UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MARCOS RAFAEL BOSCHETTI

Mecanismos de Reconfiguração Dinâmica Aplicados ao Projeto de um Processador de Imagens Reconfigurável

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof Dr. Sergio Bampi
Orientador

Porto Alegre, novembro de 2004

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Boschetti, Marcos Rafael

Mecanismos de Reconfiguração Dinâmica Aplicados ao Projeto de um Processador de Imagens Reconfigurável / Marcos Rafael Boschetti. – Porto Alegre: PPGC da UFRGS, 2004.

103 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2004. Orientador: Sergio Bampi.

1. Arquiteturas reconfiguráveis. 2. Processamento de imagens. 3. FPGA. 4. CSoC. I. Bampi, Sergio. II. Título.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

SCORE	Stream Computations Organized for Reconfigurable Execution
DRIP	Dynamically Reconfigurable Image Processor
CSoC	Configurable System-on-Chip
PLICE	Programmable Low Impedance Circuit Element
CAD	Computer Aided Design
ASIC	Application Specific Integrated Circuit
IP	Intellectual Property
RTR	Run-time Reconfigurable
ASP	Application Specific Processor
GPP	General Purpose Processor
RISC	Reduced Instruction Set Computer
VLSI	Very Large Scale Integration
SoC	System-on-Chip
PDA	Personal Digital Assistant
DSP	Digital Signal Processing
SIMD	Single Instruction Multiple Data
ULA	Unidade Lógica e Aritmética
DMA	Direct memory Access
I/O	Input/Output
LE	Logic Element
ISP	Instruction Set Processor
USB	Universal Serial Bus
HAL	Hardware Abstraction Layer
VHDL	VHSIC Hardware Description Language
LUT	Look-up Table
FIFO	First in First out

MAC	Multiplica Acumula
RAM	Random Access Memory
SRAM	Static Random Access Memory
FIR	Finite Impulse Response
RTOS	Real-Time Operating System
VDR	Visual interface for Dynamic Reconfiguration
MCMU	MultiContext Management Unit

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

As modernas aplicações em diversas áreas como multimídia e telecomunicações exigem arquiteturas que ofereçam altas taxas de processamento. Entretanto, os padrões e algoritmos mudam com incrível rapidez o que gera a necessidade de que esses sistemas digitais tenham também por característica uma grande flexibilidade.

Dentro desse contexto, tem-se as arquiteturas reconfiguráveis em geral e, mais recentemente, os sistemas reconfiguráveis em um único chip como soluções adequadas que podem oferecer desempenho, sendo, ao mesmo tempo, adaptáveis a novos problemas e a classes mais amplas de algoritmos dentro de um dado escopo de aplicação. Este trabalho apresenta o estado-da-arte em relação a arquiteturas reconfiguráveis nos meios acadêmcio e industrial e descreve todas as etapas de desenvolvimento do processador de imagens reconfigurável DRIP (*Dynamically Reconfigurable Image Processor*), desde suas origens como um processador estático até sua última versão reconfigurável em tempo de execução.

O DRIP possui um *pipeline* composto por 81 processadores elementares. Esses processadores constituem a chave do processo de reconfiguração e possuem a capacidade de computar um grande número de algoritmos de processamento de imagens, mais específicamente dentro do domínio da filtragem digital de imagens. Durante o projeto, foram desenvolvidos uma série de modelos em linguagem de descrição de hardware da arquitetura e também ferramentas de software para auxiliar nos processos de implementação de novos algorimos, geração automática de modelos VHDL e validação das implementações.

O desenvolvimento de mecanismos com o objetivo de incluir a possibilidade de reconfiguração dinâmica, naturalmente, introduz *overheads* na arquitetura. Contudo, o processo de reconfiguração do DRIP-RTR é da ordem de milhões de vezes mais rápido do que na versão estaticamente reconfigurável implementada em FPGAs Altera. Finalizando este trabalho, é apresentado o projeto lógico e elétrico do processador elementar do DRIP, visando uma futura implementação do sistema diretamente como um circuito VLSI.

Palavras-chave: Arquiteturas reconfiguráveis, Processamento de imagens, FPGA, CSoC.

Dynamic Reconfiguration Mechanisms Applied to a Reconfigurable Image Processor Design

ABSTRACT

Contemporary applications in different fields like multimedia and telecommunications demand efficient architectures capable to provide high processing power. However, standards and algorithms are changing at a fast pace, which makes flexibility an important design parameter.

In this context, reconfigurable architectures in general and, more recently, configurable systems-on-chip are suitable solutions that can offer performance, and, at the same time, adaptability to new tasks and other algorithm classes within a certain application scope. This work presents the state-of-art in reconfigurable architectures in the academic and industrial environment and describes all the steps in the development of the dynamically reconfigurable image processor DRIP from an initial static implementation to a run-time reconfigurable version.

The DRIP pipeline is composed by 81 processor elements. These processors are the key for the reconfiguration process. They can process a large number of digital image processing algorithms, more specifically in the application domain of digital image filtering. During the development several models for the architecture were built in hardware description language. Moreover, software tools were created to help in the implementation, validation and automatic generation of algorithms for the DRIP architecture.

The design of mechanisms to include dynamic reconfiguration certainly introduces a set of overheads. Nevertheless, the reconfiguration process of DRIP-RTR is in the order of million times faster than in its statically reconfigurable version. In the final part of this work a logic and electrical design of the processor element is presented targeting a future VLSI implementation.

Keywords: Reconfigurable Architectures, Digital Image Processing, CSoC, Dynamic Reconfiguration.

1 INTRODUÇÃO

Este trabalho apresenta a evolução e desenvolvimento da arquitetura reconfigurável DRIP (*Dynamically Reconfigurable Image Processor*). O DRIP está focado na execução eficiente de algoritmos de processamento de imagens de baixo nível (exemplos desses algoritmos são citados na seção 4.1). Ao longo deste trabalho, serão mostrados os esforços realizados na construção e implementação da arquitetura com mecanismos de reconfiguração dinâmica e também os aspectos relacionados ao desenvolvimentos das ferramentas de software que auxiliam no processo de síntese do processador. Este capítulo apresenta a motivação, objetivos do trabalho e a maneira como este texto está organizado.

1.1 Motivação

Nos útltimos anos as arquiteturas reconfiguráveis introduziram um novo conjunto de alternativas para os projetistas de hardware. O campo de pesquisa vem sendo impulsionado pelas altas densidades dos dispositivos FPGAs modernos e pela combinação de software com sistemas reconfiguráveis mais complexos e de alto desempenho chamados CSoCs (*Configurable System-on-Chip*).

Os CSoCs oferecem grandes possibilidades de inovações arquiteturais. Em virtude de sua capacidade de customizar módulos de hardware, é possível otimizar controle, parte operativa e interconexões de acordo com os requisitos específicos de determinado algoritmo.

Sistemas reconfiguráveis em tempo de execução (RTR), particularmente, podem adquirir benefícios excepcionais desse paradigma, adaptando-se para as necessidades instantâneas de uma aplicação. As arquiteturas reconfiguráveis trazem a possibilidade de combinar a programabilidade pós-fabricação dos processadores com o desempenho, economia de hardware e, muitas vezes, baixo consumo de energia de arquiteturas de propósito especial (ASICs e ASIPs). Como uma solução intermediária entre uma abordagem de computação genérica (software) e específica (hardware), arquiteturas reconfiguráveis constituem-se em uma ferramenta valiosa de exploração do espaço de projeto.

As aplicações contemporâneas de multimídia e telecomunicações trazem novos desafios aos projetistas. Além do alto desempenho exigido por algoritmos complexos, dois outros aspectos de projeto são cada vez mais importantes: consumo de energia e flexibilidade do hardware em adaptar-se para outras tarefas.

O emprego da capacidade de reconfiguração pode ajudar o projetista a alcançar esses parâmetros. O sucesso de muitos sistemas embarcados, por exemplo, está diretamente relacionado a um projeto de baixo consumo que maximize a autonomia do dispositivo. Em (DAVID et al., 2002) e (RABAEY, 1997) a computação reconfigurável é apresentada como um mecanismo para construção de arquiteturas eficientes do ponto de vista de

energia.

Flexibilidade do sistema reconfigurável é também um fator chave. Uma arquitetura flexível permite a modificação pós-venda de produtos e sistemas digitais através da adição de novos serviços ou pela redefinição da funcionalidade dos algoritmos, resultando em importantes vantagens comerciais não encontradas em ASICs.

Operações multimídia incluem algoritmos que exigem processamento eficiente e em tempo real. Soluções em análise de imagens e visão computacional são importantes em diversas aplicações industriais como robótica, segurança, imagens médicas e inspeção de cenas. Todas essas aplicações possuem características regulares e um nível inerente de paralelismo que podem ser explorados por sistemas reconfiguráveis.

O problema do processamento digital de imagens, mais especificamente, possui uma série de características que favorecem uma implementação reconfigurável como as já citadas alta regularidade e paralelismo. Nesse contexto a implementação espacial da computação traz grandes vantagens sobre uma implementação temporal.

No desenvolvimento de uma aplicação, analisando-se os requisitos do projeto, é realizada a escolha de implementação em hardware ou software. Em alguns sistemas, essas decisões são feitas considerando cada subtarefa, de modo que algumas são alocadas em hardware e outras em software. Hardware oferece desempenho, pois :

- É customizado para o problema. Não existe a latência extra para interpretação ou circuitos capazes de soluções mais genéricas.
- É rápido, devido ao paralelismo e execução espacial.

Implementações em software exploram um mecanismo de computação mais genérico, o qual interpreta as instruções e gera os sinais para a parte operativa, assim o software tem como características:

- Flexibilidade: as tarefas podem ser alteradas por uma simples mudança na seqüência de bits da instrução na memória.
- Relativamente lento: em virtude da execução temporal.
- Relativamente ineficiente: uma vez que operações não podem ser diretamente mapeadas para tarefas computacionais.

Em implementações espaciais (hardware), cada operador existe em um diferente ponto do espaço, permitindo a exploração de paralelismo além de alta taxa de processamento e baixa latência computacional.

Em implementações temporais (software), por sua vez, um pequeno número de recursos computacionais mais genéricos são reutilizados no tempo, permitindo que o processamento ocorra de maneira compacta. A figura 1.1 (DEHON; WAWRZYNEK, 1999) ilustra a computação espacial e a computação temporal de uma expressão matemática.

Um benefício chave da computação reconfigurável é que introduz uma classe de arquiteturas programáveis pós fabricação que suportam computação espacial.

Em arquiteturas puramente espaciais os bits para cada operação podem ser definidos uma vez e usados por um grande tempo de processamento. Isso permite aos blocos lógicos armazenar uma única instrução local aos operadores computacionais e interconexões.

Em arquiteturais temporais baseadas em software, o operador deve modificar-se a cada ciclo (muitas instruções), a fim de implementar a computação como uma seqüência

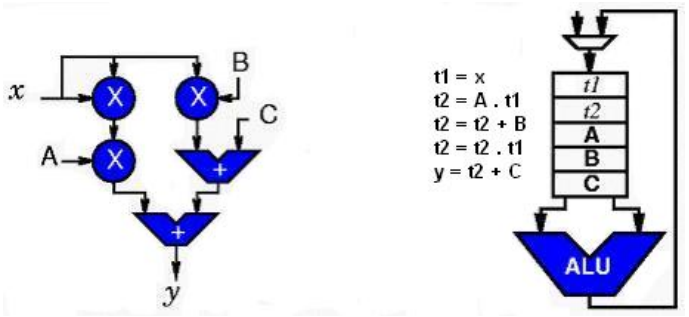

Figura 1.1: Execução espacial e temporal de Ax2 + Bx + C

de operações em um número pequeno de operadores ativos. Entretanto, quanto mais cedo forem definidas as operações menores podem ser os custos (*overhead*) de implementação. Um ASIC pode implementar apenas os circuitos e portas necessárias para o problema, não é preciso memória extra para as instruções ou circuitos para executar operações que não são necessárias para uma tarefa em particular.

Assim, um FPGA ou dispositivo reconfigurável precisa, por sua vez, armazenar uma única instrução (a configuração do circuito), enquanto que um processador necessita redefinir sua operação a cada ciclo, tendo de pagar um preço em mecanismos de distribuição de instruções e armazenamento.

Contudo, um tempo ”mais tarde“ para definição das operações traz a possibilidade de especializar o projeto para as necessidades de momento de uma dada aplicação. Se a operação tem de ser definida antes do conjunto de operadores, o projeto tem de ser muito mais geral para suportar todos os requisitos da aplicação.

Um exemplo apresentado em (DEHON; WAWRZYNEK, 1999) é o da filtragem digital. Um ASIC deve incluir multiplicadores de propósito geral e alocá-los de maneira a suportar as mais diferentes configurações de filtro. Um FPGA ou processador reconfigurável, pode esperar até que os requisitos de determinada aplicação sejam bem conhecidos. Como o problema específico sempre será mais simples que o geral, o dispositivo programável pode explorar essa vantagem e proporcionar uma implementação mais otimizada.

Na situação citada, os coeficientes do filtro poderiam estar embutidos nos multiplicadores do dispositivo reconfigurável reduzindo, assim, a área. Um multiplicador especializado pode ter até um quarto da área de um mais genérico e para casos particulares pode ser ainda menor.

Dessa forma, ao mesmo tempo em que uma arquitetura reconfigurável pode ser um dispositivo espacial de computação, ela tem a capacidade de esperar até o tempo mais tarde na definição das operações. Essas duas características juntas favorecem o desenvolvimento de aplicações de computação configurável de alto desempenho.

Essas arquiteturas oferecem um caminho a ser explorado no mundo da engenharia de processadores. A grande capacidade de adaptação que um computador possui como uma ferramenta, via programação, existe também para os dispositivos reconfiguráveis, não apenas para ser configurado a agir como um hardware de propósito especial para um problema particular, mas para ser reconfigurado para diferentes aplicações.

1.2 Objetivos e Organização do Texto

O objetivo deste trabalho é apresentar o caminho completo de desenvolvimento do processador reconfigurável DRIP a partir dos fundamentos lançados na arquitetura NP9

(ADÁRIO, 1997). Serão mostrados todos os passos de projeto que transformaram uma arquitetura inicialmente sem capacidades de reconfiguração em um processador reconfigurável em tempo de execução (DRIP-RTR). O processador atinge taxas de processamento que garantem operação em tempo real para uma ampla quantidade de algoritmos de processamento de imagens. Além disso, foi realizado o projeto lógico e elétrico do processador elementar do DRIP, visando uma implementação *full-custom* VLSI. Os trabalhos de projeto de hardware e desenvolvimento de software estão ilustrados na figura 1.2.

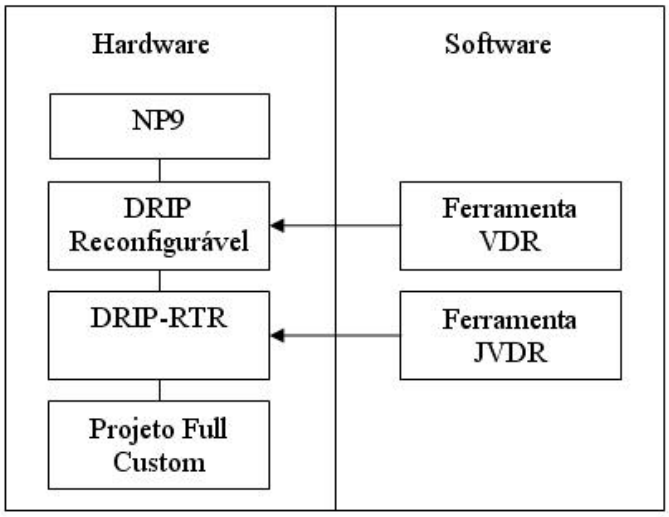

Figura 1.2: Hardware e Software no projeto na evolução do processador DRIP

O texto está estruturado da seguinte maneira:

Capítulo 2 Apresenta uma revisão sobre a evolução dos dispositivos programáveis de hardware, os quais impulsionaram o surgimento dos sistemas reconfiguráveis. Descreve também uma série de conceitos e caracteristicas referentes às arquiteturas reconfiguráveis como a taxonomia de classes de programabilidade, granularidade e métricas de avaliação. Por fim, apresenta as técnicas genéricas de reconfiguração dinâmica mais empregadas nos diversos projetos. De uma forma geral, uma série de conceitos apresentados neste capítulo são utilizados extensamente ao longo deste trabalho.

