*miss*) na *cache*, e a instrução deve ser buscada em níveis mais altos da hierarquia de memória, como *caches* nível 2 (L2 *cache*) ou memória principal. A falta na *cache* deve ser minimizada, uma vez que o acesso a níveis mais altos de memória são normalmente lentos e custosos e, consequentemente, reduzem o desempenho do processador (Lee, Evans e Cho, 2009).

2.1.2 Decodificação e Despacho

No estágio de decodificação, as instruções são retiradas da fila de busca e informações sobre a execução são associadas a cada instrução. Tais informações tratam da operação a ser executada e de identificadores dos operadores de entrada e saída. Em arquiteturas que permitem execução fora de ordem, também é inserida uma entrada em uma fila de reordenação para cada instrução decodificada. A fila de reordenação é um *buffer* FIFO (*First In First Out*) circular que mantém as instruções ativas até que elas sejam completadas. A Figura 2.2 apresenta a estrutura de uma fila de reordenação.

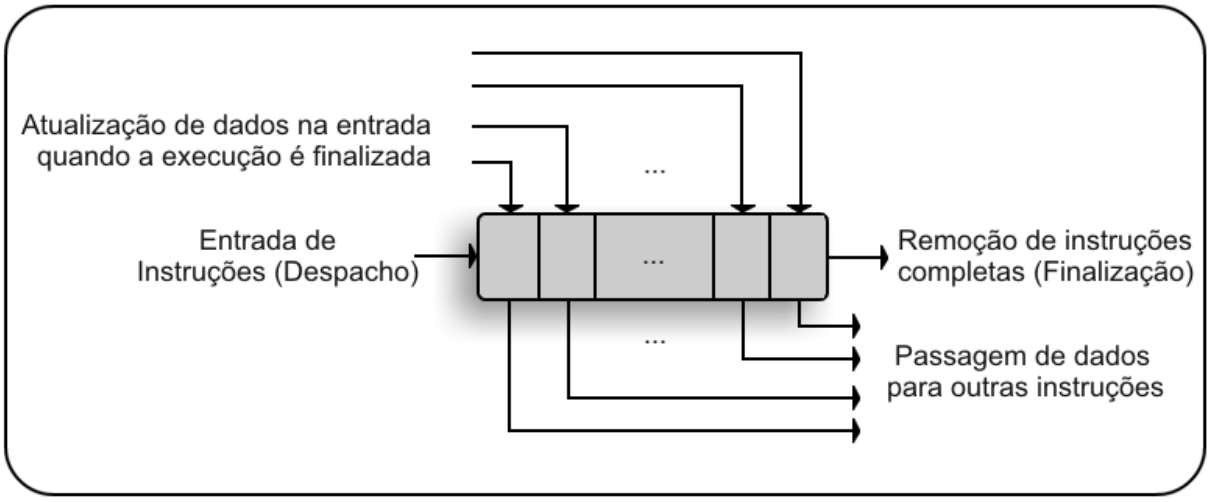


Figura 2.2. Fila de reordenação

Conforme as instruções vão sendo despachadas, uma entrada de cada instrução é inserida no final da fila de reordenação mantendo a ordem de execução do programa. Quando uma instrução completa sua execução, seu resultado é inserido na sua entrada na fila de reordenação, independente de sua posição. No momento que uma instrução atinge o início da fila, caso ela tenha sido completada, sua entrada é retirada da fila de reordenação e seu resultado transferido para o registrador correspondente. Já se a instrução estiver no início da fila, mas não tiver sido completada, ela permanecerá na fila e as instruções posteriores não poderão ser finalizadas. Isso garante a finalização em ordem das instruções.

Ainda na fase de decodificação, as instruções são examinadas para a verificação e tratamento de dependências. As dependências são classificadas em (i) dependência de dados, (ii) de controle e (iii) de hardware. As dependências de dados ocorrem quando duas instruções tentam acessar (ler ou escrever) um mesmo local de armazenamento (memória ou registradores). Dessa forma uma instrução *I0* pode armazenar o resultado de determinada operação em um registrador e uma instrução *I1* poderia ler esse mesmo registrador momentos depois. Assim, para que *I1* não leia um valor desatualizado do registrador, ela deve esperar a finalização de *I0* para só então acessar o dado requerido.

O exemplo anteriormente citado trata de uma dependência de dados verdadeira, na qual uma instrução de escrita é seguida por uma instrução de leitura. Isso causa um risco denominado RAW (*read-after-write*) (Utamaphethai, Blanton e Shen, 2001). Outra espécie de dependência de dados é a artificial, que ocorre quando uma operação de escrita procede a uma de leitura (WAR – *write-after-read*) ou quando uma instrução de escrita procede outra instrução do mesmo tipo (WAW – *write-after-write*) (Sima, 2000). Um risco WAR ocorre quando uma instrução precisa escrever um determinado valor em um local de armazenamento, mas deve esperar que todas as instruções precedentes leiam o antigo valor desse local. Já um risco WAW acontece quando várias instruções atualizam o mesmo local de armazenamento. Assim, a sequência que essas escritas ocorrem deve ser respeitada. A Figura 2.3 apresenta exemplos de dependências de dados utilizando registradores.

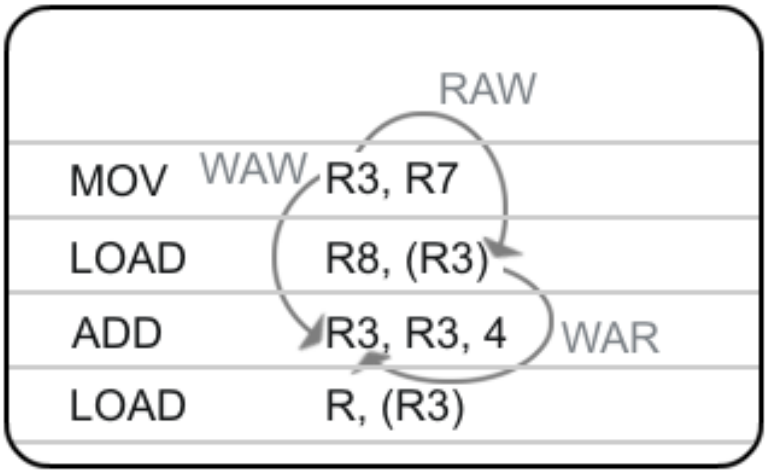


Figura 2.3. Exemplos de dependências de dados

O trecho de código exibido na Figura 2.3 apresenta uma instrução *mov* que irá atualizar o valor do registrador *r3*. Na sequência da execução, o mesmo registrador *r3* será acessado pela instrução *load* e *add*, o que corresponde a um risco RAW. A execução

dinâmica deve garantir que o valor lido por essas instruções corresponda ao valor atualizado pelo *mov*. A instrução *add* ainda utilizará o registrador *r3* para armazenar o resultado de sua operação, ocasionando um risco WAW, já que *r3* será escrito pela instrução *mov* e um risco WAR, uma vez que esse mesmo registrador será lido pela instrução *load*. Nesse sentido, a execução das instruções deve garantir que uma instrução posterior ao *add* acesse o valor de *r3* correto, escrito por essa instrução.

Já uma dependência de controle ocorre quando uma instrução de desvio é buscada, pois é o resultado da execução dessa instrução que determinará a próxima instrução a ser executada. Dessa forma, se o desvio não for tomado, a instrução seguinte na memória deve ser executada. Já se o desvio for tomado, a próxima instrução deverá ser buscada a partir de outro endereço, alterando o fluxo de execução das instruções. Somente com o resultado obtido após a execução da instrução de desvio é possível saber se o fluxo de controle de busca das instruções deve ser mantido ou alterado. Para resolver o problema da dependência de controle utiliza-se a previsão de desvios, técnicas utilizadas para antever se esses desvios serão tomados ou não.

A dependência de hardware ocorre quando duas ou mais instruções competem ao mesmo tempo por um mesmo recurso, como *caches* ou uma unidade funcional. Uma forma de resolver esse problema é aumentar os recursos disponíveis, por exemplo, utilizar várias unidades funcionais para a execução de um mesmo tipo de instrução.

A eliminação de dependências artificiais pode ser resolvida com técnicas de renomeação de registradores (Sima, 2000), que elimina dependências WAW e WAR e aumenta a média de instruções executadas por ciclo. Na renomeação de registradores, uma unidade de renomeação realiza a troca de registradores que causam conflitos por outros que não estão sendo utilizados, pertencentes a um conjunto especial de registradores (registradores lógicos) invisíveis ao programador.

Sempre que uma falsa dependência for encontrada, o registrador destino da instrução dependente é renomeado para um registrador de destino lógico disponível. O destino da instrução renomeada ficará armazenado em um desses registradores, que mantém contadores de acesso para indicar se o registrador está sendo utilizado e apontadores para indicar o registrador físico da instrução. Quando nenhum processo estiver mais utilizando o registrador físico, o conteúdo do registrador lógico é transferido para ele, e o registrador lógico torna-se livre para ser utilizado novamente.

Ao final da fase de decodificação, as instruções já decodificadas são despachadas para as estações de reserva apropriadas, para então serem transferidas para a execução em um estágio posterior do *pipeline*. Os estágios de decodificação e despacho normalmente são unificados.

2.1.3 Remessa e Execução

A fase de remessa é responsável pela verificação da disponibilidade de recursos para a execução das instruções inseridas nas estações de reserva, tais como as unidades funcionais e os operandos. As instruções ficam armazenadas nas estações de reserva aguardando a resolução de dependências verdadeiras de dados e dependências de recursos.

Uma estação de reserva pode ser associada a uma única unidade funcional ou compartilhada por duas ou mais unidades. As estações de reserva permitem que o resultado de uma instrução recém executada possa ser imediatamente utilizado por outras instruções que aguardam por esse resultado nas estações de reserva, não sendo necessário esperar até que o resultado seja armazenado no registrador destino. Essa técnica de realimentar as estações de reserva com o resultado de uma instrução recém executada é denominada *Forwarding* (Önder e Gupta, 1998). Cada entrada em uma estação de reserva armazena as informações apresentadas na Figura 2.4.

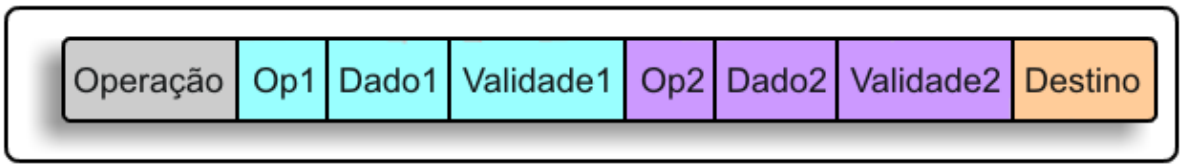


Figura 2.4. Informações presentes em cada entrada da estação de reserva

As estações de reserva mantêm, para cada instrução, a operação a ser realizada, o operando destino e informações sobre o valor e a validade de cada um dos operandos fonte. Quando uma instrução é executada, seu operando destino é comparado com os operandos fonte de instruções armazenadas nas estações de reserva para ver se existe alguma instrução que depende do resultado calculado. Em caso positivo, o valor é escrito na entrada do respectivo operando dessa instrução e o bit de validade é ativado para ‘1’. Quando todos os operandos forem válidos e existirem recursos disponíveis, a instrução é enviada para a unidade funcional responsável por executá-la. Se mais de uma instrução está pronta para a execução, alguma política de arbitragem deve ser utilizada para determinar qual instrução será

executada. Um exemplo de política é enviar a instrução que está a mais tempo na estação de reserva para a execução.

As arquiteturas superescalares possuem unidades funcionais especializadas que possibilitam a execução de instruções diferentes simultaneamente. A fase de execução recebe as instruções prontas para o processamento, executando-as em sua respectiva unidade funcional. Após a execução de cada instrução, seu resultado é inserido na fila de reordenação, na entrada associada à respectiva instrução, sendo esta marcada como pronta para a finalização. O resultado da execução também é repassado para as estações de reserva que contêm instruções aguardando esse resultado.

2.1.4 Conclusão

O último estágio do *pipeline* de uma arquitetura superescalar é o estágio de conclusão. O propósito dessa fase é implementar um modelo no qual as instruções pareçam ter sido executadas sequencialmente, apesar de elas poderem ter sido executadas fora de ordem ou especulativamente. É nessa fase que os resultados são escritos no banco de registradores ou na memória de dados. Existem três formas básicas de realizar o despacho das instruções e a conclusão dos resultados: (i) despacho em ordem, conclusão em ordem; (ii) despacho em ordem, conclusão fora de ordem; e (iii) despacho fora de ordem, conclusão fora de ordem.

Manter o despacho e a conclusão em ordem é a maneira mais simples de tratar as instruções. O despacho e a conclusão respeitam a ordem que as instruções aparecem. Nesse caso, não é necessária nenhuma unidade de reordenação, já que uma instrução só pode ser despachada se sua precedente também for. Ambas podem até ser despachadas em um mesmo momento. O mesmo ocorre com a conclusão, na qual duas ou mais instruções podem ser finalizadas ao mesmo tempo.

Uma alternativa é manter o despacho em ordem, mas permitir que a conclusão seja realizada fora de ordem. Assim, o resultado pode ser concluído em qualquer ordem, diminuindo a latência que algumas operações mais demoradas podem causar. Tal técnica possui um custo maior para o sistema, pois pode causar dependência de dados artificial (WAR e WAW) e é necessário hardware complementar para tratar tais dependências.

Com o despacho em ordem, o processador para de decodificar as instruções sempre que um conflito de dados ou hardware ocorre, uma vez que não é possível despachar instruções subsequentes que não possuem dependências. O despacho fora de ordem consegue

fazer isso, buscando e decodificando instruções posteriores que estão prontas para execução. Isso aumenta a vazão de instruções e diminui interrupções causadas por dependências de dados e hardware. Tal técnica necessita de uma etapa de reordenação após a conclusão das instruções para parecer que elas foram executadas em ordem. A fila de reordenação é comumente utilizada para esse propósito, na qual a análise começa sempre pela primeira posição da fila e para quando é encontrada a primeira instrução não pronta. Se a instrução no início da fila estiver pronta para ser finalizada, sua entrada na fila de reordenação é retirada e seu resultado gravado no registrador destino.

Caso a instrução seja de desvio, o resultado do desvio é examinado. Se a previsão do desvio foi realizada corretamente, a análise da fila prossegue normalmente a partir da segunda posição. Porém, se a previsão de desvio tenha ocorrido de maneira errônea todas as instruções buscadas posteriores à do desvio deverão ser descartadas, ocasionando uma penalidade ao desempenho do processador.

O descarte das instruções buscadas depois da instrução de desvio ocorre em todos os estágios do *pipeline*. Todos os *buffers* e as estações de reserva precisarão ser esvaziados, todas as entradas da fila de reordenação posteriores à entrada da instrução do desvio previsto errado deverão ser retiradas e os bits de ocupação dos registradores deverão ser desmarcados. O endereço de busca correto é passado para a fase da busca, que redireciona o fluxo de controle para trazer as instruções do caminho correto, voltando a preencher o *pipeline*. Após a finalização de todas as instruções prontas contidas na fila de reordenação, os ponteiros que indicam o início e final da fila são atualizados.

2.2 Cache

Embora o escalonamento dinâmico de instruções provido pelas arquiteturas superescalares tenha certo grau de tolerância à latência, algumas operações provocam uma latência muito grande, causando atraso no processamento de instruções. Um dos principais desafios dos projetistas é otimizar o tempo de operações de acesso à memória, não deixando que este fator torne-se o gargalo do sistema.

Operações de acesso à memória podem ser bem lentas. Um acesso à memória principal (DRAM – *Dynamic Random Access Memory*) pode demorar de 20 a 30 ciclos de *clock* para ser realizado (Diefendorff, 1999). Para minimizar a latência causada pela realização de uma operação de memória são utilizadas memórias *caches* (SRAM – *Static*

Random Access Memory) (Smith, 1982). As *caches* são unidades pequenas de armazenamento temporário e com tempo de acesso rápido. Quando um bloco da memória é acessado pela primeira vez, ele é trazido para uma linha da *cache* e um rótulo, baseado no endereço do bloco, é associado a essa informação. A partir de então, cada vez que o processador necessite acessar a memória, o rótulo do endereço é comparado com os rótulos armazenados na *cache* para checar se a informação requisitada está na *cache*. Em caso positivo ocorre um acerto e o dado é fornecido ao processador. Caso contrário ocorre uma falta e o dado deve ser buscado na memória principal para então ser fornecido ao processador e atualizado na *cache*.

Vários níveis de *cache* normalmente são utilizados para garantir menor latência nas operações de acesso à memória. A *cache* primária (*cache* de nível 1 ou *cache* L1) é rápida, porém pequena, enquanto a *cache* secundária (*cache* de nível 2 ou *cache* L2) é mais lenta, entretanto maior. Um nível só é acessado, caso a informação requisitada não seja encontrada no nível mais baixo, por definição aquele mais próximo ao processador. A Figura 2.5 apresenta a organização geral dessa hierarquia de memória.

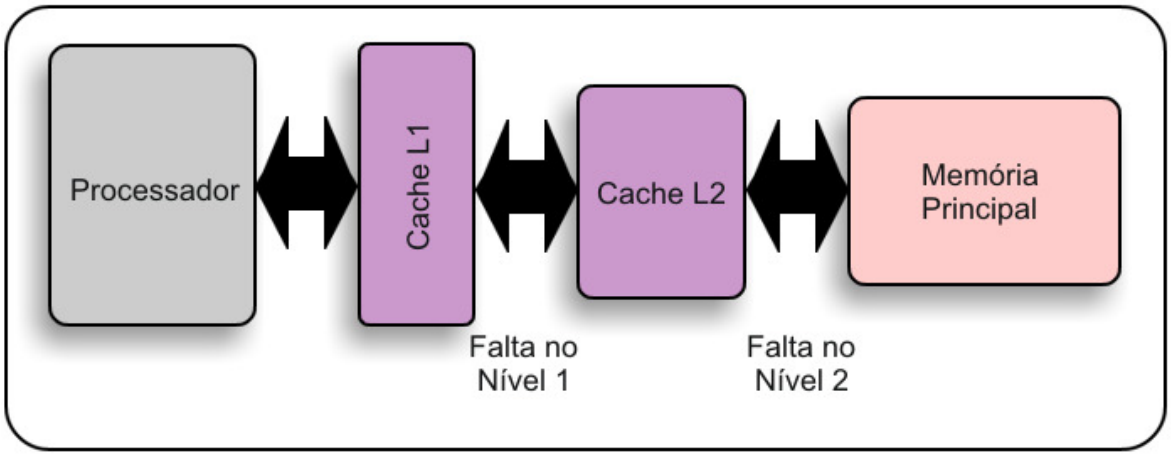


Figura 2.5. Organização geral da hierarquia de memória

Uma forma de melhorar o desempenho no acesso à memória é permitir que múltiplas requisições sejam atendidas simultaneamente, o que exige várias portas na hierarquia de memória. Normalmente é suficiente que apenas o nível mais baixo da hierarquia, *caches* primárias, seja multiporta, uma vez que nem todas as requisições são transferidas para os outros níveis (Smith e Sohi, 1995).

2.2.1 Organização da Cache

A maneira como a *cache* é organizada pode ajudar a reduzir faltas ocasionadas por conflitos, quando dois dados vindos da memória tendem a ocupar a mesma posição na *cache*. A forma mais simples de organização é a *cache* com mapeamento direto, na qual cada bloco da memória endereça exatamente uma única linha da *cache*. A Figura 2.6 apresenta o mapeamento de uma memória com 16 posições para uma *cache* de 4 linhas.

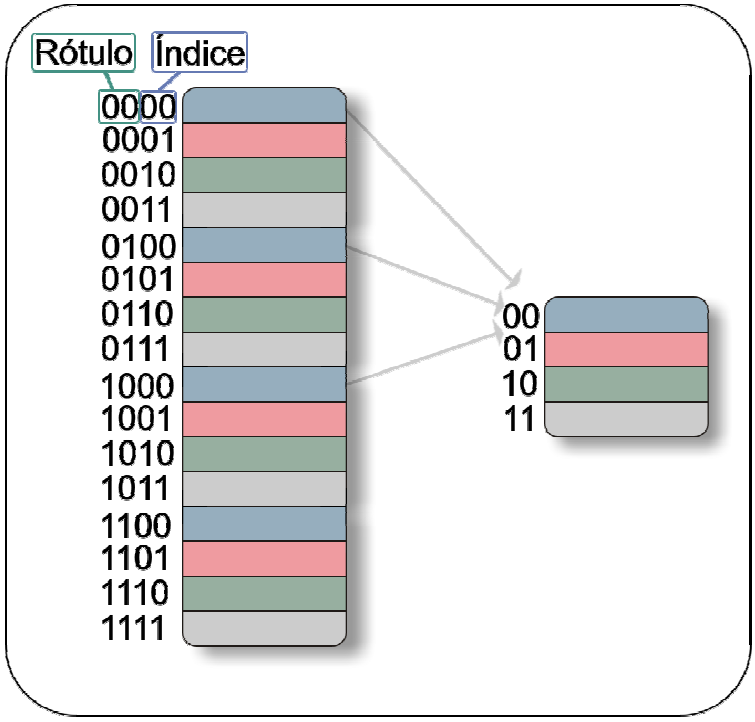


Figura 2.6. Mapeamento em uma cache com mapeamento direto

Como pode ser observado na Figura 2.6, mais de um bloco da memória pode ser mapeado para uma mesma posição da *cache*. Para determinar em qual linha da *cache* o bloco da memória será colocado utiliza-se os bits menos significativos do endereço, que formam o campo denominado índice. Esse campo também é utilizado durante o acesso para leitura de um dado na *cache*. A quantidade de bits utilizada para endereçar cada linha da *cache* é igual a *ln (l),* sendo *l* a quantidade de linhas da *cache*. Como diferentes blocos da memória são mapeados para uma mesma posição na *cache*, é necessário determinar se a informação contida na *cache* realmente é a informação requisitada. Para isso cada informação recebe um rótulo, composto pelos bits do endereço que não estão sendo utilizados como índice na *cache*. Dessa forma, a combinação índice e rótulo formam um identificador único, que pode ser utilizado para comparar informações na *cache*. A Figura 2.7 apresenta tal organização.

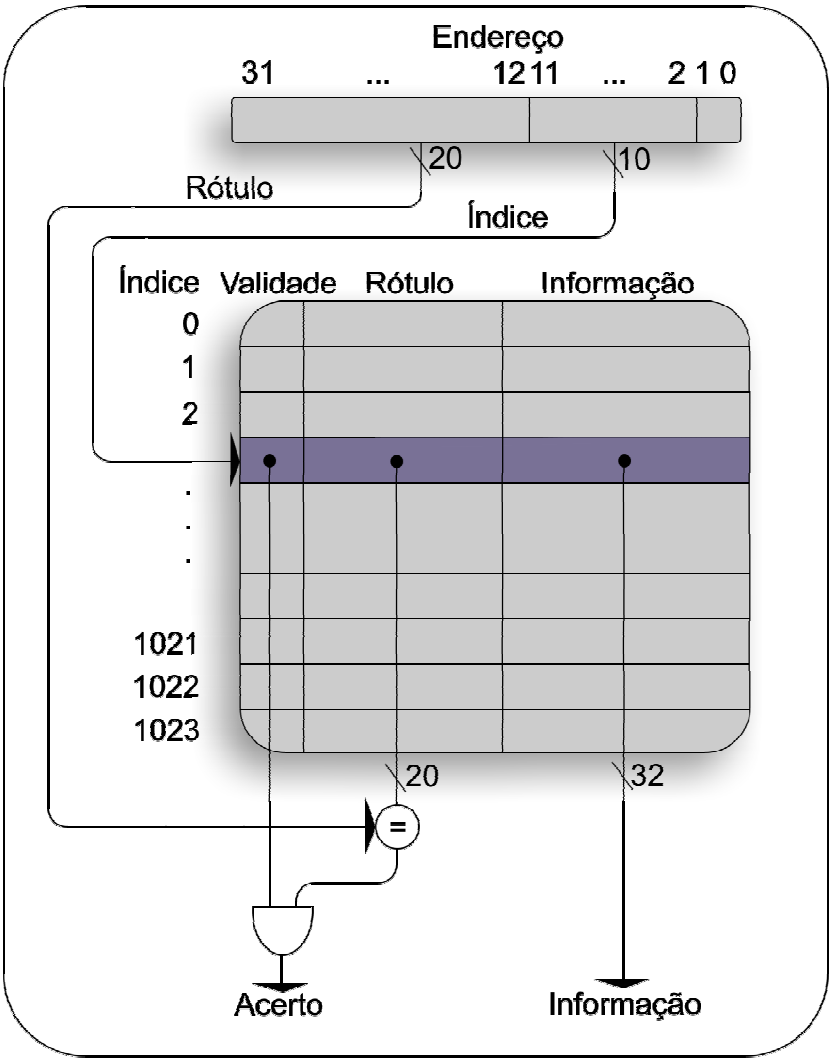


Figura 2.7. Organização de uma cache com mapeamento direto

Outra forma de organizar a *cache* é determinar que os blocos da memória não tenham posição definida durante o mapeamento. Na *cache* com mapeamento totalmente associativo, apresentada na Figura 2.8, os blocos da memória são inseridos na primeira posição desocupada na *cache*, independente de qual posição seja.

Na *cache* com mapeamento totalmente associativo, para verificar se um dado se encontra na *cache*, os rótulos de todas as posições são verificados e comparados com o rótulo da informação requerida. Para que isso seja possível um comparador para cada linha da *cache* é utilizado. Uma operação *AND* é realizada entre o resultado do comparador e o bit de validade da linha. O resultado dessa operação é utilizado como controle de um multiplexador, que irá selecionar a informação correta e transferi-la ao processador.

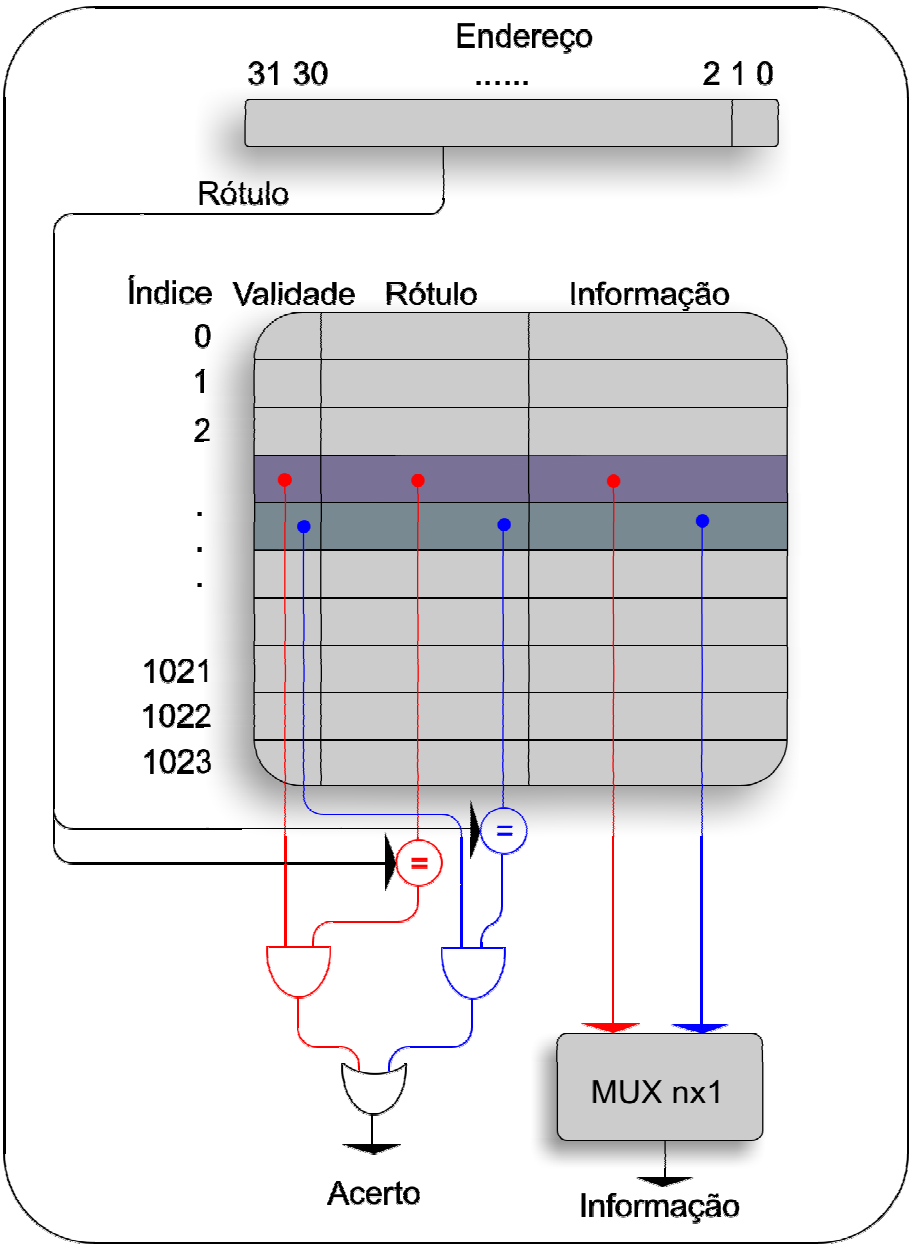


Figura 2.8. Cache com mapeamento totalmente associativo

A vantagem desse tipo de organização é que há melhor aproveitamento das posições livres da *cache*, uma vez que um dado só será substituído se a *cache* estiver cheia. O problema, porém, é o custo de implementação e operação de tal organização, que requer mais hardware e acesso a todas as linhas da *cache* para a busca de uma informação.

Para amenizar os problemas de conflito com o mapeamento direto e a complexidade e custo do mapeamento totalmente associativo, uma técnica intermediária pode ser utilizada, a *cache* com mapeamento associativo por conjunto, apresentada na Figura 2.9.

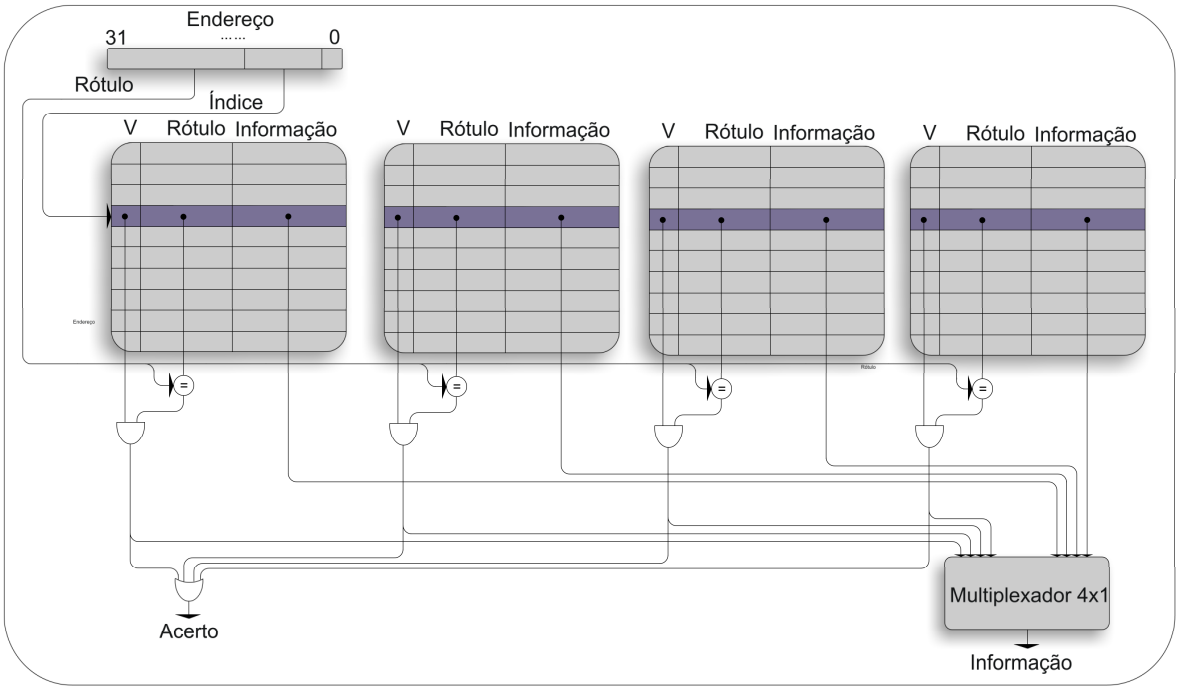


Figura 2.9. Cache com mapeamento associativo por conjunto com 4 vias

Nesse tipo de organização cada bloco pode ser armazenado em um determinado conjunto de blocos da *cache*. O número de posições que existe em cada conjunto é definido como a associatividade ou quantidade de vias da *cache*.

A vantagem de utilizar tal técnica é que há uma consequente redução nas faltas, pois mais de um bloco da memória pode ser armazenado com um mesmo índice. O índice indica em qual conjunto o dado requisitado está. Para verificar em qual via o dado se encontra compara-se o rótulo de cada bloco do conjunto acessado com o rótulo do dado requisitado. Em relação ao mapeamento totalmente associativo, a associatividade por conjunto traz uma redução na quantidade de comparações que devem ser realizadas para encontrar um dado e uma redução no custo de implementação.

Uma forma de melhor explorar a organização da *cache* é utilizar blocos com mais de uma palavra. Assim, sempre que ocorrer uma falta várias palavras adjacentes são buscadas da memória e trazidas para a *cache*, tirando proveito do princípio da localidade espacial, que afirma que se um dado é referenciado, dados próximos a ele também tendem a ser referenciados. A Figura 2.10 apresenta um exemplo de *cache* com mapeamento direto que endereça quatro palavras por bloco.

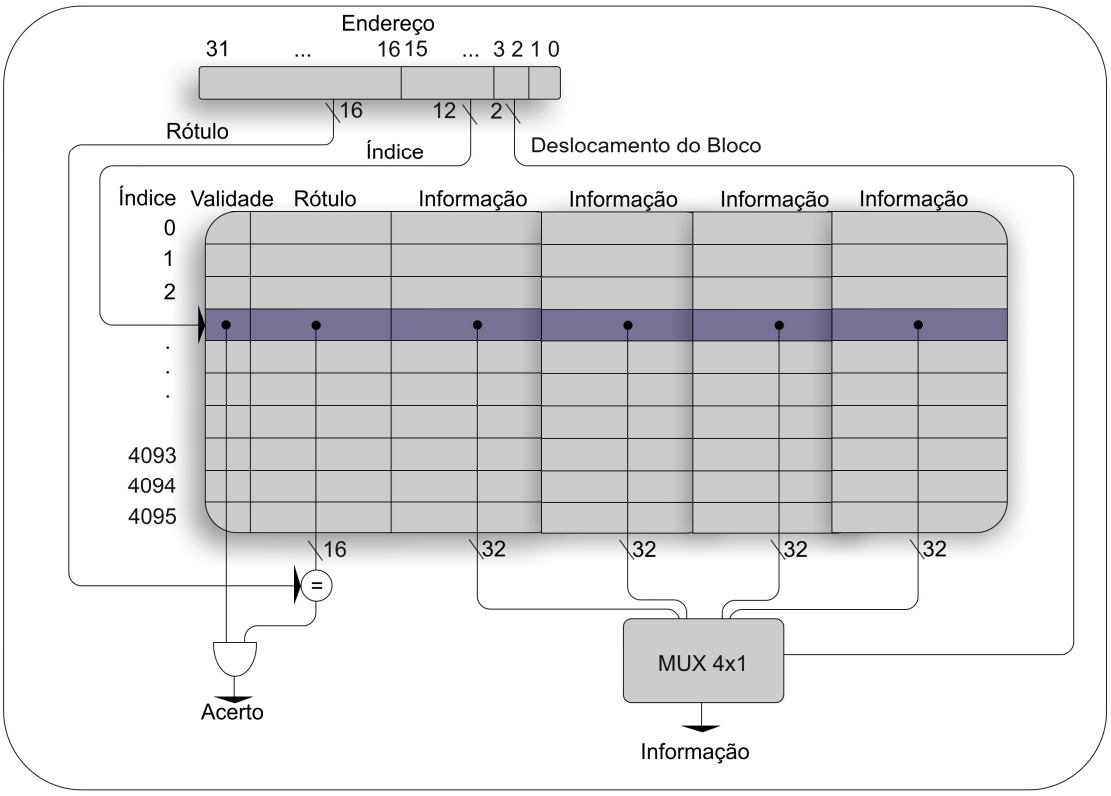


Figura 2.10. Cache com mapeamento direto e 4 palavras por bloco

Para que seja possível utilizar a configuração de *cache* apresentada na Figura 2.10 é necessário o uso de um campo extra para indexar o bloco. Esse campo é empregado como entrada de controle de um multiplexador, que selecionará a informação requisitada dentre as quatro informações presentes no bloco acessado.

Esse tipo de organização faz uso mais eficiente do espaço de armazenamento da *cache*, pois o número de bits de validade e bits de rótulo é menor que em uma *cache* de capacidade equivalente, mas com uma única palavra por bloco. Isso porque *caches* com mais de uma palavra por bloco utilizam um único rótulo e bit de validade por bloco. No exemplo apresentado na Figura 2.10, quatro palavras diferentes compartilham um mesmo rótulo.

2.2.2 Políticas de Substituição

Quando uma informação é requisitada pelo processador e a *cache* está cheia, alguma informação presente na *cache* deve ser escolhida para ser substituída. As principais políticas de substituição na *cache* são: (i) LRU (*Least Recently Used*), (ii) LFU (*Least Frequently Used*), (iii) FIFO ou (iv) aleatório.

A política LRU substitui o bloco que está a mais tempo sem ser referenciado. Ela

utiliza um ou mais bits para indicar quando um bloco foi acessado em cada linha da *cache*. Esses bits indicam a ordem de acesso de cada bloco. Quando um bloco é acessado, os bits de uso são atualizados. No início todos os bits estão em '0', indicando que qualquer um dos blocos pode ser substituído. A implementação da LRU é cara para *caches* com alto grau de associatividade, uma vez que o controle das informações que permitem determinar qual bloco será substituído possui um custo alto para a implementação em hardware. Mesmo assim, por apresentar bom desempenho é uma das técnicas mais utilizadas em processadores comerciais (Patterson e Hennessy, 2000).

Já a política LFU substitui o bloco que tem sido menos acessado. Para isso contadores são associados a cada bloco, sendo incrementados a cada acesso. Dessa forma, ao acessar um conjunto da *cache* para substituir um dado, o bloco substituído é aquele cujo contador possui o menor valor. A política FIFO substitui o bloco que está a mais tempo na *cache*. A política não faz distinção da frequência de acesso ao bloco, podendo substituir um bloco que está sendo constantemente utilizado.

Por fim, a política aleatória substitui qualquer um dos blocos do conjunto, independente de sua situação. A substituição aleatória é de simples implementação e, para *caches* com muitas vias, seu desempenho se equivale ao do LRU (Monteiro, 2007).

2.2.3 Políticas de Escrita

Nos processadores que fazem uso de memórias *cache*, sempre que uma operação de escrita é realizada, ela reflete imediatamente no conteúdo da *cache*. Porém, é necessário manter uma integridade de dados entre a *cache* e os outros níveis da memória, propagando as alterações realizadas na *cache* para que dados posteriormente acessados não estejam inconsistentes. Dá-se o nome de política de escrita à técnica utilizada nessa transferência de dados entre a *cache* e os outros níveis da memória durante operações de escrita. As principais políticas de escrita empregadas em arquiteturas superescalares são: (i) *write through* e (ii) *write back*.

A política *write through* faz a atualização nos outros níveis de memória no momento em que é realizada a escrita na *cache*. A vantagem desse método é que ele é de fácil implementação e mantém os níveis de memória sempre atualizados. O problema porém é que cada operação de escrita requer um acesso a todos os níveis de memória, inclusive à memória principal, que possui grande latência de acesso. Muitos desses acessos são desnecessários e acabam reduzindo o desempenho do sistema.

Já a política *write back* realiza a atualização apenas quando o bloco na *cache* for substituído e se houver alguma modificação nele. Um bit de modificação é adicionado a cada bloco da *cache* e sempre que houver uma operação de escrita esse bit é atualizado. Assim, quando o bloco armazenado na *cache* for substituído, é feita uma verificação nesse bit, que indicará se o bloco deve ser escrito no nível de memória imediatamente acima ou não.

A vantagem dessa técnica é que ela evita acessos desnecessários aos níveis mais altos da hierarquia de memória. O problema é que a memória principal fica desatualizada em relação à *cache* e, se existir outro dispositivo que acesse a memória, ele pode receber dados inconsistentes. Uma forma de solucionar isso é selecionar áreas da memória principal que não vão para a *cache*.

2.3 Previsão de Desvios

Em arquiteturas superescalares, instruções de desvios condicionais são constantemente encontradas sem que os dados que determinam a direção do desvio estejam prontos. Como as instruções são buscadas antes da execução, prever corretamente a direção desses desvios não resolvidos permite a busca de instruções com alta probabilidade de serem utilizadas e, consequentemente, não interrompe o fluxo de busca de instruções.

Alguns processadores chegam a executar essas instruções especulativamente evitando latência ocasionada por dependências de controle caso a previsão tenha sido realizada corretamente. Entretanto, quando uma previsão é feita incorretamente, o *pipeline* deve ser reiniciado, com todas as instruções presentes neles descartadas e o fluxo de busca redirecionado. Ao circuito responsável por determinar se um desvio será ou não tomado dá-se o nome de previsor de desvios (Lee e Smith, 1984). As técnicas de previsão de desvio são classificadas em estáticas e dinâmicas.

2.3.1 Técnicas de Previsão Estáticas

As técnicas estáticas mantêm sempre a mesma decisão para determinada situação. Alguns exemplos de previsores estáticos são: considerar que um desvio sempre será tomado ou considerar que o desvio nunca será tomado. É possível também prever o desvio dependendo do endereço do salto. A técnica BT/FNT (*backward taken/forward not taken*) (Calder, Grunwald e Emer, 1995), por exemplo, assume que desvios cujo destino está endereçado para

antes do endereço da instrução atual sejam sempre tomados e instruções cujo destino do desvio é posterior ao endereço da instrução de desvio não sejam tomados.

Outro exemplo de técnica de previsão estática é considerar a operação a ser realizada pela instrução. A técnica baseada no código da operação considera que instruções que fazem parte de um conjunto de *opcodes* saltem para o endereço do desvio e instruções que não pertencem a esse conjunto mantenham o fluxo de busca de instruções na sequência, não tomando o desvio. Tais técnicas requerem pouco hardware e são de fácil implementação.

2.3.2 Técnicas de Previsão Dinâmicas

Já as técnicas dinâmicas observam o comportamento das instruções de desvios executadas anteriormente para prever se o desvio deve ou não ser tomado. A implementação de um previsor dinâmico requer hardware dedicado que ocupa bastante espaço em chip, o que o torna caro. Todavia, o aumento no desempenho que ele proporciona é um atrativo para projetistas utilizarem tais técnicas na grande maioria das arquiteturas modernas (Guy e Haggard, 1996).

Previsores dinâmicos normalmente utilizam uma BTB (*Branch Target Buffer*), uma pequena memória *cache* utilizada para manter informações relacionadas aos desvios, tais como informações sobre a previsão, endereço alvo do desvio e o endereço da instrução. Quando uma instrução é buscada, seu endereço é comparado com o conteúdo da BTB. Caso o endereço da instrução buscada esteja na BTB, a previsão de desvio é realizada utilizando informações de previsão armazenadas na BTB e associadas àquele endereço. Quando a instrução é executada, seu resultado é comparado com o resultado previsto. Se a previsão foi realizada corretamente, a execução continua sem qualquer interrupção. Já se o resultado da previsão for diferente do resultado previsto, o *pipeline* deve ser reiniciado, as instruções dentro do *pipeline* descartadas e novas instruções devem ser buscadas a partir do fluxo correto da execução.

Técnicas dinâmicas devem possuir uma forma de atualizar sua previsão sempre que uma instrução de desvio é executada. Para isso é possível utilizar um registrador de histórico global (GHR – *global history register*), uma tabela de histórico local (LHT – *local history table*) ou uma tabela de histórico de padrões (PHT – *pattern history table*).

O GHR é um registrador de *n* bits que mantém os resultados dos últimos *n* desvios. O registrador é independente do endereço do desvio, o que significa que o resultado de todas as

instruções de desvio é armazenado no GHR. Dessa forma a previsão pode ser realizada baseada no comportamento dos últimos desvios executados.

A LHT é um vetor com registradores de *k* bits incorporados à BTB, na qual cada entrada contém informações sobre o resultado das *k* instruções de desvio anteriores para cada endereço. Já a PHT é um vetor de contadores de saturação que podem ser indexados de diversas maneiras, sendo inclusive incorporados a cada endereço da BTB. Um contador de saturação é um contador que possui um limite máximo e mínimo. Por exemplo, um contador de saturação de dois bits atingirá seu limite máximo ao atingir o valor 3 (“11” em binário). Assim, se o contador for incrementado novamente ele permanecerá com o valor 3. Tais contadores são utilizados porque conseguem reconhecer os padrões dos desvios.

2.4 Considerações Finais

O presente capítulo introduziu conceitos relacionados às arquiteturas superescalares e suas principais estruturas, tais como *cache* e previsor de desvios. O emprego de técnicas superescalares em processadores modernos está relacionado ao avanço no processamento de alto desempenho, uma vez que o tempo de execução de um conjunto de instruções é otimizado quando utilizadas tais arquiteturas.

O uso desses métodos requer o uso de maior quantidade de hardware e consequentemente, maior quantidade de transistores para sua implementação. Isso ocasiona um aumento na dissipação de potência de tais sistemas, o que pode acabar interferindo em seu desempenho e tempo de vida. Assim, um estudo mais aprofundado no comportamento do consumo de potência de arquiteturas superescalares é requerido a fim de obter melhores resultados no desempenho e manter o consumo em níveis aceitáveis.

CAPÍTULO

3

Consumo de Potência

O aumento da complexidade da arquitetura, da frequência de *clock* e da quantidade de transistores em busca de melhores desempenhos tem aumentado consequentemente as taxas no consumo de energia e potência dos sistemas de computadores, o que eleva os custos de manutenção e operabilidade do sistema.

Esse crescente aumento no consumo tem levado as técnicas de resfriamento a chegarem próximas de seus limites. Técnicas de refrigeração e resfriamento líquido serão indispensáveis caso o consumo e a dissipação de calor continuem crescendo (Brooks et al., 2003). Neste sentido, é importante ter estimativas desse consumo, pois ele possui forte influência em diversos fatores do projeto tais como capacidade de fonte de alimentação, tempo de vida de bateria e definição dos mecanismos de dissipação de calor. O presente capítulo apresenta uma visão geral dos principais conceitos relacionados à potência e seu consumo, apresentando técnicas para reduzir e ferramentas para estimar o consumo de potência.

3.1 Potência

Segundo Venkatachalan e Franz (2005), potência e energia são definidas em termos do trabalho que o sistema realiza. A potência é a taxa na qual o sistema realiza o trabalho, já a

energia é a quantidade total de trabalho que o sistema realiza durante um período de tempo. A potência é mensurada em *watts* enquanto a energia é medida em *joules*.

Os conceitos de energia, potência e trabalho são utilizados em diferentes contextos quando relacionados a sistemas de computadores. O trabalho envolve tarefas diretamente associadas à execução de programas, tais como adição, subtração e acesso e escrita na memória. Já a potência é a taxa a qual o sistema consome energia elétrica enquanto realiza essas atividades e a energia é a quantidade total de energia elétrica que o sistema consome no decorrer de um período de tempo.

É importante a distinção desses conceitos, uma vez que técnicas que reduzem o consumo de potência nem sempre reduzem o consumo de energia. Quando a frequência de *clock* é reduzida, há uma redução na potência dissipada, mas o mesmo trabalho computacional levará mais tempo para ser realizado, o que manterá o mesmo consumo de energia ao final da realização da atividade. Por exemplo, uma tarefa demora 10 ciclos para ser realizada com um ciclo de *clock* de 50MHz. Uma redução de 50% na frequência do *clock* implica que a atividade levará 20 ciclos para ser completada.

A redução do consumo de energia ou potência depende do contexto em que o sistema está inserido. Para aplicações móveis, como telefones celulares, *laptops* e PDAs (*Personal Digit Assistants*), é importante uma redução no gasto de energia, uma vez isso afeta o tempo de duração da carga de uma bateria. Já para outros sistemas, como servidores e computadores pessoais, um alto consumo de potência eleva a temperatura do sistema e diminui seu desempenho e confiabilidade. Logo, é importante a redução da dissipação de potência instantânea para que o sistema opere em temperaturas aceitáveis, independente do gasto final que o sistema terá com energia (Lu e De Micheli, 2001).

O consumo de potência de um circuito CMOS (*Complementary Metal Oxide Semiconductor*) é determinado primariamente por uma potência estática dissipada por correntes de fuga e pela potência dinâmica, que é constituída por uma dissipação de potência de curto-circuito e pela potência de chaveamento consumida durante o carregamento ou descarregamento de cargas de capacitância (Mathew, 2004).

3.2 Potência Dinâmica

A parcela dinâmica é a principal fonte de dissipação de potência na grande maioria dos circuitos CMOS (Chabini, Aboulhamid e Savaria, 2001), sendo quadraticamente proporcional

à voltagem fornecida. Ela é composta pela potência de chaveamento (*switching power*) e pela potência de curto circuito (*short-circuit power*).

A equação que determina a potência dinâmica em um circuito CMOS pode ser determinada pela Equação 3.1 (Devadas e Malik, 1995).

$$P = \alpha \cdot C \cdot f \cdot V_{DD}^2 + Q_{SC} \cdot V_{DD} \cdot f \cdot \alpha \quad \text{... (3.1)}$$

O primeiro termo na Equação 3.1 corresponde à potência de chaveamento, na qual α é a atividade de chaveamento (*switching activity*), causada pela troca de valores do em uma porta lógica, ocasionando carga ou descarga nos capacitores deste. *C* é a capacitância de cada nodo, *f* é a frequência do *clock* e V_{DD} é a tensão fornecida. Em circuitos CMOS, a potência dinâmica é dissipada quando a energia é retirada da fonte de potência para carregar a capacitância de saída do nodo. Durante a fase de carregamento, a voltagem de saída do nodo faz uma transição de 0 para V_{DD}, e a energia utilizada para essa transição é relativamente independente da função executada pelo circuito (Akita e Asada, 1995). Para ilustrar a dissipação de potência dinâmica, considere o circuito apresentado na Figura 3.1.

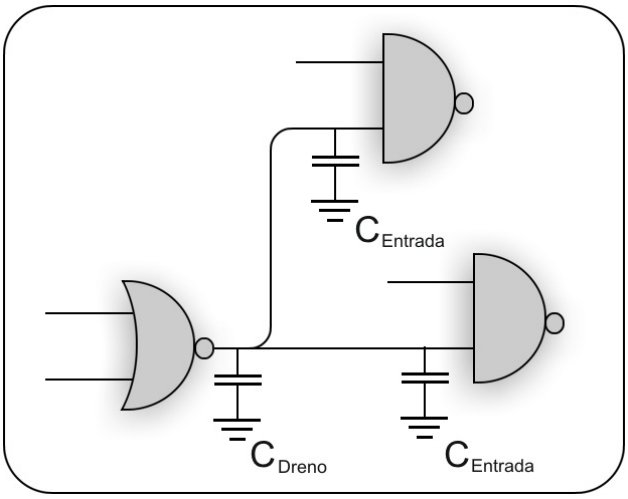


Figura 3.1. Portas NAND tendo como entrada sinal recebido de uma porta NOR

Como apresentado na Figura 3.1, o resultado de uma porta NOR é utilizado como entrada de duas portas NAND. A conexão entre as portas é realizada por meio de linhas de interconexão. Nesse sentido, a carga capacitiva da saída da porta NOR consiste (i) da capacitância da porta lógica, (ii) da capacitância total da linha de interconexão e (iii) das capacitâncias de entrada das portas NAND.

A dissipação da potência de chaveamento de uma porta CMOS é essencialmente independente das características e tamanho dos transistores. Assim, dado um padrão de dados de entrada, o atraso na propagação do resultado da porta lógica não tem relevância na quantidade de potência consumida durante uma atividade de chaveamento. A potência de chaveamento é a principal fonte de dissipação de potência e chega a corresponder a mais de 90% da potência de circuitos anteriores a nanotecnologia (Fornaciari et al., 1997).

O segundo termo da Equação 3.1 representa a potência de curto circuito, uma fonte secundária de dissipação de potência. Ela ocorre porque, durante a transição de uma porta CMOS, dois transistores complementares nMOS (*Negative Complementary Metal Oxide Semiconductor*) e pMOS (*Positive Complementary Metal Oxide Semiconductor*) podem conduzir carga simultaneamente durante um curto período de tempo, estabelecendo um fluxo direto de corrente da fonte para o dreno (terra). A variável Q_{SC} indica a quantidade de carga carregada pela corrente de curto circuito por transição na Equação 3.1.

Considerando o inversor apresentado na Figura 3.2, durante a troca do sinal de saída da porta lógica, o transistor nMOS começará a conduzir energia quando a tensão de entrada (Vin) exceder sua tensão de limiar (VTn). O transistor pMOS permanecerá ligado até que a entrada atinja o nível de tensão Vdd - |VTp|, sendo VTp a tensão de limiar do transistor pMOS. Assim, há um curto período de tempo em que ambos os transistores estão conduzindo energia.

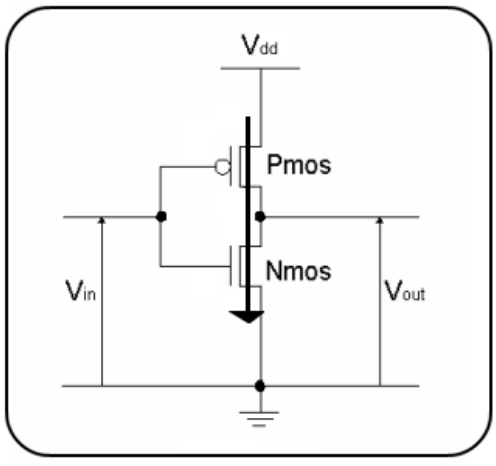


Figura 3.2. Inversor CMOS

A corrente de curto-circuito é terminada quando a transição da tensão de entrada é completada e o transistor pMOS é desligado. Um evento similar é responsável por estabelecer uma corrente de curto circuito durante uma transição de 0 para 1, quando a tensão de saída começa a aumentar enquanto ambos os transistores estão ligados.

A magnitude de uma corrente de curto-circuito é praticamente a mesma durante transições de subida ou descida de uma saída lógica, assumindo que o tempo de subida e descida do sinal seja o mesmo. Essa corrente não contribui com a capacitância do circuito.

3.3 Potência Estática

Já a potência estática é a potência dissipada quando o circuito está inativo. Ela é composta pela corrente de fuga (*leakage current*) do circuito, obtida quando não há variações nos sinais aplicados às entradas de uma porta (Fornaciari et al., 1997). A fórmula para a obtenção da potência estática é apresentada na Equação 3.2:

$$P_{estática} = I_{fuga} \cdot V_{DD} \quad \text{... (3.2)}$$

Como pode ser observado na Equação 3.2, a potência estática corresponde ao produto entre a corrente de fuga, I_{fuga}, e a tensão fornecida, V_{DD}. A corrente de fuga é composta por seis tipos diferentes de correntes. São elas: (i) corrente de polarização reversa (*reverse-biased-junction*), (ii) corrente de sub-limiar (*subthreshold*), (iii) corrente de porta (*gate-current*), (iv) corrente de dreno induzida pela porta (*gate-induced-drain*), (v) corrente de perfuração (*punchthrough*) e (vi) corrente de óxido da porta (*gate-oxide*).

A corrente de sub-limiar corresponde a maior parcela da corrente de fuga (Mamidipaka et al., 2003). Essa corrente é gerada entre o dreno e a fonte mesmo quando o transistor está desligado. Ela depende de alguns parâmetros como o tamanho da porta lógica, a tensão fornecida, a tensão limiar e a temperatura.

A potência estática é dependente da tecnologia empregada na fabricação do circuito e é significativa somente em circuitos CMOS modernos, pois a redução no tamanho dos transistores desses circuitos fez com que os isolantes entre a base e o substrato fossem reduzidos, aumentando assim a corrente de fuga.

3.4 Redução no Consumo de Potência

Na constante busca por melhores desempenhos nos sistemas de computador, o aumento na quantidade de hardware tem sido uma das alternativas mais aceitas, pelo fato de proporcionar paralelismo em nível de instrução e possibilitar a execução simultânea de diversas instruções.

Entretanto, essa adição de componentes pode trazer efeitos negativos, como o aumento no consumo de energia ou da dissipação do calor. Assim, o ganho de desempenho deve ser alcançado sem comprometer o funcionamento do sistema, devendo manter o consumo de energia a níveis aceitáveis.

Neste âmbito, a redução do consumo de potência tem sido alvo de diversas pesquisas atualmente. Tanto na indústria quanto na área acadêmica, técnicas que buscam reduzir a potência e a energia dissipada nos processadores estão sendo estudadas e incorporadas aos sistemas computacionais.

3.4.1 Redução no Consumo da Potência Dinâmica

Existem quatro formas principais para reduzir a dissipação dinâmica de potência (Venkatachalam e Franz, 2005). A primeira é reduzir a capacitância física de um circuito. A capacitância física depende de parâmetros de projeto em baixo nível tais como tamanho dos transistores ou comprimento dos fios. A redução na capacitância física pode ser obtida com a redução do tamanho dos transistores. Isso reduz a dissipação de potência, porém também reduz o desempenho do circuito.

A segunda maneira de reduzir o consumo de potência dinâmica é reduzir a atividade de chaveamento. Como uma das principais fontes de dissipação de potência é a troca de valores nos sinais de uma porta lógica, técnicas que reduzem a ocorrência dessa troca de valores reduzem o consumo total da potência dissipada.

A terceira forma é a redução da frequência do *clock*. Entretanto, essa técnica reduz o desempenho do sistema e pode ainda não reduzir a energia total gasta, uma vez que o sistema demorará mais ciclos para realizar uma atividade.

A quarta maneira de reduzir o consumo de potência é diminuir a fonte de tensão do circuito. Como essa redução provoca atraso nas portas, também é necessário reduzir a frequência do clock para o circuito operar corretamente. A utilização de tal técnica, denominada DVS (Dynamic Voltage Scaling) (Snowdon, Ruocco e Heiser, 2005) deveria fornecer idealmente uma redução de ordem cúbica no consumo de potência, uma vez que a quantidade de potência dissipada é quadraticamente proporcional à tensão aplicada ao processador e linearmente e inversamente proporcional à voltagem.

3.4.2 Redução no Consumo da Potência Estática

Para a redução no consumo de potência estática também existem quatro formas principais (Venkatachalam e Franz, 2005). A primeira consiste em reduzir a fonte de tensão do circuito. A segunda maneira é reduzir o tamanho do circuito, uma vez que a corrente de fuga total de um circuito é proporcional à corrente de fuga dissipada em todos os transistores do circuito. Uma forma de conseguir essa redução é projetar o circuito com menos transistores por meio da omissão de componentes redundantes. Todavia, isso pode limitar o desempenho do sistema.

A terceira forma de reduzir a corrente de fuga é refrigerando o circuito. A principal vantagem da redução de temperatura é que ela não influencia negativamente o desempenho do circuito. Pelo contrário, o circuito pode trabalhar mais rápido, uma vez que a eletricidade encontra menos resistência em temperaturas mais baixas. Outra vantagem é que a refrigeração do circuito elimina alguns problemas provocados por temperaturas altas, como a durabilidade e confiabilidade do circuito. O problema, porém é que técnicas de refrigeração normalmente aumentam bastante o custo do sistema (Ellsworth, 2004). Além disso, técnicas de refrigeração devem ser eficientes a ponto de não causar grandes variações de temperatura no circuito e de distribuir igualmente o calor por todo o circuito para não criar pontos de concentração de altas temperaturas, denominados *Hotspots* (Link e Vijaykrishnan, 2006).

Por fim, a quarta maneira de reduzir a corrente de fuga é aumentar a tensão de limiar. A tensão de limiar é a tensão mínima aplicada em um transistor para que ele seja ativado. A redução no consumo de potência ocorre porque o aumento na tensão de limiar diminui a dissipação da corrente de sub-limiar, um dos principais componentes da corrente de fuga. O uso dessa técnica provoca um aumento no tempo de troca de valores de um transistor e, como consequência, há uma redução na frequência de *clock* de operação do circuito (Intel, 2005).

3.5 Estado da Arte na Redução do Consumo de Potência

A redução e a estimativa do consumo de energia têm sido alvo de diversas pesquisas atualmente. Tanto na indústria quanto na área acadêmica, técnicas que buscam reduzir e analisar a potência e a energia dissipada nos processadores estão sendo estudadas e incorporadas aos sistemas computacionais. Esta seção apresenta algumas das principais técnicas utilizadas para reduzir a potência dissipada.

3.5.1 *Pipeline Gating*

As técnicas de previsão de desvios e execução especulativa aumentam o IPC, mas também trazem custos pelo trabalho gasto inutilmente, já que um erro na previsão faz com que muitas instruções pertencentes ao caminho errado nunca sejam finalizadas, ocupando de forma desnecessária os recursos do processador. Desta forma, se o número de instruções inúteis dentro do *pipeline* for menor, o consumo de potência do processador também será reduzido. Nesse âmbito um mecanismo de hardware denominado *Pipeline Gating* (Manne, Grunwald e Klauser, 1998) foi proposto, o qual controla a especulação no *pipeline* reduzindo a potência consumida.

A Figura 3.3 mostra a distribuição da potência dissipada pelos componentes de um processador PentiumPro. Como pode ser observada, a atividade no *pipeline* é responsável pela porção dominante na dissipação da potência, sendo a busca e a decodificação os estágios que mais dissipam potência.

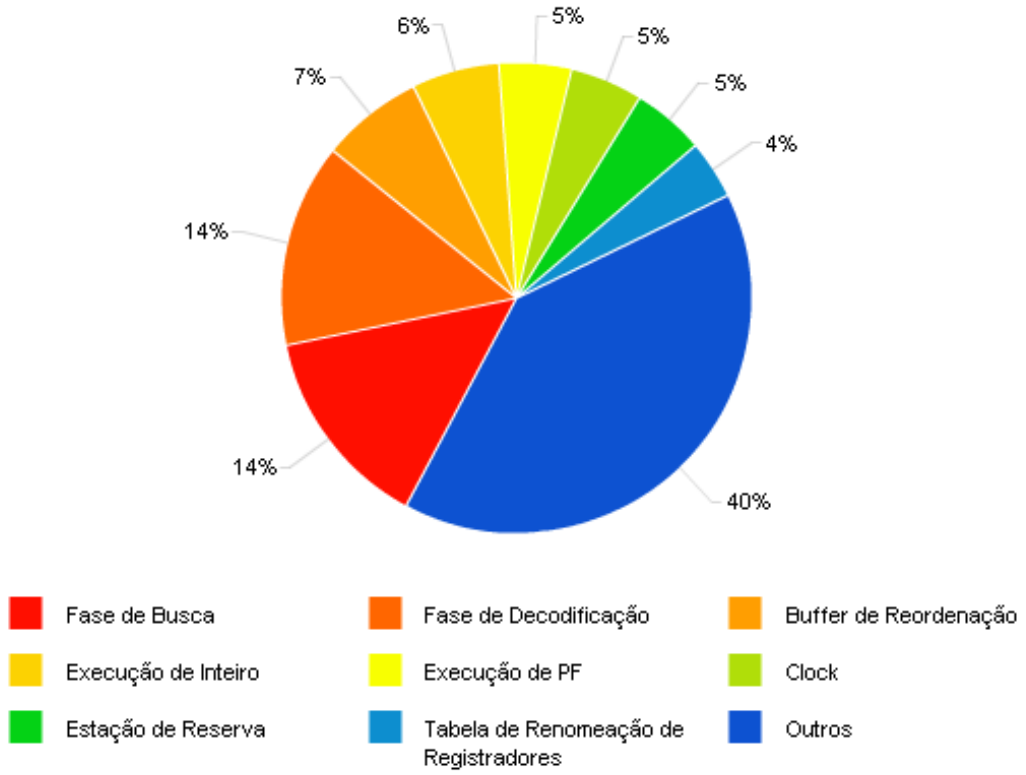


Figura 3.3. Consumo de potência por componentes em um processador PentiumPro

O mecanismo proposto utiliza um "estimador de confiança" para ter certeza que o fluxo de execução do programa determinado pela previsão de um desvio é correto. Caso uma baixa confiança seja estimada, há grandes chances da previsão ter sido incorreta. Quando o

número de desvios de baixa confiança excede um limite estipulado, o estágio de busca do *pipeline* é fechado, permanecendo neste estado de 2 a 4 ciclos.

Os mecanismos para estimar a confiança existem a um custo baixo, reduzindo o trabalho de programas com grandes taxas de erro na previsão sem causar impacto no desempenho. Os testes mostraram que a maior parte do trabalho extra no *pipeline* ocorre nos estágios de busca e decodificação. Então, desligando o estágio de busca o impacto na redução de consumo será maior. Com o estágio de busca fechado, não é possível que novas instruções sejam trazidas da memória. Apesar de não ser possível realizar novas buscas, as instruções que já estavam na fila de busca continuam sendo decodificadas e remetidas normalmente.

Os resultados obtidos com o mecanismo de *Pipeline Gating* mostraram uma redução de 38% de instruções do caminho errado dentro do *pipeline* com uma redução de desempenho de aproximadamente 1%.

3.5.2 *Branch Cache*

Seguindo o mesmo foco de redução de consumo por meio da redução de instruções especulativas que são buscadas pelo processador, mas nunca finalizadas, o mecanismo *Branch Cache* (Jaggar, 1996) foi desenvolvido. O mecanismo propõe desabilitar o bloco do previsor de desvio baseado na informação de quando ocorrerá o próximo desvio. Essa informação é fornecida pelo próprio previsor de desvios em uma previsão anterior e indica quando o previsor poderá ser religado. Considerando que a previsão esteja correta, entre o acesso anterior ao previsor e o momento em que a próxima instrução será trazida da memória, nenhum acesso ao previsor será requerido e ele pode ser desligado.

Para o funcionamento do mecanismo, o processador deve prover uma *cache* de desvio, na qual cada bloco deve manter um rótulo, o valor do próximo desvio, o endereço da instrução alvo e a própria instrução alvo. O valor do próximo desvio indica quando a próxima instrução de desvio será encontrada no fluxo de instruções inseridas no *pipeline* do processador. Esse valor corresponde aos bits menos significativos do PC. A Figura 3.4 ilustra a representação em blocos deste mecanismo.

A *Branch Cache* usualmente permanece em um estado de baixo consumo de potência. Um detector de instruções de desvio monitora as instruções que entram no estágio de busca do *pipeline*. Quando uma instrução de desvio é encontrada, o detector dispara uma série de ações que resultam na escrita dos dados referentes a essa instrução na *branch cache*. Primeiro, os bits menos significativos do PC da instrução de desvio anterior à atual no fluxo de

instruções são armazenados. Então, quando a instrução chega a um estágio do *pipeline* no qual é possível determinar o endereço do desvio, esse endereço também é armazenado. Quando esse endereço é acessado para acessar a instrução alvo, essa instrução também é recuperada e guardada, junto com o PC completo da instrução de desvio. Quando esses quatro dados estiverem disponíveis, eles são escritos na *branch cache*.