Capítulo 3 Analisa arquiteturas reconfiguráveis estado-da-arte. A maioria dos sistemas estudados neste capítulo podem ser categorizados como CSoCs, sendo, portanto, representantes do que existe de mais atual no campo de pesquisa. A maioria das arquiteturas apresentadas busca obter um desempenho superior em conjunto com uma ampla flexibilidade. Este capítulo mostra também os avanços realizados no sentido de definir sistemas operacionais que possibilitam a relocação de tarefas diretamente em hardware. Finalizando, analisa-se algums tendências no desenvolvimento de arquiteturas eficientes com destaque para os futuros sistemas reconfiguráveis multigranulares e adaptáveis para um maior número de classes de aplicações.

Capítulo 4 Mostra o projeto, evolução e implementação do processador DRIP, desde os conceitos inicias da arquitetura NP9 até o desenvolvimento dos mecanismos de reconfiguração dinâmica do DRIP-RTR e suas estruturas de controle. Este capítulo

apresenta ainda as ferramentas de CAD criadas para auxiliar o processo de síntese e implementação dos algoritmos, além de um estudo que deu origem ao projeto lógico e elétrico de uma construção VLSI do processador elementar da arquitetura.

Capítulo 5 A parte final deste estudo apresenta as conclusões realizadas e também enumera uma série de trabalhos futuros a serem realizados com a continuidade dos esforços de projeto. O desenvolvimento do DRIP-RTR abre caminho para diversos caminhos que podem ser explorados e assim viabilizar a realização de novas contribuições ao campo de pesquisa.

2 HARDWARE PROGRAMÁVEL

A computação reconfigurável iniciou seu desenvolvimento a partir do momento que tornou-se possível modificar a funcionalidade implementada em hardware. Este capítulo, primeiramente, apresenta a evolução dos dispositivos programáveis do tipo FPGA e, em seguida, descreve uma série de conceitos importantes no projeto de arquiteturas reconfiguráveis utilizados ao longo deste trabalho.

2.1 FPGAs: Evolução

Um grande passo para o desenvolvimento de sistemas programáveis de hardware ocorreu em 1986 com o advento dos FPGAs. Esses dispositivos proporcionam a definição da funcionalidade do circuito após o encapsulamento. Desde 1986 diversos FPGAs foram desenvolvidos por empresas como Altera, Xilinx, Actel e outras.

O progresso dos FPGAs tem sido impulsionado por três fontes principais:

- A tecnologia de circuitos integrados proporciona geometrias cada vez menores, transistores mais rápidos e menor custo por função.
- A arquitetura dos FPGAs tem evoluído através da incorporação de funcionalidades especiais e através de uma melhor estrutura hierárquica de interconexões.
- As condições de projeto tem avançado, com a popularização de módulos de propriedade intelectual (IP *cores*) e melhores ferramentas de projeto, possibilitando, inclusive, esforços de desenvolvimento distribuídos através da Internet

Além dos avanços na tecnologia dos FPGAs, esses dipositivos apresentam uma outra característica muito importante: são sistemas digitais de computação espacial que podem ser configurados para as necessidades imediatas de uma tarefa.

Fatores como tamanho, estrutura, número e conectividade dos blocos variam consideravelmente entre arquiteturas de FPGAs. A principal diferença, no entanto, é a tecnologia de programação. Assim, através do método de programação pode-se realizar uma classificação que divide os FPGAs em três grupos básicos: baseados em SRAM, programáveis por anti-fusível e estilo EPROM - EEPROM.

2.1.1 FPGAs Baseados em Anti-Fusível

Um anti-fusível é um elemento que pode ser programado uma única vez. A utilização de anti-fusíveis possui algumas vantagens:

- Anti-fusíveis possuem resistência e capacitância parasita significativamente inferior à transistores de passagem, diminuindo os atrasos RC nas linhas.

- Os anti-fusíveis são pequenos, tipicamente do tamanho de uma via, possibilitando grande densidade.

Os anti-fusíveis podem ser de duas categorias possíveis: silício amorfo ou dielétrico. Um anti-fusível de silício amorfo pode ser depositado no espaço entre duas camadas de metal, proporcionando isolação elétrica. Um pulso de programação de tensão (tipicamente 10V-12V) e duração suficientes podem ser aplicados através do elemento anti-fusível, fazendo com que as camadas superiores e inferiores de metal penetrem no silício amorfo, criando um canal condutivo bidirecional com uma resistência de cerca de 50 Ω (SKAHILL, 2002) (BROWN; VRANESIC, 2000).

Um anti-fusível baseado em dielétrico, consiste em uma camada de material dielétrico (chamado ONO - Oxide-Nitride-Oxide) colocado entre uma difusão N+ e polisilício (figura 2.1). Um sistema de anti-fusível desta categoria utilizado pela ACTEL é chamado de PLICE (*Programmable Low Impedance Circuit Element*).

Antes de ser programado um PLICE possui alta resistência. A programação consiste na aplicação de uma tensão de cerca de 16 V (TRIMBERGER, 1994) durante 1 milisegundo. Esse pulso de programação derrete o dielétrico, criando um canal condutivo de silício policristalino entre os eletrodos. O raio do canal é proporcional à corrente de programação, ou seja quanto maior for a corrente maior será o raio do canal e, consequentemente, menor a resistência.

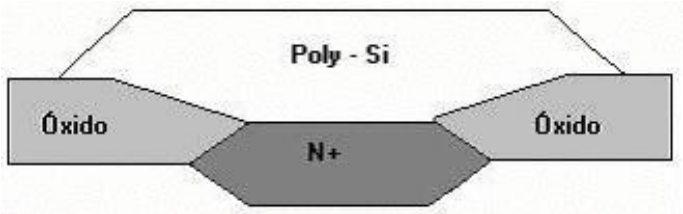

Figura 2.1: Exemplo de Plice

2.1.2 Tecnologia de Programação EPROM e EEPROM

Uma outra classe de FPGAs é baseada em tecnologia EPROM e EEPROM. As técnicas de programação neste caso operam através da injeção de carga em transistores com *gate* flutuante. O processo de programação, consiste em aplicar uma voltagem entre o dreno e o *gate* do transistor de modo a criar um campo elétrico suficiente para permitir que elétrons energizados saltem da região de dreno para o *gate* flutuante.

Os elétrons atraídos para o *gate* flutuante permanecem mesmo com a remoção da tensão, ou seja, trata-se de um modo não volátil de programação. As cargas armazenadas no *gate* flutuante alteram o *threshold* do transistor de um valor baixo para um valor mais alto.

Dessa forma um transistor EPROM pode estar em modo programado ou não programado. Na situação em que não houve programação, o funcionamento é como o de um transistor normal. Quando uma tensão maior que a tensão de *threshold* é aplicada no *gate*, um canal é induzido sob a região de *gate* permitindo que a corrente flua entre as regiões de fonte e dreno. Quando programado, devido ao maior valor de *threshold*, mesmo aplicando-se a tensão necessária no *gate* não ocorre a formação de canal entre fonte e dreno.

2.1.3 Programação por SRAM

Os FPGAs baseados em SRAM são os mais utilizados atualmente. Eles são programados através da carga de suas células de configuração a partir de uma fonte externa. Essa memória de configuração controla a lógica e as interconexões que definem a funcionalidade do circuito.

As células de SRAM que armazenam as configurações podem ser projetadas de diferentes maneiras. Uma possibilidade, apresentada na figura 2.2, mostra uma estrutura de 6 transistores. Essa estrutura possui os dois inversores CMOS e dois transistores de passagem (R/W) , os quais são utilizados para a leitura e escrita da célula.

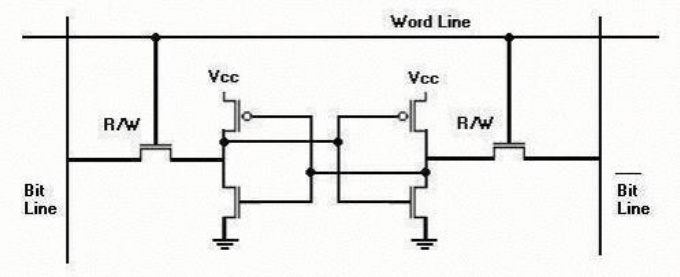

Figura 2.2: Célula SRAM de 6 transistores

Ao contrário de FPGAs baseados em anti-fuse ou EEPROM, os FPGAs com memória de configuração SRAM são voláteis, ou seja, perdem a configuração do circuito quando a energia é desligada. Muitos FPGAs possuem circuitos para detectar quando o sistema é ligado e automaticamente iniciar o processo de inicialização.

Durante a inicialização é preciso que o FPGA tenha acesso aos bits de programação para realizar a carga. Isso torna necessária a existência de uma memória externa que pode ser uma memória do tipo EEPROM. Essa memória, pode, inclusive, ser compartilhada por diversos FPGAs.

Ao mesmo tempo que a volatilidade pode ser considerada uma desvantagem, ela permite o benefício da reprogramabilidade. Um FPGA baseado em anti-fuse, pode ser programado uma única vez, assim, um protótipo com erro deve ser necessariamente substituído, enquanto que com a tecnologia de SRAM, basta corrigir o projeto com as ferramentas de CAD e gerar um novo arquivo de configuração.

A reprogramabilidade é cada vez mais uma característica fundamental. Além da facilidade de realizar mudanças em tempo de projeto, os FPGAs com reconfiguração rápida permitem que os produtos possam ser atualizados após a entrega ao usuário final com o *bitstream* de reconfiguração vindo, por exemplo, de um computador remoto.

O processo utilizado para construir os FPGAs é o mesmo processo CMOS utilizado na produção de ASICs e é bastante similar ao processo usado em memórias CMOS. Dessa forma, FPGAs baseados em SRAM estão entre os primeiros produtos a beneficiarem-se das melhorias nos processos de fabricação, tornando-se, cada vez mais densos e rápidos.

Na seção 3.6 são analisadas arquiteturas de FPGAs atuais, sendo todas programáveis por SRAM. O restante deste capítulo foca em importantes conceitos relacionados à arquiteturas reconfiguráveis, conceitos esses que são utilizados ao longo de todo este estudo.

Tabela 2.1: Resumo das classes de programabilidade

Tipo	Configurações	Momento
Estático	1	Projeto
Estaticamente Reconfigurável	N	Fim da tarefa
Dinamicamente Reconfigurável	N	Durante execução

2.2 Classes de Programabilidade

É possível analisar o projeto da plataforma de hardware reconfigurável segundo classes de programabilidade (ADÁRIO; ROEHE; BAMPI, 1999). Essas classes dividem os projetos em três categorias levando em consideração o número de configurações e o instante de mudança da tarefa/algoritmo em execução:

- Projeto Estático: O circuito utiliza apenas 1 programação, a qual nunca é alterada seja durante ou após o processamento. O componente é programado integralmente, para realizar apenas uma função que não é mais alterada durante toda a atividade do sistema. Esse tipo de projeto não explora a flexibilidade de programação da arquitetura, apenas a facilidade de implementação e/ou prototipação rápida.

- Projeto Reconfigurável Estaticamente: Há varias (N) configurações possíveis para o circuito, entretanto a reconfiguração da arquitetura ocorre apenas ao final do processamento de uma dada tarefa. Assim, os componentes são melhor aproveitados e o hardware pode ser particionado, visando a reusabilidade de recursos. Neste caso, cada configuração corresponde a uma transação longa. Um exemplo que pode ser citado são os sistemas de roteamento de informações (processadores de IP) multiprotocolo, nos quais a configuração é modificada após uma tarefa completa. A partir da reconfiguração, o dispositivo passa a dar suporte a um novo formato de pacote, o qual representa o novo protocolo a ser processado.

- Projeto Reconfigurável Dinamicamente: Nesta modalidade também existem N configurações para o circuito, e a reprogramação ocorre durante a execução. Esse método é conhecido como RTR (Run-Time Reconfiguration). É nesse tipo de aplicação que a arquitetura reconfigurável é utilizada de maneira mais eficiente, extraindo toda a flexibilidade disponível. Todas as arquiteturas discutidas no capítulo 3 pertencem a esta abordagem.

A tabela 2.1 sumariza as características das classes de programabilidade.

2.3 Características das Arquiteturas Reconfiguráveis

Conforme mencionado, uma arquitetura reconfigurável pode ser considerada uma solução intermediária entre as abordagens de hardware e software. A abordagem de hardware situa-se no universo dos ASICs (*Application Specific Integrated Circuits*) e ASPs (*Application Specific Processors*), os quais proporcionam alto desempenho, área reduzida de silício e, em muitos casos, um projeto de baixo consumo. O paradigma do software, por sua vez, baseia-se nos GPPs (*General Purpose Processors*) sob o controle de um programa. GPPs oferecem flexibilidade para programação genérica de tarefas computacionais.

O enfoque principal do projeto de um GPP está concentrado no desempenho e na funcionalidade geral do mesmo, contudo os custos de projeto e fabricação de GPPs RISC são elevados. Eles envolvem (ADÁRIO; ROEHE; BAMPI, 1999):

- Custos de hardware: Em geral, os GPPs são maiores (possuem mais unidades funcionais) do que o necessário para a execução de uma determinada tarefa computacional, havendo desperdício de área/recursos do circuito.
- Custos de projeto: Muitas unidades funcionais são raramente utilizadas, mas devem estar presentes devido ao compromisso com a generalidade e isso implica gastos de recursos financeiros e tempo no projeto.
- Custos de energia: atualmente tem crescido a utilização de sistemas eletrônicos portáteis, conduzindo a preocupação com o projeto de circuitos com baixo consumo de potência. Em um GPP, muita potência é desperdiçada com unidades funcionais que poderão não ser usadas durante uma grande parte do tempo de processamento

Além dos custos mencionados, muitas vezes, os GPPs não são adequados para aplicações de alto desempenho, as quais tratam de problemas de computação intensiva. Nesses casos, os projetistas normalmente recorrem a arquiteturas de aplicação específica como os ASPs. Os ASPs caracterizam-se por serem adequados a um tipo de aplicação ou otimizados para um dado conjunto de requisitos de projeto. Entretanto, essa abordagem carece do fator flexibilidade, em geral, não é possível realizar a inclusão de novas funções que estendam o conjunto original, como também não se pode atualizar a estrutura do dispositivo com o surgimento de uma nova técnica. A execução de melhorias em um sistema de propósito especial implica a substituição do mesmo.

As arquiteturas reconfiguráveis enquadram-se nesse contexto como uma abordagem alternativa. Oferecem a possibilidade de reunir as características de flexibilidade e programabilidade dos processadores gerais ao desempenho das arquiteturas e circuitos específicos. A reconfiguração dos sistemas pode ocorrer em um nível granular fino ou grosso. A reconfigutação de granularidade fina é semelhante ao FPGA, cada bit é configurável. Na abordagem de granularidade grossa, as unidades reconfiguráveis podem ter apenas sua funcionalidade e interconexões alteradas.