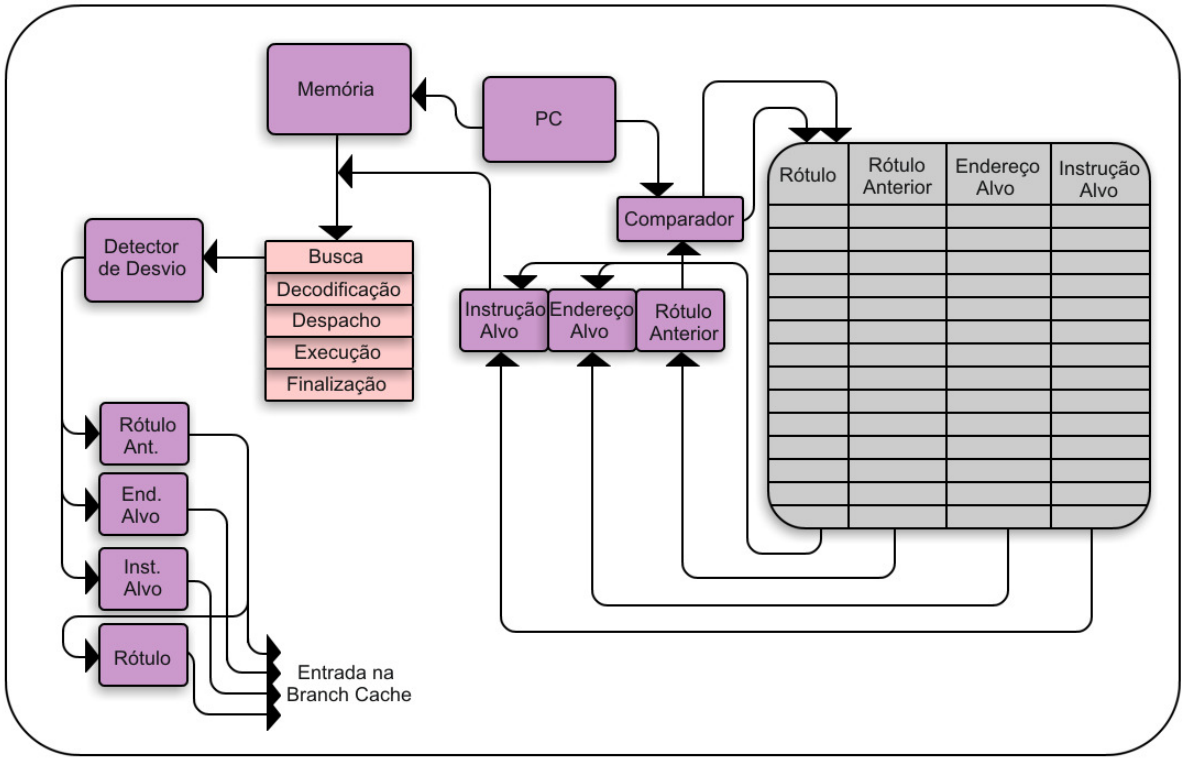


Figura 3.4. Branch Cache

Ao encontrar uma instrução de desvio, a *branch cache* é acessada e seus dados são trazidos para registradores. A cada alteração no PC, o valor da próxima instrução que foi buscada da *branch cache* é comparado com os bits menos significativos do PC. Caso sejam diferentes, o comparador dispara um sinal para desabilitar a *branch cache*. Dessa forma, a *branch cache* não é acessada e a potência não é dissipada por tentativas desnecessárias de encontrar uma instrução de desvio.

Caso os bits menos significativos do PC e o valor da próxima instrução sejam iguais, então a *branch cache* é acessada e os valores da linha correspondente são carregados nos registradores e comparados, verificando se ocorreu um *hit* na *branch cache*. Em caso afirmativo, os dados são trazidos para os registradores, uma vez que foi encontrada a próxima instrução de desvio do fluxo de instruções.

3.5.3 *Pipeline Balacing*

Uma solução arquitetural denominada *Pipeline Balancing* (PLB) foi proposta baseada no ajuste da capacidade de remessa e execução de instruções de acordo com as necessidades do programa em execução (Bahar e Manne, 2001).

A premissa básica do algoritmo é que o comportamento passado do programa indica suas necessidades futuras. Então, baseado no monitoramento da quantidade de instruções que são remetidas por ciclo, a previsão da necessidade futura do programa é realizada. Assim, quando um programa não precisa da capacidade total de remessa do processador, a largura de remessa é reduzida, entrando em um estado de baixo consumo de energia. E, quando o comportamento do programa muda, o processador volta para o estado normal.

O *Pipeline Balancing* necessita de alguns recursos adicionais. O esquema de monitoração deve ser simples a ponto de consumir o mínimo de potência necessária. Adicionalmente, a técnica empregada não pode sacrificar o desempenho do processador para reduzir a potência. Nesse sentido, os dispositivos no modo de baixo consumo devem garantir o funcionamento de todas as operações definidas no conjunto de instruções do processador. Então, pelo menos uma unidade funcional para cada tipo de instrução deve permanecer ligada.

Os valores limite que determinam quando o estado do processador deve ser alterado foram obtidos através de experimentos e análises do número de instruções remetidas por ciclo. O processador utilizado como base foi o Alpha 21264, cujas unidades funcionais foram duplicadas para os testes. Dessa forma, a largura de remessa passou de 4 para 8 com 8 instruções de inteiros, 4 de ponto flutuante e 4 operações de memória podendo ser executadas por ciclo. A fila de remessa e as unidades funcionais foram divididas em 2 agrupamentos. Cada agrupamento contém 4 posições da fila de remessa, 4 unidades de execução de inteiros, 2 de ponto flutuante e 2 de memória. Assim, quando o processador entra no modo de baixo consumo, parte ou todo um agrupamento é desabilitado, resultando em economia de potência.

A técnica PLB empregada no processador base pode reduzir a largura de remessa de 8 para 6 ou 4. Reduções maiores ocasionam perdas de desempenho. A maior economia de potência é alcançada com largura 4, mas o processador entra menos vezes nesse estado. O contrário ocorre com largura 6, a qual poupa menos potência, mas frequentemente os programas entram nesse modo. Para evitar perdas de desempenho, quando o processador está em um dos modos de baixa potência e precisa aumentar a largura de remessa, o processador sempre volta para o modo de operação normal, com a largura de remessa igual a 8.

Os testes mostraram que, utilizando o modo de baixo consumo de potência com largura 4, foi possível economizar 20% de potência na unidade de execução e 35% na fila de remessa, resultando em uma redução de 12% total no chip. Já com largura 6, as taxas de economia na potência foram de 10%, 17% e 6% nas unidades de execução, fila de remessa e chip respectivamente.

3.5.4 *Clock Gating*

Uma das maiores preocupações em relação ao consumo de potência de um processador é em relação à dissipação de potência gerada pelo sinal de *clock*. Isso porque o *clock* é responsável por alimentar a maioria dos componentes do processador e seu valor é alterado a cada ciclo, gerando muita potência de chaveamento. Considerando todos os sinais de *clock* de um processador, o consumo pode chegar a 30%-35% do consumo total do sistema (Gowan, Biro e Jackson, 1998). Nesse sentido, uma forma bastante difundida para a redução desse consumo é a utilização da técnica *Clock Gating* (Wu, Pedram e Wu, 1997).

A representação da não utilização do *Clock Gating* é apresentada na Figura 3.5, na qual um registrador recebe dados oriundos de um multiplexador. Os valores no registrador são atualizados com os valores do circuito combinacional caso o sinal de habilitação do mux (*ENB*) esteja ativo e o laço indo de *Q* até *E1* não esteja ativo. Caso contrário, o valor do *flip-flop* não será alterado, entrando em modo de manutenção (*hold*). A implementação do laço faz com que potência desnecessária seja consumida.

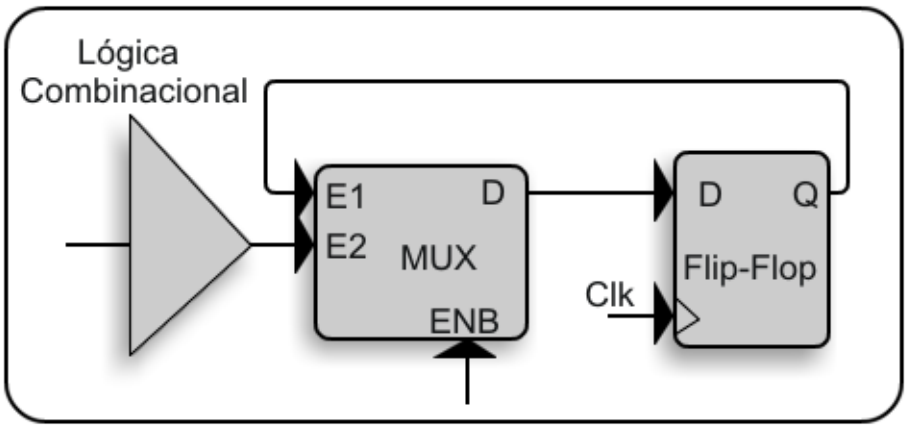


Figura 3.5. Exemplo de circuito que não utiliza Clock Gating

Já com a utilização do *Clock Gating*, apresentada na Figura 3.6, o multiplexador é removido e o próprio sinal de *clock* forma um circuito com o controle *ENB*, fazendo com o que o *clock* só chegue até o registrador se *ENB* estiver ativo. Dessa forma o sistema consegue

selecionar partes do *clock* para serem interrompidas e não realizar a troca de valores do *clock* na parte selecionada. Isso reduz as atividades de chaveamento e, consequentemente, o consumo de potência.

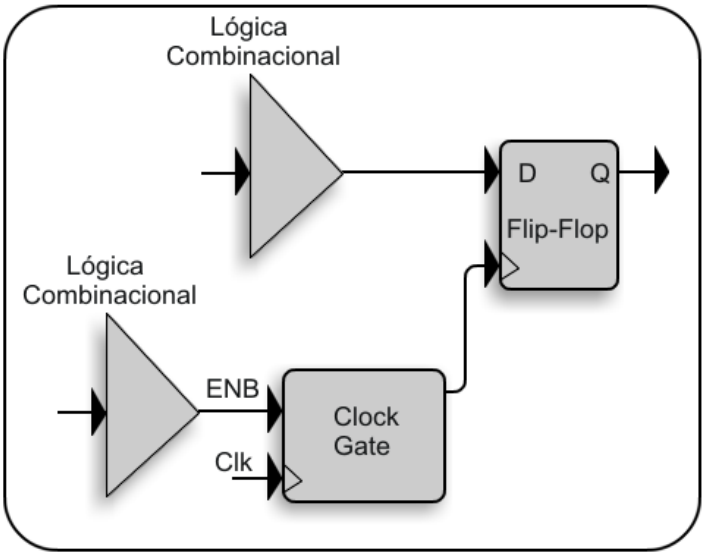


Figura 3.6. Exemplo de circuito que utiliza Clock Gating

O próprio circuito que implementa o *Clock Gating* ocupa espaço e dissipa calor. Então, uma seleção de quais partes do circuito deve receber a implementação da técnica deve ser realizada. Normalmente não é desejável utilizar a técnica em um conjunto pequeno de *flip-flops*, uma vez que a área e o gasto com o circuito responsável por realizar o *Clock Gating* tornam-se mais substanciais do que a redução de consumo proporcionada pelo emprego da técnica nesses casos.

O *Clock Gating* vem sendo bastante explorado. Huda, Mallick e Anderson (2009) propõe a utilização de tal técnica para a redução do consumo de potência em FPGAs (*Field Programmable Gate Array*). A proposta é a implementação do *Clock Gating* com pequenas alterações de hardware, nas quais subconjuntos da rede de *clocks* são controlados por um sinal de habilitação. Esse sinal é adicionado na própria árvore do *clock*, o que representa uma menor demanda por roteamento que se fosse implementado nos blocos lógicos do FPGA.

O controle dinâmico da técnica é apresentado em (Chang et al., 2007). Para a utilização do *Clock Gating* dinâmico, cada bloco do sistema é modelado como uma máquina finita de estados, na qual os estados que o bloco pode assumir são classificados em duas classes: (i) estado ocioso e (ii) estado de trabalho. Se o bloco estiver no estado ocioso por alguns ciclos, ele não precisa de *clock*. Assim, o sistema desabilita o *clock* do bloco caso, além do bloco estar no estado ocioso, não houver requisição de utilização de componentes do bloco no barramento do sistema.

Já Li et al. (2004) utiliza uma técnica determinística para determinar quando habilitar o *Clock Gating*. A abordagem parte da premissa de que para muitos estágios do *pipeline* de um processador moderno, é possível determinar quantos ciclos passarão até que alguns blocos de circuito sejam utilizados. Assim, o *Clock Gating* determinístico desabilita blocos não utilizados no momento. Os problemas desse método é que ele é completamente dependente da arquitetura utilizada e os projetistas devem ter completo conhecimento da arquitetura, principalmente do *pipeline*.

3.5.5 Economia por Meio da Redução de Acessos Desnecessários

Outra maneira de obter redução no consumo de potência é evitar que acessos desnecessários a determinados blocos do processador ocorram. Em (Hu et al., 2003) a *trace cache* é utilizada para reduzir a energia gasta na unidade de busca. A *trace cache* armazena instruções na ordem dinâmica de execução para agilizar o processo de busca. Para minimizar a perda de desempenho ocasionada por uma falta na *trace cache*, tanto a *trace cache* quanto a *cache* de instruções são acessadas simultaneamente, aumentando o consumo de energia na unidade de busca.

Para evitar este acesso simultâneo e consequentemente diminuir o consumo, foi proposta uma previsão da direção dinâmica baseada na *trace cache*, a qual é potencializada por um previsor dinâmico de direção de busca. Esse previsor antevê se o próximo traço na *trace cache* está certo ou não, baseado na informação histórica obtida em tempo de execução. É esta previsão que controla a unidade de busca para acessar somente a *trace cache* ou a *cache* de instruções, evitando o acesso simultâneo. Os experimentos mostraram uma média de redução no consumo de energia na unidade de busca de 38,5% com perda de desempenho de 1,8% quando comparado às *trace caches* tradicionais.

Outro trabalho relacionado propõe utilizar previsores para cada bloco do processador que indicam se o processamento realizado por esse bloco será útil ou não (Musoll, 2003). Como o resultado da previsão pode não ser correto, o previsor deve ter capacidade de reiniciar o acesso ao bloco caso faça uma previsão incorreta. A ocorrência de tal procedimento não deve ser frequente, uma vez que a latência para reiniciar o bloco pode aumentar muito o número de instruções executadas por ciclo.

3.6 Estimativa do Consumo de Potência

Com o consumo de potência tornando-se o maior fator limitante para o desempenho dos computadores, torna-se importante precisar a quantidade de potência dissipada pelo sistema em estágios iniciais do projeto. Decisões de projeto também devem ser feitas com base em estimativas do consumo, tais como propriedades e configurações de hardware. Decisões no nível de projeto arquitetural possuem grande influência na dissipação de potência, sendo responsáveis pela maior parte do consumo do sistema implementado (Stammermann, 2001).

O presente trabalho se enquadra justamente nesse escopo, buscando formas de estimar o consumo de diferentes configurações arquiteturais para auxiliar o projeto e a fabricação de processadores superescalares. Nesse sentido, as seções a seguir apresentam alternativas para a estimativa de consumo de potência em diferentes níveis de abstração.

3.6.1 CACTI

A CACTI (Thoziyoor, Muralimanohar e Jouppi, 2007) é uma biblioteca que integra modelos de acesso, área e consumo de potência de memórias SRAM e DRAM. A biblioteca foi escrita na linguagem C++ e possui modelos analíticos para medir a potência dinâmica e a corrente de fuga em nível arquitetural. A estrutura de memória utilizada pela CACTI é apresentada na Figura 3.7.

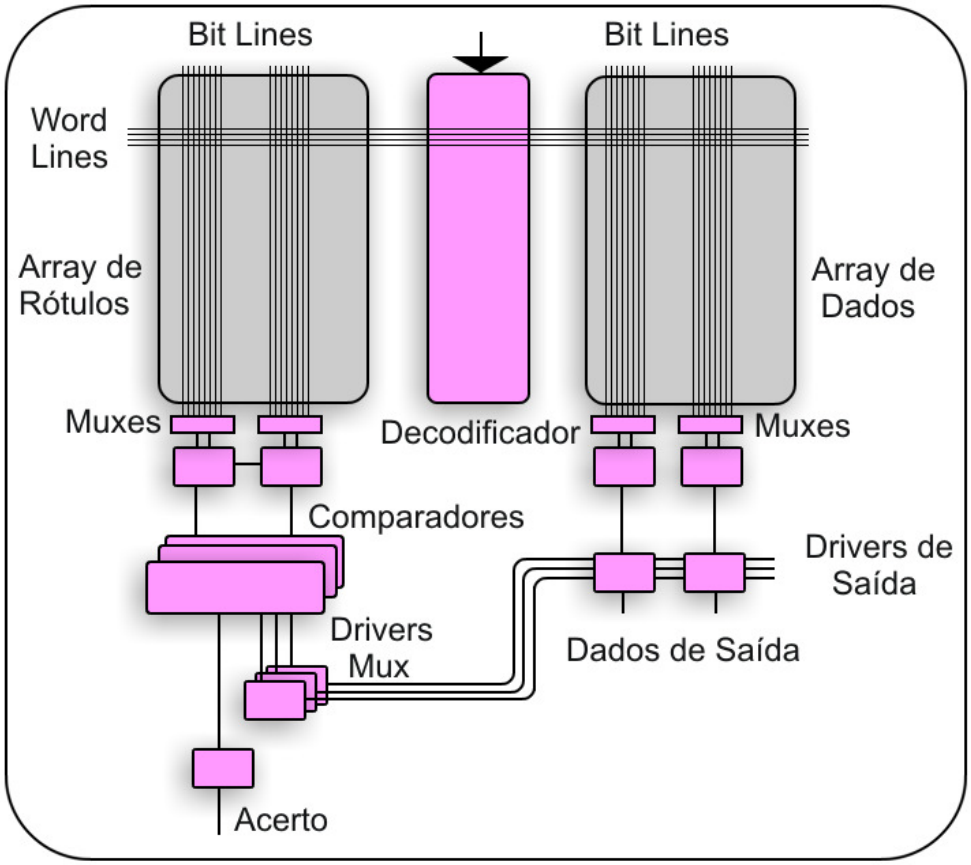


Figura 3.7. Modelo de memória cache utilizada pela CACTI

Cada bloco da *cache* é organizado em forma de vetor e é acessado por meio de *wordlines*, ativadas por um decodificador. A quantidade de *wordlines* na organização é indicada pela quantidade de linhas na *cache*. Apenas uma *worline* é ativada por vez, indicando um acesso àquela linha. As células da memória são associadas à *bitlines*, cujos valores dependem das células ativadas pela *wordline*. Esses valores são monitorados por amplificadores, que verificam as alterações de sinal nas *bitlines*.

As informações lidas no vetor de rótulos são comparadas com o rótulo do endereço. Uma *cache* com *n* conjuntos, necessita de *n* comparadores. Os resultados dessa comparação indicarão se houve um acerto (*hit*) ou uma falta (*miss*) na memória e enviarão as informações para multiplexadores de saída, responsáveis por transmitir o dado de saída da *cache*.

Os modelos analíticos da biblioteca foram obtidos por meio da decomposição de todo o circuito em circuitos resistor-capacitor (RC) equivalentes, com modelos de temporização e consumo baseados em simples equações RC.

A CACTI possui uma versão *online*, o que a torna acessível para a comunidade. Além disso, o código dela é disponível para alterações ou integração com outras ferramentas, como o *Sim-Wattch*, apresentado na Seção 3.6.2.

3.6.2 *Sim-Wattch*

A ferramenta *Sim-Wattch* (Brooks, Tiwari e Martonosi, 2000) permite a análise e a otimização do consumo de energia em processadores superescalares em nível arquitetural, estimando o consumo de um processador MIPS IV. O *Sim-Wattch* é uma extensão do *Sim-Outorder*, presente no *framework Simplescalar* (Burger e Austin, 1997), que possibilita a análise de diferentes parâmetros de arquiteturas superescalares, como previsão de desvios, execução especulativa e *cache*.

O *Sim-Wattch* fornece as mesmas análises do *Simplescalar*, com o adicional da estimativa do consumo de potência das estruturas envolvidas durante a simulação. A estimativa de potência fornecida pela ferramenta é baseada em um conjunto de modelos parametrizados para diferentes estruturas de hardware e contadores de recursos utilizados por ciclo, gerando uma estimativa em nível de ciclo. A Figura 3.8 apresenta a estrutura geral do *Sim-Wattch*, bem como as interfaces de integração do simulador de desempenho e dos modelos de consumo. Alguns desses modelos são obtidos a partir da biblioteca CACTI.

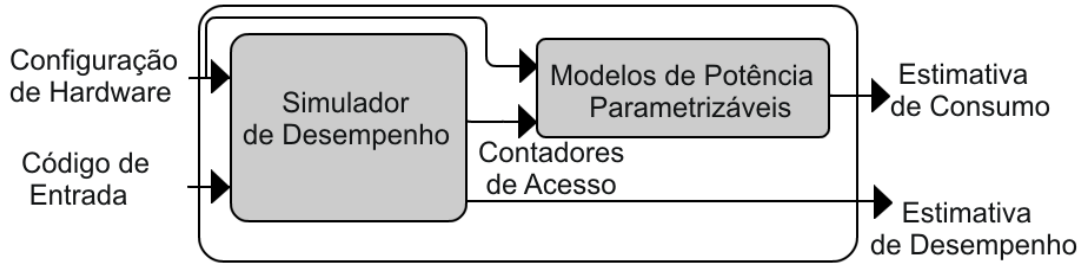


Figura 3.8. Estrutura geral do Sim-Wattch

O simulador de desempenho utilizado pela ferramenta é baseado em um processador com execução fora de ordem e *pipeline* de cinco estágios: busca, decodificação, remessa, execução e finalização. Execução especulativa também é suportada. A penalidade para previsões de desvio incorretas é de sete estágios.

Ao iniciar a simulação da execução de um programa, é computada a dissipação de potência base de cada unidade modelada pelo *Sim-Wattch*. Esses valores são escalados com os contadores de acesso ao final da simulação, indicando a potência final calculada pela ferramenta.

3.6.3 *PowerSMT*

Outra ferramenta voltada para a análise do consumo de potência de processadores em nível arquitetural é o *PowerSMT* (Gonçalves, 2008), voltado para a análise de processadores SMT (*Simultaneous Multithreading*), capazes de executar instruções provenientes de diferentes fluxos simultaneamente.

O *PowerSMT* foi desenvolvido com base no simulador SS_SMT (Gonçalves, 2000), uma alteração do *Simplescalar* utilizado para simular e avaliar arquiteturas SMT. O SS_SMT replica algumas das estruturas existentes no *Sim-Outorder* para possibilitar a execução de diferentes programas paralelamente, compartilhando alguns recursos entre esses programas, assim como uma arquitetura SMT real.

O *PowerSMT* utiliza o mesmo padrão de implementação do modelo de consumo de potência do *Sim-Wattch*, entretanto esse modelo sofreu alguns ajustes como a atualização da biblioteca CACTI utilizada para a versão 4.0, uma vez que a utilizada pelo *Sim-Wattch* é a versão 1.0. É possível afirmar que o *PowerSMT* é uma integração entre o modelo de consumo proposto pelo *Sim-Wattch* e o modelo arquitetural herdado do SS_SMT (Gonçalves e Gonçalves, 2008).

3.6.4 *PowerTimer*

A ferramenta *PowerTimer* (Brooks et al., 2003), desenvolvida pela IBM, também realiza a estimativa do consumo de energia em nível arquitetural. Ela é utilizada para análise de consumo e desempenho na fase inicial do projeto de um processador. O *PowerTimer* é uma extensão do simulador *Turandot* (Moudgill, Wellman e Moreno, 1999), que modela um processador superescalar com execução fora de ordem, *caches* nível 1 e 2 e memória principal. O componente principal da ferramenta é um conjunto de funções de consumo de energia a ser utilizado juntamente com um simulador microarquitetural.

No *PowerTimer*, os modelos de potência são baseados em simulações realizadas em nível de circuito em um processador PowerPC. Esses modelos são controlados pela tecnologia empregada na simulação e por parâmetros arquiteturais, tais como tamanho de *buffers*, largura de banda e latência do *pipeline*. O modelo de potência pode ser utilizado de duas formas diferentes. Na primeira, o simulador de desempenho é usado sozinho produzindo estatísticas de utilização de recursos. Então essas estatísticas são processadas pelo modelo de potência para gerar a média do consumo obtido. No outro modo, os modelos de potência são acessados durante a simulação ciclo a ciclo, provendo características de consumo mais detalhadas.

A ferramenta abordada utiliza um conjunto de fases de funções de potência que são refinadas conforme os modelos de projeto e simulação evoluem. Com o progresso do projeto, equações analíticas derivadas dos dados de potência dos blocos construídos anteriormente são utilizadas pelas estruturas de circuitos mais novas dentro da estrutura alvo.

3.6.5 *DesignPower*

DesignPower (Toshiba, 1997) é uma ferramenta desenvolvida pela Synopsys para prover um ambiente de análise de potência em diferentes estágios de um projeto. A ferramenta fornece estimativas de projetos em HDL (*Hardware Description Language*) durante a fase de projeto por meio de análises probabilísticas e estimativas mais precisas em nível de portas lógicas por meio de análises baseadas em simulação.

A ferramenta faz a estimativa da potência de chaveamento e da potência de curto-circuito. Ela recebe como entrada uma *netlist* em nível de porta, uma biblioteca específica da tecnologia utilizada e atividade de chaveamento ocorrida durante uma simulação. *DesignPower* pode operar em modo probabilístico, no qual as atividades de chaveamento são monitoradas apenas nas entradas por simulação em nível RTL (*Register Transfer Level*) ou em modo de simulação, que recebe a atividade de chaveamento de todo o circuito projetado a partir de simulações em nível de porta. Como resultado, a ferramenta fornece relatórios de consumo de todo o projeto, de blocos específicos ou mesmo de células e interconexões individuais. As informações necessárias para a execução do simulador e sua saída são apresentadas na Figura 3.9.

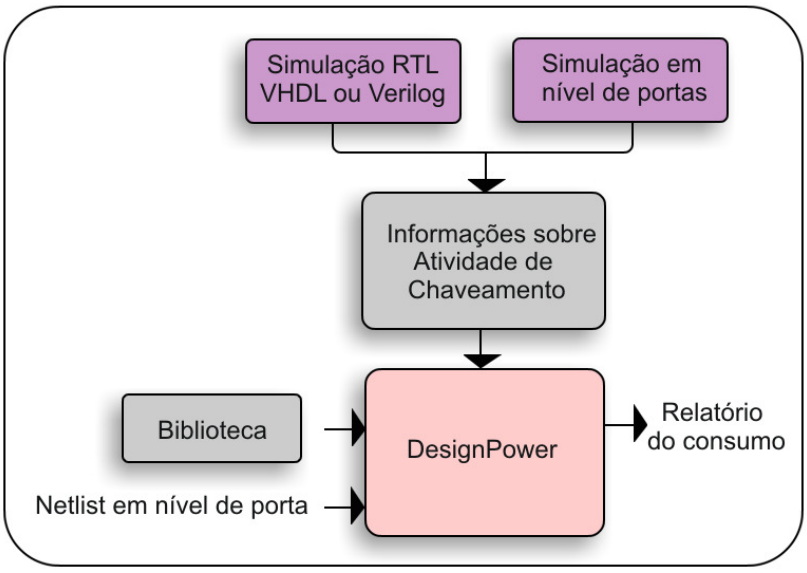


Figura 3.9. Entradas e saída do DesignPower

O *DesignPower* faz a análise do consumo em nível de portas lógicas. A ferramenta não consegue fazer a análise de um circuito descrito em RTL mesmo no modo de operação probabilístico. Mesmo nesse modo, é necessário o mapeamento das portas lógicas do circuito para que a análise do consumo possa ser realizada.

3.6.6 *XPower*

A ferramenta gráfica *XPower* (Wenande e Chidester, 2001) da empresa Xilinx é utilizada para a estimativa de consumo de potência em nível de porta lógica. O cálculo do consumo é realizado com base na observação de que a maior parte do consumo em circuitos CMOS ocorre por meio da atividade de chaveamento. Nesse sentido, outra ferramenta é necessária para gerar arquivos com o mapeamento de todas as entradas e interconexões do circuito, além de todos os estímulos gerados durante uma determinada simulação. Isso significa que a síntese do circuito deve ser realizada antes da utilização do *XPower*.

O *XPower* recebe o mapeamento das entradas e um arquivo com os parâmetros tecnológicos do circuito. Com essas informações, a ferramenta cria uma representação hierárquica do projeto, dividida em sinais, *clock*, lógica e entrada e saída. A cada um desses níveis hierárquicos é associada uma capacitância correspondente. Então, o *XPower* calcula o consumo somando a potência dissipada por cada elemento utilizado no projeto associando o modelo de capacitância aos estímulos da simulação. A Figura 3.10 exemplifica o método de estimativa utilizado pela ferramenta.

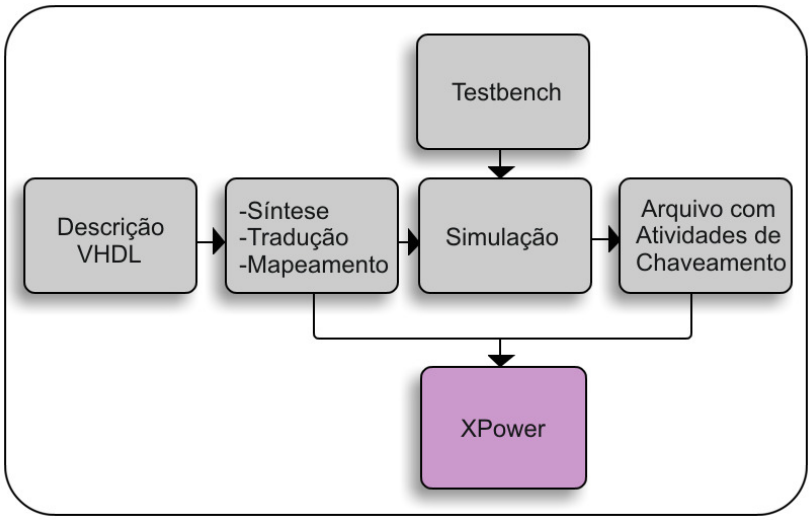


Figura 3.10. Modelo de estimativa de consumo do XPower

O *XPower* provê resultados precisos para projetos pequenos, mas o aumento na complexidade e número de sinais do circuito pode tornar a análise impraticável (Becker, Huebner e Ullmann, 2003).

3.6.7 *SPICE*

SPICE (OrCAD, 1998) é uma ferramenta utilizada para a simulação de circuitos MOS (*Metal Oxide Semiconductor*) em nível de transistor. Por realizar simulações no nível mais baixo possível, o simulador consegue prover resultados mais precisos que ferramentas que operam em outros níveis. O *SPICE* é um simulador de propósito geral, não sendo específico para a previsão de consumo de potência, mas também utilizado para este fim. Além da potência também é possível prever o atraso e o comportamento do circuito.

Um arquivo de entrada contendo uma *netlist* do circuito deve ser especificado e passado como parâmetro para o *SPICE* iniciar o processo de simulação. A estrutura básica de uma *netlist* de entrada do *SPICE* consiste de um programa principal e um ou mais sub-módulos opcionais. Os sub-módulos podem ser utilizados para facilmente alterar configurações e opções da *netlist* de entrada no momento da realização de testes.

O *SPICE* é utilizado como base para diversos simuladores comerciais tais como o *HSPICE* ou o *PSPICE*, que adicionam interface gráfica, visualização gráfica dos resultados e outros recursos. Apesar de apontar resultados mais precisos que ferramentas que fazem a análise em outros níveis do projeto, as simulações utilizando o *SPICE* podem ser demasiadamente longas. Os padrões de entradas para as simulações também podem ser difíceis de prever dependendo do circuito.

3.7 Considerações Finais

O presente capítulo apresentou os principais conceitos relacionados ao consumo de potência em sistemas de computadores. Técnicas que buscam reduzir o consumo também foram introduzidas bem como ferramentas que fazem a análise da potência dissipada em diversos níveis de um projeto.

É válido ressaltar que os softwares disponíveis para a estimativa do consumo de potência não estão limitados às ferramentas aqui apresentadas. Procurou-se introduzir ferramentas que realizam a previsão em diferentes níveis do projeto ou que apresentam diferentes técnicas para realizar esta previsão.

Com base nas ferramentas analisadas notou-se a escassez de ferramentas *open-source* que fazem a análise do consumo com base em descrições em linguagem de descrição de hardware. Neste contexto é apresentada uma ferramenta contendo componentes arquiteturais VHDL para a estimativa do consumo de potência dinâmica em processadores superescalares.

CAPTULO

4

D-Power

O consumo de potência é um dos principais fatores que limitam o desenvolvimento de sistemas de computadores restringindo características como frequência de *clock*, quantidade de transistores no chip, tempo de autonomia de bateria em sistemas portáveis, entre outras. Assim, é importante estimar a potência consumida em estágios preliminares do desenvolvimento para garantir que as restrições do sistema sejam atingidas.

Como apresentado na Seção 3.6, diversas ferramentas buscam prever o consumo de potência em diferentes fases de um projeto. Algumas ferramentas propõem a estimativa em níveis mais baixos, como de porta e transistores. Embora seja fundamental o uso de tais ferramentas, é importante que as restrições de consumo já estejam previstas e analisadas quando o desenvolvimento do projeto atingir tais níveis. Como tais ferramentas requerem descrições em nível de transistor ou porta, a estimativa de potência ocorre em um estágio tardio do projeto, podendo causar diversas penalidades de tempo caso as características de consumo estejam longe das ideais (Jevtic, Carreras e Caffarena, 2007). Possíveis alterações no projeto em nível de porta podem significar que todo o projeto tenha que ser refeito.

Ferramentas de análise do consumo em baixo nível também consomem bastante tempo do projeto, então é importante que problemas arquiteturais de consumo já tenham sido identificados e a análise do consumo nessa fase seja apenas refinada e focada em efeitos secundários, tais como dimensões dos transistores, tempo de transição e técnicas de resfriamento.