Em comparação com as arquiteturas de processadores tradicionais pode-se citar algumas diferenças importantes dos sitemas reconfiguráveis (BONDALAPATI; PRASANNA, 2002):

- Computação espacial: Os dados são computados através da distribuição espacial da computação ao invés do sequenciamento temporal através de uma unidade computacional compartilhada.
- Parte operativa configurável: A funcionalidade das unidades computacionais e a rede de interconexões podem adaptar-se em tempo de execução utilizando-se mecanismos de configuração.
- Controle distribuído: As unidades computacionais processam os dados de acordo com uma configuração local ao invés de instruções distribuídas a todas unidades funcionais.
- Recursos distribuídos: Os recursos necessários para a computação como as unidades funcionais e memória podem estar distribuídos ao longo do dispositivo, ao invés de estarem localizados em uma única área.

O aspecto da distribuição espacial da computação, bem como o da distribuição do controle e dos recursos, resultam em uma maior eficiência da força computacional (BONDALAPATI; PRASANNA, 2002) para arquiteturas reconfiguráveis em comparação aos microprocessadores, processadores DSP e ASICs. Eficiência da força computacional é definida como o número de portas lógicas ativamente operando em um dado ciclo de relógio para resolver um problema em relação ao número total de portas existentes no dispositivo.

2.3.1 Granularidade

Granularidade é um dos conceitos mais importantes para a computação reconfigurável. A granularidade da lógica reconfigurável é normalmente definida como o tamanho da menor unidade funcional que pode ser configurada pelas ferramentas de mapeamento. Granularidade menor permite maior flexibilidade na adaptação do hardware para a estrutura de computação. Entretanto, granularidade baixa é penalizada com uma maior latência quando são construídos módulos de computação maiores a partir das unidades funcionais pequenas. Algumas arquiteturas de granularidade fina possuem características específicas para combater esse problema. A maioria dos FPGAs, por exemplo, possuem estruturas de cadeias de *carry* para permitir a construção de blocos aritméticos maiores utilizando seus elementos lógicos pequenos.

Dessa forma, em arquiteturas de granularidade fina os operadores, normalmente, são portas lógicas. Esses circuitos são construídos utilizando-se *look-up-tables*, multiplexadores e flip-flops. No outro extremo, os operadores de granularidade grossa são mais complexos e estão conectados a barramentos capazes de transportar dados maiores (palavras de 16 e 32 bits na maioria das vezes). Em muitos casos os blocos reconfiguráveis são ULAs, multiplicadores e mais recentemente unidades ainda mais complexas como as que serão descritas nas arquiteturas apresentadas no capítulo 3.

Desde o final da década de 90 a maioria das arquiteturas apresentadas na literatura exploram o conceito de granularidade grossa. O motivo para o fortalecimento dessa corrente de desenvolvimento de arquiteturas está no fato de que é possível construir dispositivos com um aproveitamento de área muito mais eficiente. As partes operativas tendem a ser bastante regulares, características de arquiteturas de maior granularidade são: unidades funcionais configuráveis em nível de operadores, partes operativas com tamanho de dados maiores e estruturas de chaveamento mais eficientes.

Dessa maneira, tem-se como resultado a redução significativa de memória de configuração e tempo de configuração, além da diminuição drástica do problema de posicionamento e roteamento (HARTENSTEIN, 2001). Para ilustrar essa situação pode-se citar a arquitetura GARP (HAUSER; WAWRZYNEK, 1997) a qual necessita de 49152 bits para um contexto, enquanto que uma arquitetura de grão maior como Remarc (MIYAMORI; OLUKOTUN, 1998) utiliza 2048 bits.

Entretanto, a granularidade ótima depende da aplicação. Quando o tamanho dos operandos é menor do que os operadores funcionais a performance tende a diminuir em razão dos recursos lógicos não utilizados. De maneira semelhante, se os dados forem maiores do que os elementos da parte operativa existirá maior *overhead* de reconfiguração, resultando também em penalidades no desempenho.

Apresenta-se como uma tendência a construção de sistemas multigranulares. Esses sistemas possuem áreas reconfiguráveis com elementos processadores de tamanhos diversos visando a adaptação para diversas classes de problemas. A seção 3.8 descreve essa tendência em maiores detalhes.

2.4 Reconfiguração Dinâmica : Métricas Aplicáveis

A capacidade de reconfigurar recusos de hardware de acordo com as necessidades de um aplicação oferece outras vantagens além da flexibilidade. Quando a aplicação pode ser reconfigurada durante a execução de uma tarefa, a plataforma em questão é a de um projeto dinamicamente reconfigurável e permite oportunidades adicionais de especialização.

Uma maneira de aumentar a eficiência de uma computação utilizando RTR é substituir hardware ocioso ou inativo por circuitos que serão efetivamente utilizados. Esse processo é na verdade uma otimização em tempo de execução e permite que se amplie as capacidades do sistema com menos recursos de hardware.

Contudo, as novas oportunidades de projeto e as melhorias em eficiência proporcionadas por sistemas dinamicamente reconfiguraveis não surgem sem custos. A utilização extra de recursos de memória e latências de configuração, por exemplo, devem ser considerados na avaliação desses sistemas digitais. A densidade funcional descrita a seguir é uma métrica bastante interessante, pois ela fornece uma base de avaliação teórica de arquiteturas reconfiguráveis relacionando conceitos como aproveitamento de recursos, taxa de processamento e tempo de configuração.

2.4.1 Densidade Funcional

Densidade Funcional (WIRTHLIN; HUTCHINGS, 1998) é uma métrica que relaciona área e tempo. Arquiteturas dinamicamente reconfiguráveis oferecem a possibilidade de especialização em tempo de execução. Essa vantagem permite que o circuito execute de maneira mais adequada determinada tarefa. A densidade funcional é uma maneira de medir os benefícios dessa especialização.

A densidade funcional (D) é definida como o custo de implementar a computacao em hardware. Para circuitos VLSI, o custo da computacao é, nomalmente, calculado através do produto da área do circuito (A) com o tempo de computação (T).

$$C = AT \tag{2.1}$$

A densidade funcional mede a taxa de processamento (operações por segundo) por unidade de recursos de hardware. Sua definição é o inverso do custo de uma computação.

$$D = \frac{1}{C} = \frac{1}{AT} \tag{2.2}$$

Na equação 2.2, T é o tempo total da operacão o que inclui o tempo de execução da tarefa, controle, inicialização e transferência de dados. A densidade funcional é bastante útil para comparar circuitos estáticos com circuitos reconfiguráveis em tempo de execução. A melhoria (I) na densidade funcional de um determinado circuito reconfigurável (Dr) sobre a versão estática (Ds) é a diferença normalizada entre Dr e Ds:

$$I = \frac{\Delta D}{Ds} = \frac{Dr - Ds}{Ds} = \frac{Dr}{Ds} - 1 \tag{2.3}$$

Para obter o mesmo dado em termos percentuais:

$$I = 100(\frac{Dr}{Ds}) - 100 \tag{2.4}$$

2.4.2 Tempo de Configuração

A métrica da densidade funcional deve ser extendida para suportar análises que levem em consideração o tempo de configuração, o qual é um *overhead* sempre presente em circuitos RTR. Para uma classe inteira de circuitos reconfiguráveis (os estaticamente reconfiguráveis) o processo de reconfiguração é completamente disjunto da execução. Ou seja, enquanto acontece a reconfiguração o circuito não está operando. Para sistemas realmente RTR, existem casos em que a latência de reconfiguração é escondida quase que completamente pela sobreposição entre configuração e execução de determinado algoritmo. De uma maneira geral, a métrica da densidade funcional considerando o *overhead* de configuração sugere alterações no tempo de computação total T, o qual é formado por uma porção referente à execução (Te) e outra referente à latência de reconfiguração (Tc). Assim:

$$T = Te + Tc \tag{2.5}$$

substituindo em 2.2, temos:

$$Dr = \frac{1}{A(Te + Tc)} \tag{2.6}$$

Como pode ser visto analisando-se a equacao 2.6, o tempo de configuração reduz a densidade funcional. Esse comportamento ocorre porque o conceito de densidade funcional está relacionado com a contribuição de cada unidade de hardware na computação total e enquanto ocorre a reconfiguração existem unidades funcionais inativas no circuito.

2.4.3 Tempo Relativo de Configuração

Outro conceito muito importante para arquiteturas reconfiguraveis é o conceito de tempo relativo de configuração (f), o qual expressa a relação entre os tempos de configuração e execução.

$$f = \frac{Tc}{Te} \tag{2.7}$$

O tempo total pode ser expresso por:

$$T = Te(1 + f) \tag{2.8}$$

A substituição desse tempo na equação da densidade funcional resulta na métrica em função de f:

$$Dr = \frac{1}{ATe(1 + f)} \tag{2.9}$$

A equação mostra que um circuito estaticamente reconfigurável deve possuir um tempo de execução longo (f pequeno) para existir uma boa densidade funcional. Existem aplicações, como processadores de protocolos (conforme exemplificado na seção 2.2 sobre classes de programabilidade) que se adaptam a essas características.

É possivel ampliar essa análise e identificar o caso ideal (Imax) da melhoria de um sistema reconfigurável sobre uma abordagem estática. O caso ideal ocorre quando f tende a 0, ou seja, quando o tempo de configuração é desprezível. Teremos então Dmax (densidade funcional máxima) e Imax como:

$$Dmax = \frac{1}{ATe} \tag{2.10}$$

$$Imax = \frac{Dmax}{Ds} - 1 \tag{2.11}$$

2.4.4 Análise: Sistemas Digitais Reconfiguráveis versus Circuitos Estáticos

Uma constatação analítica importante é identificar sob quais condições ou parâmetros uma implementação RTR ou estaticamente reconfigurável pode oferecer benefícios em relação a uma abordagem puramente estática do ponto de vista da densidade funcional. Para que essa sistuação ocorra Dr deve ser maior ou igual a Ds (Dr >= Ds). Nas equações abaixo Ar e Tr representam, respectivamente, a área e o tempo de execução do circuito reconfigurável.

$$\frac{1}{ArTr(1+f)} >= Ds \tag{2.12}$$

$$\frac{1}{DsArTr(1+f)} >= 1 + \frac{Tc}{Tr} \tag{2.13}$$

Substituindo-se a equação 2.10 temos que:

$$\frac{Dmax}{Ds} - 1 >= 1 + \frac{Tc}{Tr} \tag{2.14}$$

O lado esquerdo da expressão representa o ganho teórico máximo possível com RTR, o que leva a uma importante equação:

$$Imax >= f \tag{2.15}$$

A equação 2.15 mostra que para um sistema reconfigurável superar um circuito estático em termos de densidade funcional o tempo relativo de configuração (f) deve ser menor que o ganho teórico máximo da implementação reconfigurável.

Para exemplificar essa relação considere um circuito estático de área α e tempo de execução τ. Uma determinada arquitetura dinamicamente reconfigurável executa a mesma computação em tempo $2\tau/3$ e em uma área $\alpha/2$. Neste caso o ganho teórico máximo do circuito reconfigurável é 2 (200%). Utilizando a equação 2.15 pode-se dizer que a arquitetura reconfigurável terá maior densidade funcional enquanto f < 2 (tempo de configuração menor que duas vezes o tempo de execução).

Essa análise mostra que as arquiteturas que através da especialização oferecem grandes melhorias na densidade funcional são menos sensíveis ao tempo de configuração do que arquiteturas que atingem pequenos ganhos. De maneira similar, essas relações matemáticas demonstram que mesmo um arquitetura com um ganho de densidade pequeno em relação a uma abordagem totalmente estática pode ser uma solução atrativa, desde que os períodos de execução sejam longos em relação o tempo de configuração. Esses conceitos são bastante úteis sempre que estudamos sistemas reconfiguráveis. Como será visto mais tarde modificações na célula que forma a parte operativa do processador DRIP possibilitou a redução drástica do tempo relativo de reconfiguração resultando em um aumento importante de densidade funcional.

2.5 Técnicas de Reconfiguração Dinâmica

Reconfiguração em tempo de execução permite ao projetista explorar de maneira mais eficiente a flexibilidade das arquiteturas reconfiguráveis. Além de minimizar os *overheads* de reconfiguração das arquiteturas estaticamente reconfiguráveis, a reconfiguração dinâmica traz a possibilidade de implementar estratégias de hardware virtual e computação de fluxo. Em hardware programável tradicional o projeto deve caber dentro do limite de recursos lógicos disponíveis. Em um ambiente dinamicamente reconfigurável, entretanto, não existe tal restrição. As unidades funcionais necessárias para uma determinada tarefa computacional podem ser divididas em partes e mapeadas para o hardware programável de acordo com a aplicação. O SCORE (*Stream Computations Organized for Reconfigurable Execution*) (DEHON et al., 2000) utiliza essa abordagem, um programa pode ser visto como um grafo de nodos de computação e blocos de memórias ligados por fluxos, novos operadores são carregados no hardware de acordo com as exigências do controle de execução.

De uma maneira geral, um processo de reconfiguração pode ser dividido em três fases distintas: desconexão/salvamento, modificação e reconexão/restauração. No primeiro passo todas as portas conectadas ao bloco dinâmico devem ser desabilitadas e os dados sequênciais salvos. Em seguida, o controlador de reconfiguração deve executar as modificações nas unidades funcionais e interconexões. Finalmente, os valores dos registradores salvos devem ser restaurados e as portas de comunicação devem voltar a interagir com o restante da lógica. Sistemas reconfiguráveis em tempo de execução podem admitir muitas reconfigurações durante a execução de um programa. Esse processo deve, portanto, acontecer da maneira mais eficiente possível, de forma que o *overhead* de reconfiguração não elimine os benefícios adquiridos pelo maior desempenho de processamento.

Para implementar reconfiguração dinâmica de maneira eficiente diversos modelos foram propostos e podem ser encontrados na literatura (COMPTON; HAUCK, 2002). Reconfiguração dinâmica parcial, multicontexto e *pipelined* são algumas das abordagens possíveis.

2.5.1 Reconfiguração Dinâmica Parcial

Reconfiguração dinâmica parcial está relacionada com a redefinição de partes de um circuito mapeado para um dispositivo reconfigurável através de modificações no *bitstream* de configuração. Partes estáticas do *array* podem prosseguir com a execução, o que leva a sobreposição de computação com reconfiguração, escondendo a latência de reconfiguração.

Os FPGAs da família Virtex da Xilinx (analisados na seção 3.6.1) estão entre os dispositivos programáveis que permitem a implementação desse modelo. Sua estrutura interna é dividida em colunas que correspondem a uma fatia vertical inteira do *chip*. É possível realizar transformações na lógica de um módulo dinâmico mapeado em uma dessas colunas através de modificações no arquivo de configuração.

Um exemplo de utilização dessa técnica é a arquitetura Recats (HORTA; KOFUGI, 2002). Recats é uma *switch* ATM reconfigurável em tempo de execução e utiliza reconfiguração parcial para realizar dinamicamente a carga de módulos de hardware pré-compilados para um FPGA Virtex (melhor detalhada na seção 3.7.2 como estudo de caso).

Uma desvantagem deste modelo de reconfiguração é que informações de endereçamento devem ser fornecidas juntamente com os dados de configuração, aumentando o número total de bytes transferidos. Para minimizar esse problema e evitar *overheads* di-

ferentes técnicas de compressão podem ser utilizadas. Uma dessas técnicas é *wildcarding* (XILINX, 2003), que proporciona um método para programar múltiplas células lógicas com um único endereço e valor de dados.