Já ferramentas em nível arquitetural normalmente são baseadas no simulador *Simplescalar*, o qual aglomera muitos elementos críticos de um *pipeline* superescalar, como as filas de remessa, *buffer* de reordenação e arquivos de registradores, em uma única estrutura unificada denominada Unidade de Atualização de Registradores (RUU – *Register Update Unit*). Tal organização possui relativa disparidade se comparada com implementações reais, nas quais o número de portas nessas estruturas constantemente difere (Ponomarev, Kucuk e Ghose, 2002). Como consequência, uma melhor avaliação do impacto que diferentes configurações desses componentes podem ter na estimativa da dissipação de potência e desempenho do sistema nem sempre é possível.

Também existe o caso de ferramentas não públicas, algumas por possuírem sua distribuição limitada a projetistas de determinada empresa, outras por serem ferramentas comerciais que requerem a aquisição de licença ou, em alguns casos, de outras ferramentas para que seu uso seja possível.

Nesse sentido, notou-se uma necessidade de ferramentas públicas que apresentem estimativas do consumo de potência, sendo possível alterar suas configurações e analisar quais implicações isso acarretaria no consumo e desempenho do sistema. A D-Power, contendo componentes descritos em VHDL para arquiteturas superescalares é então proposta, visando criar uma fonte de consulta para que projetistas e acadêmicos possam analisar o impacto do emprego de diferentes parâmetros em estágios primários do desenvolvimento de processadores.

Embora a arquitetura superescalar desenvolvida seja completamente funcional, a estimativa do consumo aqui representada é voltada para as *caches* e para o estágio de busca do *pipeline*, estruturas que juntas podem corresponder a mais de 30% do consumo total de uma arquitetura superescalar (Manne, Grunwald e Klauser, 1998). Como a fase de busca é a maior responsável individual pelo consumo nessas arquiteturas, junto com a fase de decodificação, optou-se por priorizar e detalhar a análise nesse estágio.

4.1 Desenvolvimento da Arquitetura Superescalar

O desenvolvimento da arquitetura base foi realizado utilizando a linguagem de descrição de hardware VHDL. Optou-se pela utilização dessa linguagem por ser um dos padrões no projeto de hardware. A principal vantagem da linguagem está relacionada à possibilidade de especificar o comportamento do sistema utilizando descrições mistas em diferentes níveis de abstração, o que provê alta flexibilidade durante a fase de projeto e simulação. Além disso, a

VHDL possui suporte a abordagens hierárquicas, nas quais diferentes elementos de um projeto podem ser desenvolvidos em diferentes níveis (Fornaciari, 1997). Enquanto alguns componentes do projeto podem apresentar descrições comportamentais, outros componentes podem ter suas especificações mais detalhadas e decompostas em unidades mais simples e de nível mais baixo.

Os elementos arquiteturais apresentados neste trabalho tiram proveito dessa funcionalidade da linguagem. Os componentes que possuem estimativa de consumo de potência apresentam descrições mais refinadas, com as unidades de armazenamento, funções lógicas e caminhos de dados minuciosamente detalhados. Já os outros componentes são apresentados em nível comportamental. A abordagem hierárquica utilizada no projeto, entretanto, permite o posterior detalhamento desses elementos, que são completamente independentes, mas que, quando propriamente conectados, formam uma arquitetura superescalar completa.

A arquitetura implementada é baseada na arquitetura do processador MIPS R10000 (Mips, 1996), um microprocessador superescalar RISC (*Reduction Instruction Set Computer*) da família MIPS IV cujo diagrama de blocos básico é apresentado na Figura 4.1.

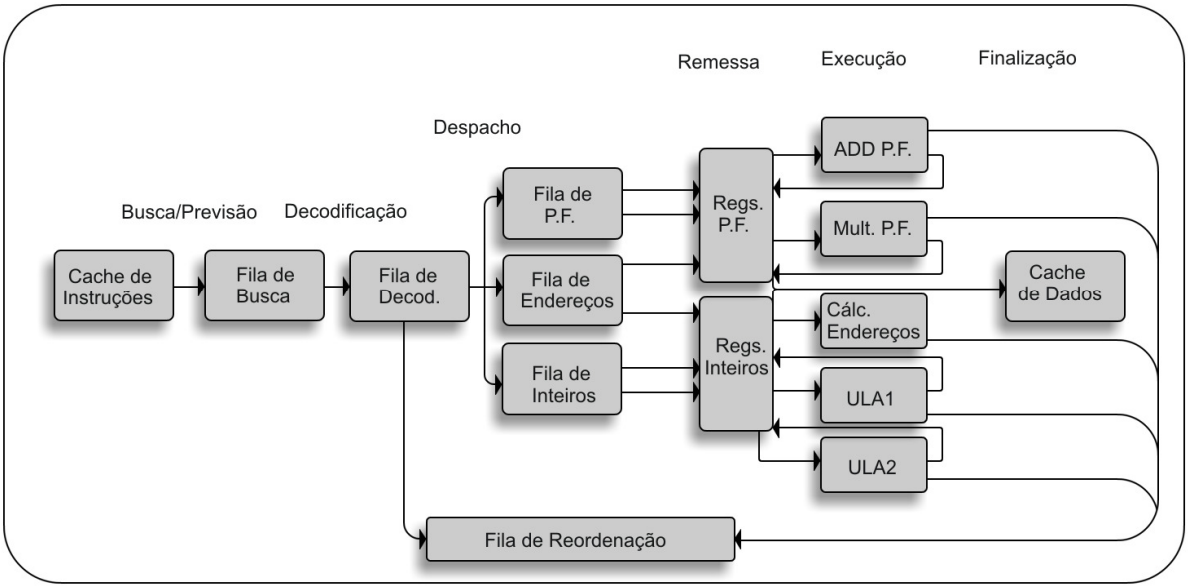


Figura 4.1. Diagrama de blocos do processador MIPS R10000

O *pipeline* do R10000 possui estágios para Busca, Decodificação, Despacho, Remesssa, Execução e Finalização de instruções. O processador busca e decodifica quatro instruções por ciclo de *clock*. Após a decodificação, as instruções são encaminhadas para uma das três estações de reserva presentes: (i) inteiro, (ii) ponto flutuante e (iii) endereços. As instruções são encaminhadas para a estação correspondente de acordo com sua operação. As

três estações de reserva podem enviar uma nova instrução para execução por ciclo e são capazes de armazenar até 16 instruções cada. As cinco unidades funcionais permitem execução especulativa e fora de ordem. Há uma fila de reordenação para manter a consistência durante a finalização das instruções.

As *caches* são associativas por conjuntos com 2 vias e 8 palavras por bloco e separadas para instruções e dados. O algoritmo LRU é utilizado como política de substituição e a técnica *write back* como política de escrita.

Apesar do R10000 ser utilizado como base, a arquitetura implementada é inteiramente configurável, com diversos parâmetros arquiteturais que podem ser definidos. A variação dos parâmetros é importante para a análise de desempenho e consumo de potência com configurações alternadas. Assim é possível estimar quais configurações são aceitáveis dadas certas restrições que um sistema pode possuir. A Tabela 4.1 apresenta os parâmetros arquiteturais que podem ser variados na ferramenta.

Tabela 4.1. Parâmetros variáveis na arquitetura implementada

Tamanho da memória	Tamanho da BTB
Tamanho da *cache* L1	Organização da BTB
Tamanho da *cache* L2	Largura de busca
Palavras por bloco	Tamanho do *buffer* de busca
Associatividade na *cache* L1	Largura de decodificação
Associatividade na *cache* L2	Tamanho do *buffer* de decodificação
Política de substituição nas *caches*	Tamanho das estações de reserva
Política de escrita nas *caches*	Tamanho do *buffer* de reordenação
Previsor de desvio	Quantidade de unidades funcionais

Os primeiros parâmetros tratam do tamanho da memória principal e das memórias *cache*. A hierarquia de memória implementada apresenta dois níveis de *cache*, nas quais é possível configurar a quantidade de palavras por bloco e a associatividade. As organizações mapeamento direto e *cache* associativas por conjunto são apresentadas e podem ser utilizadas. Três políticas de substituição também são descritas: LRU, LFU e FIFO. As técnicas *write through* e *write back* podem ser usadas como políticas de escrita.

A ferramenta também disponibiliza dois previsores de desvio estáticos: Sempre Tomado e Sempre Não-Tomado e o previsor de desvio dinâmico BTB-PHT que faz uso de *branch target buffers* e contadores de saturação para determinar se um desvio será ou não

tomado. Quando utilizado o previsor BTB-PHT ainda é possível definir o tamanho da BTB e sua organização, que pode ser diretamente mapeada ou associativa por conjunto. Além disso, é possível configurar as larguras de busca e decodificação, bem como as quantidades de entradas nos *buffers* de busca, decodificação, reordenação e nas estações de reserva. A quantidade de unidades funcionais também pode ser determinada.

4.1.1 *Caches*

A definição das *caches* partiu de estruturas matriciais utilizadas para o armazenamento de dados. A representação dos *flip-flops* de cada célula das *caches* é realizada por um bit do tipo STD_LOGIC. A escolha por esse tipo se deve ao fato dele ser mais flexível do que o tipo BIT, sendo capaz de representar outros valores além dos valores lógicos '0' e '1', como o valor 'Z', representando alta impedância na célula.

A definição de cada bloco da *cache* é apresentada na Figura 4.2. O bloco é constituído pela informação armazenada, pelo rótulo e um bit de validade. Quando a *cache* possui mais de uma palavra por bloco, todas as palavras são mantidas no campo *informacao*, que é um vetor utilizado para guardar as *n* palavras do bloco.

```
TYPE T_palavra IS ARRAY (0 TO palavras_por_bloco - 1) OF dado;

TYPE T_Cache_Instruction IS
  RECORD
      informacao:      T_palavra;
      rotulo:          T_rotulo_dados;
      bit_validade:    STD_LOGIC;
      bit_modificacao: STD_LOGIC;
      bit_uso:         STD_LOGIC_VECTOR ((bit_uso_dado_width - 1) DOWNTO 0);
      count:           STD_LOGIC_VECTOR (1 TO count_width);
  END RECORD;
```

Figura 4.2. Definição de tipo utilizado como bloco da cache

A estrutura do bloco possui campos que são utilizados de acordo com as configurações da simulação. O campo *bit_modificacao* é utilizado para indicar se o bloco foi ou não modificado quando a política de escrita utilizada é a *write back*. Nessa política, quando um bloco *B1* é alterado na *cache* por uma operação de *store* o bit de modificação desse bloco é ativado para '1'. Assim, quando um novo bloco buscado de um nível superior da *cache* tiver que sobre-escrever *B1*, o bit de modificação será verificado e, como ele possui valor '1', *B1* terá que ser escrito no nível imediatamente superior de *cache* antes de ser substituído.

As políticas de substituição FIFO e LRU utilizam o campo *bit_uso* para verificar qual bloco será substituído. Na política FIFO, sempre que um bloco é buscado, o campo é atualizado, indicando que o bloco que acabou de ser adicionado na *cache* é o mais recente. Já na LRU, o campo é atualizado a cada acerto em um bloco da *cache*, indicando a ordem de acesso aos blocos que pertencem a um conjunto da *cache*. Já o campo *count* é utilizado quanto a política de substituição LFU é utilizada. Esse campo é incrementado sempre que há um acesso no bloco. Ele é utilizado como contador de frequência para que o bloco menos frequentemente usado possa ser substituído quando a técnica LFU é empregada.

A construção de uma *cache* é realizada por meio da interligação de várias estruturas do tipo *T_Cache_Instruction*. Cada bit STD_LOGIC desse tipo representa uma célula da *cache*, que é controlada por um sinal de habilitação. Quando esse sinal está desativado, o dado já armazenado na *cache* permanece em um estado estável, não sofrendo qualquer alteração. A Figura 4.3 apresenta a organização utilizada na construção das memórias *cache*.

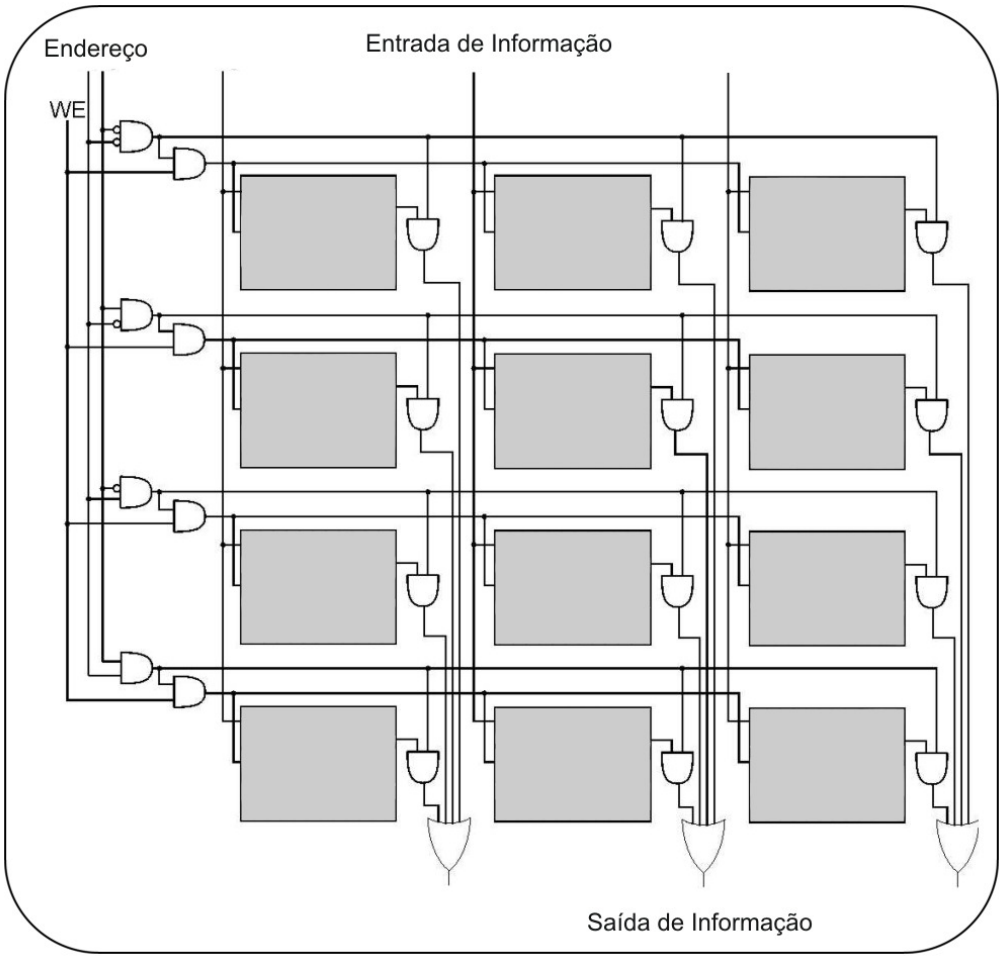


Figura 4.3. Organização de uma memória cache

Para a seleção do bloco é utilizado um decodificador, que ativará uma das linhas da *cache* de acordo com o endereço a ser acessado. A saída do decodificador é ligada a um sinal de escrita, que indica se a operação a ser realizada é de leitura ou escrita. No caso de uma operação de escrita, todos os sinais de habilitação das células da linha acessada serão ativados e os dados na porta de escrita da *cache* serão carregados nessas células. Caso a operação seja de leitura, o sinal de habilitação permanecerá desativado, mantendo os dados na linha acessada imutáveis, mas transferindo-os para a porta de leitura da *cache*.

Os comparadores que verificam se o rótulo do endereço é o mesmo que o rótulo de uma informação na *cache* foi desenvolvido utilizando portas XNOR, que comparam bit a bit ambos os rótulos e retorna '1' caso ambos os sinais de entrada sejam iguais. Se os resultados de todas as portas XNOR do comparador forem '1', assim como o bit de validade, ocorreu um acerto e o dado está presente na *cache*.

A estrutura implementada prevê *caches* L1 separadas para dados e instruções e uma *cache* L2 unificada. As latências para a realização de cada operação nos diferentes níveis da *cache* também foram definidas e parametrizadas. O número de palavras por bloco na *cache* é igual ao número de palavras de cada bloco da memória principal. Dessa forma, quando há troca de dados entre diferentes níveis da memória, todo o bloco é transferido, explorando a localidade espacial normalmente oferecida pelos programas.

4.1.2 Busca

O estágio de busca traz instruções da *cache* e as coloca na fila de busca. A quantidade de instruções buscadas depende da largura desse estágio e da quantidade de posições livres na fila. A largura de busca indica também a quantidade de portas de escrita presente na fila de busca, que é implementada como um *buffer* FIFO circular. Um ponteiro indica a primeira posição vaga na fila, a qual será ocupada quando uma instrução é inserida no *pipeline*. Durante o estágio de busca, as instruções são pré-analisadas para determinar se a operação a ser realizada é um desvio condicional. A forma que a arquitetura trata esses desvios depende da técnica de previsão utilizada.

O previsor Sempre Não-Tomado não realizará qualquer ação caso a instrução apontada pelo PC seja um desvio. Esse registrador simplesmente será incrementado e a próxima instrução a ser trazida da *cache* será a próxima instrução no fluxo de execução do programa. Se o previsor utilizado for o Sempre Tomado, ao detectar uma instrução de desvio, no próximo ciclo o PC passará a apontar para o endereço indicado na instrução.

Já se a técnica de previsão utilizada for a BTB-PHT, ao ser feita a busca de uma instrução, é verificada se há uma entrada para o endereço da instrução buscada na BTB. Para isso, utiliza-se *n* bits menos significativos do PC para indexar a BTB, sendo 2^n o número de linhas do *buffer*. Isso é feito comparando o PC com o endereço da instrução armazenada na linha acessada da BTB. Caso a comparação retorne um resultado positivo há um acerto na BTB, o que indica que a instrução é de desvio e que o endereço alvo dele é conhecido. Dessa forma, a previsão é realizada de acordo com a entrada da PHT no endereço da BTB. O previsor implementado utiliza contadores de saturação de dois bits, que relacionam cada instrução a quatro estados, como apresentado na Figura 4.4.

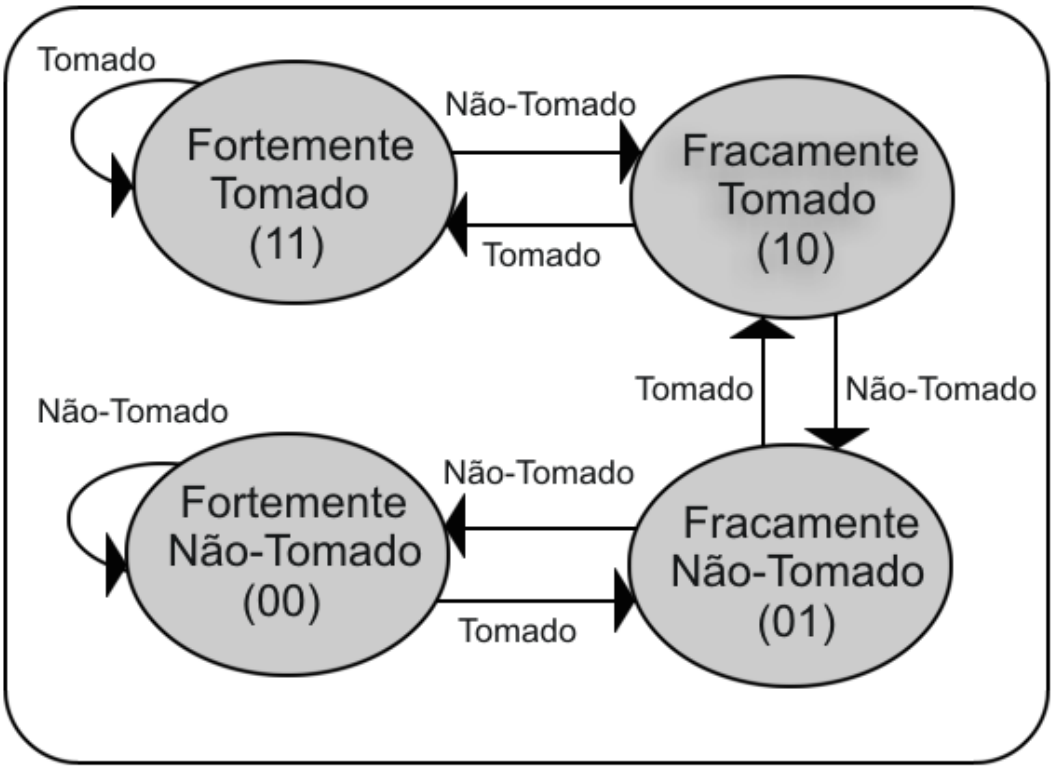


Figura 4.4. Máquina de estados do contador de saturação de 2 bits

Cada contador pode assumir quatro estados: fortemente tomado (“11”), fracamente tomado (“10”), fracamente não-tomado (“01”) e fortemente não tomado (“00”). Os estados “11” e “10” consideram que o desvio será tomado, enquanto os estados “01” e “00” consideram que o desvio não será tomado. O estado inicial de cada contador é “01”. Dessa forma a primeira previsão de cada instrução é não tomar o desvio. O contador é atualizado após a execução da instrução de desvio. Se a previsão foi correta e o desvio realmente não deveria ter sido tomado, o contador é decrementado e o estado do contador passa a ser “00”. Se o desvio foi previsto incorretamente e deveria ter sido tomado, o contador é incrementado

e passa para o estado “10”. Dessa forma, em uma próxima busca dessa instrução, o desvio será previsto como tomado. A utilização de dois bits no contador garante que desvios previstos como fortemente tomado e fortemente não-tomado necessitem de duas previsões incorretas para que a previsão seja alterada. A estrutura utilizada pela BTB é apresentada na Figura 4.5.

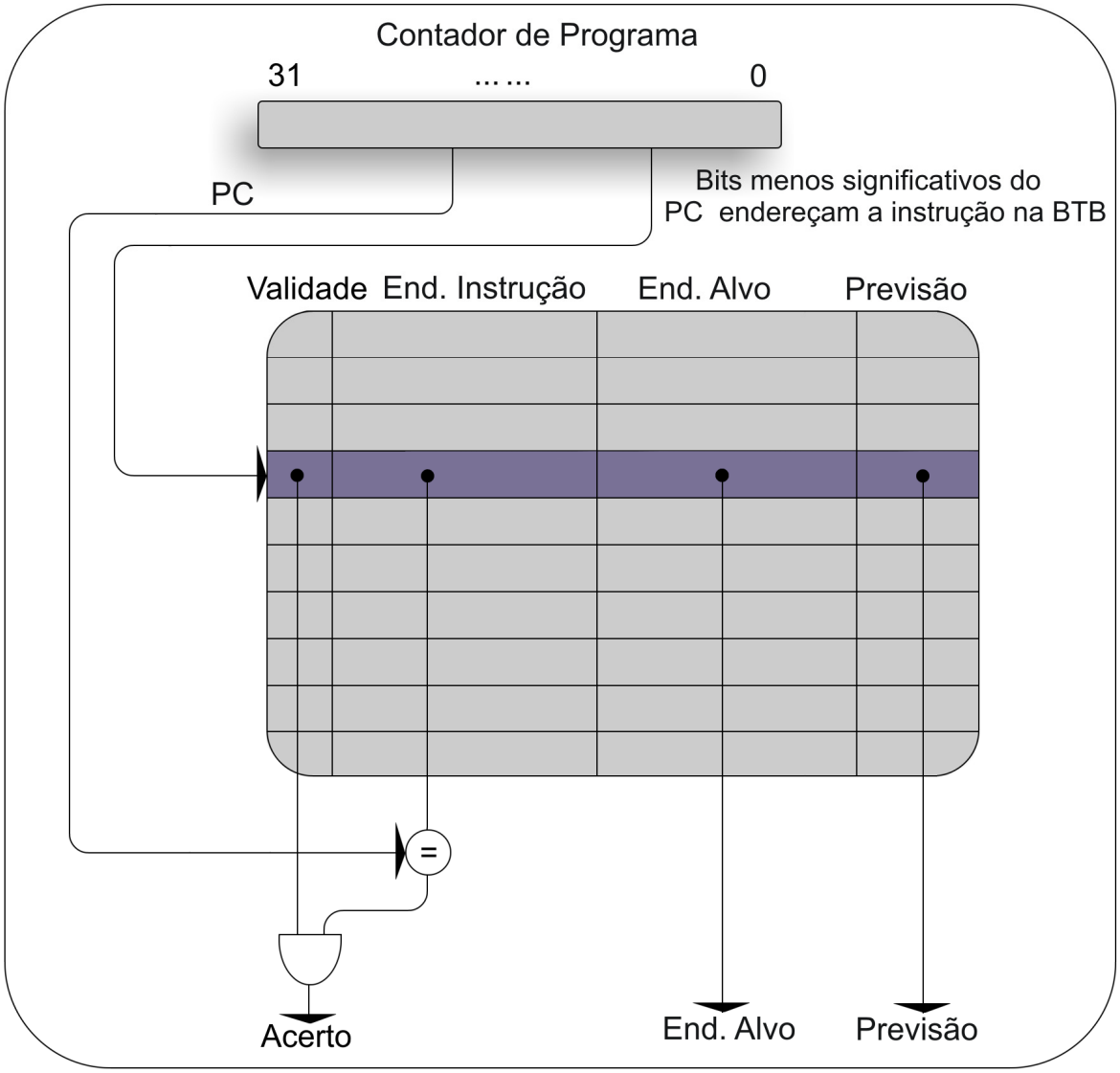


Figura 4.5. Estrutura utilizada na BTB

Cada linha da BTB contém o endereço da instrução, o endereço alvo do desvio e os 2 bits da previsão. A BTB pode ser mapeada diretamente ou utilizar associatividade por conjunto. No último caso a política de substituição pode ser escolhida entre a FIFO, a LFU e a LRU, assim como nas *caches* implementadas. A definição de cada bloco da BTB na arquitetura implementada é apresentada na Figura 4.6.

```
TYPE T_btb_row IS
 RECORD
     next_pc:  T_program_counter;
     rotulo:   T_program_counter;
     previsor: STD_LOGIC_VECTOR ((qt_bit_previsao - 1) DOWNTO 0);
     bit_uso:  STD_LOGIC_VECTOR ((bit_uso_btb_width - 1) DOWNTO 0);
     count:    STD_LOGIC_VECTOR (1 TO count_width);
 END RECORD;
```

Figura 4.6. Definição de tipo utilizado como bloco da BTB

O endereço da instrução é armazenado no campo *rotulo*. O endereço alvo da instrução é indicado por *next_pc*. O resultado do contador de saturação do bloco é armazenado no campo *previsor*, que é atualizado sempre que o resultado de uma previsão é conhecido. Assim como na definição da *cache*, o bloco da BTB também mantém o campo *bit_uso*, para armazenar informações a respeito da substituição quando utilizada as técnicas LFU ou LRU, e o campo *count*, utilizado durante o uso da técnica de substituição FIFO.

Depois que as instruções são armazenadas na fila de busca elas são encaminhadas para o estágio de decodificação. A quantidade de instruções enviadas à decodificação depende da largura de decodificação do processador. No D-Power, essa largura é parametrizável, podendo ser configurada pelo usuário. Quando esse parâmetro é configurado, é indicada também a quantidade de portas de leitura que a fila de busca possuirá. Os dados são encaminhados a partir de um ponteiro que indica a primeira posição ocupada na fila. A partir desse ponteiro, as instruções são enviadas para o estágio de decodificação na quantidade correspondente a largura da decodificação.

4.1.3 Decodificação e Despacho

Após inseridas no *pipeline*, as instruções devem ser decodificadas para que dependências possam ser tratadas e para que elas possam ser distribuídas corretamente para as unidades funcionais responsáveis por sua execução. Para isso uma fila de decodificação é utilizada pela arquitetura.

Ao decodificar uma instrução e inseri-la na fila de decodificação, informações adicionais são associadas a cada instrução para que as dependências possam ser resolvidas. A Figura 4.7 apresenta a definição das informações armazenadas por cada posição da fila de decodificação.

```
TYPE Inst_Decod IS
   RECORD
     id_inst:         INTEGER;
     opcode:          STD_LOGIC_VECTOR (opcode_width - 1 DOWNTO 0);
     op1:             STD_LOGIC_VECTOR (operando_width - 1 DOWNTO 0);
     op2,op3:         STD_LOGIC_VECTOR (dado_width - 1 DOWNTO 0);
     ready_op2:       STD_LOGIC;
     ready_op3:       STD_LOGIC;
     bit_previsao:    STD_LOGIC;
     pc_instruction: T_program_counter;
  END RECORD;
```

Figura 4.7. Definição de uma instrução decodificada

A operação a ser realizada é armazenada no campo *opcode*. O registrador destino da operação é armazenado no campo *op1*. No campo *id_inst* é atribuído um identificador à instrução, o que possibilita a implementação adaptada do algoritmo de Tomasulo (Tomasulo, 1967). Durante a decodificação o banco de registradores é acessado para verificação da disponibilidade do dado e consequente detecção de dependências de dados RAW.

Ao acessar o banco de registradores, o registrador indicado no campo *op1* é marcado como ocupado, indicando que seu valor será alterado. Além disso, o registrador mantém um campo *tag_exec*, que recebe o *id* da instrução que irá modificá-lo. Então ocorre a verificação de dependências, por meio da averiguação do bit de ocupação dos registradores fonte. Se esse bit possuir valor ‘0’, indica que o registrador não está sendo utilizado por nenhuma outra instrução e não há nenhuma dependência verdadeira referente ao registrador acessado. Então o valor do registrador é associado ao campo *op2* ou ao campo *op3* da fila de decodificação, dependendo de qual operando está sendo verificado, e os campos *ready_op2* ou *ready_op3* recebem o valor ‘1’, indicando que o respectivo operando possui um valor válido.

Já se o bit de ocupação do registrador fonte for ‘1’, indica uma detecção de dependência RAW e que uma instrução anteriormente decodificada já está utilizando o registrador. Isso implica que o registrador terá seu valor alterado e ler o valor agora faria com que, no momento da execução, a instrução utilizasse valores ultrapassados. É nesse sentido que o algoritmo de Tomasulo é empregado, para permitir que a instrução acesse o valor real do registrador assim que a instrução que está utilizando-o seja executada.

O que a decodificação faz então, é associar o campo *tag_exec* do registrador, que contém o *id* da instrução que irá atualizar seu valor, ao campo do operando buscado, *op2* ou *op3,* e atualizar os campos *ready_op2* ou *ready_op3* com o valor ‘0’, indicando que o dado referente àquele operando ainda não é válido.

Quando a instrução que atualizará o registrador terminar de executar, seu resultado será repassado para as outras instruções armazenadas nas estações de reserva. Para identificar quais instruções nas estações de reserva estão aguardando por esse resultado, cada valor dos operandos fonte cujo dado ainda não é válido é comparado com o *id* da instrução recém executada. Caso os valores sejam iguais, o valor do operando é atualizado com o resultado e o campo *ready_op* do respectivo operando é atualizado para válido.

A associação do *id* de uma instrução ao *tag_exec* do registrador é realizada mesmo se o registrador já estiver marcado como ocupado. Isso garante a resolução de várias dependências relacionadas a um mesmo registrador.

A utilização do algoritmo de Tomasulo permite que os valores dos operandos fonte que possuem alguma dependência sejam atualizados antes da conclusão da instrução, otimizando o tempo total de execução de instruções que apresentem dependências.

O campo *bit_previsao* é utilizado apenas em instruções de desvio condicionais, indicando qual foi a previsão realizada para essa instrução. Esse bit indica se o desvio foi ou não tomado e é comparado com o resultado da execução da instrução para verificar se a previsão foi ou não realizada corretamente.

Ainda nessa fase, uma entrada para cada instrução decodificada é inserida no final da fila de reordenação. A função da fila de reordenação é manter a integridade de instruções executadas fora de ordem, finalizando-as na ordem que aparecem no fluxo de um programa. A estrutura dessa fila é apresentada na Figura 4.8.

```
Type dado_reordenacao is
    RECORD
      id_inst:        INTEGER;
      resultado:      STD_LOGIC_VECTOR (dado_width - 1 DOWNTO 0);
      op1:            STD_LOGIC_VECTOR (operando_width - 1 DOWNTO 0);
      opcode:         STD_LOGIC_VECTOR (opcode_width - 1 DOWNTO 0);
      ready:          STD_LOGIC;
      pc_instruction: T_program_counter;
      bit_previsao:   STD_LOGIC;
    END RECORD;
```

Figura 4.8. Definição da fila de reordenação

A fila de reordenação foi implementada como um *buffer* FIFO circular, com ponteiros indicando a primeira e última posição da fila. O campo *id_inst* mantém o identificador (*id*) da instrução. O campo *pc_instruction* armazena o endereço da instrução. Essa informação deve ser salva para o caso de o *pipeline* precisar ser reiniciado por causa de uma previsão incorreta.

Nesse caso, é possível determinar o valor correto do PC após a reinicialização do *pipeline*. *Op1* indica o registrador que receberá o resultado no momento da finalização da memória. Os valores desses campos são determinados durante a decodificação da instrução. Já os campos *resultado* e *ready* são preenchidos apenas após a execução da instrução. *Resultado* mantém o resultado da operação realizada pela instrução após sua execução. Já o campo *ready* indica que uma instrução já foi executada e pode ser finalizada.

Após a decodificação as instruções são encaminhadas para suas respectivas estações de reserva. Antes de inserir uma instrução na estação de reserva correspondente, é verificada se a estação está cheia. Caso esteja, o despacho não ocorrerá e a instrução permanecerá na fila de decodificação até que haja uma posição desocupada na estação de reserva. No entanto, como o despacho ocorre fora de ordem, pode ser que instruções posteriores possam ser enviadas a outras estações que não estejam cheias. Para solucionar esse problema, o D-Power implementa a fila de decodificação como um *buffer* SAMQ (*Statically Allocated Multi-Queue*), como apresentado na Figura 4.9.

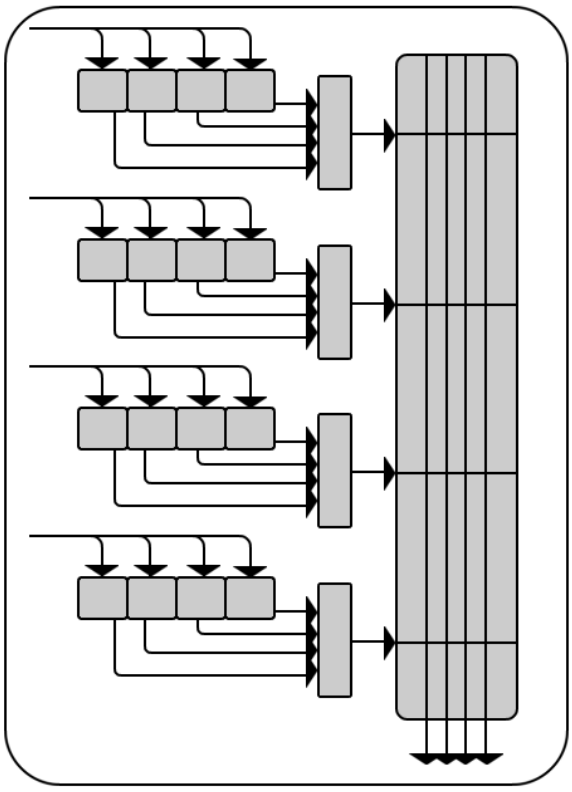


Figura 4.9. Buffer SAMQ

Com a utilização do *buffer* SAMQ, instruções armazenadas em qualquer posição podem ser encaminhadas para uma porta de leitura. Um multiplexador seleciona as instruções com base na ordem de chegada e na estação de reserva que ela deve ser inserida. A

quantidade de portas de leitura da fila de decodificação depende da quantidade de estações de reserva existentes na arquitetura.

O despacho não analisa se a instrução possui todos os dados prontos. Assim, o único fator que determinará se a instrução será ou não despachada é a disponibilidade da estação de reserva. Caso a instrução esteja aguardando por algum resultado de outra instrução anterior, ela permanecerá aguardando dentro da estação de reserva.

4.1.4 Remessa e Execução

A arquitetura implementada utiliza estações de reserva especializadas, portanto, cada unidade funcional possui uma estação de reserva dedicada. As instruções armazenadas nas estações ficam aguardando a disponibilidade de recursos ou dados para serem executadas. A arquitetura implementada apresenta unidades funcionais para execução de operações com inteiro e operações de acesso à memória. Nas estações de reserva das unidades funcionais de operações de memória a remessa é realizada em ordem. Apenas a instrução na primeira posição da estação pode ser enviada à execução.

Já as estações de reserva das unidades de inteiro podem enviar instruções para execução fora de ordem, desde que ela esteja pronta. Para que isso seja possível, estações de reserva de diferentes tipos de unidades funcionais foram definidas como *buffers* de organizações distintas. As estações de reserva para operações de memória possuem organização FIFO enquanto as estações de reserva para instruções de inteiro possuem organização SAMQ. Nas estações de inteiro, a política de remessa utilizada possui prioridade fixa, na qual a prioridade de execução é dada à instrução pronta que foi inserida a mais tempo na estação.

Apenas uma instrução por estação de reserva pode ser encaminhada à execução, mantendo uma única porta de leitura por estação. Cada unidade funcional executa uma instrução por vez. O paralelismo acontece quando mais de uma unidade funcional está ativa, executando instruções. A cada instrução é atribuído um atraso para sua execução. Em arquiteturas reais, por exemplo, instruções de divisão têm latência superior a instruções de adição. Nessa fase da implementação do D-Power, o atraso é simulado podendo ser configurado pelo usuário.

Ao final da execução de uma instrução, o resultado da operação é salvo na fila de reordenação na entrada da instrução, que é marcada como pronta para a finalização. As estações de reserva e fila de decodificação também são averiguadas, a fim de verificar se há

alguma instrução esperando o resultado dessa operação. Isso é realizado comparando o *id* da instrução com o valor dos operandos fonte das instruções presentes nessas estruturas que possuam operandos marcados como não válidos. Caso alguma instrução seja encontrada, o valor resultante da operação é passado à entrada da instrução e o respectivo operando é marcado como válido.

O conjunto de instruções implementado no D-Power é um subconjunto do conjunto de instruções do MIPS IV (Price, 1995). Instruções para operações de inteiro e operações de memória fazem parte do grupo de instruções do D-Power. Um posterior suporte a operações de ponto flutuante é planejado. Dentre as instruções estão implementadas instruções com operações matemáticas e lógicas, operações de leitura e escrita na memória e instruções de desvio condicionais e incondicionais, provendo uma ampla variedade de instruções que tornam possível a implementação de uma grande diversidade de programas para serem executados.

4.1.5 Conclusão

O último estágio do *pipeline* implementado no D-Power é o estágio de conclusão. Nessa etapa o conteúdo da fila de reordenação é examinado. A análise começa sempre a partir da primeira posição da fila e para quando a primeira instrução não pronta é encontrada ou quando a largura de *commit* é atingida. Se a instrução estiver pronta para ser finalizada, seu resultado será gravado no registrador destino e o bit de ocupação desse registrador é marcado como falso caso a instrução em questão tenha sido a última a ocupar tal registrador. O ponteiro que indica a primeira posição da fila será decrementado, indicando a retirada da entrada dessa instrução da fila de reordenação.

Caso a instrução concluída seja de desvio, o resultado do desvio é examinado: se o desvio foi tomado corretamente, a análise da fila prossegue normalmente a partir da próxima posição. Mas, caso a previsão de desvio tenha ocorrido de maneira errônea, todas as instruções buscadas posteriores à do desvio deverão ser descartadas.

O descarte das instruções buscadas depois da instrução de desvio ocorre em todos os estágios do *pipeline*. Todas as filas e as estações de reserva precisarão ser esvaziadas, todas as entradas da fila de reordenação posteriores a entrada da instrução do desvio previsto errado deverão ser retiradas e os bits de ocupação dos registradores deverão ser desmarcados. O endereço de busca correto é passado para o PC, que redireciona o fluxo de controle para trazer

as instruções do caminho correto, voltando a preencher o *pipeline*.

Caso o previsor utilizado for o BTB-PHT, a BTB é acessada nesse estágio. Caso não exista uma entrada para a instrução na BTB, a instrução é inserida com os bits de previsão assumindo o valor padrão ‘01’. Caso já exista uma entrada para essa instrução na BTB, os bits de previsão são atualizados com o incremento ou decremento do contador de saturação de acordo com o resultado da previsão.

4.2 Estimativa do Consumo de Potência

O principal objetivo do trabalho é oferecer uma arquitetura superescalar configurável com capacidade de estimativa de consumo de potência. Por representar a maior parcela na dissipação de potência de um processador, decidiu-se por investigar o consumo obtido pela potência dinâmica. Para que isso seja possível, o D-Power precisa apresentar meios para monitorar e contabilizar a atividade de chaveamento presente nos elementos que compõem a arquitetura e estimar o consumo com base nessa atividade.

4.2.1 Detecção da Atividade de Chaveamento

A estimativa é realizada por meio do monitoramento dos sinais de entrada e saída de cada porta lógica representada na arquitetura. A Figura 4.10 apresenta a metodologia empregada para a detecção das atividades de chaveamento, tomando como exemplo um circuito somador de 1 bit.

No código apresentado na Figura 4.10 cada sinal *Xi* equivale à saída de uma porta lógica no somador. Assim, sempre que houver uma alteração em um desses sinais, um processo é ativado para contabilizar a atividade de chaveamento. Essa detecção é permitida graças à estrutura PROCESS aliada ao atributo de sinais EVENT, presentes na linguagem VHDL. Um processo VHDL é ativado sempre que um dos sinais presentes em sua lista de sensibilidade é ativado. Na Figura 4.10, um único sinal faz parte da lista de sensibilidade de cada processo, que é ativado quando esse sinal é referenciado.

```
ENTITY somador IS
  PORT (a, b, cin: IN BIT;
        s, cout: OUT BIT;
        power: OUT REAL);
END somador;

ARCHITECTURE structural OF somador IS
  SIGNAL X1, X2, X3, X4, X5, X6: BIT;
  SHARED VARIABLE powerCont: INTEGER := 0;
  BEGIN

    X1 <= a AND b;
    X2 <= a AND Cin;
    X3 <= b AND Cin;
    X4 <= X1 OR X2 OR X3;
    Cout <= X4;
    X5 <= A XOR B XOR Cin;
    S<= X5;

    PROCESS (X1)
    BEGIN
       IF (X1'EVENT) THEN
           powerCont := powerCont + 1;
       END IF;
    END PROCESS;
    .
    .
    PROCESS (X5)
    BEGIN
       IF (X5'EVENT) THEN
           powerCont := powerCont + 1;
       END IF;
    END PROCESS;

END structural;
```

Figura 4.10. Somador de 1 bit com detecção de atividade de chaveamento

Quando um processo é ativado, dentro dele é verificado se houve alteração no valor do sinal passado na lista de sensibilidade. Isso é possível por meio do uso do atributo EVENT, que retorna o valor ‘verdadeiro’ caso o sinal tenha o seu valor alterado. Assim, se ocorreu uma alteração no valor do sinal, ocorreu uma atividade de chaveamento, que deve ser computada. A contabilidade das atividades de chaveamento ocorre com o uso de SHARED_VARIABLES, variáveis que podem ser compartilhadas por mais de um processo.

Estruturas matriciais, como as *caches* e a BTB implementadas na arquitetura também utilizam o mesmo princípio para a detecção das atividades de chaveamento. A diferença é que para cada linha da estrutura é gerado um processo que é disparado quando essa linha é acessada. Após ativado, o processo então percorre todos os elementos do vetor verificando se

houve alguma alteração em seus bits. A Figura 4.11 mostra como é realizada a detecção da atividade de chaveamento nas células da *cache* L1.

```
L1: FOR i IN 0 TO qt_linhas_caches_N1 GENERATE
    PROCESS (cache_instruction(i))
    BEGIN

        FOR j in 0 to palavras_por_bloco-1
          FOR k IN 0 to dado_width
                IF cache_instruction(i).informacao(j)(k)'EVENT THEN
                        PowerCell := PowerCell + 1;
                END IF;
          END FOR; -- j
        END FOR; -- k

        FOR j in 0 to rotulo_width
                IF cache_instruction(i).rotulo(j)'EVENT THEN
                        PowerCell := PowerCell + 1;
                END IF;
        END FOR;

        IF politica_substituicao = "LFU" THEN
            FOR j IN 1 to count_width
                    IF cache_instruction(i).count(j)'EVENT THEN
                            PowerCell := PowerCell + 1;
                    END IF;
            END FOR;
        ELSE
            FOR j IN bit_uso_dado_width - 1 DOWNTO 0
                    IF cache_instruction(i).bit_uso(j)'EVENT THEN
                            PowerCell := PowerCell + 1;
                    END IF;
            END FOR;
        END IF;

        IF politica_escrita = "WB" THEN
            IF cache_instruction(i).bit_modificacao'EVENT THEN
                PowerCell := PowerCell + 1;
            END IF;
        END IF;

        IF cache_instruction(i).bit_validade'EVENT THEN
            PowerCell := PowerCell + 1;
        END IF;

    END PROCESS
END GENERATE L1;
```

Figura 4.11. Detecção da atividade de chaveamento na cache L1

Como apresentado na Figura 4.11, diversos processos são criados com a utilização da cláusula GENERATE presente na linguagem VHDL. Isso faz com que a quantidade de processos criados que detectam a atividade de chaveamento tenha relação direta com a quantidade de linhas presente nas estruturas matriciais, permitindo que a detecção da atividade de chaveamento siga a parametrização da arquitetura.

A implementação do D-Power busca a estimativa do consumo de potência dinâmica no estágio de busca e em dois níveis de *cache* por meio da detecção da atividade de chaveamento que ocorre durante a execução de programas na arquitetura implementada. Para possibilitar a detecção da atividade de chaveamento de forma precisa, todas as estruturas presentes tanto nesse estágio do *pipeline* quanto nas caches, foram minuciosamente detalhadas, permitindo que qualquer alteração no sinal de uma porta lógica presente na estrutura pudesse ser detectado.

O consumo estimado em uma *cache*, por exemplo, observa a troca de valores no registrador de endereço, no decodificador que determina a linha da *cache* a ser acessada, as entradas e saída da comparação entre o rótulo contido no registrador e o rótulo do bloco acessado na *cache*, os valores da determinação de um acerto na *cache* por meio do resultado do comparador e do bit de validade e o gasto com o multiplexador, em organizações que utilizam tal estrutura. Além disso, o D-Power determina a atividade de chaveamento que ocorre quando uma informação presente na *cache* é trocada. Atividades de chaveamento em células de *cache* e de *buffer* são tratadas de forma diferente pelo D-Power em relação às atividades de chaveamento que ocorrem nas portas lógicas. Considerando uma célula de *cache* que utiliza *flip-flops* do tipo D para armazenamento de um bit, como apresentado na Figura 4.12, por exemplo. A troca de um valor de '0' (Figura 4.12a) para '1' (Figura 4.12b), implica mais de uma atividade de chaveamento dentro dessa célula.

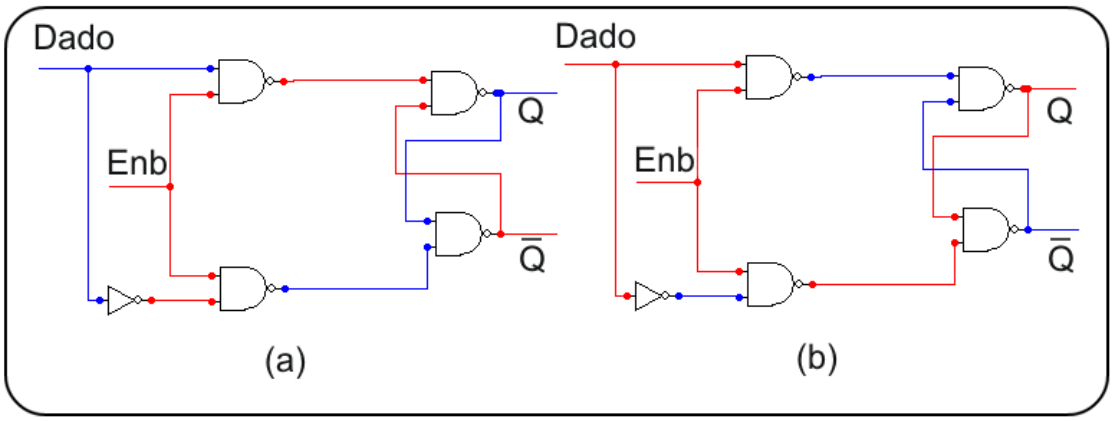


Figura 4.12. Troca de valores em um flip-flop do tipo D

Como pode ser observado na Figura 4.12, o resultado de quatro portas lógicas NAND e um inversor têm seus valores alterados quando ocorre a troca de valores em uma célula da *cache*. O D-Power prevê esse comportamento, mantendo uma correspondência entre a troca de um valor em uma célula e a quantidade de portas presentes em um *flip-flop* no momento do cálculo da estimativa do consumo de potência.

Além do gasto com a troca da informação armazenada, são contabilizados gastos com outros dados salvos, como o gasto na troca do rótulo quando uma nova informação é escrita na *cache*, os gastos para manter as estruturas responsáveis pelo controle das políticas de substituição e o gasto com o bit de validade.

O mesmo estudo minucioso dos outros componentes envolvidos durante a inserção de instruções no *pipeline* foi realizado para que todas as transições de valores neles fossem detectadas. Tanto a BTB, quanto a fila de busca, o PC e as interligações de todas as estruturas foram descritas de forma detalhada para possibilitar a detecção precisa das atividades de chaveamento.

4.2.2 Estimativa do Consumo de Potência Dinâmico

O cálculo da potência dinâmica dissipada a partir da atividade de chaveamento detectada durante a execução de um programa pela arquitetura é realizado com base na Equação 4.1 (XILINX, 2002).

$$P = C \cdot V^2 \cdot F \cdot E \cdot 1000 \qquad \text{...(4.1)}$$

Onde P representa potência final dissipada, medida em miliWatts (mW). C é a carga capacitiva do circuito, que é medida em Farads (F). V é tensão fornecida pela fonte de alimentação em Volts (V). F é a frequência de operação do circuito, medida em Hertz (Hz). E é a atividade de chaveamento. A Equação 4.1 leva em conta a média de transições de chaveamento que ocorreram por ciclo de *clock*. Dessa forma, o D-Power precisa medir a quantidade de ciclos gastos pela simulação. Percebe-se pela equação que a atividade de chaveamento é um fator crucial para uma estimativa precisa do consumo de potência dinâmica. Por isso, a ferramenta aqui apresentada busca especificar, de maneira detalhada, as estruturas implementadas.

Os parâmetros C, V e F são dependentes da tecnologia empregada. O D-Power não é uma ferramenta de síntese de circuitos, o que implica que ele assume que o usuário conheça, ou pelo menos possa estimar, a capacitância, a tensão e a frequência que seu projeto irá utilizar. A responsabilidade do D-Power é prever de forma precisa a atividade de chaveamento de acordo com os parâmetros arquiteturais fornecidos pelo usuário, utilizando como entrada determinada aplicação, e assim estimar o consumo de potência com base nos parâmetros tecnológicos.

Independente dos parâmetros tecnológicos utilizados, a ferramenta parte do princípio que uma precisão relativa no consumo de potência é mais importante do que uma precisão absoluta em níveis mais altos do projeto (Landman, 1996). Isso porque o que o projetista realmente deseja saber nessa fase é se uma configuração arquitetural é melhor do que outra e quais implicações diferentes alternativas de configurações podem ter no consumo e no desempenho do sistema.

4.2.3 Medição de Desempenho

Para permitir melhor análise da relação consumo de potência/desempenho, o D-Power oferece ainda como saída alguns resultados relacionados ao desempenho obtido pela arquitetura durante a execução de determinado programa. A Tabela 4.2 apresenta quais informações de desempenho são oferecidas pelo D-Power.

Tabela 4.2. Informações de desempenho no D-Power

Acessos à *cache* de instruções	Ciclos
Acertos na *cache* de instruções	Instruções Buscadas
Acessos à *cache* de dados	Instruções Executadas
Acertos na *cache* de dados	Instruções Finalizadas
Acessos à *cache* L2	Instruções de Desvios Executadas
Acertos na *cache* L2	Desvios incorretos
Acessos à BTB	

Dessa forma, é possível analisar o desempenho de diferentes técnicas implementadas pela arquitetura, por exemplo, qual a influência das políticas de escrita e substituição na quantidade de acertos na *cache*, a relação que a quantidade de palavras por bloco pode ter sobre a quantidade de acessos a níveis superiores da *cache* e eficiência dos previsores de desvio.

4.2.4 Validação do Consumo

Após a implementação da arquitetura, foi necessário validar a metodologia de estimativa de consumo empregada de forma a verificar se os consumos apresentados pela D-Power condizem com a realidade. A validação da ferramenta ocorreu por meio da comparação das estimativas obtidas pelas ferramenta D-Power e SPICE para os mesmos circuitos sob as

mesmas condições de testes. Optou-se pela comparação da estimativa obtida para circuitos essenciais utilizados pela arquitetura.

A Tabela 4.3 apresenta a estimativa do consumo de potência de um *flip-flop* do tipo D, comparando os resultados obtidos pelo D-Power, com os resultados obtidos por uma descrição do mesmo circuito no *SPICE* de acordo com a variação na tensão fornecida. Os parâmetros tecnológicos utilizados na comparação são de um processo CMOS 0.18µm. Para a simulação dos valores de entrada foram criados cinco vetores de dados aleatórios e a frequência de operação fixada em 1 MHz. Os resultados apresentados correspondem à média das simulações.

Tabela 4.3. Estimativa do consumo de potência dinâmico de um flip-flop do tipo D

Consumo / Tensão	D-Power (mW)	SPICE (mW)	Diferença (%)
3.3V	1,263	1,387	8,94
2.5V	0,725	0,793	8,57
1.8V	0,375	0,417	10,07
1.5V	0,261	0,282	7,44
1.2V	0,167	0,189	11,64
0.9V	0,094	0,101	6,93

Como pode ser observado na Tabela 4.3, a diferença do consumo estimado entre o *SPICE* e o D-Power fica entre 6,93% e 11,64%, com uma média geral de erro de 8,93% nos exemplos simulados. Essa diferença na precisão do consumo é completamente aceitável considerando a disparidade de nível de projeto que ambas as ferramentas consideram no momento da estimativa.

Outro exemplo de validação é apresentado na Tabela 4.4. Nela, são comparados os valores de implementações de um circuito somador no D-Power e no *SPICE*. As simulações apresentam variação na frequência e mantêm a tensão fixa em 1.8V. Os resultados representam a média de cinco conjuntos aleatórios de entradas para cada circuito simulado.

Como apresentado na Tabela 4.4, o D-Power manteve uma taxa de erro média de 5,15% em relação ao *SPICE*. A diferença máxima foi de 9,43% e a mínima foi 2,75%. Optou-se pela validação por meio da comparação da estimativa de potência obtida para circuitos que compõem a arquitetura, pois uma validação de toda arquitetura implementada em baixo nível seria impraticável, uma vez que a ferramenta *SPICE* necessita de tempos de projeto e simulação elevados.

Tabela 4.4. Estimativa do consumo de potência dinâmico de um somador

Consumo / Frequência	D-Power (mW)	SPICE (mW)	Diferença (%)
1 MHz	0,225	0,232	3,01
5 MHz	0,826	0,912	9,43
10 MHz	1,804	1,931	6,58
20 MHz	3,758	3,903	3,72
50 MHz	9,959	10,241	2,75
100 MHz	20,821	22,016	5,43

A validação de componentes menores da arquitetura implementada indica que o consumo da arquitetura como um todo também é válido, já que a técnica empregada para estimar o consumo desses componentes é a mesma aplicada na estimativa do consumo de potência de toda a arquitetura.

4.3 Considerações Finais

O presente capítulo apresentou a ferramenta D-Power, um simulador de arquiteturas superescalares com capacidade para estimar o consumo de potência dinâmica dissipado pelo processador. A ferramenta faz a estimativa por meio da detecção e contagem de atividades de chaveamento que ocorrem nos componentes de um processador superescalar que executa determinado programa, aliado a parâmetros tecnológicos fornecidos ao D-Power. A arquitetura foi validada por meio de comparações de resultados do consumo de elementos que a compõem em relação à ferramenta *SPICE*.

Vale ressaltar que nessa fase de implementação, a avaliação do consumo refere-se às estruturas ligadas ao estágio de busca e hierarquias de *caches*. São essas estruturas que, se avaliadas individualmente, são responsáveis pela maior parcela do consumo de um processador superescalar. As outras estruturas terão sua estimativa de consumo detalhadas em fases posteriores do projeto do D-Power, a fim de possibilitar a previsão do consumo em todo o processador.

CAPÍTULO

5

Experimentos e Resultados

Neste capítulo são apresentados alguns experimentos realizados com a ferramenta D-Power, visando a exposição dos diversos parâmetros configuráveis na arquitetura, e os resultados obtidos nesses testes, buscando mostrar a capacidade da ferramenta em estimar a potência dissipada por um processador superescalar. Os experimentos retratam a simulação de dois programas executados por uma arquitetura superescalar com a variação de diversos parâmetros arquiteturais. Parâmetros como o tamanho das *caches*, associatividade, quantidade de palavras por bloco, política de substituição, previsor de desvio e largura de busca foram variados no intuito de avaliar o impacto que esses parâmetros possuem no consumo de potência de um processador superescalar.

5.1 Descrição dos Experimentos

Para avaliar o modelo arquitetural implementado no D-Power buscou-se a resolução de problemas reais. Desta forma foram implementados algoritmos para o cálculo do fatorial de um número, para determinar o n-ésimo elemento da sequência de Fibonacci e algoritmo para multiplicação de matrizes. Como o D-Power ainda não é compatível com nenhum *benchmark* disponível, optou-se pela implementação desses algoritmos para avaliar a ferramenta.

O fatorial de um número positivo *n*, representado por *n!*, é obtido por meio do produto entre todos os inteiros positivos menores ou iguais a *n*. O algoritmo desenvolvido é apresentado na Figura 5.1.

```
lui $20, 0x1001 -- posição da memória que receberá o resultado
lui $10, 12     -- o número para calcular o Fatorial
lui $1, 1
lui $4, 1
mov $3, $10
sub $5, $10, $1
blez $5, sleese
sub $2, $10, $1
mul $3, $2, $10
loop: beq $2, $4, sleese
sub $2, $2, $1
mul $3, $3, $2
j loop
sleese: sw $3, 0($20)
```

Figura 5.1. Algoritmo para cálculo do fatorial

O algoritmo implementado parte do princípio que *n! = n * (n-1)!* . Assim, o valor de *n* é armazenado no registrador $10 e o valor de *n-1* no registrador $2. O produto do valor de ambos os registradores é armazenado em $3. Então, o conteúdo do registrador $2 é decrementado novamente e multiplicado pelo conteúdo de $3. Isso é repetido até que $2 atinja o valor 1. O resultado estará no registrador $3 que é então armazenado na memória.

Como observado na Figura 5.1, o cálculo do fatorial de um número pode ser realizado com pouco mais de 14 instruções, considerando as instruções para controle do laço. Nesse sentido, para possibilitar a melhor análise da arquitetura sob um fluxo maior de instruções, optou-se por realizar o cálculo do fatorial de uma sequência de números sorteados de forma pseudorandômica. Os números positivos foram sorteados aleatoriamente e seus fatoriais então calculados. A mesma sequência de números foi então inserida em todos os testes, para que as diferentes configurações pudessem ser avaliadas de acordo com a mesma entrada de dados.

Outro experimento implementado realiza o cálculo do n-ésimo elemento da sequência de Fibonacci, sendo *n* a entrada do programa. A sequência de Fibonacci é iniciada com 0 e 1 e cada um dos outros elementos da sequência é calculado por meio da soma dos dois elementos anteriores a ele, como apresentado na Equação 5.1.

$$F(n) = \begin{cases} 0, & se\ n = 0; \\ 1, & se\ n = 1; \\ F(n-1) + F(n-2) & se\ n \geq 2. \end{cases} \qquad ...(5.1)$$

Os primeiros números da sequência são 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144. O algoritmo desenvolvido utiliza como entrada um valor *n* que indicará a posição do número de Fibonacci a ser calculo. Por exemplo, *n=8* produzirá 13 como resultado, uma vez que 13 é o oitavo elemento da sequência. O algoritmo desenvolvido é apresentado na Figura 5.2.

```
lui $20, 0x1001 -- posição da memória que receberá o resultado
lui $3, 8       -- n-ésimo nº da sequência de Fibonacci a ser encontrado
lui $4, 1
add $3, $3, $4
lui $5, -1
lui $1, -1
loop: beq $3 $0, sleese
add $4, $4, $5
sub $5, $4, $5
add $3, $3, $1
j loop
sleese: sw $4, 0($20)
```

Figura 5.2. Algoritmo para cálculo do n-ésimo elemento de Fibonacci

No algoritmo descrito, *n+1* é armazenado no registrador $3 e os valores 1 e -1 são armazenados nos registradores $4 e $5, respectivamente. Então um laço é executado até que $3 atinja o valor zero. Nesse laço o registrador $4 recebe seu valor somado ao valor do registrador $5 e o registrador $5 recebe o valor do registrador $4 menos o seu próprio valor. Quando $3 for igual a zero o número de Fibonacci calculado estará armazenado em $4.

Assim como no experimento que calcula o fatorial de um número positivo, também foi gerada uma sequência de valores de entrada para que o cálculo de diversos elementos presentes na sequência fosse realizado. Isso propiciou um conjunto maior de instruções executadas e uma melhor análise dos resultados. Considerando o espaço utilizado em memória, o experimento Fatorial possui cerca de 1KB enquanto o experimento Fibonacci cerca de 5KB.

O algoritmo da multiplicação de matriz recebe como parâmetro o tamanho de cada matriz *A* e *B* que serão multiplicadas e gera uma sequência de valores de entrada para cada uma das matrizes. Os valores são armazenados na memória e trazidos dela quando o valor for requisitado pelo algoritmo. Os valores da matriz *C* resultante também são armazenados na memória. Pelo fato de ler e gravar diversos dados na memória, esse algoritmo é caracterizado pelo intenso acesso à memória de dados. O algoritmo que realiza a multiplicação de matrizes é apresentado na Figura 5.3.

```
lui $1, linha_a --- número de linhas de a
lui $2, coluna_a --- número de colunas de a
lui $3, linha_b --- número de linhas de b
lui $4, coluna_a --- número de colunas de b
lui $5, inicio_a --- início da região de memória da matriz a
lui $6, inicio_b --- início da região de memória da matriz b
lui $7, inicio_c --- início da região de memória da matriz c
lui $18, 1
lui $8, 0 -- controle do Loop da Matriz A
lui $9, 0 -- controle do Loop da Matriz B
lui $10, 0 -- controle do FOR INTERNO
lui $11, 0 -- controle das Loop da Matriz C
loop_a: beq $1, $8, end_program
loop_b: beq $4, $9, acrescenta_a
lui $12, 0
lui $13, 0
loop_interno: beq $2, $10, acrescenta_b
mul $14, $8, r2
add $14, $14, $10
add $5, $14, $10
lw $16, $14
mul $15, $10, $4
add $15, $15, $9
add $15, $15, $6
lw $17, $15
mul $13, $16, $17
add $12, $12, $13
add $10, $10, $18
j loop_interno
acrescenta_b: add $9, $9, $18
add $19, $11, $7
sw $12, 0($19)
add $11, $11, $18
lui $10, 0
j loop_b
acrescenta_a: add $8, $8, $18
lui $9, 0
j loop_a
end_program
```

Figura 5.3. Algoritmo para multiplicação de matrizes

O algoritmo descrito é composto por três laços aninhados. O primeiro laço percorre os elementos de cada linha da Matriz *A* enquanto o segundo laço percorre os elementos de uma coluna da matriz *B*. O laço mais interno multiplica um elemento da Matriz *A* com um

elemento da Matriz *B* e acumula o valor no registrador $12. Ao fim do laço mais interno, o registrador $12 armazena o elemento resultante da Matriz *C* que é então armazenado na memória de dados.

Para cada um dos algoritmos implementados, diversas configurações arquiteturais foram utilizadas. Os parâmetros tecnológicos foram fixados, considerando um circuito implementado com processo CMOS 18μm operando a uma frequência de 250MHz e alimentado por uma tensão de 1.5V. Alguns parâmetros arquiteturais foram fixados, como apresentado na Tabela 5.1.

Tabela 5.1. Parâmetros fixos nos experimentos

Parâmetro	**Valor**
Tamanho dos dados e instruções	64 bits
Largura de Remessa	3
Entradas nas Estações de Reserva	16
Quantidade de Unidades Funcionais	3
Largura de Finalização	4
Entradas na Fila de Reordenação	64
Quantidade de Registradores	64

Os outros parâmetros arquiteturais foram variados e os valores por eles assumidos são descritos de acordo com os resultados apresentados na Seção 5.2

5.2 Resultados

Os resultados aqui listados apresentam o impacto no consumo de potência dinâmica e no desempenho de um processador de acordo com a variação de diversos parâmetros arquiteturais.

5.2.1 Tamanho das *Caches*

A variação do tamanho das *caches* foi realizada no intuito de prover uma análise do impacto do tamanho da *cache* no consumo. Para isso, outros parâmetros foram fixados. A Tabela 5.2 apresenta a configuração utilizada.

Tabela 5.2. Configuração base para análise do tamanho da cache

Parâmetro	**Valor**
Palavras/Bloco	1
Organização	Mapeamento Direto
Largura de Busca	4
Largura de Decodificação	4
Entradas na Fila de Busca	8
Entradas na Fila de Decodificação	8
Política de Substituição	LFU
Política de Escrita	*Write Through*
Previsor de Desvio	Tomado

Para as *caches* L1, foram analisados os gastos com os seguintes tamanhos: 256 Bytes, 512 Bytes, 1KB, 2KB, 4KB, 8KB e 16KB. O tamanho da *cache* L2 foi configurado como o dobro do tamanho definido para a *cache* L1 em cada teste. Portanto, para os tamanhos de *cache* L1 citados acima, as *caches* secundárias possuem 512 Bytes, 1KB, 2KB, 4KB, 8KB, 16KB e 32KB, respectivamente. A Figura 5.4 apresenta os resultados do consumo obtido com essa variação no tamanho da *cache* de instruções para os experimentos Fatorial, Fibonacci e Multiplicação de Matrizes.

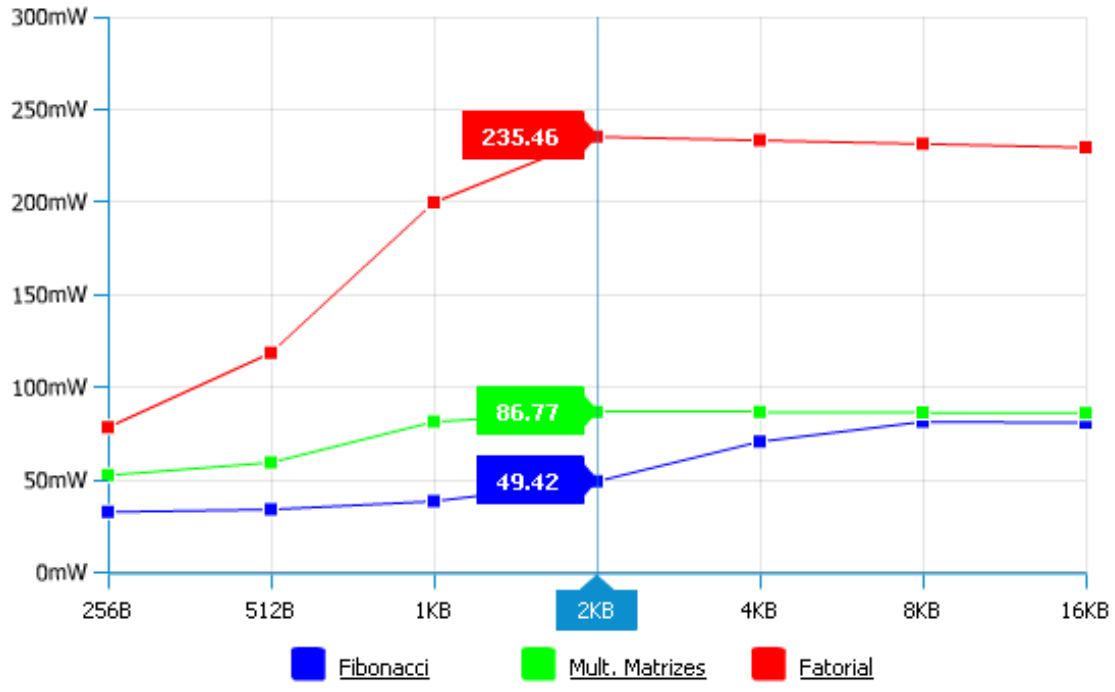


Figura 5.4. Consumo na cache de instruções com variação no tamanho

Como pode ser observado na Figura 5.4, o experimento Fatorial possuiu maior consumo de potência que os demais experimentos, mesmo ocupando menos espaço na memória. Isso porque o cálculo do Fatorial apresenta maior atividade de chaveamento por ciclo. Os menores consumos foram obtidos com os menores tamanhos de *caches*.