2.5.2 Reconfiguração Dinâmica Multicontexto

Um contexto representa uma configuração possível do dispositivo reconfigurável. Em um ambiente multicontexto existem múltiplos bits de programação para um dado recurso reconfigurável. Esses bits de memória podem ser vistos como planos de configuração. O controlador de reconfiguração pode realizar o chaveamento entre os diferentes contextos de acordo com as necessidades da aplicação.

No processador de comunicação reconfigurável Chameleon (SALEFSKI; CAGLAR, 2001) existem dois planos de configuração principais. O plano ativo representa o algoritmo em execução, enquanto que o plano reserva armazena a configuração de outra tarefa. O controlador pode carregar uma nova configuração no plano reserva enquanto a parte operativa executa conforme o plano ativo. O sistema pode ser reconfigurado como um todo em um único ciclo de *clock*. Outros sistemas explorando estilos similares de reconfiguração podem ser encontrados em (CARDOSO; WEINHARDT, 2003) e (DEHON, 1994). Uma desvantagem da reconfiguração multicontetxo está na quantidade de bytes necessários para armazenar cada configuração. Esse problema pode tornar proibitivo um grande número de planos diferentes no hardware, diminuindo, assim, a flexibilidade total do sistema.

2.5.3 Reconfiguração Dinâmica Pipelined

Neste modelo, através de reconfiguração dinâmica uma estrutura de pipeline implementada em hardware pode suportar mais estágios do que fisicamente disponíveis. Em um primeiro passo, um estágio do pipeline é adaptado para computar parte da aplicação. Na sequência, o processamento efetivamente ocorre (como na figura 2.3). Este modelo pode ser observado na arquitetura Piperench (SCHMIT et al., 2002) e torna a computação possível mesmo que a configuração completa não esteja presente no dispositivo ao mesmo tempo.

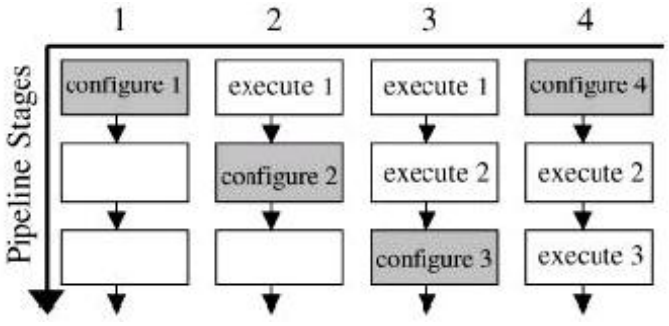

Figura 2.3: Reconfiguração pipelined

3 CSOCS : SISTEMAS CONFIGURÁVEIS EM UM ÚNICO CHIP

Os sistemas em um único chip (SoCs) surgiram para enfrentar os desafios apresentados pelas exigentes aplicações contemporâneas em áreas como multimídia e telecomunicações. Os SoCs consistem na integração, em um único chip, de circuitos de uma complexidade significativa como processadores, controladores e memórias.

A popularização dos SoCs traz como consequência impactos na metodologia de projeto. Os *IP-Cores*, módulos de propriedade intelectual descrevendo circuitos prontos para serem sintetizados, abrem uma possibilidade enorme de reuso de blocos desenvolvidos em projetos anteriores dentro dos meios acadêmico e industrial. Além disso, através dos SoCs existe a diminuição drástica de latências de comunicação, uma vez que os módulos de processamento não estão mais integrados através de placas de circuito impresso.

Em (GUPTA; ZORIAN, 1997) um núcleo de propriedade intelecutal é descrito como um módulo de hardware pré-projetado e pré-verificado que pode ser utilizado no desenvolvimento de sistemas maiores e mais complexos. Um *IP-Core* pode ser classificado em diferentes classes como:

- *soft cores*: Normalmente desenvolvidos em linguagem de descrição de hardware, seu funcionamento pode ser estudado e inclusive modificado. Oferece independência de tecnologia.
- *firm cores*: Representa a descrição de um circuito em um nível mais baixo como *netlist*. Neste ponto, geralmente, já estão associados a uma tecnologia e a flexibilidade de manipulação é bem mais restrita que no caso dos *soft cores*.
- *hard cores*: Núcleos *hard* são otimizados para uma tecnologia específica e não podem ser modificados. Possuem garantias mais precisas em relação a questões de temporização e desempenho.

Dentro desta tendência houve, nos últimos anos, o surgimento dos *CSoCs* (BECKER, 2002) (Sistemas Configuráveis em um único *Chip*). Um CSoC consiste nos componentes integrados de um SoC aliados a partes de hardware reconfiguráveis residentes dentro do *chip*. A adição de partes reconfiguráveis traz diversas vantagens como por exemplo:

- Customizar o CSoC da melhor maneira possível de acordo com as necessidades da aplicação a ser executada.
- Maior adaptatividade frente a novas especificações.
- Redução de custos em relação a um SoC, conforme o desenvolvimento continuado do SoC (inflexível) necessita de novos conjuntos de máscaras.

3.1 CSoCs no Contexto dos GPPs, DSPs, ASICs e FPGAs

Uma das áreas em que a utilização de CSoCs é mais promissora é no desenvolvimento dos dispositivos portáteis de última geração, como por exemplo PDAs e celulares com capacidades multimídia. Este tipo de aplicação é, portanto, o estudo de caso utilizado para a análise realizada nesta seção.

Os dispositivos portáteis são pequenos, utilizam bateria como fonte de alimentação e devem possuir um custo baixo para obter competitividade nos mercados atuais. Dessa maneira, torna-se inviável, por exemplo, incorporar a um dispositivo dessa natureza um processador de propósito geral como Pentium ou Sparc.

Uma solução possível é o emprego de processadores DSP. Os DSPs são programáveis e existem modelos que podem oferecer taxas de desempenho satisfatórias para algumas aplicações, entretanto, são caros e consomem muita energia. A relação performance versus consumo é um ponto cada vez mais importante, muitos sistemas atuais procuram utilizar processadores DSP otimizados para um menor consumo com a finalidade de aumentar o tempo de vida das baterias. Esses processadores, no entanto, não terão condições de cumprir as demandas de processamento das novas aplicações. Como exemplo pode-se citar que um receptor UMTS trabalhando a uma taxa de 384 kbps necessita de cerca de 6000 MIPS, a próxima geração deverá manipular taxas de até 2 Mb/s indicando a necessidade de migração da força computacional dos DSPs para estruturas mais bem adaptadas para essa natureza de processamento (BECKER; PIONTECK; GLESNER, 2000a).

Outra alternativa possível são circuitos integrados específicos (ASICs) atuando como coprocessadores. ASICs produzidos em larga escala possuem um custo aceitável e oferecem o desempenho e consumo adequados para atender exigentes algoritmos. O desenvolvimento de ASICs, contudo, tende a ser um processo longo, além do fato de que o dispositivo não possui flexibilidade para alterações. Esse último ponto é particularmente importante em uma indústria na qual os produtos e padrões mudam em uma velocidade enorme, com cliclos de vida inferiores a 1 ano. Dessa forma, ASICs tendem a ser uma boa solução, mas de alto risco.

Os FPGAs, por sua vez, apresentam uma densidade de portas lógicas e recursos especializados cada vez maior. São altamente flexíveis em virtude de sua alta programabilidade. Os projetistas possuem a sua disposição bibliotecas grandes de *IP-cores* que possibilitam a agilização das etapas de projeto e validação. As desvantagens da utilização de FPGAs, no contexto deste estudo de caso, está, primeiramente, no custo por unidade e no consumo de energia que impede uma boa autonomia em sistemas que operam com auxílio de baterias.

Em relação aos GPPs, os CSoCs oferecem vantagens de desempenho e custos inferiores de engenharia. O enfoque principal do projeto de um GPP está concentrado na funcionalidade geral do mesmo, contudo os custos de projeto e fabricação de GPPs RISC são consideráveis conforme descrito na seção 2.3.

A tabela 3.1 (adaptada de) oferece uma visão geral sobre arquiteturas reconfiguráveis de granularidade grossa.

As seções seguintes deste capítulo analisam uma série de CSoCs estado-da-arte, explicitando as características e potencialidades desses sistemas.

Tabela 3.1: Sumário de arquiteturas de granularidade grossa

Projeto	Arquitetura	Granularidade	Aplicação-alvo
PADDI	*crossbar*	16 bit	DSP
PADDI-2	*crossbar*	16 bit	DSP e outros
DP-FPGA	*array* 2-D	1 e 4 multigranular	partes operativas regulares
Kressarray	*mesh* 2-D	Selecionável	adaptável
Colt	*array* 2-D	1 e 16 bit multigranular	aplicações altamente dinâmicas
RaPID	*array* 1-D	16 bit	pipelining
Matrix	*mesh* 2-D	8 bit multigranular	propósito geral
RAW	*mesh* 2-D	8 bit multigranular	experimental
Garp	*mesh* 2-D	2 bit	aceleração de loops
Pleiades	*mesh* / *crossbar*	multigranular	multimídia

3.2 MorphoSys

O MorphoSys (SINGH et al., 2000) é um processador reconfigurável concebido por pesquisadores da universidade da Califórnia em Irvine. O MorphoSys é composto de células reconfiguráveis de granularidade grossa em um arranjo 8x8 (RC *array*) organizadas segundo um modelo de computação SIMD acopladas a um processador RISC semelhante ao MIPS chamado TinyRISC. Implementado em tecnologia 0.35μm , o MorphoSys opera em uma frequência de 100 MHz

O processador TinyRISC consiste em um módulo programável por software o qual está ligado a componentes de hardware reconfiguráveis. A funcionalidade dos elementos e suas interconexões é definido pelo contexto carregado em determinado momento.

3.2.1 Arquitetura do Sistema

A figura 3.1 ilustra a arquitetura básica do sistema. Os principais componentes são:

- Processador TinyRISC
- Memória de contextos
- Controlador de DMA
- *Array* de células reconfiguráveis
- *Frame Buffer*
- *Cache* de dados e instruções
- Controlador de memória

Os elementos de processamento do MorphoSys são as células reconfiguráveis. Cada célula reconfigurável possui uma ULA-multiplicador, um deslocador e um pequeno banco de registradores. Cada célula é configurada através de uma palavra de contexto de 32 bits. O *datapath* reconfigurável (ou RC *array*) completo é formado por 64 células em uma matriz 8x8.

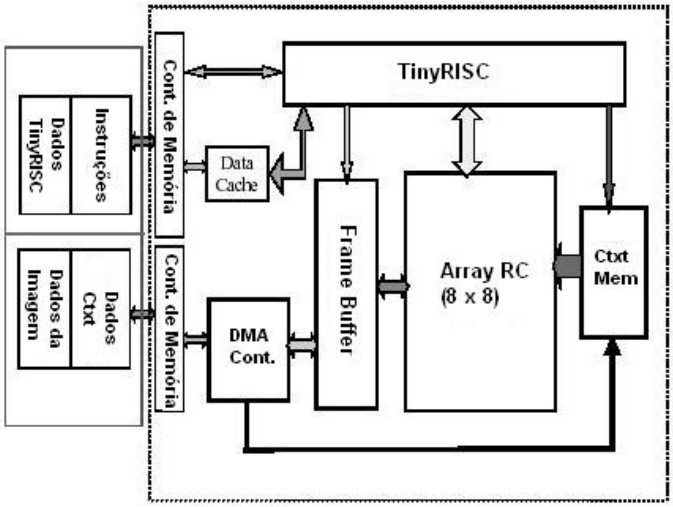

Figura 3.1: Arquitetura MorphoSys

3.2.2 A Célula Reconfigurável (RC)

A célula reconfigurável (figura 3.2) é uma estrutura de granularidade grossa (a palavra mínima de processamento é de 16 bits). Conforme mencionado, cada célula compreende uma ULA-multiplicador, uma unidade de deslocamentos e 4 registradores. Além desses circuitos, existe ainda um registrador de saída, dois multiplexadores para as entradas da ULA e circuitos especiais para aplicações voltadas ao processamento de imagens (implementados como funções da ULA).

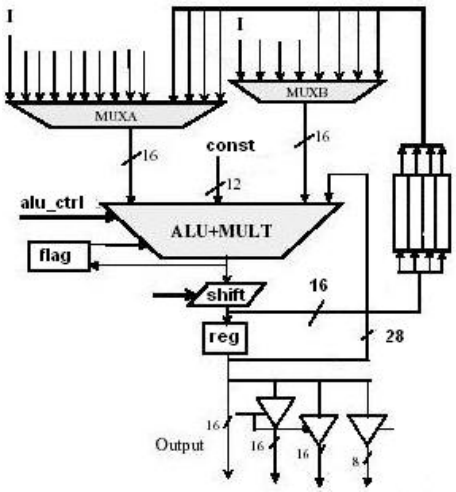

Figura 3.2: Célula Reconfigurável

O multiplicador existente na célula é um multiplicador *array* de 28 bits (16 bits x 12 bits), sendo o operando de 12 bits um valor constante vindo diretamente do registrador de contexto. Ele permite operações entre números com sinal e números sem sinal, o que satisfaz diferentes classes de aplicações. Para computar eficientemente aplicações de DSP, a ULA-multiplicador possui a capacidade de executar uma operação de múltiplica-acumula (MAC) em um único ciclo de relógio. A unidade de deslocamento, por sua vez, possui 28 bits e a capacidade de deslocar para a direita e para esquerda até 15 bits de uma só vez.

Os multiplexadores de entrada selecionam uma entre diversas entradas para a ULA-multiplicador. O multiplicador MUXA (que pode ser visto na figura 3.2) seleciona uma entrada que pode ser:

- Um dos quatro vizinhos mais próximos no *array*.
- Outros RCs na mesma linha e coluna dentro do mesmo quadrante.
- O barramento de operandos.
- Registradores internos.

O multiplexador MUXB, por sua vez, seleciona entre:

- Três dos vizinhos mais próximos.
- Registradores internos.
- O barramento de operandos.

3.2.3 O *Array* de Células Reconfiguráveis

O *array* é composto por uma matriz 8x8 de células reconfiguráveis. Um dos pontos fundamentais da estrutura é sua estratégia de interconexão em três camadas. As primeiras duas camadas conectam os RCs como uma matriz bidimensional como pode ser visto na figura 3.3.

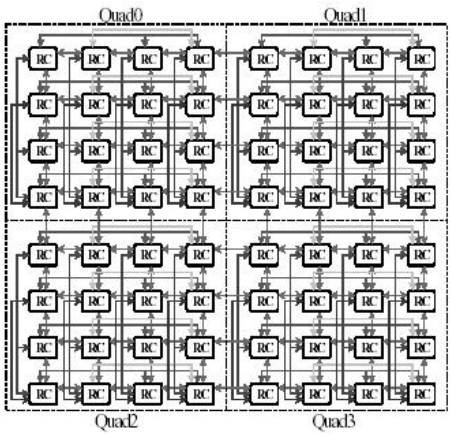

Figura 3.3: *Array* de processamento

A primeira camada de interconexão garante que cada RC pode acessar dados de qualquer outro RC vizinho em sua linha e coluna. A segunda camada proporciona conectividade total intra-quadrante, ou seja, permite que um RC acesse qualquer outro dentro da mesma linha e coluna desde que esteja no mesmo quadrante.

A terceira camada de conectividade é ilustrada na figura 3.4. Essa camada de interconexão é também chamada de conexão inter-quadrante. Ela consiste de barramentos que percorrem toda a extensão das linhas e colunas, passando pelas fronteiras entre os quadrantes. Os barramentos verticais permitem que um RC envie/receba dados de qualquer outro RC na mesma coluna do *chip*, de maneira análoga os barramentos horizontais permitem a comunicação com qualquer outro RC na mesma linha.