O aumento no tamanho da *cache* provoca um aumento no consumo de potência, uma vez que há um aumento na quantidade de linhas. As principais variações ocorrem quando a *cache* de instruções é aumentada de 512B para 1KB nos experimentos Fatorial e Multiplicação de Matrizes e de 2KB para 4KB no experimento Fibonacci. No Fatorial o consumo sobe de 118mW para 199mW, apresentando uma variação de cerca de 68%. Na Multiplicação de Matrizes, o consumo de potência apresenta aumento de 40,6% quando o tamanho da *cache* passa de 512B para 1KB. Já no Fibonacci o consumo aumenta de 49mW, na *cache* de 2KB, para 70mW, na *cache* de 4KB, proporcionando um aumento de cerca de 42% no consumo da *cache*. Esse aumento no consumo de acordo com o aumento no tamanho da *cache* ocorre até que o tamanho da *cache* seja igual ou maior que o tamanho do programa executado. Nos experimentos realizados isso ocorre com *caches* de instruções de 2KB no Fatorial, *caches* de instruções de 8KB no Fibonacci e de 1KB na Multiplicação de Matrizes. Após esses valores, a *cache* de instruções passa a conter todo o programa e o consumo é estabilizado, uma vez que há pouca diferença nas atividades de chaveamento. Já a Figura 5.5 apresenta o consumo na *cache* L2 de acordo com a variação do tamanho.

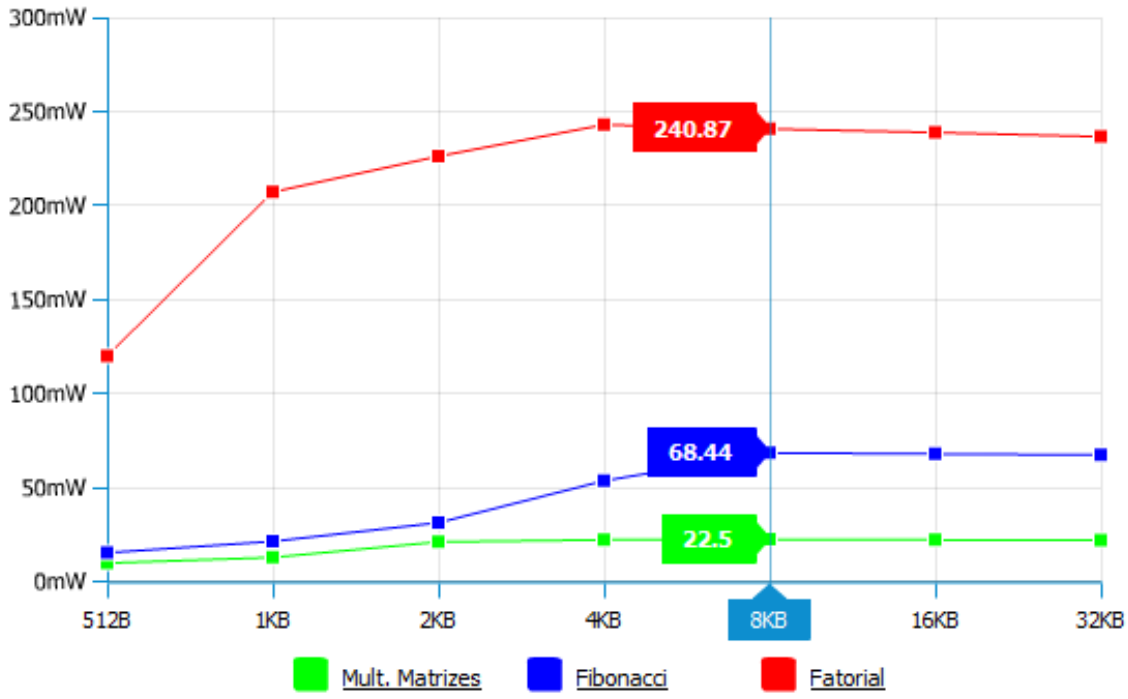


Figura 5.5. Consumo na cache L2 com variação no tamanho

Como pode ser observado na Figura 5.5, o comportamento do consumo na *cache* secundária é parecido com o comportamento da *cache* de instruções. O aumento no tamanho da *cache* provoca também aumento no consumo de potência. O consumo também é estabilizado após determinado tamanho. Para o experimento Fatorial, isso ocorre com *cache* L2 de 4KB, que atinge um pico de 242,93mW, enquanto que para o experimento Fibonacci isso ocorre quando a *cache* L2 possui 8KB, que provoca um gasto de 68,44mW. Na Multiplicação de Matrizes o consumo se estabiliza com *cache* L2 de 2KB, apresentando consumo de 21,09mW.

Com o aumento de consumo provocado pelo aumento do tamanho das *caches*, têm-se um consequente aumento na potência dissipada por todo o processador. A Figura 5.6 apresenta o consumo total medido pelo D-Power de acordo com a variação no tamanho das *caches* primárias e secundária. O eixo *x* do gráfico apresenta o tamanho utilizado para a *cache* L1.

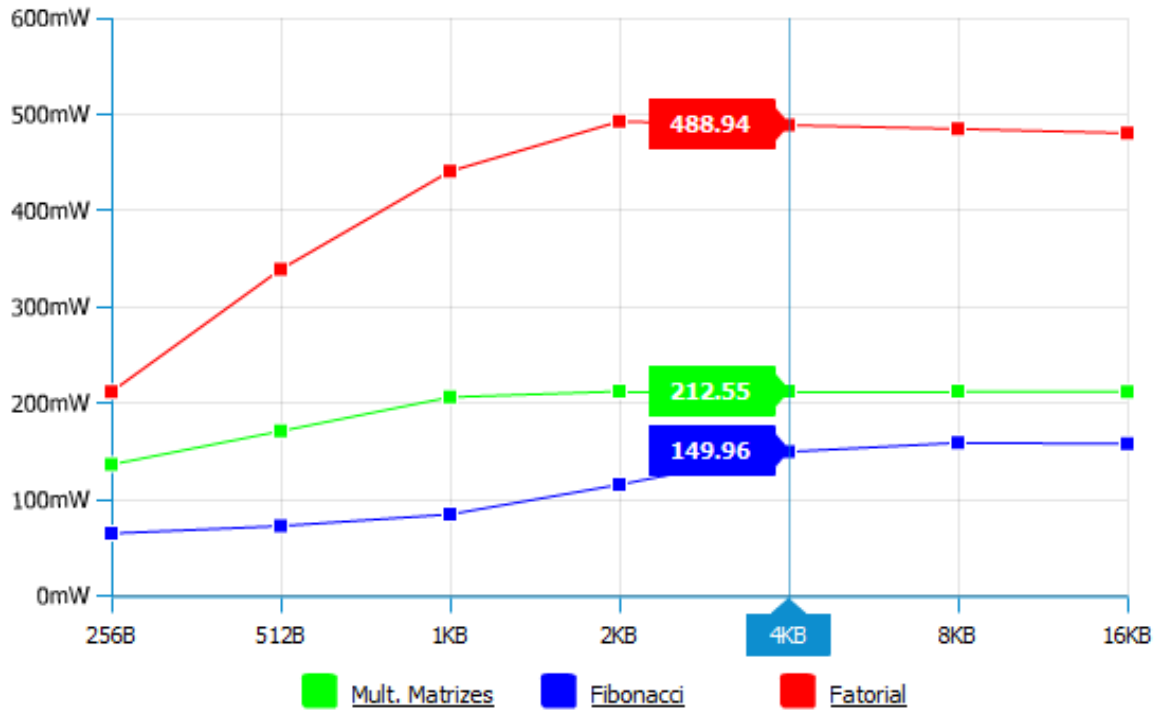


Figura 5.6. Consumo na fase de busca com variação no tamanho das caches

Como pode ser observado na Figura 5.6, o mesmo comportamento previsto no consumo das *caches* é percebido no consumo geral de todo o estágio de busca. Nesses resultados estão previstos o consumo em ambos os níveis de *cache*, na fila de busca, no contador de programa, todas as interligações entre essas estruturas, sinais e circuitos de controle ligados a esses componentes. O consumo na fase de busca atinge seu pico no

experimento Fibonacci quando utilizadas *caches* L1 de 8KB e *cache* L2 de 16KB. Nesse caso o consumo estimado é de 159,12mW. Já nos experimentos Fatorial e Multiplicação de Matrizes, os testes que mais dissiparam potência foram quando utilizadas *caches* L1 de 2KB e *cache* L2 de 4KB. Para essa configuração, o gasto estimado é de 492,98mW para o Fatorial e de 212,66mW para o experimento da Multiplicação de Matrizes. Já os menores consumos foram estimados com o menor tamanho de *cache* simulado. O experimento Fatorial prevê um consumo de potência dinâmica de 211,75mW utilizando *caches* primárias de 256B e *cache* secundária de 512B, enquanto o experimento Fibonacci estima um consumo de 64,95mW e na Multiplicação de Matrizes o consumo estimado foi de 136,52mW utilizando os mesmos tamanhos.

O gasto provocado pela utilização de uma *cache* maior pode ser justificado, caso essa *cache* provoque ganhos de desempenho consideráveis ao processador. Nesse sentido, é considerada a quantidade de ciclos de *clock* que o D-Power levou para executar os experimentos de acordo com essa variação no tamanho da *cache*. A Figura 5.7 apresenta esses resultados considerando o experimento de Fibonacci.

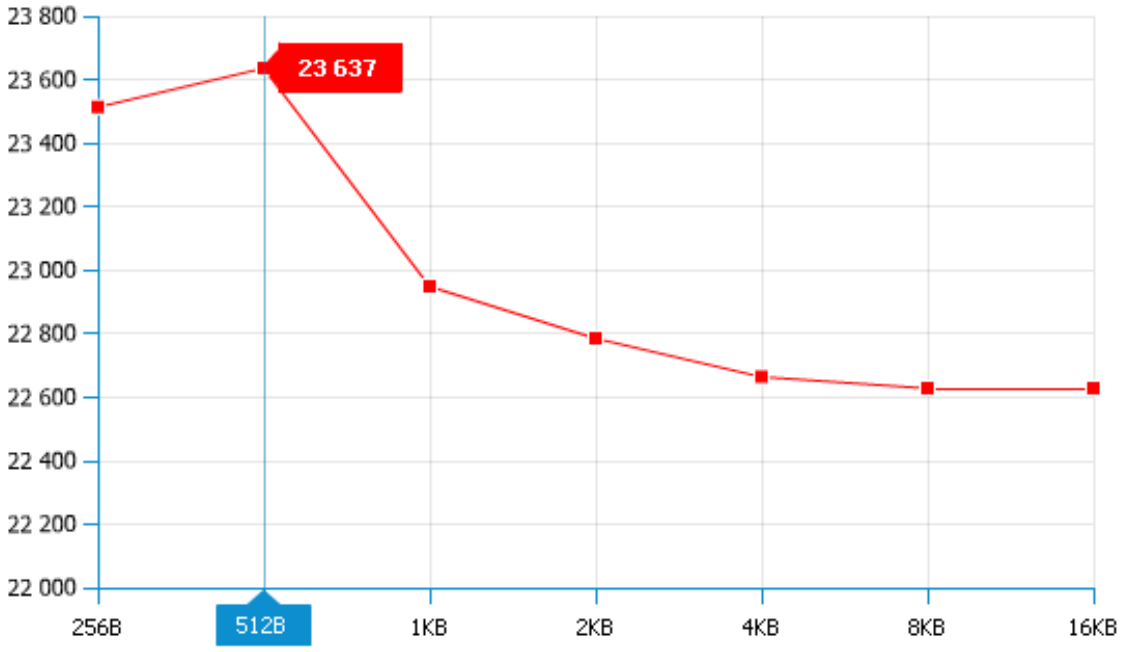


Figura 5.7. Ciclos para a execução do experimento de Fibonacci

A Figura 5.7 apresenta a quantidade de ciclos gastos para a execução do experimento de Fibonacci de acordo com a variação no tamanho da *cache* de instruções L1. É possível observar que na maioria dos casos, o aumento no tamanho da *cache* provocou um ganho de

desempenho. A exceção ocorre quando o tamanho da *cache* passa de 256B para 512B. Nesse caso, o comportamento do programa causa uma pequena variação no desempenho, provocando uma perda de aproximadamente 0,4% de desempenho. Em todos os outros casos, entretanto, ocorre ganhos de performance quando a *cache* é aumentada. O maior ganho ocorre quando a *cache* passa de 512B para 1KB. Isso provoca um ganho de cerca de 3% no desempenho, o que corresponde a 688 ciclos a menos para terminar a execução do programa.

Nesse sentido é importante que o projetista tenha consciência das restrições do projeto e do domínio e características das aplicações executadas para que possa existir um equilíbrio entre o consumo de potência esperado e o melhor desempenho possível.

5.2.2 Políticas de Substituição

O D-Power implementa três diferentes políticas de substituição. A utilização de cada uma delas foi testada nos experimentos implementados e o impacto que cada uma apresenta no consumo de potência do processador foi analisado. Para isso foi utilizada uma configuração base, sendo alterada apenas a política de substituição e a associatividade das *caches*, que eram associativas por conjunto. Dessa forma é possível analisar o consumo provocado por cada uma das políticas sob os mesmos aspectos. A Tabela 5.3 apresenta essa configuração.

Tabela 5.3. Configuração base para análise das políticas de substituição

Parâmetro	**Valor**
Tamanho da *cache* no Fibonacci	8KB
Tamanho da *cache* no Fatorial	2KB
Palavras/Bloco	16
Largura de Busca	4
Largura de Decodificação	4
Entradas na Fila de Busca	8
Entradas na Fila de Decodificação	8
Política de Escrita	*Write Through*
Previsor de Desvio	Tomado

As simulações buscavam observar o consumo estimado na arquitetura base com a utilização das diferentes técnicas de substituição de blocos na *cache* implementadas no D-Power. A Figura 5.8 apresenta a variação no consumo da *cache* de instruções com o emprego de cada uma das técnicas variando a associatividade das *caches* no experimento Fatorial.

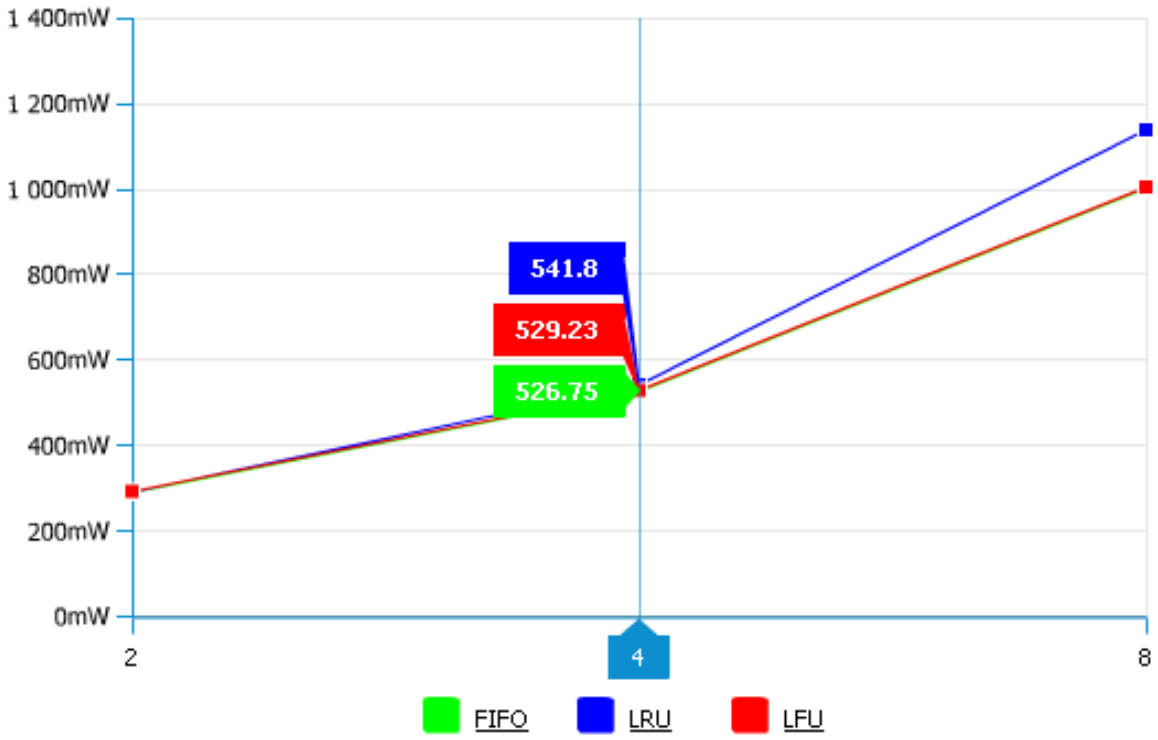


Figura 5.8. Consumo na cache de instruções variando política de substituição - Fatorial

Os gastos com o experimento Fatorial utilizando associatividade 2 mantiveram-se bem próximos, independente da política de substituição utilizada. O consumo estimado é cerca de 155mW com essa quantidade de vias e a variação entre as três técnicas é de aproximadamente 0,3%. Uma variação maior no consumo de potência passa a ocorrer com o aumento da associatividade. As políticas LFU e FIFO, entretanto, mantêm valores aproximados para o consumo, enquanto que uma maior disparidade ocorre com o emprego da técnica LRU. Em relação à LFU, a LRU dissipou aproximadamente 2,3% a mais de potência com associatividade 4 e 11,7% a mais com associatividade 8. Os mesmos resultados são apresentados para o experimento Fibonacci na Figura 5.9.

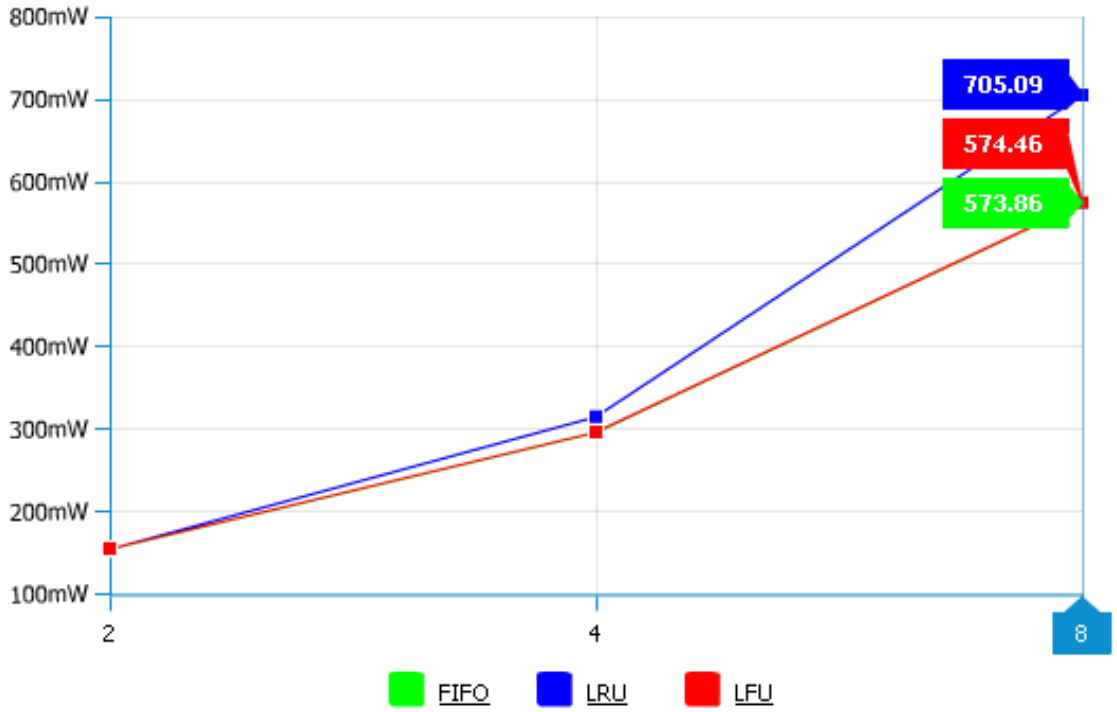


Figura 5.9. Consumo na cache de instruções variando política de substituição - Fibonacci

Como apresentado na Figura 5.9, o comportamento do consumo dissipado no experimento Fibonacci é semelhante ao comportamento do consumo no experimento Fatorial apresentado na Figura 5.8. A variação entre o FIFO e o LFU é mínima enquanto que o LRU apresenta um gasto maior. O aumento no consumo com o uso da LRU chega a 18,5% na *cache* de instruções em relação ao LFU se utilizada associatividade 8.

Os resultados indicam maior consumo no algoritmo LRU uma vez que o controle utilizado mantém a ordem de acesso a cada bloco de diferentes vias da *cache*. Esse controle requer mais atividades de chaveamento, uma vez que a cada acesso à *cache*, o LRU precisa atualizar campos referentes a todos os blocos, indicando a ordem em que eles foram acessados. Essa quantidade de atividade de chaveamento aumenta proporcionalmente ao aumento da associatividade na *cache*, pois uma associatividade maior indica manter a ordem de mais blocos. Já as outras técnicas utilizam menos atividade de chaveamento, pois requerem a atualização apenas do bloco acessado.

5.2.3 Associatividade e Palavras por Bloco

O sistema de *caches* implementado pelo D-Power permite a variação da organização da *cache* e da quantidade de palavras que compõem cada bloco da *cache*. Essa variação foi analisada de

forma a prover o impacto no consumo que um maior número de vias e uma maior quantidade de palavras por bloco apresentam no consumo de uma *cache*. Para esses testes foi utilizada uma configuração arquitetural base, apresentada na Tabela 5.4.

Tabela 5.4. Configuração base para análise da organização da cache

Parâmetro	**Valor**
Tamanho da *cache* no Fibonacci	8KB
Tamanho da *cache* no Fatorial	2KB
Tamanho da *cache* na Multiplicação de Matrizes	1KB
Largura de Busca	4
Largura de Decodificação	4
Entradas na Fila de Busca	8
Entradas na Fila de Decodificação	8
Política de Substituição	LFU
Política de Escrita	*Write Through*

Utilizando como base a configuração arquitetural apresentada na Tabela 5.4 foi variada a associatividade da *cache* e a quantidade de palavras por bloco. Foram utilizadas *cache* com mapeamento direto e *cache* associativa por conjunto com associatividade 2, 4 e 8. A quantidade de palavras por bloco foi variada em 1, 2, 4, 8 e 16. A Figura 5.10 apresenta a variação do consumo para essas configurações no experimento Fatorial. O mapeamento direto é indicado pela associatividade 1 na figura e o eixo *x* apresenta a variação na quantidade de palavras por bloco na *cache*.

É possível observar na Figura 5.10 que o aumento na associatividade implica em maior consumo de potência independente da quantidade de palavras utilizadas. Entretanto, essa variação no consumo é maior conforme um maior número de palavras por bloco é utilizado na organização. Utilizando uma palavra por bloco, por exemplo, ao mudar a associatividade de 4 para 8, têm-se um aumento de 15,46% no consumo, que passa de 300,63mW para 355,6mW. Já se for realizada essa mesma alteração na associatividade utilizando 16 palavras por bloco, o aumento no gasto é de 47,33%, passando de 529,23mW para 1.004,75mW.

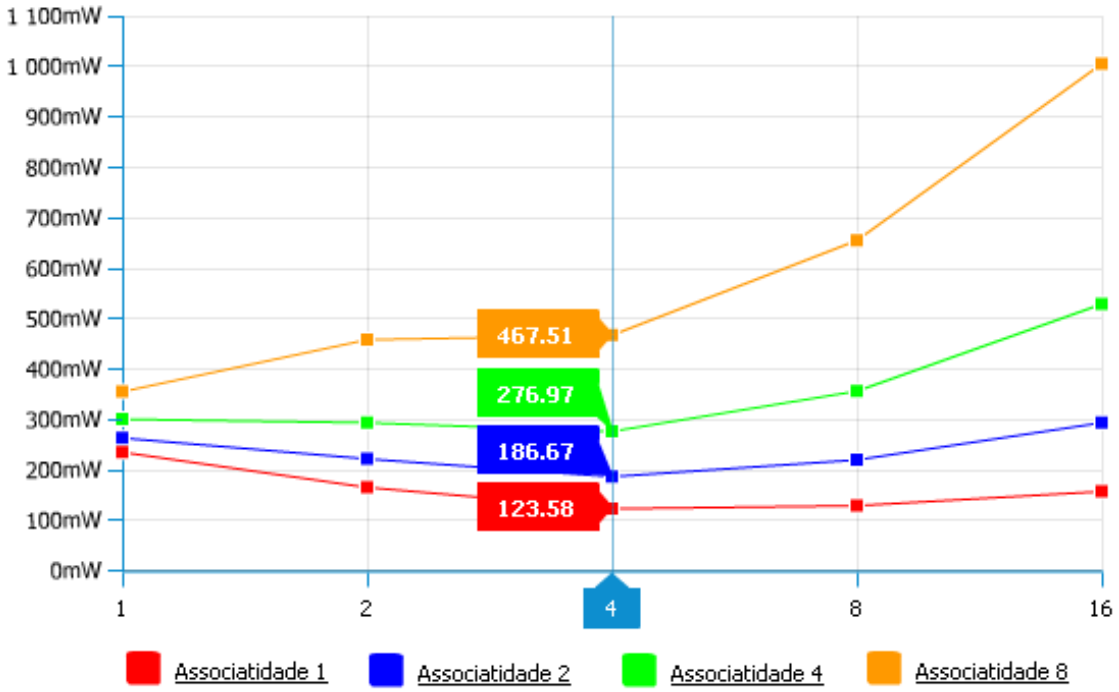


Figura 5.10. Consumo na cache L1 com variação na associatividade e no número de palavras por bloco - Fatorial

Já a variação na quantidade de palavras por blocos com associatividade 1, 2 e 4 apresentou redução no consumo de potência quando utilizadas 1, 2 e 4 palavras por bloco. Isso ocorreu porque o aumento na quantidade de palavras em *caches* de mesmo tamanho provocou uma redução de bits de controle armazenados na *cache*. Por exemplo, uma *cache* com duas linhas armazenando uma palavra por bloco requer rótulos diferentes para cada uma das palavras armazenadas. Já uma *cache* com uma linha que armazena duas palavras por bloco utiliza um único rótulo para as duas palavras. Há uma redução na quantidade de bits utilizados para controle, mas em contrapartida são necessários circuitos multiplexadores adicionais para selecionar a palavra requerida pertencente a um bloco. Essa redução na quantidade de bits de controle armazenados na *cache* fez com que ocorresse uma redução no consumo de potência. A *cache* com mapeamento direto apresentou uma redução de 47,52% no consumo aumentando as palavras por bloco de 1 para 4 palavras. Porém, com mais de 4 palavras, a quantidade de controles utilizados para selecionar a palavra dentro de um bloco, que é proporcional ao número de vias da *cache*, passou a dissipar muita potência, elevando o consumo proporcionalmente ao aumento no número de palavras por bloco. Esse comportamento não foi observado com *caches* com associatividade 8, uma vez que nessa configuração já há um alto número de sinais responsáveis por selecionar de qual bloco o dado

será buscado. Assim, o aumento no número de palavras por bloco sempre causa um aumento no consumo de potência com essa configuração.

O aumento no número de palavras por bloco normalmente é explorado para que o processador tire proveito da localidade espacial presente nos programas. O princípio da localidade diz que se um dado é referenciado, dados próximos a ele também tendem a ser referenciados. Nesse sentido, o aumento no consumo só é justificado se trouxer um ganho de desempenho ao processador. A Figura 5.11 apresenta a quantidade de acertos na *cache* L1 com variação no número de palavras e associatividade fixada em 4.

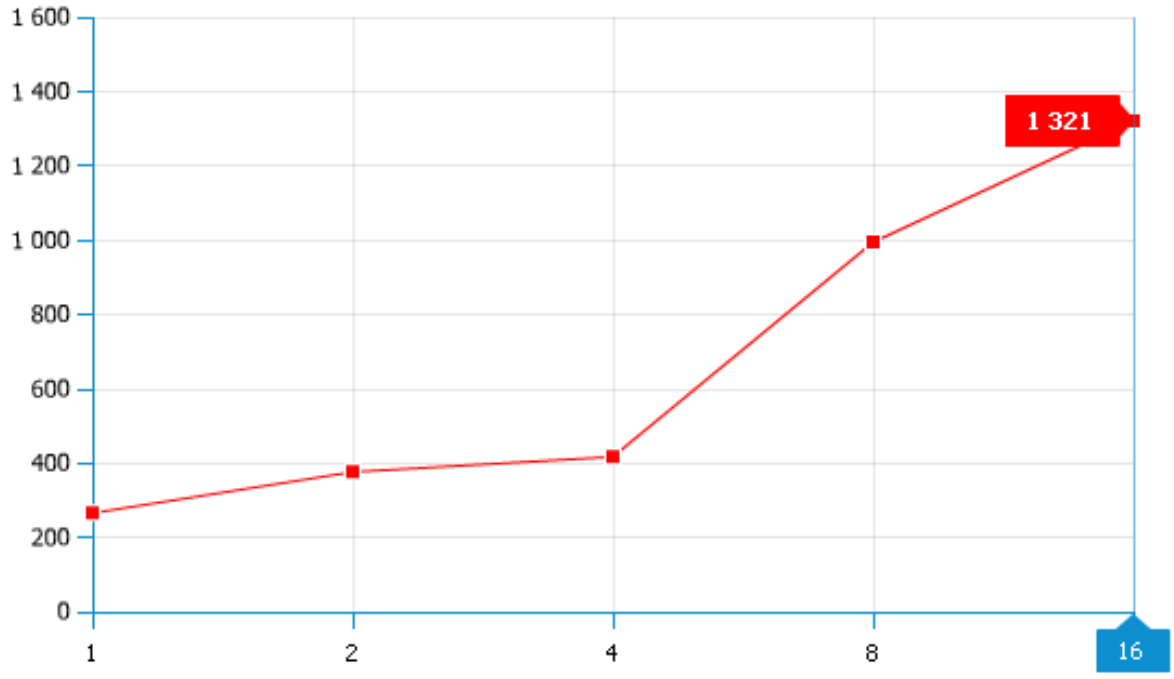


Figura 5.11. Acertos na cache L1 com variação no número de palavras por bloco - Fatorial

Como é possível observar na Figura 5.11, a quantidade de palavras por bloco possui bastante influência no número de acertos na *cache*. O principal ganho nesse experimento ocorre quando a quantidade de palavras por bloco passa de 4 para 8. Isso provoca um aumento de 58% na quantidade de acertos na *cache* primária. Consequentemente há uma menor taxa de acessos à memória principal que passa de 40 para 22 acessos.

A mesma comparação de consumo na variação da associatividade e palavras por bloco foi realizada para o experimento Fibonacci. Os resultados são apresentados na Figura 5.12.

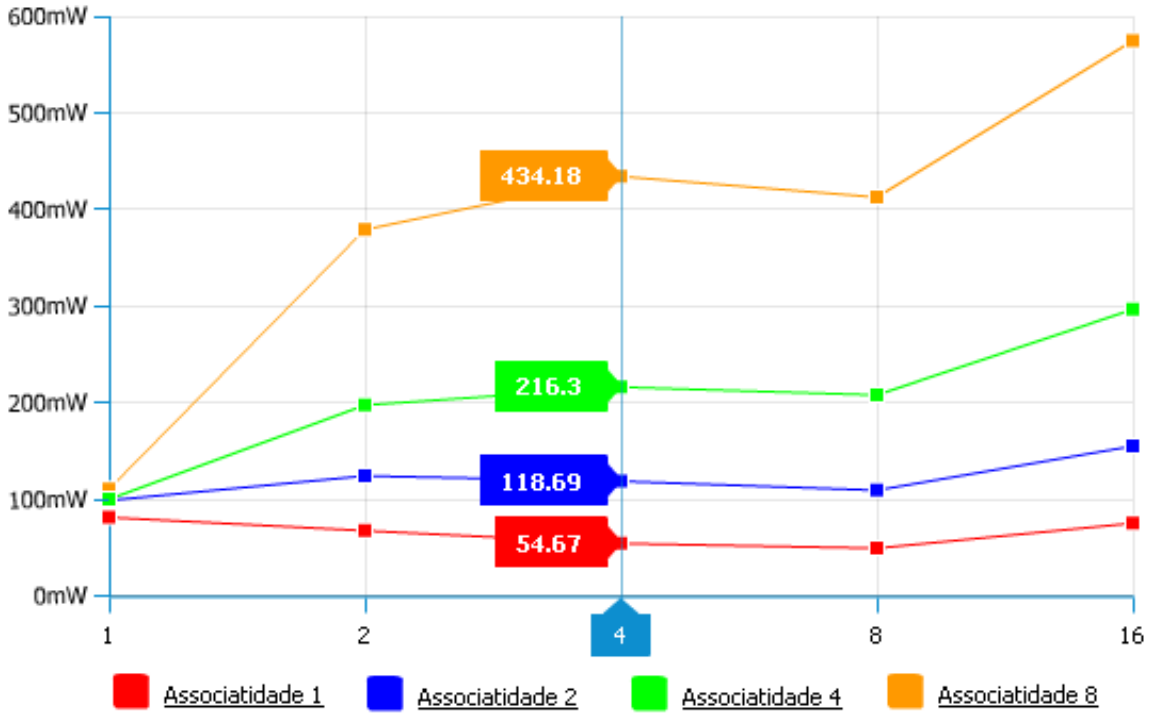


Figura 5.12. Consumo na cache L1 com variação na associatividade e no número de palavras por bloco - Fibonacci

Assim como no experimento Fatorial, fica claro o aumento no consumo com o aumento na associatividade. Já em relação ao aumento da quantidade de palavras por blocos, nota-se um comportamento um pouco diferente em relação ao primeiro experimento. No experimento Fibonacci, existe uma redução no consumo principalmente quando se altera o número de palavras de 4 para 8. Isso porque o uso de 8 palavras por bloco é a configuração que melhor explora a localidade espacial presente no algoritmo de Fibonacci implementado. Isso pode ser observado por meio da Figura 5.13, a qual apresenta a quantidade de acertos na *cache* L1 utilizando associatividade 4.

Quando a associatividade é alterada de 4 para 8 se obtém um ganho de 33,67% na quantidade de acertos na *cache* de instruções, que passa de 9.554 para 14.404. Essa maior quantidade de acertos faz com que existam menos acessos aos níveis secundários da memória e, por isso, menos troca de valores na *cache*. Isso implica em menos atividade de chaveamento e, consequentemente, menos potência dinâmica dissipada.

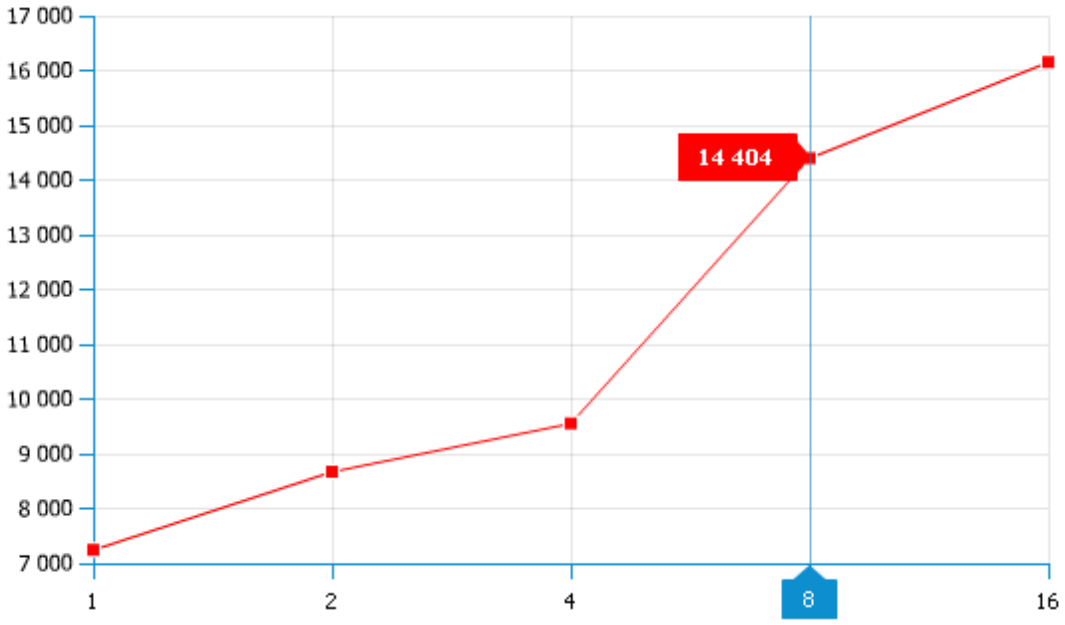


Figura 5.13. Acertos na cache de instruções com variação no número de palavras por bloco - Fibonacci

A comparação de consumo na variação da associatividade e palavras por bloco também foi realizada para o experimento de Multiplicação de Matrizes. Os resultados são apresentados na Figura 5.14.

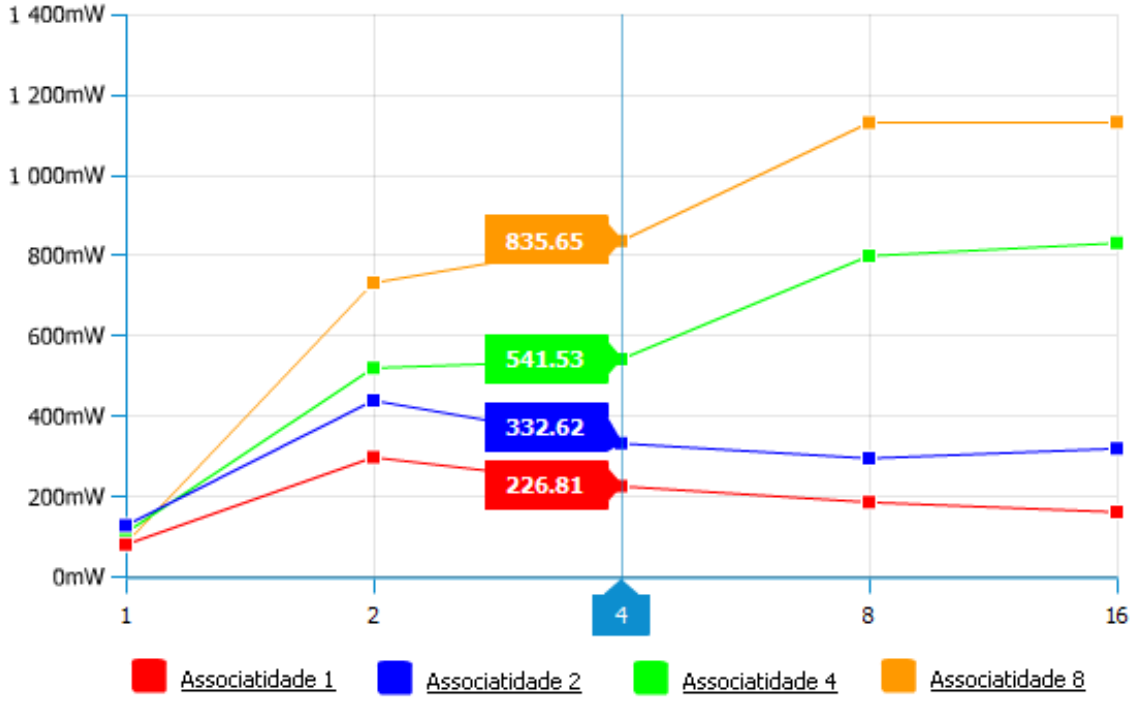


Figura 5.14. Consumo na cache L1 com variação na associatividade e no número de palavras por bloco - Multiplicação de Matrizes

Para esse experimento, notou-se também o aumento no consumo de potência ocasionado pelo aumento na associatividade. Em relação à variação no número de palavras por bloco, observou-se que, a partir de duas palavras por bloco, o consumo diminui para associatividades menores enquanto aumenta para maiores associatividades. Isso demonstra a melhor configuração para o experimento da Multiplicação de Matrizes, que alcança menores consumos com baixa associatividade e mais palavras por bloco.

Os experimentos mostram que a quantidade de palavras por bloco e a associatividade na *cache* possuem dependência em relação à aplicação executada. Nesse contexto, o D-Power pode ser uma ferramenta importante para que uma melhor configuração seja encontrada em sistemas específicos à aplicação. Um maior número de acertos na *cache* pode ajudar a reduzir o consumo, já que implica em menos atividade de chaveamento para armazenar novos dados buscados de outros níveis de memória.

5.2.4 Previsor de Desvio

Previsores de desvio são fundamentais em processadores superescalares, permitindo a execução especulativa de instruções. O descarte de instruções incorretas executadas especulativamente, entretanto, causa um atraso no fluxo de execução do processador, que deve invalidar as instruções erroneamente executadas e retomar o caminho correto da execução. Nesse sentido, um estudo do comportamento de três técnicas de previsão, duas estáticas e uma dinâmica, implementadas no D-Power é apresentado, comparando o desempenho de cada uma delas bem como o consumo por elas proporcionado. Para isso, uma configuração base foi utilizada, variando apenas o previsor. A Tabela 5.5 apresenta essa variação.

Tabela 5.5. Configuração base para análise das técnicas de previsão de desvio

Parâmetro	**Valor**
Tamanho da *cache* no Fibonacci	8KB
Tamanho da *cache* no Fatorial	2KB
Tamanho da *cache* na Multiplicação de Matrizes	1KB
Palavras/Bloco	16
Associatividade	4
Largura de Busca e Decodificação	4
Entradas na Fila de Busca e Decodificação	8
Política de Substituição	LRU
Política de Escrita	*Write Through*

Essa configuração base foi testada utilizando os previsores sempre-tomado, sempre não-tomado e BTB-PHT. O uso do previsor dinâmico BTB-PHT requer a utilização de uma BTB para manter o endereço alvo do desvio e a previsão. Nesse caso, foi utilizada uma BTB mapeada diretamente com tamanho de 256 Bytes, 512 Bytes e 2KB para os experimentos Multiplicação de Matrizes, Fatorial e Fibonacci respectivamente. Os resultados da comparação do consumo provocado pelo uso dos três previsores na fase de busca são apresentados na Figura 5.15, considerando o experimento Fatorial.

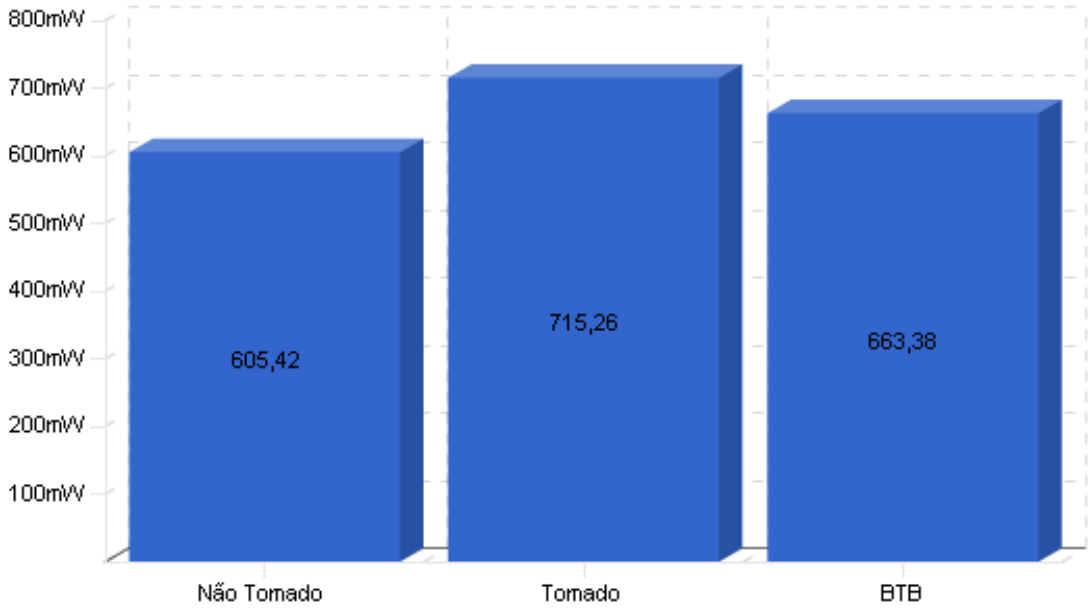


Figura 5.15. Consumo do Fatorial com variação no previsor de desvio

Como pode ser observado na Figura 5.15, um gasto maior é obtido pelo previsor tomado, que consumiu 715,26mW. O previsor BTB-PHT teve um consumo 7,25% menor, com 663,38mW enquanto que o previsor não-tomado obteve uma redução de 15,36% no consumo, com uma dissipação de potência de 605,42mW. Vale ressaltar que o previsor BTB-PHT utiliza uma estrutura adicional para realizar a previsão do desvio, enquanto que os previsores estáticos não utilizam qualquer estrutura. Nesse sentido, o gasto maior provocado pelo previsor tomado é justificado por um grande número de instruções buscadas de forma incorreta. A Figura 5.16 apresenta a taxa de acerto de cada um dos previsores analisados considerando o experimento Fatorial.

Como pode ser observado na Figura 5.16, a taxa de acerto nas previsões com o uso do previsor tomado é significativamente menor do que a taxa apresentada pelos outros previsores. A baixa quantidade de previsões corretas dessa técnica faz com que 42,28% a

mais de instruções sejam inseridas no *pipeline* se comparada com a técnica BTB-PHT. Instruções que posteriormente são descartadas por terem sido buscadas a partir de um endereço incorreto do contador de programa. Já o previsor não-tomado obteve uma taxa de acerto cerca de 6% menor, mas, em contrapartida, apresentou um consumo 8,74% menor que o previsor BTB-PHT.

Figura 5.16. Taxa de acerto dos previsores de desvio no experimento Fatorial

Os mesmos testes foram realizados considerando o experimento Fibonacci, que apresenta um número maior de instruções a serem executadas e comportamento diferente do algoritmo Fatorial. Os resultados referentes ao consumo de potência desse experimento com variação nas técnicas de previsão de desvio são apresentados na Figura 5.17.

A Figura 5.17 indica uma maior diferença no consumo provocado pelo previsor tomado em relação às outras técnicas. O gasto da fase de busca com o uso do previsor tomado é 54, 39% maior do que com o uso do previsor BTB-PHT. Já o previsor não tomado apresenta um gasto de 5,09% menor do que o gasto apresentado pelo previsor dinâmico. Essa maior diferença apresentada pelo previsor tomado ocorre graças a pouca eficiência que a técnica obteve durante a execução dos experimentos Fibonacci.

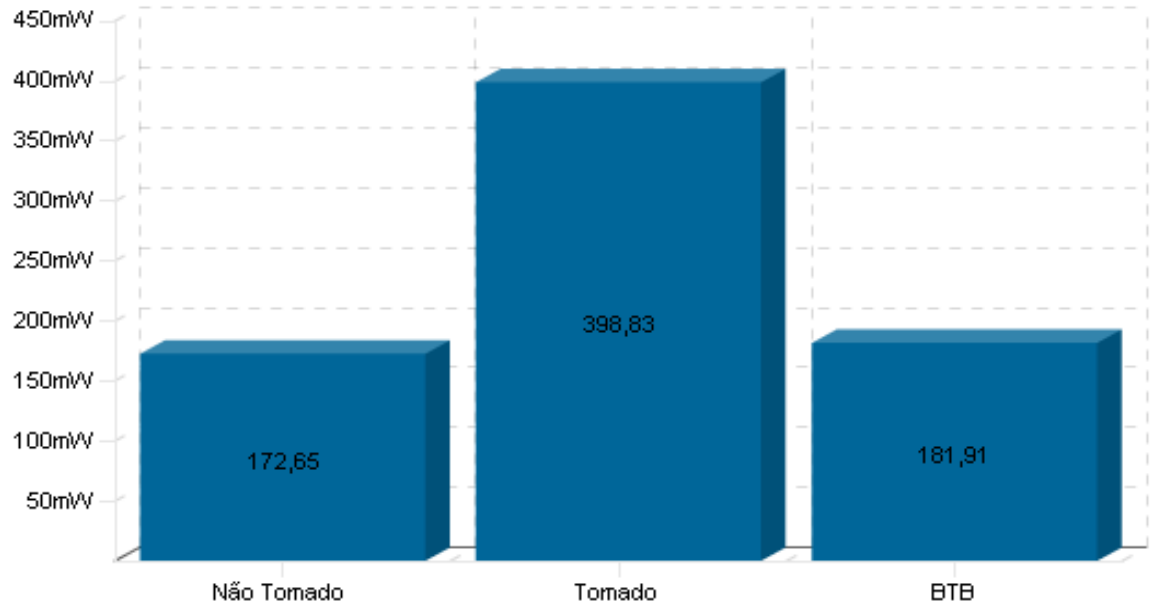


Figura 5.17. Consumo do Fibonacci com variação no previsor de desvio

A Figura 5.18 apresenta a taxa de acerto de cada uma das técnicas empregadas. O emprego do previsor tomado se mostra ineficiente para o experimento executado, causando um tráfego desnecessário de instruções no *pipeline*. Tal previsor busca 71,75% a mais de instruções se comparado com o previsor BTB-PHT, que se mostrou mais eficiente, com uma taxa de acerto de 96,08%.

Figura 5.18. Taxa de acerto dos previsores de desvio no experimento Fibonacci

A variação no consumo de potência ocasionada pela utilização de diferentes previsores de desvios para o experimento Multiplicação de Matrizes é apresentada na Figura 5.19. É possível observar que o previsor tomado apresentou um consumo de potência muito superior aos demais previsores testados. O consumo estimado para esse previsor foi de 1435,15mW enquanto o consumo obtido para o BTH-PHT foi de 187,59mW. O menor consumo foi alcançado pelo previsor estático não-tomado, que obteve consumo de 181,60mW, cerca de 3% a menos que o consumo estimado para o previsor BTB-PHT.

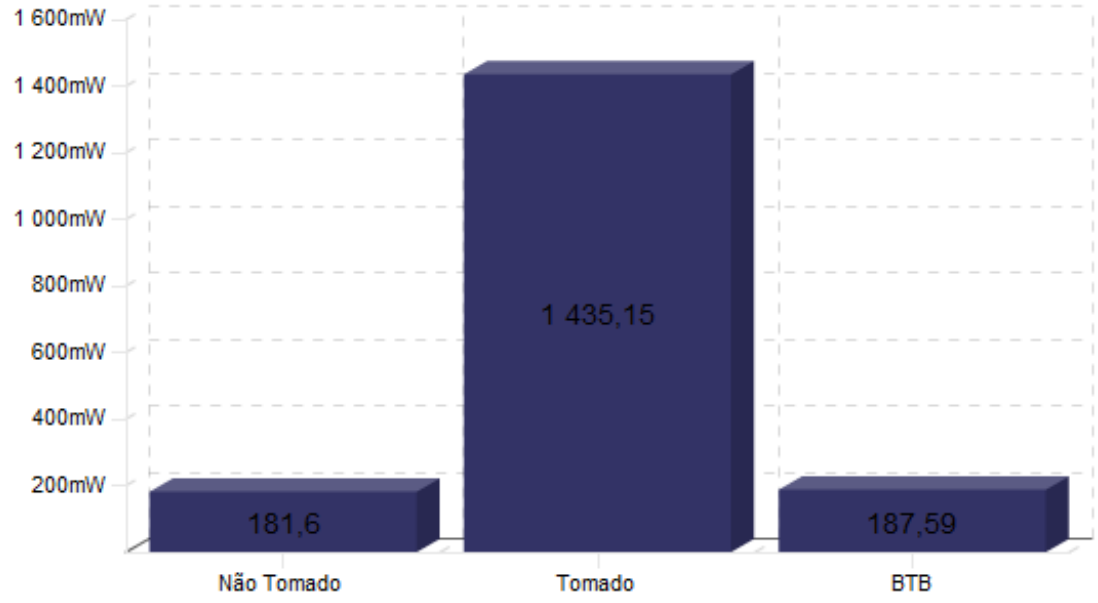


Figura 5.19. Consumo da Multiplicação de Matrizes com variação no previsor de desvio

A taxa de acerto de cada uma das técnicas empregadas no experimento Multiplicação de Matrizes é apresentada na Figura 5.20. O alto consumo observado para o previsor tomado pode ser justificado por sua baixa eficácia para o experimento em questão, uma vez que tal técnica apresentou taxa de acerto de apenas 11,11%. O previsor não-tomado, que apresentou os menores consumos, não foi a técnica mais precisa, obtendo taxa de acerto superior a 88%. Enquanto a técnica BTB-PHT obteve as melhores taxas de acerto em suas previsões, com 93,7% de acerto, com uma taxa de precisão cerca de 5% maior em relação ao previsor não-tomado.

Figura 5.20. Taxa de acerto dos previsores de desvio na Multiplicação de Matrizes

Os experimentos indicam que técnicas de previsão de desvio ineficientes provocam um consumo maior de potência, uma vez que há um gasto com instruções buscadas incorretamente. Além de provocar um gasto maior, há uma perda de desempenho no processador que deve tratar o descarte de instruções. Nesse sentido é importante o uso de técnicas que maximizam o uso eficiente do *pipeline* trazendo apenas instruções que realmente sejam utilizadas e não fazendo com o processador desperdice processamento com instruções inúteis.

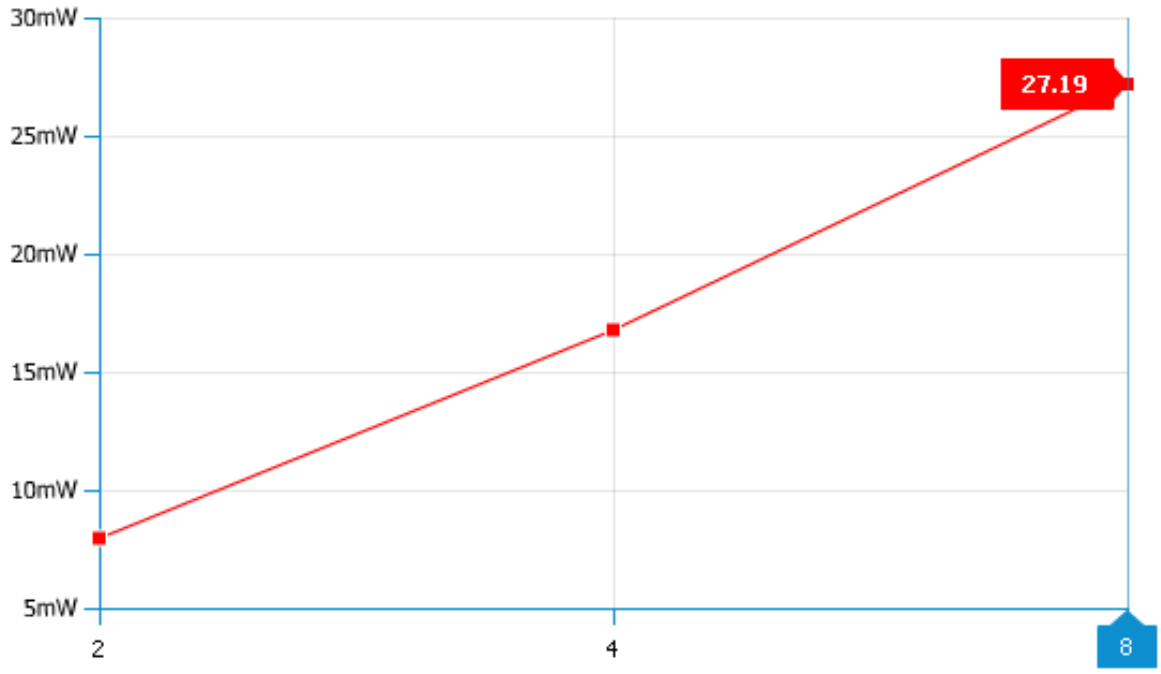
5.2.5 Largura de Busca

Buscando analisar o impacto que diferentes quantidades de instruções buscadas simultaneamente da *cache* de instruções ocasionam no consumo de potência e no desempenho de uma arquitetura superescalar, optou-se por realizar experimentos com diferentes larguras de busca. Para isso utilizou-se a configuração base apresentada na Tabela 5.6.

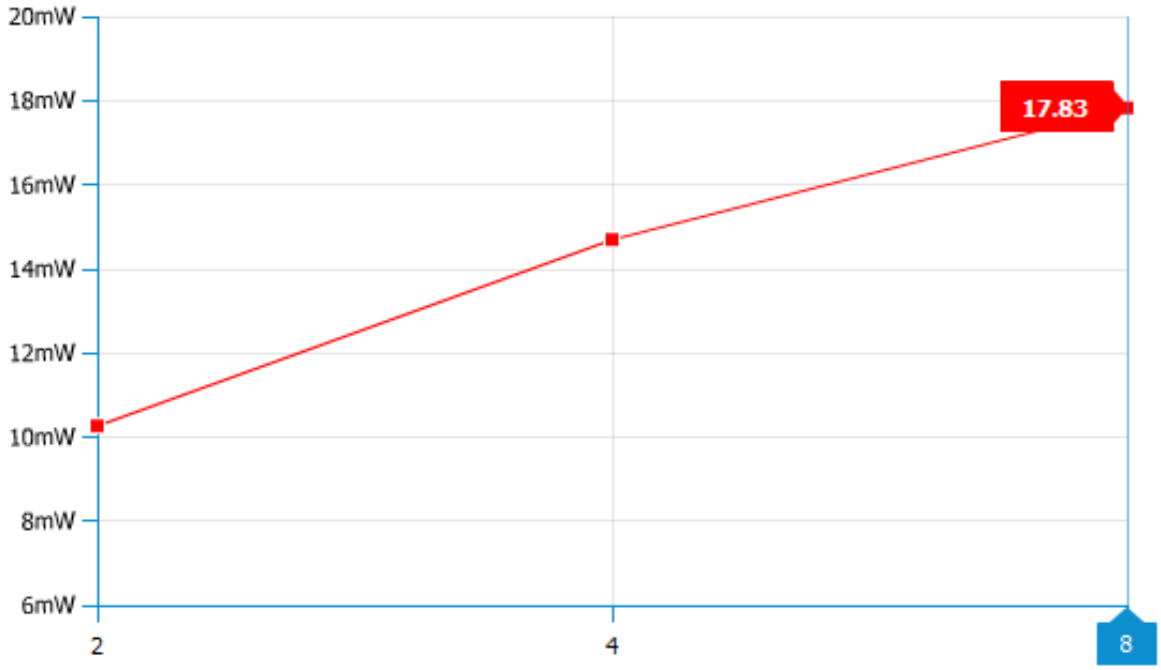
Tabela 5.6. Configuração base para análise da variação da largura de busca

Parâmetro	**Valor**
Tamanho da *cache* L1	2KB
Palavras/Bloco	2
Associatividade	4
Entradas na Fila de Busca	16
Entradas na Fila de Decodificação	16
Política de Substituição	LRU
Política de Escrita	*Write Through*
Previsor de Desvio	Tomado

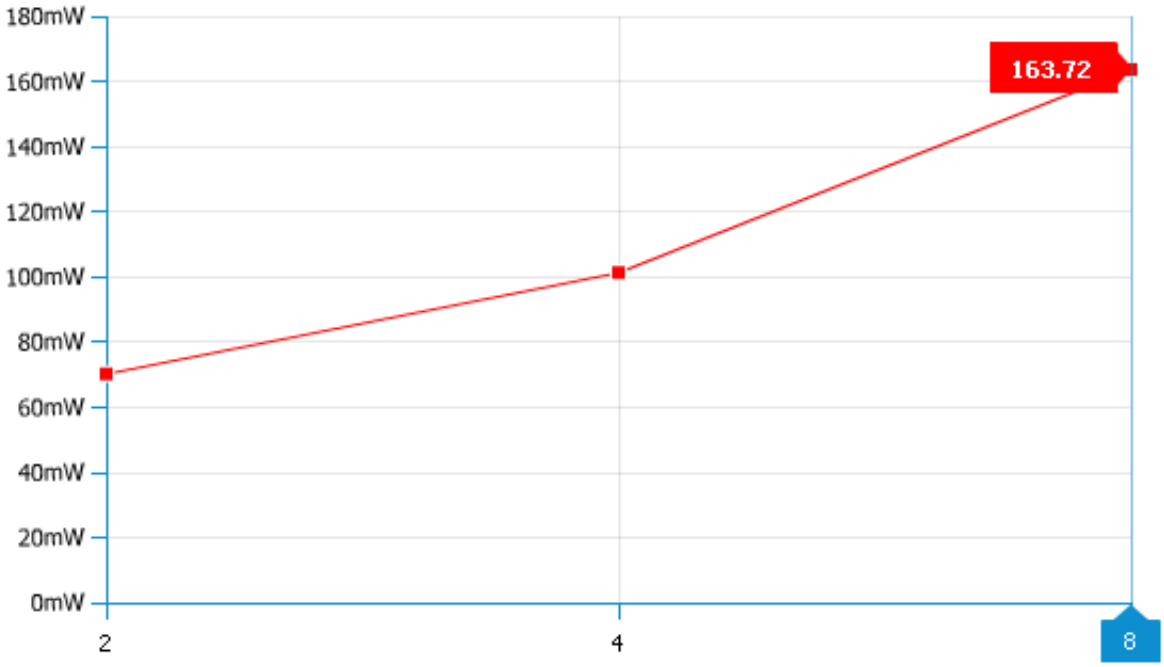
Utilizando essa configuração base, foi variada a largura de busca que assumiu os valores 2, 4 e 8. Valores que representam a quantidade de instruções inseridas no *pipeline* por ciclo de *clock*. Nos testes realizados, a largura de decodificação acompanhou os valores da largura de busca. A Figura 5.21 apresenta os resultados do consumo de potência obtidos na fila de busca com a variação na largura. A Figura 5.21a apresenta os resultados do experimento Fatorial enquanto que a Figura 5.21b exibe os resultados do experimento Fibonacci e a Figura 5.21c do experimento Multiplicação de Matrizes.



(a)



(b)

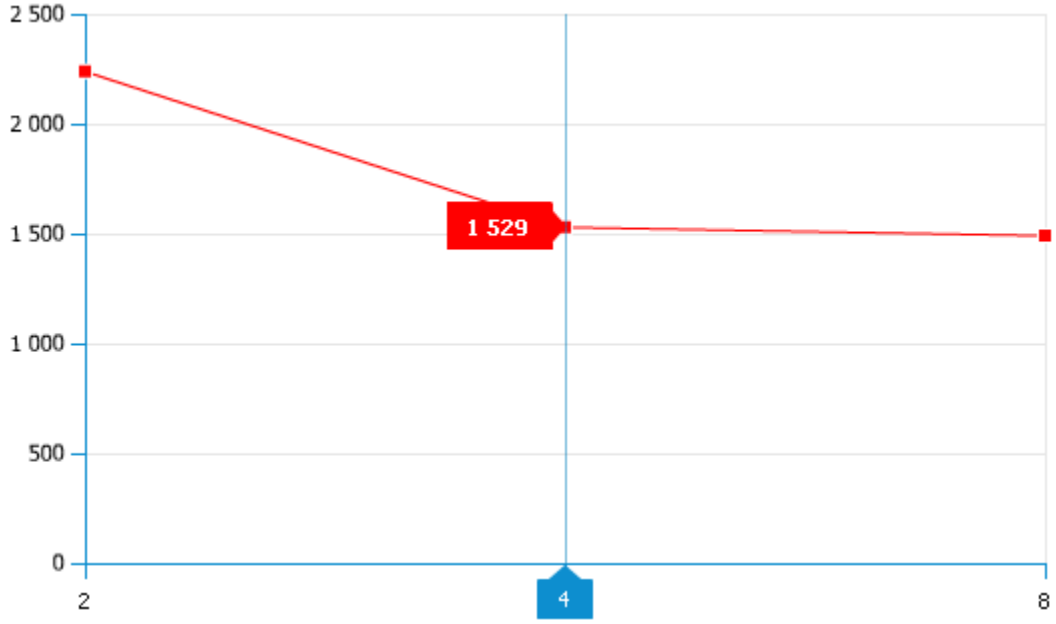


(c)

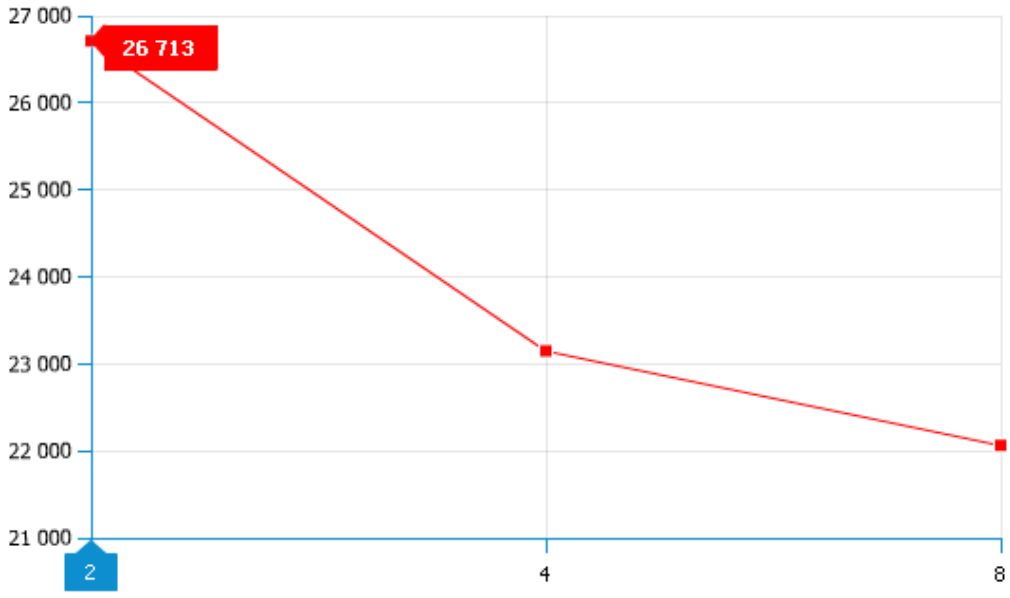
Figura 5.21. Consumo na fila de busca com variação na largura de busca para Fatorial (a), Fibonacci (b) e Multiplicação de Matrizes (c)

É possível observar que o aumento na largura de busca provoca uma maior dissipação de potência dinâmica. No experimento Fatorial ocorreu um aumento de 110,8% no consumo da fila de busca se alterada a largura de busca de 2 para 4 e um aumento de 62% quando a

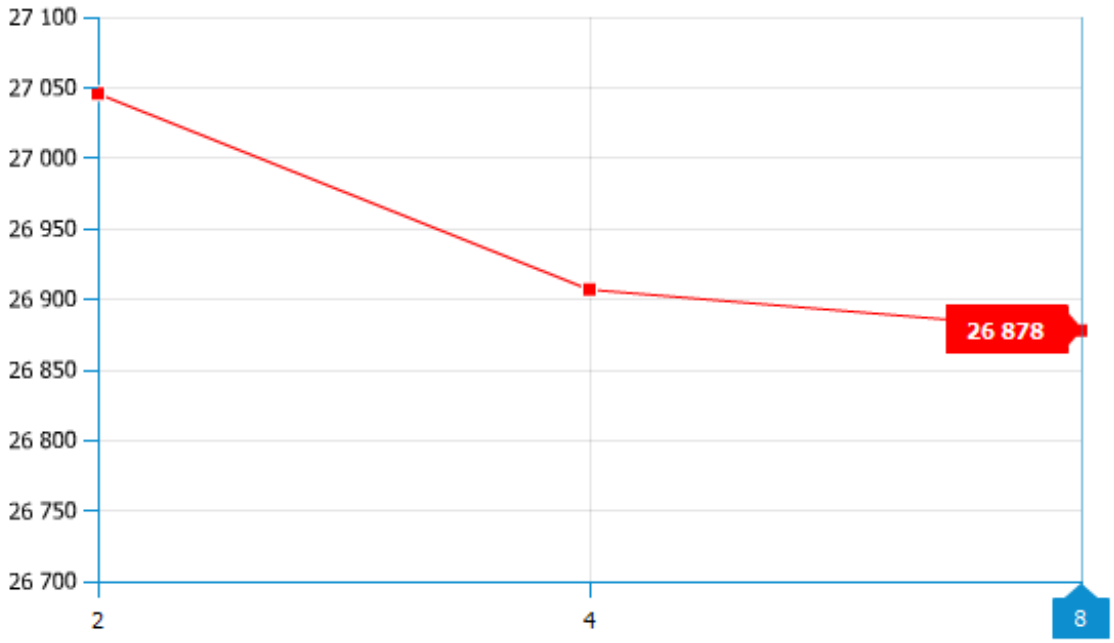
largura passa de 4 para 8. É com essa largura que a fila de busca apresenta o maior consumo, com 27,19mW. Já para o experimento Fibonacci ocorre um aumento de 43,13% no consumo da fila de busca quando a largura passa de 2 para 4 e um aumento de 21,3% no consumo quando a largura é alterada de 4 para 8. No experimento Multiplicação de Matrizes, o consumo de potência na fila de busca aumenta 44,47% quando a largura de busca passa de 2 para 4. Quando a largura de busca aumenta de 4 para 8, o aumento observado foi de 61,53%. Apesar do aumento no consumo, larguras maiores obtiveram melhores desempenhos nos testes, como apresentado na Figura 5.22.