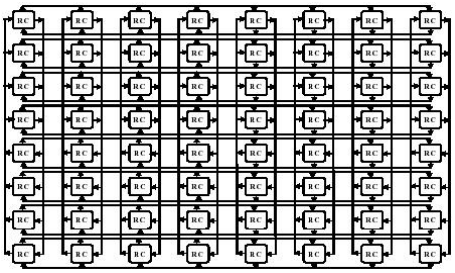

Figura 3.4: Célula Reconfigurável

3.2.4 Processador TinyRISC

O TinyRISC é uma versão 32 bits simplificada do MIPS (PATTERSON; HENESSY, 1997). Possui 4 estágios de pipeline: busca, decodificação, execução e escrita. Ao contrário do MIPS não existe o estágio separado de acesso à memória, o endereço de memória vem diretamente do banco de registradores. O TinyRISC possui 16 registradores disponíveis para o programador, sendo o registrador R0 fixo com o valor 0. O processador compreende um subconjunto das instruções MIPS com a adição de algumas instruções dedicadas para controlar o *array* reconfigurável, memória de contexto, *frame buffer* e DMA.

Cada registrador do TinyRISC é de 32 bits. Existem ainda 3 unidades funcionais: uma ULA de 32 bits, uma unidade de *shift* também de 32 bits e uma unidade de memória. Com a finalidade de minimizar acessos à memória externa, o TinyRISC inclui uma memória *cache* de 1Kbyte.

3.2.5 A Memória de Contexto

A memória de contexto armazena o programa de configuração para o *array* reconfigurável. A organização lógica da estrutura é composta por 2 blocos de contexto, cada bloco contendo 8 conjuntos de contexto e cada contexto, por sua vez, sendo formado por 16 palavras.

O contexto é propagado considerando a separação entre linhas e colunas. As palavras de contexto de um bloco são transferidas pelas linhas, enquanto que palavras do outro bloco são passadas através das colunas. Cada um dos 8 conjuntos de contexto de um bloco está associado a uma linha/coluna específica do *array*.

A palavra de contexto de um conjunto é propagada para os 8 RCs na linha/coluna correspondente. Dessa forma, todos os RCs em uma mesma linha/coluna dividem a mesma palavra e executam a mesma operação. Um conceito relacionado, portanto, é o de plano de contexto do MorphoSys. Cada plano é formado pelas palavras de cada conjunto. Como existem 16 palavras em em conjunto, até 16 planos podem estar residentes ao mesmo tempo em cada um dos 2 blocos da memória.

3.2.6 Modelo de Execução

O modelo de execução do MorphoSys é baseado no particionamento das aplicações em tarefas sequenciais e paralelas. O processador TinyRISC executa as tarefas sequenciais, enquanto que as tarefas para execução paralela são mapeadas para o *array* reconfigurável.

O TinyRISC gerencia todas as transferências de dados e configurações. As instruções especiais para o MorphoSys incluidas no TinyRISC residem em duas categorias:

instruções relacionadas ao DMA e instruções voltadas para o *array* reconfigurável. As instruções de DMA iniciam transferências de dados entre a memória principal e o *frame buffer*, além de carga de contextos da memória principal para a memória de contextos. As instruções para o *array* reconfigurável são instruções que especificam o contexto a ser selecionado para execução e o modo de propagação da palavra de contexto.

3.3 Arquitetura DReAM

A arquitetura DReAM (BECKER; PIONTECK; GLESNER, 2000b)(*Dynamically Reconfigurable Architecture for future Mobile Communication Systems*) foi projetada para atender aos requisitos da terceira geração de dispositivos de comunicação (3G), os quais além de oferecer os serviçoes existentes nos antecessores apresenta capacidade de comunicação de dados de alta velocidade e execução em tempo real de aplicações multimídia. A computação reconfigurável se adapta de maneira adequada a este contexto, possibilitando uma maior flexibilidade para mudanças/alterações de padrões, além de menor consumo de potência (RABAEY, 1997) (DAVID et al., 2002) e aumento de performance.

A idéia principal da arquitetura DReAM está em retirar responsabilidades de computação intenciva do DSP e transferi-las para hardware dedicado, o qual pode realizar as tarefas com melhor desempenho e menor consumo. A parte reconfigurável garante que o dispositivo seja flexível podendo receber novos padrões através de configurações externas.

Em virtude da natureza das aplicações alvo do DReAM, não existe a necessidade de operadores de pequena granularidade. Multiplicações e adições são operações típicas de sistemas de comunicãção CDMA e UMTS, o que leva a definição de unidades funcionais de granularidade grossa. Um outro ponto importante que deve estar presente na definição da arquitetura é que nem todos os problemas computacionais se adaptam de forma eficiente a uma implementação reconfigurável. Dessa forma, o sistema permite o acoplamento de módulos como DSPs ou microcontroladores visando aplicações ou algoritmos com fluxo irregular.

Na figura 3.5 tem-se uma representação da arquitetura geral DReAM. Ela possui componentes individuais de SoCs como DSP, memória e microcontrolador. Esses blocos são conectados por um barramento AHB. O barramento permite transferências em rajada e múltiplos mestres.

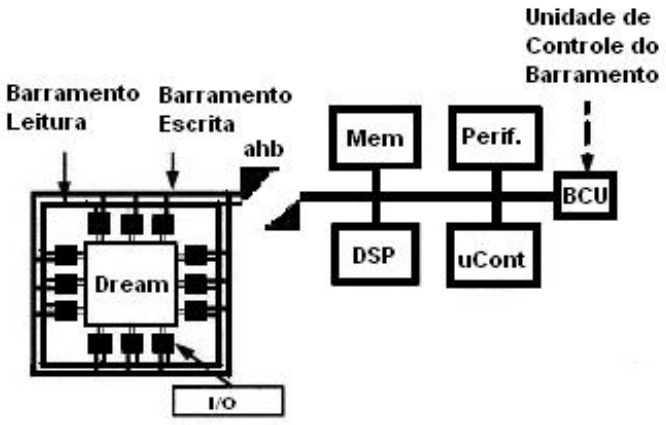

Figura 3.5: CSoC DReAM

O *array* DReAM não está diretamente conectado ao barramento. A interconexão

ocorre através de uma *bridge*. O objetivo dessa abordagem de acoplamento é manter a flexibilidade, podendo o *array* reconfigurável ser adaptado para outros barramentos de SoC. A *bridge*, por sua vez, está ligada a um barramento de leitura e escrita que se comunica com as portas dedicadas de I/O do DReAM.

3.3.1 O *array* DReAM

A estrutura da arquitetura (figura 3.6) é formada por um *array* de unidades de granularidade grossa chamadas RPUs (*Reconfigurable Processing Units*), as quais estão conectadas a redes de comunicação globais e locais.

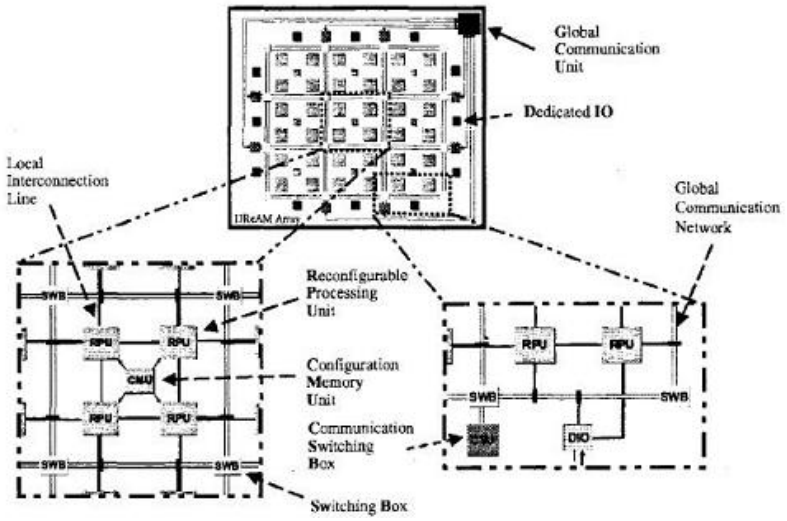

Figura 3.6: Estrutura DReAM

A rede de conexão global divide-se em segmentos pequenos e consiste em dois barramentos de 16 bits. Esses segmentos podem conectar-se de maneira independente com cada um dos barramentos de 16 bits com o auxílio das chaves SWBs (*Switching Boxes*). A rede local, conecta cada RPU com seus quatro vizinhos diretos, permitindo uma comunicação síncrona e rápida. Quatro RPUs compartilham uma mesma CMU (*Configuration Memory Unit*). A CMU armazena os dados de configuração para cada um dos quatro vizinhos. A CMU é controlada pela GCU (*Global Communication Unit*). A GCU representa o nível de controle mais alto do *array* é a unidade que comanda as CSUs (controladora das chaves SWB) e é também a unidade responsável pela comunicação com outros componentes de hardware do SoC.

Durante o processo de reconfiguração dinâmica, a CMU transfere códigos de configuração para as RPUs. Se uma configuração necessária não está presente na CMU, o GCU deve comunicar-se com o microcontrolador do SoC para receber os dados de configuração correspondentes. É possível, portanto, efetuar a reconfiguração em tempo de execução.

3.3.2 O RPU

O RPU é a maior unidade da arquitetura DReAM. Uma RPU pode executar operações aritméticas (estilo *dataflow*) e pode também executar operações de máquinas de estado para as partes orientadas por controle.

Cada RPU é composta por duas unidades aritméticas chamadas RAPs (*Reconfigurable Arithmetic Processing Units*), uma unidade chamada SDP (*Spreading Datapath*), duas RAMs dupla-porta e um controlador de comunicação.

O controlador do RPU é responsável por gerenciar as operações e transferências de dados, além de, juntamente com a CMU, atuar no processo de reconfiguração dinâmica.

As duas memórias RAM são utilizadas como LUTs quando a unidade executa multiplicação de 8 bits, mas pode ser utilizada como memória normal ou FIFO. A unidade SDU é um bloco especial utilizado para operações de correlação. Cada RPU possui 5 entradas de dados e duas saídas, todas de 16 bits. Os dados de entrada vêm da rede de comunicação global e dos quatro RPUs vizinhos. O código de configuração, por sua vez, é transmitido usando uma linha de comunicação extra.

3.3.3 A unidade RAP

A unidade RAP é construída em torno de um multiplicador de 8 bits (figura 3.7) especializado para aplicações de comunicação. Nesse tipo de problema a maioria das multiplicações ocorrem com operandos fixos.

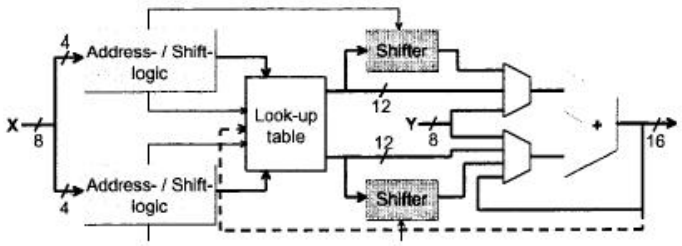

Figura 3.7: Parte operativa do multiplicador

Dessa maneira, o multiplicador utiliza uma LUT para calcular os produtos. A idéia é gerar todos os possíveis múltiplos de um operando fixo e armazená-los na LUT. O operando variável é então utilizado para endereçar a LUT.

As demais operações da unidade RAP derivam-se da estrutura do multiplicador. Operações de MAC são tão rápidas quanto a multiplicação se o segundo operando for fixo, divisão é obtida através dos elementos de *shift* e comparação. Adição e subtração em 16 bits são implementações diretas na estrutura.

Cada RAP dentro do RPU possui memória dedicada. A memória é utilizada como LUT para multiplicação e operação MAC. A RAM possui uma porta de escrita síncrona e duas de leitura assíncrona, podendo efetuar leitura e escrita simultâneas. A segunda porta somente é utilizada quando a memória funciona em modo de LUT.

3.3.4 Controlador da RPU

O controlador é responsável por guiar todas as manipulações de dados e transferências dentro da RPU e também enviar e receber dados através das redes de comunicação. A comunicação entre dois módulos dentro de uma RPU e a comunicação entre duas RPUs distintas seguem o mesmo conceito. Existe um bit de controle que vai da unidade de envio até a unidade de recebimento sinalizando a existência de dados válidos na unidade de envio. Um segundo bit de controle realiza o trajeto entre a unidade de recebimento até a unidade de envio indicando a necessidade de novos dados. O controlador da RPU também realiza operações condicionais e gerencia a reconfiguração em conjunto com a GCU.

3.4 Plataforma XPP: eXtreme Processing Plataform

O SMeXPP (PACT, 2004) (BAUMGARTE et al., 2003) é fabricado pela empresa PACT. O SMeXPP é um processador dinamicamente reconfigurável que substitui o conceito de sequenciamento de instruções por sequenciamento de configurações. As aplicações-alvo do XPP são sistemas multimídia que requerem um processamento relativamente uniforme sobre grandes quantidades de dados como codificação e decodificação de vídeo. A plataforma XPP foi concebida com a idéia de que uma grande vantagem de desempenho pode ser obtida com o uso de um *array* orientado para o processamento específico de fluxo de dados, esse tipo de processamento consome cerca de 80% do tempo de um processador normal em aplicações de mídia. Com o XPP computando esse tipo de tarefa, códigos menos regulares, com estruturas de controle complexas podem ser processados por microcontroladores com frequência de relógio mais moderadas.

3.4.1 O *array* XPP

O núcleo da arquitetura SMeXPP é seu *array* de processamento. Ele é composto por uma matriz de 5 x 4 processadores elementares chamados de ALU-PAEs (*ALU-Processing Array Elements*), 8 unidades especializadas de memória (RAM-PAEs) e 4 elementos de I/O, confome mostrado na figura 3.8.

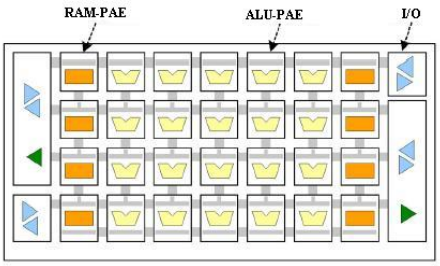

Figura 3.8: O *array* reconfigurável

Esta estrutura homogênea facilita a programação e mapeamento dos algoritmos. As unidades de processamento podem ser configuradas para operar sobre largura de dados que variam de 8 até 32 bits. Acoplado ao *array* tem-se ainda o gerenciador de configurações (*Configuration Manager*) que é o responsável pelo sequenciamento das configurações na parte operativa. De uma maneira geral, as funções dos blocos são:

- **ALU-PAEs**: São a unidade básica de computação.
- **RAM-PAES**: Elementos voltados ao armazenamento de dados.
- **Elementos de I/O**: Conectam as estruturas internas a RAMs externas ou portas de dados.

3.4.2 ALU-PAEs

Os ALU-PAES são compostos por três blocos e uma matriz interna de interconexões. O objeto central (que pode ser visto na figura 3.9) possui uma ULA com capacidades especiais para processamento digital de sinais. Dentre as funções realizáveis por esta ULA estão multiplicação (16x16), soma, comparação, classificação e operações booleanas. Todas as operações podem ser realizadas em um único ciclo de relógio.

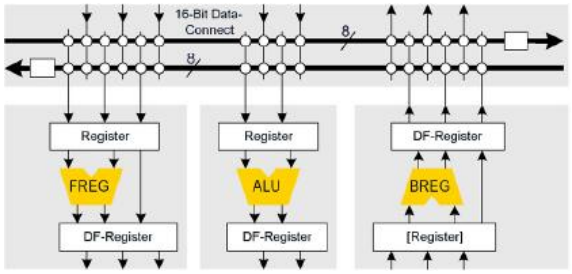

Figura 3.9: ALU-PAE

O bloco mostrado no lado direito da figura 3.9 mostra o BREG (*BackRegister-Object*). Este elemento possui canais de roteamento verticais e uma unidade especial para operações de adição, deslocamento e normalização.