(a)



(b)



(c)

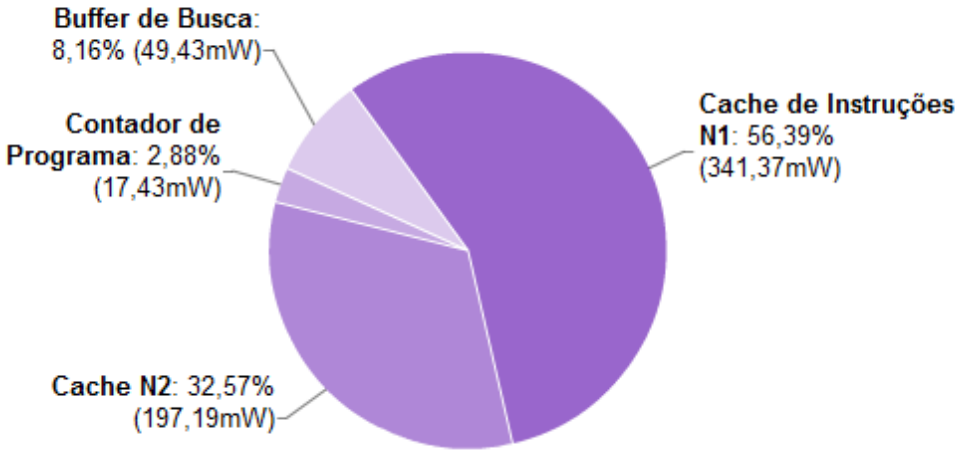
Figura 5.22. Ciclos para execução dos experimentos Fatorial (a), Fibonacci (b) e Multiplicação de Matrizes (c) com variação na largura de busca

Os experimentos Fatorial, Figura 5.22a, Fibonacci, Figura 5.22b, e Multiplicação de Matrizes, Figura 5.22c, mostram que há uma redução na quantidade de ciclos necessários para a execução de acordo com o aumento na largura de busca. Isso provocou um ganho de desempenho de cerca 33,3% no experimento Fatorial e um ganho de 17,4% no experimento Fibonacci. No experimento Multiplicação de Matrizes, a quantidade de ciclos para a execução passou de 27046 com largura de busca igual a 2 para 26878 com largura de busca igual a 4.

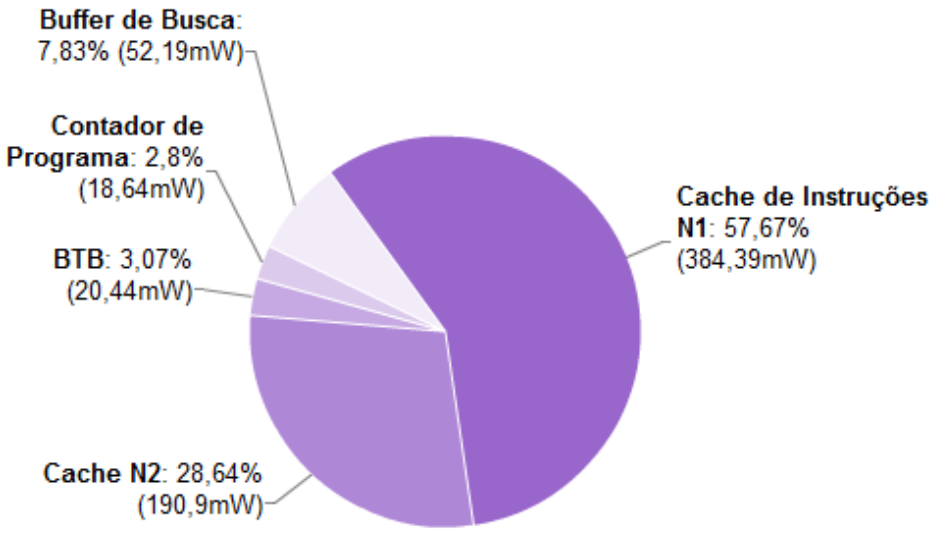
5.2.6 Distribuição do Consumo

Os experimentos realizados buscaram atingir uma gama de configurações a fim de apresentar o potencial da ferramenta na análise e estimativa do consumo de potência dinâmica. A capacidade da ferramenta de permitir a simulação de diferentes configurações arquiteturais pode ser explorada de modo a verificar quais estruturas são responsáveis pela maior parcela do consumo e buscar uma redução na dissipação de potência nelas caso seja necessário. A Figura 5.23 apresenta a distribuição no consumo do experimento Fatorial, considerando *caches* L1 de 2KB, *cache* L2 de 4KB, associatividade 4, 16 palavras por bloco, política de escrita LRU e largura de busca e decodificação igual a 4. Os testes variaram o previsor de

desvio, utilizando tanto o previsor não-tomado, Figura 5.23a, quanto o previsor BTB-PHT, Figura 5.23b, com uma BTB de 512 Bytes.



(a)

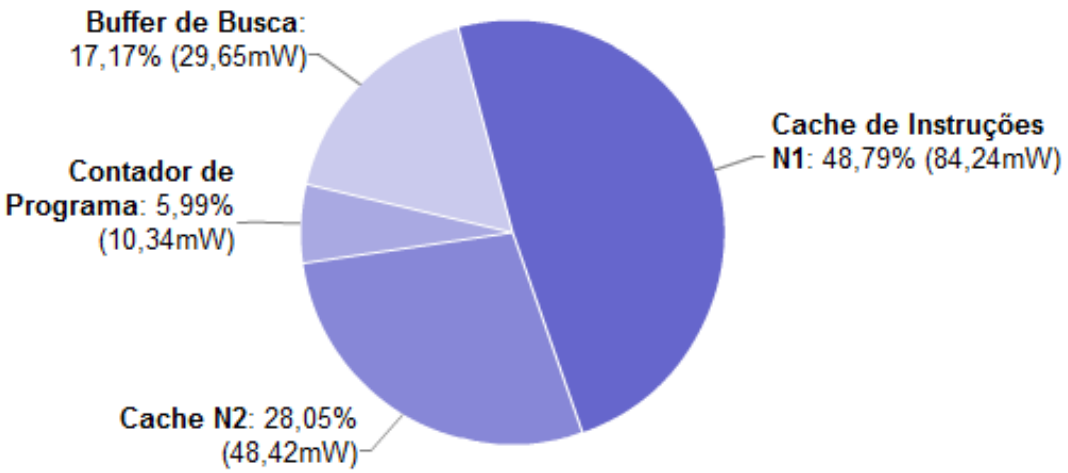


(b)

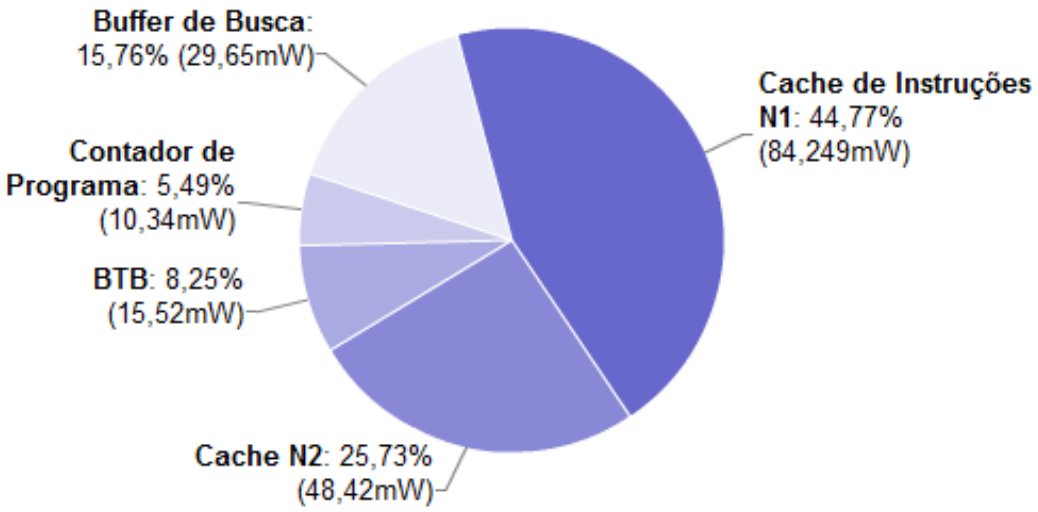
Figura 5.23. Distribuição do consumo no experimento Fatorial com previsor não-tomado (a) e BTB-PHT (b)

É possível observar por meio da Figura 5.23 que a maior parte do consumo da fase de busca se concentrou na *cache* de instruções para o experimento Fatorial. Mais da metade do gasto ocorreu nessa estrutura independente da técnica de previsão utilizada. O uso do previsor BTB-PHT faz com que uma nova estrutura seja adicionada ao processador, a BTB. O consumo que ela provocou foi pequeno se comparada com os outros elementos dessa fase, com 20,44mW.

Já a Figura 5.24 apresenta a distribuição do consumo considerando o experimento Fibonacci. A configuração utilizada apresenta *cache* L1 de 8KB, *cache* L2 de 16KB, associatividade 4, 16 palavras por bloco, política de substituição LRU e larguras de busca e decodificação 4. Também foram variados os previsores de desvio. A Figura 5.24a apresenta a distribuição no experimento de Fibonacci utilizando o previsor não-tomado e a Figura 5.24b exibe a distribuição do consumo com o uso do previsor BTB-PHT com uma BTB de 512Bytes.



(a)



(b)

Figura 5.24. Distribuição do consumo no experimento Fibonacci com previsor não-tomado (a) e BTB-PHT (b)

O experimento de Fibonacci exige mais de todas as estruturas presentes no estágio de busca, o que faz com que a distribuição do consumo seja mais equilibrada. Enquanto no experimento Fatorial o consumo nos dois níveis de *cache* correspondeu a 86,31% do consumo total da fase de busca com o uso do previsor BTB-PHT, no experimento Fibonacci, esse consumo correspondeu a 70,5%. Há um aumento no percentual gasto na BTB se comparado os dois experimentos. No experimento Fatorial a BTB corresponde a 3,07% do consumo e no Fibonacci o gasto com a BTB é de 8,25%.

O mesmo comportamento foi observado com o experimento Multiplicação de Matrizes. Os testes realizados com esse experimento apresentam a seguinte configuração: *cache* L1 de 1KB, *cache* L2 de 2KB, associatividade 4, 16 palavras por bloco, política de substituição LRU e larguras de busca e decodificação igual a 4. A Figura 5.25a apresenta a distribuição desse experimento utilizando o previsor não-tomado e a Figura 5.25b exibe a distribuição do consumo com o uso do previsor BTB-PHT com uma BTB de 256Bytes.

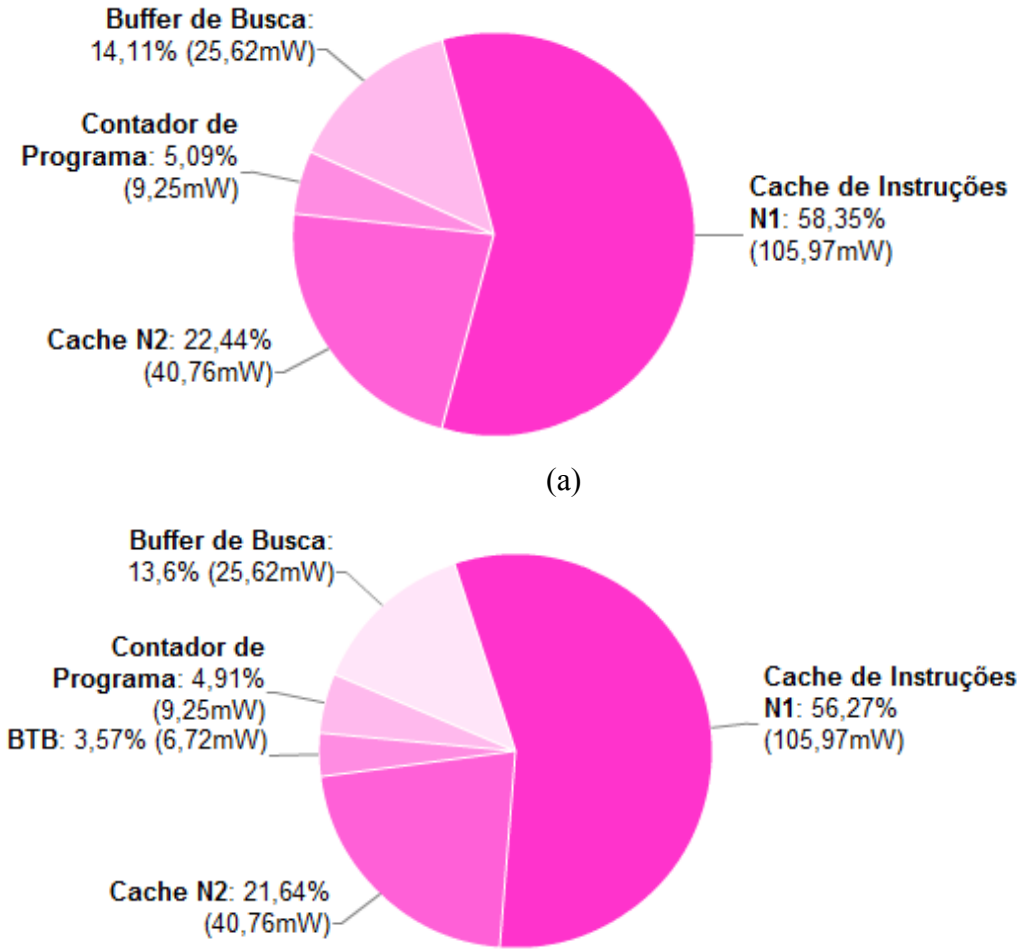


Figura 5.25. Distribuição do consumo no experimento Multiplicação de Matrizes com previsor não-tomado (a) e BTB-PHT (b)

As *caches* L1 e L2 representam mais de 77% do consumo total do experimento quando utilizado o previsor de desvios BTB-PHT e cerca de 80% com o previsor não-tomado. O consumo estimado para a BTB foi de 6,72mW, cerca de 40% menor se comparado com o experimento Fibonacci, que apresentou consumo de 15,52mW. Ainda assim, as *caches* se mostraram como as maiores responsáveis pela dissipação de potência dinâmica em um processador superescalar.

5.3 Considerações Finais

O presente capítulo apresentou alguns experimentos realizados com a ferramenta D-Power. O objetivo dos experimentos foi mostrar a variabilidade de configurações que a ferramenta apresenta, bem como as possíveis análises que podem ser realizadas. Análises não ficam restritas às aqui descritas, pois diferentes aplicações podem possuir diferentes comportamentos e mais dados podem ser coletados da ferramenta.

CAPÍTULO

6

Conclusões e Trabalhos Futuros

O consumo de potência vem sendo alvo de diversas pesquisas, pois, em sistemas computacionais, esse consumo é um fator limitante em relação à frequência do *clock*, quantidade de transistores no chip, tamanho de dispositivos portáveis, tempo de carga de baterias, entre outros. Nesse sentido é importante prover formas de estimar o consumo de potência de um processador em níveis iniciais do projeto para que as restrições de consumo possam ser atingidas sem comprometer o andamento do projeto.

A ferramenta D-Power foi apresentada, provendo estimativa do consumo de potência dinâmica com base na detecção da atividade de chaveamento que ocorre nas estruturas e conexões internas de um processador superescalar. O presente capítulo apresenta as conclusões referentes à implementação e aos experimentos realizados com a ferramenta e trabalhos a serem realizados na continuidade da pesquisa.

6.1 Conclusões Gerais

Desde a invenção do computador diversas técnicas e métodos vêm sendo pesquisados e aprimorados no intuito de se conquistar algum ganho no desempenho dessas máquinas. Arquiteturas superescalares são utilizadas para melhorar o desempenho dos sistemas computacionais, uma vez que utilizam hardwares adicionais que permitem a execução paralela de instruções.

Essa adição de hardware, porém, provoca um aumento no consumo de potência do processador, podendo causar sérios problemas como redução do tempo de autonomia de baterias, aumento na dissipação de calor e diminuição da vida útil dos componentes. Nesse sentido, durante o projeto de um sistema, além da área e do desempenho, o consumo de potência também passa a ser um importante fator a ser considerado. Isso implica em uma necessidade de ferramentas que possam prover estimativas do consumo em fases iniciais do projeto.

É justamente essa a principal contribuição do trabalho, que apresentou o D-Power, uma ferramenta VHDL com capacidade de estimar o consumo de potência dinâmica em arquiteturas superescalares. Maior responsável pelo consumo de potência em circuitos CMOS, a potência dinâmica é dissipada quando ocorrem trocas de valores nos sinais do circuito. Essa troca de valores, as atividades de chaveamento, é monitorada pelo D-Power para que o consumo possa ser estimado com base em parâmetros tecnológicos fornecidos pelo usuário. A ferramenta foi validada por meio da comparação do consumo de circuitos por ela utilizados com resultados obtidos em descrições desses mesmos circuitos na ferramenta SPICE.

A ferramenta permite a variação de diversos parâmetros arquiteturais para que seja possível analisar diferentes configurações e quais implicações essas configurações terão no consumo e desempenho de um sistema a ser projetado. Tamanho e organização de *caches*, políticas de substituição e escrita na *cache*, largura de busca, decodificação e remessa, previsor de desvio e quantidade de unidades funcionais são alguns exemplos de parâmetros que podem ser configurados durante a simulação, bem o tamanho de todas as estruturas utilizadas no *pipeline*.

Experimentos foram introduzidos com o objetivo de apresentar o potencial da ferramenta. Diversas configurações foram utilizadas nesses experimentos e algumas conclusões puderam ser obtidas por meio da análise dos resultados. Os resultados mostraram que a maior parte do consumo do estágio de busca ocorre nas *caches*. O constante acesso a essas estruturas fez com que o consumo nelas fosse maior do que o consumo apresentado por outros componentes desse estágio.

Outros comportamentos que puderam ser observados com o uso da ferramenta foram a variação da quantidade de palavras por bloco e a associatividade da *cache*, fatores que possuem relação estreita com a aplicação a ser executada. O aumento na quantidade de palavras por bloco muitas vezes resultou em uma redução no consumo de potência, uma vez

que, com mais palavras, há um menor número de linhas na *cache* e uma consequente redução no número de bits utilizados para o armazenamento do rótulo da informação.

Os resultados dos experimentos mostraram que o número de acertos na *cache* pode beneficiar tanto o desempenho quanto o consumo do sistema. Isso porque quanto maior for a taxa de acerto, menos vezes serão buscados dados em níveis secundários da memória. O acesso a esses níveis é mais lento, nesse sentido há uma melhora no desempenho. Aumentando a taxa de acertos, percebeu-se também que menos informações eram trocadas na *cache*, o que resultava em menos atividade de chaveamento e uma menor quantidade de potência dissipada.

A mesma relação desempenho/consumo foi percebida nos testes que promoveram a variação dos previsores de desvio. Técnicas pouco eficientes aumentam muito o número de instruções inseridas no *pipeline*, já que várias instruções são buscadas de forma incorreta. Essas instruções requerem um gasto para serem processadas, o que aumenta bastante o consumo de potência.

As análises permitidas pela ferramenta mostram-se de grande utilidade para avaliar o comportamento de arquiteturas superescalares de acordo com determinada aplicação. A avaliação do consumo por ela estimado pode ser benéfica para a verificação do comportamento do consumo e desempenho do sistema sob diferentes configurações. Embora os consumos apresentados pelo D-Power possam não ter precisão absoluta, a validação da ferramenta apresentou resultados muito próximos dos reais, mesmo realizando a estimativa em um nível inicial do projeto. Ainda assim, é importante considerar o consumo relativo apresentado pela ferramenta, que produzirá valores proporcionais à configuração utilizada. A variação nas configurações permitirá uma análise sobre a dissipação de potência, indicando a diferença no consumo e desempenho de cada configuração e provendo informações sobre quais estruturas concentram a maior parte do consumo, tornando o D-Power uma importante ferramenta no auxílio à tomada de decisão durante o projeto de um processador superescalar.

6.2 Trabalhos Futuros

A ferramenta desenvolvida apresenta grande potencial se comparada a outras ferramentas que realizam a estimativa do consumo de potência. Na continuação do trabalho, pretende-se realizar o mapeamento das atividades de chaveamento do restante das estruturas que compõem o *pipeline* superescalar implementado, no intuito de prover a estimativa de consumo de todo o processador.

O D-Power considera o gasto com a potência dinâmica, responsável pela maior parte da dissipação de potência em um processador superescalar. Mesmo assim, pretende-se adicionar modelos do consumo estático com base em parâmetros tecnológicos, para que valores mais precisos do consumo possam ser estimados pela ferramenta.

Com o objetivo de melhorar a usabilidade da ferramenta, uma interface gráfica na qual os parâmetros possam ser inseridos também está prevista, facilitando tanto o uso do D-Power quanto o acesso aos dados resultantes das simulações por ele realizadas.

Referências

AKITA, J.; ASADA, I. A Method for Reducing Power Consumption of CMOS Logic Based on Signal Transition Probability. *IEICE Transactions on Electronics*, v. E78-C, n. 4, pp. 436-440, 1995.

BAHAR, R.I.; MANNE, S. Power Energy Reduction via Pipeline Balancing. In: *Proceedings of the 28th International Symposium on Computer Architecture (ISCA, 2001)*, Gõtenborg, pp. 218-229, 2001.

BECKER, J.; HUEBNER, M.; ULLMANN, M. Power Estimation and Power Measurement of Xilinx Virtex FPGAs: Trade-offs and Limitations. In: *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI' 03)*, São Paulo, pp. 8-11, 2003.

BROOKS, D. et al. New Methodology for Early-Stage, Microarchitecture-Level Power-Performance Analysis of Microprocessors. *IBM Journal of Research and Development*, v. 47, n. 5/6, pp. 653-670, 2003.

BROOKS, D.; TIWARI, V.; MARTONOSI, M. Wattch: a Framework for Architectural-Level Power Analysis and Optimizations. In: *Proceedings of the 27th International Symposium on Computer Architecture*, Vancouver, pp. 83-94, 2000.

BURGER, D.; AUSTIN, T.M. *The SimpleScalar Tool Set, Version 2.0.* Technical Report, n.1342, Madison, 1997.

CALDER, B.; GRUNWALD, D.; EMER, J. A System Level Perspective on Branch Architecture Performance. In: *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Ann Arbor, pp. 199-206, 1995.

CHABINI, N.; ABOULHAMID, E.M.; SAVARIA, Y. Determining Schedules for Reducing Power Consumption Using Multiple Supply Voltages. In: *IEEE International Conference on Computer Design (ICCD'01)*, Austin, pp. 546-552, 2001.

CHANG, X. et al. Adaptive Clock Gating Technique for Low Power IP Core in SoC Design. In: *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'07)*, New Orleans, pp. 2120-2123, 2007.

DEVADAS, S.; MALIK, S. A Sourvey of Optimization Techniques Targeting Low Power

VLSI Circuits. In: *Proceedings of 32th ACM/IEEE Conference on Design Automation*, San Francisco, pp. 242-247, 1995.

DIEFENDORFF, K. PC Processor Microarchitecture: A Concise Review of the Techniques Used in Modern Pc Processors. *Microprocessor Report*, v. 13, n. 9, pp. 68-82, 1999.

ELLSWORTH, M.J. Chip Power Density and Module Cooling Technology Projections for the Current Decade. In: *Proceedings of the 9th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITHERM'04)*, Las Vegas, v. 2, pp. 707- 708, 2004.

FORNACIARI, W. et al. A VHDL-Based Approach for Power Estimation of Embedded Systems**.** *Journal of Systems Architecture: the EUROMICRO Journal*, v. 44, n. 1, pp. 37-61, 1997.

GARBUS, E. Designing Performance Into i960 Superscalar Microprocessors. In: *Computer Society International Conference (COMPCON)*, San Francisco, pp. 354-357, 1992.

GONÇALVES, R.A. *PowerSMT: Ferramenta Para Análise de Consumo de Potência em Arquiteturas SMT*. Tese de Mestrado, Maringá: UEM, 2008. 160 p.

GONÇALVES, R.A.; GONÇALVES, R.A.L. PowerSMT: Ferramenta para Análise de Consumo de Potência em Arquiteturas SMT. In: *IX Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD-SSC'08)*, Campo Grande, pp. 177-184, 2008.

GONÇALVES, R.A.L. *Arquiteturas Multi-Tarefas Simultâneas: SEMPRE – Arquitetura SMT com Capacidade de Execução e Escalonamento de Processos*. Tese de Doutorado, Porto Alegre: UFRGS, 2000. 134 p.

GOWAN, M.K.; BIRO, L.L.; JACKSON, D.B. Power Considerations in the Design of the Alpha 21264 Microprocessor. In: *Proceedings of the 35th Annual Design Automation Conference*, San Francisco, pp. 726-731, 1998.

GUY, B.M.; HAGGARD, R.L. High Performance Branch Prediction. In: *Proceedings of the 28th Southeastern Symposium on System Theory (SSST '96)*, Baton Rouge, pp. 472-476, 1996.

HU, J.S. et al. Using Dynamic Branch Behavior for Power-Efficient Instruction Fetch**.** In: *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, Tampa**,** pp. 127-132, 2003.

HUDA, S.; MALLICK, M.; ANDERSON, J.H. Clock Gating Architectures for FPGA Power Reduction. In: *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL'09)*, Praga, pp. 112-118, 2009.

INTEL CORPORATION. *Designing for Power*. Santa Clara: Intel Corporation, 2005, 8 p.

JAGGAR, D. *Branch Cache*. G06F 01 3/00. US Patent nº. 5506976. 09 ago. 1994, 9 abr. 1996. Advanced Risc Machines Limited.

JEVTIC, R.; CARRERAS, C.; CAFFARENA, G. Switching Activity Models for Power Estimation in FPGA Multipliers. In: *Proceedings of the 3th International Workshop on Applied Reconfigurable Computing*, Mangaratiba, pp. 27-29, 2007.

LANDMAN, P. High-level Power Estimation. In: *Proceedings of the 1996 International Symposium on Low Power Electronics and Design*, Monterey, pp. 29-35, 1996.

LEE, J.K.F.; SMITH, A.J. Branch Prediction Strategies and Branch Target Buffer Design**.** *Computer*, v. 17, n. 1, pp. 6-22, 1984.

LEE, K.; EVANS, S.; CHO, S. Accurately Approximating Superscalar Processor Performance from Traces. In: *Proceedings of the IEEE Int'l Symposium on Performance Analysis of Systems and Software (ISPASS)*, Boston, pp. 238-248, 2009.

LI, H. et al. DCG: Deterministic Clock-Gating for Low-Power Microprocessor Design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 12, n. 3, pp. 245-254, 2004.

LINK , G.M.; VIJAYKRISHNAN, N. Thermal Trends in Emerging Technologies. In: *Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED'06)*, San Jose, pp. 632-639, 2006.

LU, Y.H.; DE MICHELI, G. Comparing System Level Power Management Policies. *IEEE Design & Test of Computers*, v. 18, n. 2, pp. 10-19, 2001.

MAMIDIPAKA, M. et al. *Leakage Power Estimation in SRAMs*. Technical Report, n.03-32, Irvine, 2003. 15p.

MANNE, S.; GRUNWALD, D.; KLAUSER, A. Pipeline Gating: Speculation Control for Energy Reduction. In: *Proceedings of the 25th International Symposium on Computer Architecture*, Barcelona, pp. 132-141, 1998.

MATHEW, B.K. *The Perception Processor*. Tese de Doutorado, Utah: Universidade de Utah, 2004. 167 p.

MIPS TECHNOLOGIES. *MIPS R10000 Microprocessor User's Manual Version 2.0*. Mountain View: MIPS Technologies, 1996. 34p.

MONTEIRO, M.A. *Introdução à Organização de Computadores*. Rio de Janeiro: LTC, 2007, 5ª ed. 696 p.

MOUDGILL M.; WELLMAN J.D.; MORENO, J.H. Environment for PowerPC Microarchitecture Exploration. *IEEE Micro*, v. 19, n. 3, pp. 15-25, 1999.

MUSOLL, E. Speculating to Reduce Unnecessary Power Consumption. ACM *Transactions on Embedded Computing Systems*, v. 2, n. 4, pp. 509-536, 2003.

ÖNDER, S.; GUPTA, R. Superescalar Execution with Dynamic Data Forwarding. In: *Proceedings of the ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, Paris, pp. 130-135, 1998.

ORCAD INC. *PSpice User's Guide*. Beaverton: OrCAD Inc, 1998, 436 p.

PATTERSON, D.A.; HENNESSY, J.L. *Organização e Projeto de Computadores: A Interface Hardware/Software*. Rio de Janeiro: LTC, 2000, 2ª ed. 551 p.

PONOMAREV, D.; KUCUK, G.; GHOSE, K. AccuPower: An Accurate Power Estimation Tool for Superscalar Microprocessors. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'02)*, Paris, pp. 124-129, 2002.

PRICE, C. *MIPS IV Instruction Set*, *revision 3.2*. Mountain View: MIPS Technologies, 1995. 334 p.

SIMA, D. The Design Space of Register Renaming Techniques. *IEEE Micro*, v. 20, n. 5, pp. 70-83, 2000.

SMITH, A.J. Cache Memories. *ACM Computing Surveys (CSUR)*, v. 14, n. 3, pp. 473-530, 1982.

SMITH, J.E.; SOHI, G.S. The Microarchitecture of Superscalar Processors. *Proceedings of the IEEE*, v. 83, n. 12, pp. 1609-1624, 1995.

SNOWDON, D.C.; RUOCCO, S.; HEISER, G. Power Management and Dynamic Voltage Scaling: Myths and Facts. In: *Proceedings of the 2005 Workshop on Power Aware Real-time Computing*, New Jersey, 2005.

STAMMERMANN, A. et al. System Level Optimization and Design Space Exploration for Low Power. In: *Proceedings of the 14th International Symposium on Systems Synthesis*, Montreal, pp. 142-146, 2001.

THOZIYOOR, S.; MURALIMANOHAR, N.; JOUPPI N.P. *CACTI 5.0*. Technical Report, n. HPL-2007-167, Palo Alto, 2007. 81 p.

TOMASULO, R.M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal*, v. 1, n. 11, pp. 25-33, 1967.

TOSHIBA CORPORATION. *Synopsys DesignPower/PowerCompiler Design Manual*. Tokio: Toshiba Corporation, 1997, 54 p.

UTAMAPHETHAI, N.; BLANTON, R.D.; SHEN, J.P. Relating Buffer-Oriented Microarchitecture Validation to High-Level Pipeline Functionality. In: *Proceedings of the 6th IEEE International High-Level Design Validation and Test Workshop*, Monterey, pp. 03-08, 2001.

VENKATACHALAM, V.; FRANZ, M. Power Reduction Techniques for Microprocessor Systems. *ACM Computer Surveys*, v. 37, n. 3, pp. 195-237, 2005.

WENANDE, S.; CHIDESTER, R. Xilinx Takes Power Analysis to New Levels with XPower. *Xcell Journal Online*, v. 41, pp. 26-27, 2001.

WU, Q.; PEDRAM, M.; WU, X. Clock-Gating and Its Application to Low Power Design of

Sequential Circuits. In: *Proceedings of the IEEE Custom Integrated Circuits Conference*, Santa Clara, pp. 479-482, 1997.

XILINX INC. *XPower Tutorial: FPGA Design*. San Jose: XILINX Inc, v 1.3, 2002, 24 p.