Em um ALU-PAE tem-se também uma estrutura chamada FREG (*ForwardRegister-Object*). Além dos canais verticais de roteamento o FREG possui elementos para operações de multiplexação e *swap*.

Todos os blocos possuem registradores de entrada que podem ser pré-carregados durante a configuração. No BREG esse registrador pode ser eliminado do caminho de dados (*bypassed*). Barramentos horizontais são utilizados para conectar os elementos de um ALU-PAE de maneira linear. A estrutura de comunicação permite conexões ponto-a-ponto e ponto-a-multiponto das saídas até as entradas de outros elementos. Chaves configuráveis existentes ao final de cada linha permitem a conexão de um determinado canal com o canal de dados do elemento vizinho.

3.4.3 RAM-PAE

Os RAM-PAEs são semelhantes aos ALU-PAEs, com a diferença de que a unidade aritmética e lógica existente no bloco central é agora substituída por um elemento de memória RAM de 512 x 16 bits (figura 3.10).

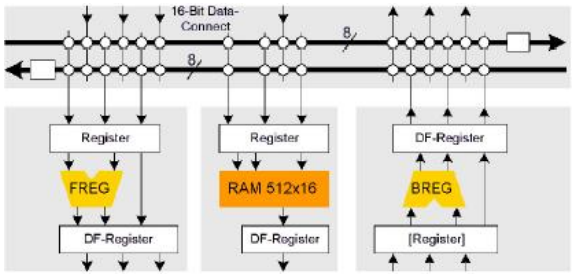

Figura 3.10: ALU-PAE

Essa memória possui duas portas independentes permitindo acessos de leitura e escrita simultâneos. Outra característica do elemento de memória é que ele é orientado à pacotes. Para ler um determinado dado, um pacote contendo o endereço desejado é enviado para a entrada de endereços da memória, a qual recebe a informação e gera um novo pacote de dados contendo o conteúdo da célula desejada. De modo semelhante, escrever na RAM exige a criação e envio de um pacote para as entradas de endereço e de dados da porta de escrita. Se a RAM estiver configurada em modo FIFO, não é necessário endereçamento, a FIFO gera pacotes de saída desde que não esteja vazia. RAMs e FIFOs podem ser pré-carregadas durante a inicialização, o que permite que sejam utilizadas como LUTs

ou que armazenem parâmetros, coeficientes e dados de inicialização. Vale ressaltar também que os RAM-PAEs podem ser combinados para formar uma memória RAM grande com um espaço de endereçamento contíguo, o que facilita o desenvolvimento de aplicações que precisam bufferizar quantidades razoáveis de dados como vários algoritmos de processsamento de imagens.

3.4.4 Gerenciador de Configurações

O gerenciador de configurações (GC) manipula todas as tarefas de configuração do *array*. Inicialmente, configurações são lidas de uma SRAM externa através da interface de comunicação para a *cache* interna. Após essa fase inicial, o gerenciador carrega a configuração (que inclui *opcodes*, canais de roteamento e constantes) para o *array*. Tão logo um dado PE esteja configurado ele inicia a operação desde que os dados estejam disponíveis.

O gerenciador é formado, portanto, por uma máquina de estados e uma RAM interna, ele se comunica com os PAEs através de um barramento de configuração. Cada PAE armazena localmente o seu estado atual, ou seja, se faz parte de uma configuração ativa ou se está livre. Toda vez que um PAE é configurado ele passa para um estado de ”objeto configurado“, indicando ao gerenciador de configurações que ele não pode ser alocado para uma nova tarefa. O gerenciador armazena os dados de configuração em sua própria *cache* até que o PAE requerido torne-se disponível.

Durante o processo de carga alguns PAEs assumirão o estado de ”objeto configurado“ antes de outros. Contudo, o processamento da tarefa não depende do término da operação de reconfiguração, uma aplicação parcialmente configurada consegue executar as instruções sem perda de pacotes. Existe, portanto, uma concorrência entre configuração e computação.

3.4.5 Gerenciamento de Pacotes e Sincronização

Os objetos PAE comunicam-se através de uma rede orientada a pacotes. Dois tipos distintos de pacotes podem ser enviados través do *array*: pacotes de dados e pacotes de eventos. Em um PAE, uma operação é realizada tão logo todos os dados (pacotes) necessários estejam disponíveis nas entradas. Os resultados são enviados para seus destinos assim que são produzidos.

Dessa forma, é possível mapear um grafo de fluxo de dados diretamente para os objetos do PAE. Os pacotes de eventos, por sua vez, possuem, normalmente, 1 bit e atuam como sinalizadores, transmitindo informações de status que são utilizadas no controle da execução das ULAs e na geração de pacotes.

3.4.6 Mapeamento de Aplicações para o XPP

O XPP conta com uma linguagem proprietária chamada NML (*Native Mapping Language*). Trata-se de uma linguagem estrutural com primitivas de reconfiguração desenvolvida para mapear as aplicações para o *array* do XPP.

Em NML, configurações consistem em módulos que são especificados em uma linguagem de descrição de hardware estrutural semelhante ao VHDL. Os objetos PAE são explicitamente alocados e podem ser posicionados pelo projetista. Módulos hierárquicos permitem o reuso de componentes, o que é especialmente importante com as pressões existentes de diminuição de tempo de projeto.

Um programa NML completo consiste de um ou mais módulos, uma sequência de

módulos inicialmente configurados, mudanças diferenciais, e comandos que mapeiam sinais de eventos para pedidos de reconfiguração, ou seja, a manipulação do processo de reconfiguração é parte explícita de um programa.

A linguagem NML no projeto XPP é mais um exemplo da necessidade de construção de ferramentas específicas quando uma arquitetura reconfigurável é desenvolvida.

3.5 Arquitetura DART

Embora desempenho tenha sido a principal motivação da maioria dos projetos de arquiteturas reconfiguráveis apresentados na literatura, cada vez mais surgem pesquisas sobre soluções reconfiguráveis voltadas para um baixo consumo de energia.

Com a chegada da necessidade de processamento multimídia em sistemas como o telefone celular, serão necessárias arquiteturas que integrem alta performance com baixo consumo de energia. Uma outra característica e necessidade dessas aplicações já discutita em seções anteriores é flexibilidade. Além das diversas funcionalidades existentes ainda é preciso oferecer suporte para a inclusão de novos serviços.

Devido à falta de flexibilidade dos ASICs e da baixa performance de processadores DSP foi proposta a arquitetura DART (DAVID et al., 2002). A arquitetura subdivide-se em *clusters* de processamento (figura 3.11) que atuam individualmente.

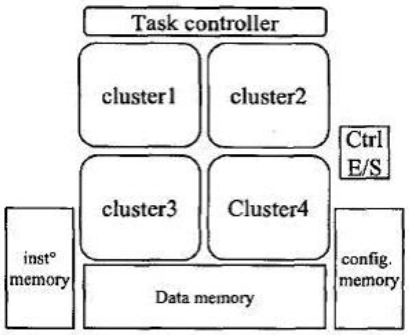

Figura 3.11: Arquitetura DART

Essa subdivisão traz como vantagem a minimização da complexidade do sistema de controle, o qual realiza o escalonamento das tarefas a serem executadas de acordo com a prioridade e a disponibilidade dos recursos. O sistema de controle é hierárquico, o controle pricipal não realiza o sequenciamento das instruções ele apenas atribui uma determinada tarefa ao *cluster*, o qual através do seu sistema de gerenciamento próprio executa o algoritmo.

3.5.1 Arquitetura do Cluster

Uma dificuldade que a arquitetura DART enfrenta é a falta de regularidade das aplicações. Para assegurar o suporte para a execução de tarefas diferentes (métodos diferentes de cálculos e tipos distintos de dados são algumas das diferenças) cada cluster DART integra duas unidades básicas: *DataPath* Reconfigurável (DPR) e um *core* FPGA. Os DPRs (figura 3.12) são reconfiguráveis no nível funcional a fim de otimizar a interconexão entre operadores aritméticos (como mulitplicadores e ULA). O *core* FPGA, por sua vez, é reconfigurável no nível de bit para suportar eficientemente processamentos lógicos.

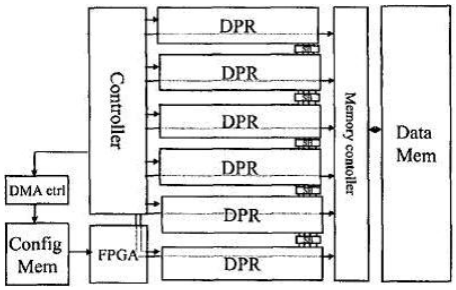

Figura 3.12: Arquitetura de um Cluster

Cada cluster integra um *core* FPGA e seis DPRs. Os DPRs podem estar interconectados entre si ou podem atuar independentemente em linhas de execução diferentes. Tanto o FPGA quanto os DPRs acessam o mesmo espaço de memória de dados, sendo do controlador a responsabilidade pela reconfiguração dessas unidades. A reconfiguração do FPGA é realizada de maneira serial, o controlador do cluster necessita apenas especificar um limite de endereço para o controlador DMA, o qual irá gerenciar as transferências de dados de uma memória de configuração. A reconfiguração dos DPRs é realizada através de instruções, cabendo ao controlador do *cluster* sequenciar essas instruções

3.5.2 O DPR

A primitiva de processamento aritmético do DART são os DPRs (figura 3.13). Cada DPR possui 4 unidades funcionais seguidas por um registrador e suporta SWP (*Sub Word Parallelism*). O uso do conceito de SWP é justificado pela grande quantidade de diferentes tamanhos de dados em um processsamento. Por exemplo, codificação de áudio e vídeo mesmo sendo processamentos aritméticos, trabalham com tamanhos de dados diferentes (respectivamente 8 e 16 bits). Dessa forma, existem operadores aritméticos otimizados para o tamanho de dado mais comum (16 bits), entretanto suporta SWP para operações em dados com menor número de bits.

As unidades funcionais são dinamicamente reconfiguráveis e atuam sobre dados armazenados em 4 memórias locais, o que permite uma taxa de 4 leituras/escritas por ciclo de relógio. Esses recursos estão interconectados através de uma rede de múltiplos barramentos. A organização hierárquica do DART permite manter esses barramentos relativamente pequenos o que favorece a diminuição do consumo de energia. Em virtude dessa rede, cada recurso pode se comunicar com outro no DPR, permitindo um *datapath* otimizado para cada tipo de padrão de cálculo. A flexibilidade da estrutura de conexões permite o compartilhamento de dados e alguma economia de energia já que a memória pode ser lida simultaneamente por 4 unidades funcionais. A estrutura permite também que através de conexões com os barramentos globais diversos DPRs possam estar ligados para processamento paralelo.

3.5.3 Reconfiguração Dinâmica no DART

A reconfiguração dinâmica dos DPRs é realizada através de instruções que trazem informações sobre os operadores e as interconexões. A reconfiguração das interconexões deve permitir a especificação da maneira como os recursos se comunicam e dessa forma, customizar o *datapath* para a execução de determinado padrão de cálculo. São necessários 88 bits para especificar as conexões dentro de um DPR e entre o DPR e os barramentos globais. Para o segundo nível na hierarquia de interconexões 40 bits são necessários para

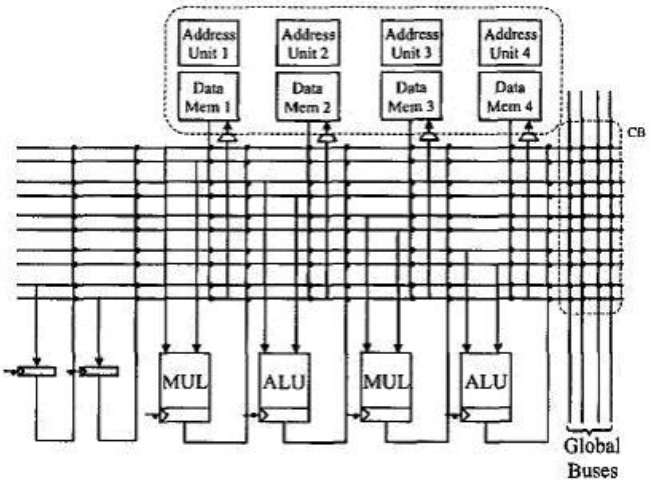

Figura 3.13: Arquitetura do DPR

especificar as transações na rede segmentada entre os DPRs.

A reconfiguração dos operadores é menos custosa do ponto de vista de volume de dados, 34 bits são utilizados para especificar as operações a serem feitas em cada unidade funcional do DPR, o tamanho de dados a ser utilizado e, eventualmente, o desligamento do *clock* de um operador que não esteja sendo utilizado.

Todos esses dados de configuração são armazenados em instruções gerenciadas pelo controlador do cluster. Com 6 DPRs em cada cluster o tamanho de dados da configuração é de 772 bits. Considerando que existem diversas aplicações com operações irregulares a arquitetura deve suportar de maneira eficiente a constante mudança de configuração, para isso o projeto DART definiu dois modos de reconfiguração: hardware e software.

A reconfiguração por hardware é utilizada em processamentos regulares, nos quais a configuração é mantida por determinado tempo. Nesses casos, a aplicação tem acesso a flexibilidade total do DPR e a reconfiguração ocorre em 4 ciclos.

A reconfiguração por software, é utilizada em processamentos irregulares. Nesses casos, é necessário reconfigurar o DPR em um ciclo com uma instrução de tamanho razoável. Para tornar isso possível a flexibilidade deste tipo de reconfiguração foi limitada. É possível modificar apenas a funcionalidade dos operadores, o tamanho dos dados e sua origem. Em virtude dessas limitações, tornou-se possível a reconfiguração em cada ciclo com uma instrução de 50 bits de largura.

3.5.4 Aplicações e Consumo de Energia

Uma das preocupações do DART é o consumo de energia. A síntese do DPR sugere uma frequência de operação de 130 MHz e do ponto de vista de instruções o DART pode processar até 520 MIPS/DPR. Dadas essas condições, o sistema deve ser eficiente quanto à potência do circuito. Existem em um cluster 4 fontes principais de consumo: os operadores, as unidades de geração de endereços, a memória e a rede de interconexões.

As unidades funcionais do DART foram projetadas também visando baixo consumo. Circuitos críticos como multiplicadores possuem latches para evitar seu funcionamento quando não estão sendo utilizados. Dessa maneira, a eficiência de energia do DART é de cerca de 9.2 MIPS/mW para operações sobre 16 bits e 15.8 MIPS/mW para operações SWP.

A tabela 3.2 apresenta dados obtidos pelo DART para três aplicações. Os dados mos-

Tabela 3.2: Aplicações implementadas com DART

Aplicação	Número de DPRs	Energia
Complex Despreading	2	435.8 nJ
DCT-2D	4	64.7 nJ
Autocorrelação	6	3.15 nJ

trados são o número de DPRs utilizados na implementação e a energia consumida na execução do algoritmo. Os valores foram obtidos por simulação após síntese física.

3.6 Sistemas em um *Chip* Programável

Os dispositivos FPGAs atuais apresentam grande densidade de componentes lógicos, oferecendo a possibilidade de desenvolvimento de projetos complexos. A família Virtex-II da Xilinx, por exemplo, possui um dispositivo que integra até 4 *cores* IBM PowerPC. Nesta seção serão analisados duas famílias de FPGAs: Stratix da Altera e Virtex-II Pro da Xilinx.

3.6.1 Família Xilinx Virtex-II Pro

A família Virtex-II Pro (XILINX, 2003) é a quarta geração da linha de FPGAs de alta densidade Virtex da Xilinx. Os dispositivos foram desenvolvidos em tecnologia 0.13μm com 9 camadas de metal. A figura 3.14 ilustra uma visão geral dos blocos e da matriz de roteamento.

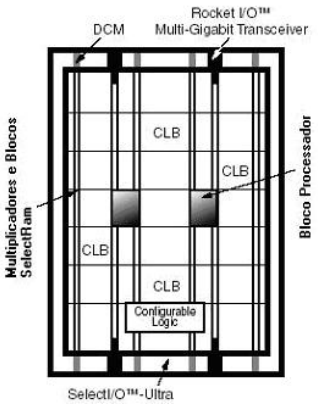

Figura 3.14: Visão geral da arquitetura Virtex-II Pro

Todos os modelos de FPGA da família (tabela 3.3) são programáveis por SRAM. A arquitetura é composta por uma série de blocos lógicos configuráveis (também chamados CLB - *Configurable Logic Blocks*) imersos em uma matriz configurável de roteamento. Os CLBs proporcionam os elementos funcionais para a implementação de lógica combinacional e sequencial. Existem ainda presentes no *chip* diversos blocos de ação especial que visam atender às necessidades de aplicações de processamento digital de sinais e a complexidade de sistemas em um único *chip*.

Dentre os blocos especializados existentes na família Virtex-II Pro podemos citar:

- *Cores* IBM RISC 405 PowerPc
- Circuitos dedicados de I/O

Tabela 3.3: Modelos da família Virtex-II Pro.

Dispositivo	**I/O Dedicado**	**PowerPC**	**Células Lógicas**	**Multiplicadores**	**SelecRAM+ (18Kb)**	**Pinos I/O**
XC2VP2	4	0	3168	12	12	204
XV2VP4	4	1	6768	28	28	348
XC2VP7	8	1	11088	44	44	396
XC2VP20	8	2	20880	88	88	564
XC2VP30	8	2	30816	136	136	692
XC2VP40	0 ou 12	2	43632	192	192	804
XC2VP50	0 ou 16	2	53136	232	232	852
XC2VP70	16 ou 20	2	74448	324	324	996
XC2VP100	0 ou 20	2	99216	444	444	1164
XC2VP125	0, 20 ou 24	4	125136	556	556	1200

- Blocos de memória (BlockSelectRAM)
- Blocos multiplicadores dedicados
- Blocos gerenciadores de *clock* (chamados DCM - *Digital Clock Manager*)

Os recursos programáveis de roteamento interconectam todos esses blocos. A matriz geral de roteamento consiste de um array de chaves de roteamento. Cada elemento programável está acoplado a uma matriz de chaves, possibilitando múltiplas conexões para com a matriz geral.

Todos os elementos programáveis, inclusive as conexões, são controlados por valores armazenados em células de memória SRAM. Esses valores são carregados nas células de memória durante a configuração e podem ser redefinidos em um processo de reconfiguração que pode ser até mesmo parcial.

3.6.2 CLBs Virtex

Conforme mencionado, os CLBs são utilizados para a definição de lógica combinacional e sequencial. Cada CLB está acoplado a uma matriz de chaves que permite o acesso à matriz de roteamento geral (3.15).

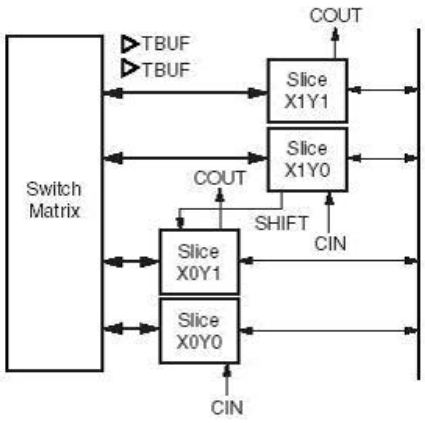

Figura 3.15: CLB Virtex-II Pro

Um CLB compreende 4 sub-blocos (*slices* na terminologia Xilinx) idênticos. Os 4 sub-blocos estão divididos em 2 colunas de 2 sub-blocos, o CLB possui cadeias de *carry* independentes em cada coluna e cadeia de *shift* comum.

Cada sub-bloco, por sua vez, contém dois geradores de funções de 4 entradas, lógica para *carry*, *gates* para lógica aritmética, multiplexadores e dois elementos de armazenamento. Cada gerador de funções pode ser programado como uma LUT de 4 entradas, 16 bits de memória distribuída ou um elemento registrador de deslocamento de 16 bits.

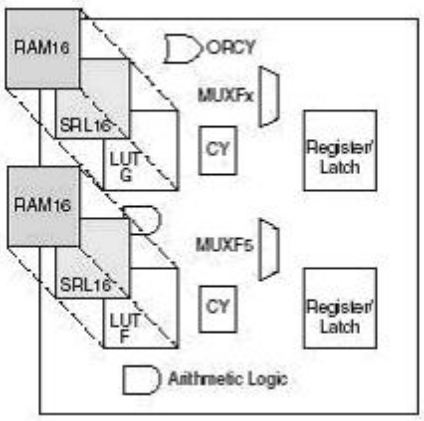

Figura 3.16: Esquemático do subbloco

A figura 3.16 mostra uma visão detalhada de uma parte do sub-bloco. Cada gerador de função é implementado como uma 4-LUT, dessa maneira são capazes de implementar qualquer função booleana de 4 entradas. O atraso de propagação é independente da função implementada e os sinais resultantes do gerador de funções possuem diferentes destinos possíveis: podem sair do sub-bloco, servir de entrada para os *gates* de lógica aritmética, servir de entrada para o multiplexador para lógica de *carry* ou alimentar o elemento de armazenamento. Além dessas funcionalidades o sub-bloco da Virtex possui muliplexadores que podem ser utilizados para o agrupamento de geradores de funções, possibilitando a definição de funções de maior número de entradas.

O elemento de armazenamento pode ser configurado como um *flip-flop* D sensível à borda ou como um *latch* sensível ao nível. Para maior flexibilidade, a entrada D pode ser alimentada tanto pela saída do gerador de funções, quanto por um sinal que entra diretamento no sub-bloco (representado pela entrada BY na figura 3.17).

Uma outro modo de operação importante do CLB é como memória. Cada gerador de funções (LUT) pode implementar um recurso de RAM síncrona de 16 x 1bit que recebe a nomenclatura de elemento SelectRAM+. A tabela 3.4 mostra as possíveis configurações.

Para configurações de porta simples, o bloco SelectRAM+ possui um endereço para escrita síncrona e para leitura assíncrona. Para configurações de porta dupla, a memória possui uma porta para escrita síncrona e leitura assíncrona e uma segunda porta para leitura assíncrona. O gerador de funções conta com entradas separadas para endereços de leitura e escrita na memória (respectivamente de A4 até A1 e de WG4 até WG1 na figura 3.17).

Dentre outras possibilidades de configuração possíveis de um CLB podemos exemplificar:

- **Registrador de deslocamento**: Cada gerador de função pode ser configurado como um registrador de deslocamento de 16 bits. A operação de escrita é síncrona com

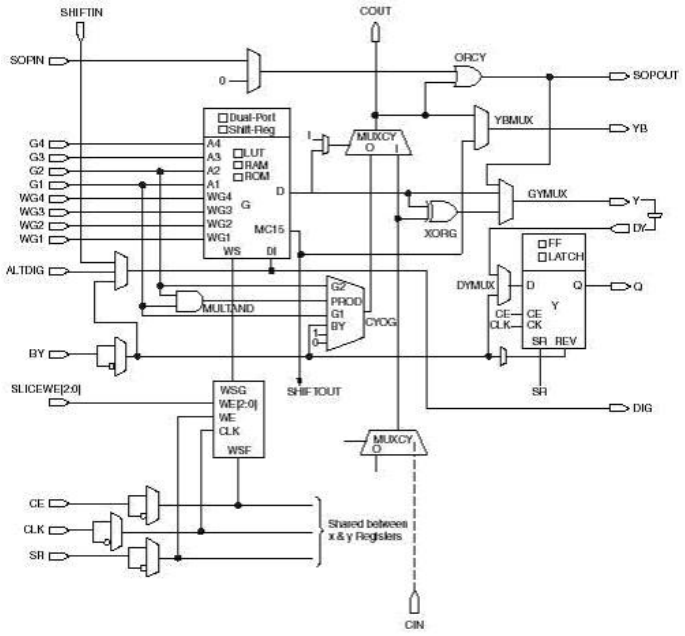

Figura 3.17: Detalhe da metade superior de um subbloco

Tabela 3.4: Configurações possíveis de memória para um CLB

RAM	**Tipo**	**Número de LUTS**
16x1	Simples	1
16x1	Dupla Porta	2
32x1	Simples	2
32x1	Dupla Porta	4
64x1	Simples	4
64x1	Dupla Porta	8
64x1	Dupla Porta	8
128x1	Simples	8

o sinal de *clock* e com um sinal opcional de habilitação de *clock*. A leitura é assíncrona, mas um esquema síncrono pode ser realizado usando-se o elemento de armazenamento.

- **Multiplexador**: Os geradores de função associados com os muliplexadores já existentes dentro de cada sub-bloco podem formar multiplexadores de diferentes tamanhos: 4:1 em um sub-bloco, 8:1 em dois sub-blocos, 16:1 em um CLB e 32:1 em 2 CLBs.

- **Lógica Aritmética**: A lógica aritmética inclui uma porta XOR que permite a implementação de um somador completo de 2 bits dentro de um único sub-bloco.

3.6.3 Estratégias de Roteamento

A matriz de roteamento segue uma estratégia hierárquica, composta basicamente por cinco diferentes modelos de interconexões:

- **Linhas Longas**: A matriz possui 24 linhas longas horizontais e verticais como as da figura 3.18-a. São fios bidirecionais que distribuem o sinal por todo o dispositivo.

- **Linhas Hex**: Roteiam o sinal para cada terceiro ou sexto bloco em todas as quatro direções (figura 3.18-b). Ao todos são 120 linhas desse tipo na horizontal e na vertical.
- **Linhas Duplas**: A figura 3.18-c mostra a estratégia das linhas duplas. Elas roteiam o sinal para cada primeiro ou segundo bloco nas quatro direções. Existem ao todo 40 linhas duplas horizontais e verticais.
- **Conexões Diretas**: As conexões diretas roteiam o sinal entre vizinhos na horizontal, vertical e diagonal (figura 3.18-d).
- **Conexões Internas**: A figura 3.18-e apresenta as conexões internas de um CLB.

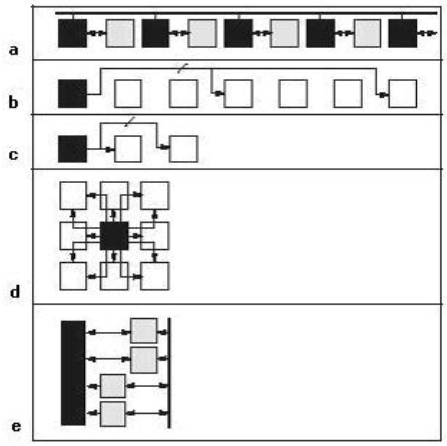

Figura 3.18: Estratégias de interconexão

3.6.4 Configuração

A família Virtex possui capacidades de configuração parcial do circuito, ou seja, é possível, obedecendo algumas restrições, redefinir partes do circuito sem influenciar no restante.

A metodologia de reconfiguração parcial é baseada em um fluxo de projeto modular. Um módulo de projeto é a unidade básica de reconfiguração, e para ser possível a reconfiguração parcial o módulo deve respeitar algumas restrições impostas pelo sistema como por exemplo:

- A altura do módulo reconfigurável deve ser a altura do dispositivo.
- As fronteiras de um módulo reconfigurável são imutáveis, ou seja, a posição e região ocupadas na planta baixa do FPGA são sempre as mesmas.
- IOBs acima e abaixo do módulo são automaticamente considerados recursos do módulo reconfigurável.
- Comunicação de um módulo reconfigurável para com qualquer outro módulo externo só é possível através de uma estrutura chamada *bus macro*.

O *bus macro* é utilizado como um caminho de dados fixo para sinais de um módulo reconfigurável que se comunicam com outro módulo. O código HDL deve, portanto, garantir que qualquer sinal utilizado para interfacear outro módulo o faça apenas através do *bus macro*.

Tabela 3.5: Características de alguns dispositivos Stratix

Características	**EP1S40**	**EP1S60**	**EP1S80**	**EP1S120**
LEs	41250	57120	79040	114140
RAM bits	3423744	5115104	7427520	10118016
Blocos DSP	14	18	22	28
Multiplicadores	112	144	176	224
Pinos de I/O	822	1022	1238	1314

3.6.5 Elementos Especiais

Devido à necessidade cada vez maior de performance, os FPGAs Virtex-II Pro incluem alguns elementos especializados. Um desses elementos é um *core* para transferências de dados em alta velocidade chamado Rocket I/O. Como pode ser visto na tabela 3.3, o modelo XC2VP125 pode ter até 24 desses comunicadores embutido. O Rocket I/O pode operar como qualquer taxa de *bauds* no intervalo de 622Mb/s até 3.125 Gb/s por canal.

Outra estrutura especial são os *cores* PowerPc. O modelo XC2VP125 possui quatro desses processadores. Com tecnologia de 0.13μm eles conseguem operar com uma frequência de até 300MHz. O Virtex-II pro ainda possui uma unidade chamada DCM para gerenciamento de *clock*, essa unidade atua evitando *skews* e pode ser utilizada para distribuição de diferentes sinais de *clock* por todo o circuito.

3.6.6 Família Altera Stratix

A família Stratix (ALTERA, 2003) baseia-se em uma tecnologia de 0.13μm, possui programação via SRAM e densidade de até 114140 elementos lógicos (LEs) e 10Mbits de RAM. Os dispositivo oferecem até 28 blocos DSP, tendo até 224 multiplicadores de 9 x 9 bits. A tabela 3.5 apresenta um sumário de alguns dos dispositivos da família.

3.6.7 Arquitetura

A família Stratix segue a arquitetura convencional dos FPGAs sendo composta por uma estrutura matricial bidimencional de blocos lógicos e interconexões (figura 3.19). A hierarquia de interconexões com tamanhos e velocidades variáveis proporciona a ligação entre os blocos de células lógicas (LABs - *logic array blocks*), estruturas de memórias e blocos DSP.

O array lógico consiste de LABs, sendo que cada LAB é composto por 10 elementos lógicos. As estruturas de memória, por sua vez, podem ser blocos M512, M4K ou M-RAM:

- **M512**: São blocos de memória RAM dupla porta com 512 bits (576 se forem contabilizados os bits de paridade). Esses blocos proporcionam memória simples ou dupla porta de até 18 bits de largura podendo ser acessada a uma taxa de até 318 MHz. Blocos M512 são agrupados em colunas ao longo do dispositivo sendo posicionados entre alguns LABs.

- **M4K**: Blocos RAM dupla porta de 4K bits (4608 com a paridade). Cada palavra pode ter até 36 bits de largura. A frequência máxima de acesso é de 291MHz.

- **M-RAM**: Blocos de memória dupla porta com 512Kbits (com a paridade totaliza 589824 bits). Palavras de até 144 bits com uma frequência de operação possível de 296MHz.

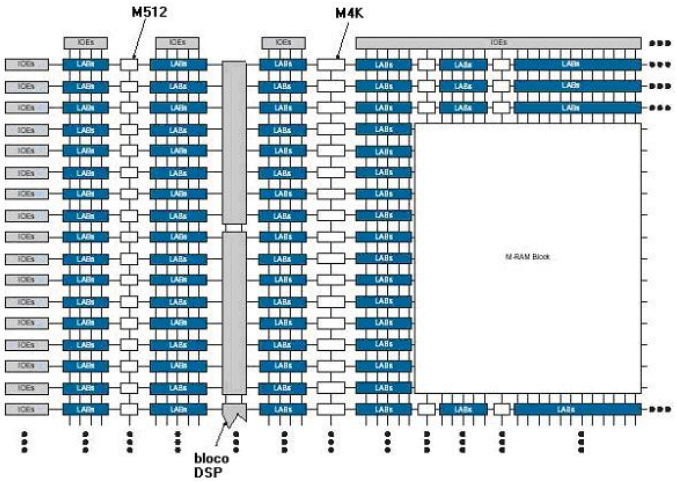

Figura 3.19: Arquitetura da família Stratix

3.6.8 Blocos de Células Lógicas (LABs)

Cada LAB é composto por 10 LEs, sinais de controle, linhas de interconexão locais e linhas especiais para a formação de cadeias de elementos. Com esse suporte, pode-se conectar diretamente LUTs e registradores de um determinado LE com LUTs e registradores de um LE adjacente. Esse mecanismo, também é utilizado em projetos *bit-slice* que necessitem propagação de sinais (como um bit de *carry*) entre os elementos lógicos.

O LE (figura 3.20), por sua vez, é a menor unidade de lógica no Stratix. Contém uma LUT de 4 entradas, implementando, portanto, qualquer função booleana de 4 variáveis. Possui também um registrador programável que pode ser configurado como flip-flop D, T, JK ou SR, tendo além dos sinais de dados e *clock* entradas também de habilitação e *preset*. Para funções puramente combinacionais o registrador não é utilizado e a saída do LE é diretamente a saída da LUT.

Cada LE tem saídas que acessam os recursos de roteamento local, em nível de linha e em nível de coluna. A saída da LUT e do registrador pode conduzir sinais para quaisquer desses recursos independentemente. Existem ainda duas saídas que podem levar os sinais para as interconexões de linha e coluna e outra para o roteamento local. Essa estratégia permite uma melhor utilização dos recursos lógicos, uma vez que torna possível que a LUT esteja ligada a um recurso de interconexão e o registrador a outro, ou seja, não necessariamente os recursos do LE tenham de participar da mesma função lógica.

3.6.9 Blocos Especiais para DSP

Dentre as mais comuns funções de processamento de sinais temos: filtro FIR, transformada rápida de Fourier (FFT) e transformada discreta do cosseno (DCT). O bloco básico para essas operações é o multiplicador. Cada dispositivo Stratix possui 2 ou mais colunas com blocos DSP (figura 3.21) que realizam essas operações mais rapidamente do que implementações baseadas em LEs. Cada bloco DSP pode ser configurado para suportar:

- Oito multiplicadores 9x9.
- Quatro multiplicadores 18 x 18.
- Um multiplicador de 36 x 36.

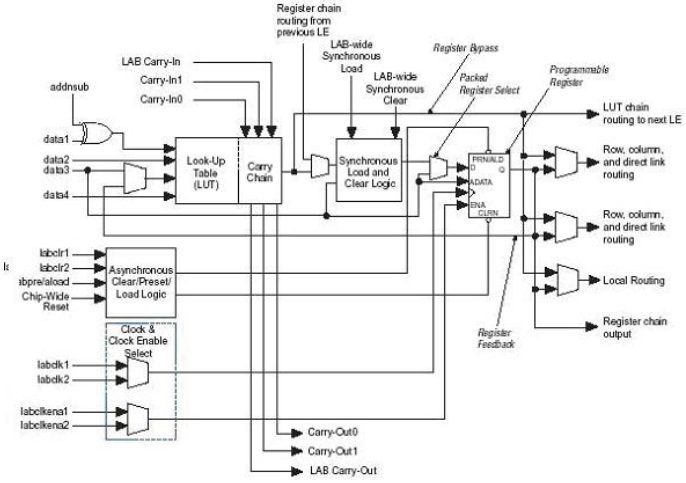

Figura 3.20: LE da família Stratix

O bloco multiplicador pode ainda estar conectado com um bloco de saída interno da unidade DSP. Esse bloco possui circuitos de adição, subtração ou acumulação do resultado para operações que são comuns em processamento de sinais como MAC.

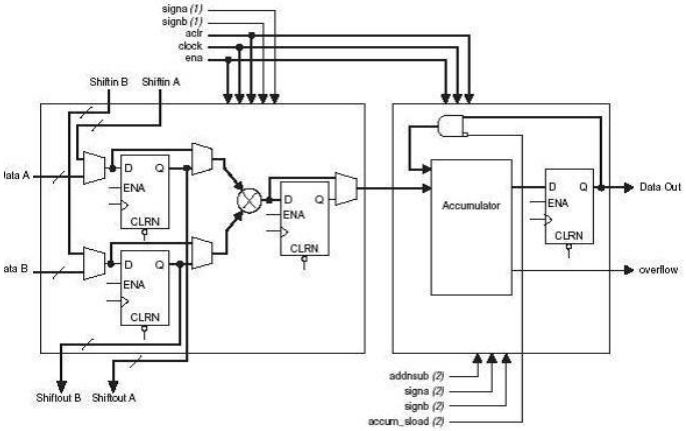

Figura 3.21: Bloco DSP configurado para operações MAC

3.6.10 Configuração

Os FPGAs Stratix são baseados em SRAM, podendo ser configurados em menos de 100 ms, utilizando uma interface paralela de 8 bits. É possível reconfiguração parcial de maneira restrita, apenas unidades de gerenciamento de *clock* chamadas PLLs podem ser modificadas em *run-time*.

3.7 Sistemas de Software para Computação Reconfigurável

A pesquisa de ferramentas de projeto para o desenvolvimento de sistemas reconfiguráveis constitui um campo que oferece grandes possibilidades de avanços. Ao contrário do que podemos chamar de sequência tradicional de projeto, a criação de uma arquitetura reconfigurável demanda o desenvolvimento de um considerável número de softwares auxiliares, tanto para a construção do sistema como para mapeamento dos algoritmos e mais

recentemente para gerenciar sua operação (sistemas operacionais e de relocação).

A criação de CADs para desenvolvimento e programação são importantes, visto que, atualmente, é necessário que o programador tenha um conhecimento razoavelmente profundo da arquitetura alvo. O software de projeto pode variar desde uma ferramenta mais simples para ajudar no mapeamento manual de um circuito até um ambiente de CAD complexo e totalmente automatizado.

Tanto se for considerado um processo ”artesanal“ ou um fluxo completamente automatizado, existem um certo número de fases a serem completadas até a obtenção da descrição final do circuito. A figura 3.22 mostra três fluxos possíveis com diferentes níveis de intervenção do projetista. O primeiro fluxo ilustra um processo automatizado no qual a participação humana está na descrição do funcionamento do circuito utilizando-se uma linguagem. Essa linguagem pode ser de alto nível como C, C++ e Java ou pode ser uma linguagem de descrição de hardware como VHDL ou Verilog. Vale ressaltar que uma tendência é a utilização cada vez maior de linguagens em nível de sistema como SystemC.

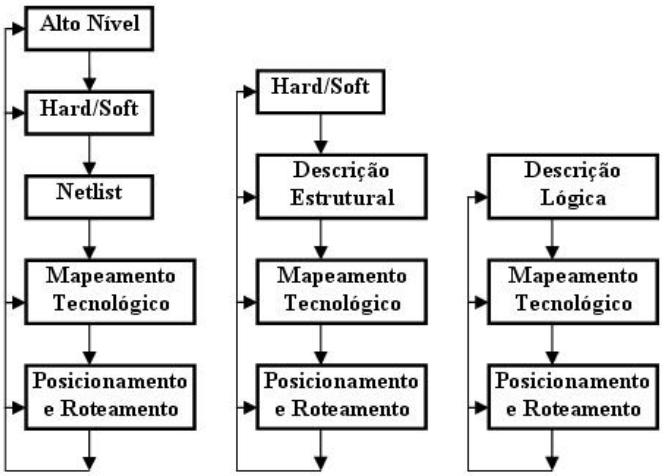

Figura 3.22: Alguns possíveis fluxos de projeto

Após a definição do circuito em alto nível, existe um processo de particionamento entre o que será implementado em hardware e o que será em software. Existem diversas técnicas e muita pesquisa tem sido realizada em projeto conjunto de hardware/software e mecanismos de síntese de alto nível. Essas técnicas não serão mencionadas aqui por fugirem do escopo deste trabalho, no entanto um vasto material pode ser encontrado na literatura especializada.

O mapeamento tecnológico é o processo pelo qual os elementos são mapeados para as estruturas lógicas existentes no dispositivo alvo (hardware), ou seja, é um processo dependente da arquitetura. Por exemplo, para uma arquitetura baseada em LUTs este estágio particiona o circuito em subfunções pequenas que são atribuídas às LUTs. As células lógicas dos FPGAs atuais possuem mais de uma LUT além de estruturas de armazenamento. Essas LUTs podem ser utilizadas separadamente ou agrupadas para gerar funções mais complexas. Para estruturas reconfiguráveis que possuem blocos de memória embarcados, a ferramenta de mapeamento pode considerar utilizar esses bits de memória como unidades lógicas quando não estiverem sendo usados para armazenamento.

Após o mapeamento do circuito, os blocos resultantes devem ser posicionados no hardware reconfigurável. Cada um desses blocos é associado a uma localização específica, o mais próximo possível aos demais recursos lógicos com os quais ele se comunica.

Com a densidade enorme dos atuais dispositivos de hardware programável como os FPGAs, o problema do posicionamento torna-se cada vez mais uma tarefa demorada e importante. O posicionamento pode ser facilitado por um projeto adequado (automatizado ou não) de planta baixa (*floorplanning*). Um algoritmo de planejamento topológico primeiro particiona as células lógicas em agrupamentos nos quais as células de um mesmo grupo possuem diversos canais de comunicação entre si. Esses grupos são posicionados como unidades no hardware reconfigurável.

Esses diferentes fluxos apresentados existem também no projeto tradicional de circuitos. No restante desta seção serão analisadas ferramentas de projeto e sistemas operacionais voltados para os problemas específicos de sistemas reconfiguráveis de computação.

3.7.1 Ferramentas para Modificação de Arquivos de Configuração

Esta classe de ferramentas é largamente utilizada no projeto de arquiteturas reconfiguráveis quando utiliza-se uma estratégia de reconfiguração parcial. A função principal é modificar o *bitstream* e assim promover alterações nos módulos dinâmicos existentes no tecido reconfigurável.

A maioria das ferramentas de manipulação de *bistream* focam nos FPGAs da família Virtex.

3.7.1.1 Parbit

Parbit (HORTA; LOCKWOOD; KOFUJI, 2002) é ferramenta utilizada para a geração de arquivos para configuração parcial em dispositivos XILINX. Esta ferramenta foi desenvolvida na universidade de Washington e utilizada, por exemplo, na implementação da arquitetura RECATS (apresentada a seguir).

Para gerar o *bitstream* parcial, Parbit realiza a leitura dos dados de configuração do *bitstream* original e copia para o arquivo parcial apenas os bits de configuração relativos a área dinâmica definida pelo usuário. A ferramenta então gera novos valores para os registradores de configuração.

Uma outra função importante da Parbit é a relocação da área reconfigurável. A ferramenta pode determinar novas coordenadas para o posicionamento da lógica em hardware de acordo com informações enviadas pelo projetista.

3.7.1.2 JBits

Jbits (GUCCIONE; LEVI; SUNDARARAJAN, 1999) é um conjunto de classes Java que oferecem uma API para acessar e modificar o arquivo *bitstream* de FPGAs Xilinx. A interface pode operar sobre *bitstreams* gerados pelas ferramentas de projeto da Xilinx ou em *bistreams* lidos do próprios dispositivo. Essa API permite, portanto, que todos os recursos reconfiguráveis como LUTs, interconexões e flip-flops possam ser individualmente (re)configurados sob o controle do software.

A API tem sido utilizada na construção de circuitos completos e na modificação de circuitos já existentes. O estilo orientado a objetos da linguagem Java permite que sejam desenvolvidos blocos de hardware e *cores* parametrizáveis prontos para serem mapeados para o dispositivo. Diversas outras ferramentas foram criadas utilizando Jbits como base, uma vez que ela pode ser utilizada tanto dentro de um fluxo de projeto tradicional para posicionamento, por exemplo, quanto para a implementação de sistemas reconfiguráveis.

3.7.2 Arquitetura RECATS como Estudo de Caso

Recats (HORTA; KOFUGI, 2002) é uma switch ATM reconfigurável. A idéia principal é utilizar uma biblioteca de módulos de hardware pré-compilados que podem ser configurados no FPGA em tempo de execução. Com o hardware adaptável, a arquitetura Recats pretende atender as necessidades existentes de switches ATM inteligentes.

O dispositivo programável utilizado pelo Recats é um FPGA Virtex da Xilinx. A capacidade de reconfiguração parcial dos FPGAs Virtex é a maneira encontrada para a obtenção de reconfiguração dinâmica do chip.

Uma representação das principais unidades da arquitetura pode ser vista na figura 3.23. Os blocos principais são:

- Módulo de entrada (*Input Module* - IM): Recebe as células e as converte em palavras de 16 bits para serem processadas pela parte operativa do Recats.
- Tabela de tradução e roteamento (*Translation Table and Routing* - TTR): Processa os cabeçalho das células ATM e faz consultas na tabela de roteamento.
- Gerenciador de fila (*Queue Manager* - QM): Gera os sinais de controle do *buffer* de dados compartilhados.
- *Buffer* de dados compartilhados (*Shared Data Buffer* - SDB): Armazena as células ATM que serão transmitidas pelos canais de saída.
- Módulo de saída (*Output Module* - OM): Este módulo está conectado a um multiplexador gerenciado pela unidade de controle. De acordo com a especificação da tarefa, o módulo de saída recebe as células do buffer de dados ou da unidade de reconfiguração parcial e os transmite através da interface de comunicação.
- Unidade de controle (CTRL): A unidade de controle escalona os acessos dos módulos de entrada e saída ao *buffer* de dados, utilizando para a realização desse escalonamento os sinais de controle gerados pelo gerenciador de fila. Este módulo também é responsável pelo controle de reconfiguração, gerando os sinais necessários para a leitura dos dados de *bitstream* parcial de uma memória externa.
- Módulo de reconfiguração parcial (*Partial Reconfigurable* - PR): É o módulo dinâmico do Recats que pode ser substituído em tempo de execução. Recebe as células do *buffer* de dados e as processa de acordo com a função atualmente carregada. Uma das funcionalidades que podem ser ativadas na PR introduz a capacidade de realizar a criptografia dos dados que passam pela *switch*.

O *chip* é programado no processo de inicialização através de uma EPROM conectada à porta serial mestre. A reconfiguração é implementada através de *bitstreams* parciais armazenados em uma memória Flash externa.

O Recats adquire capacidade de reconfiguração em tempo de execução utilizando a possibilidade de reconfiguração parcial dos dispositivos Virtex. Para configurar cada recurso no Virtex uma série de bits, divididos em campos de comandos e dados, são carregados no dispositivo. Conforme visto na seção que trata da arquitetura Virtex, a reprogramação deve acontecer sobre uma fatia completa do circuito, ou seja, pelo menos uma coluna que ocupe a altura total do dispositivo. A construção de uma estratégia de RTR no Recats segue os seguintes passos:

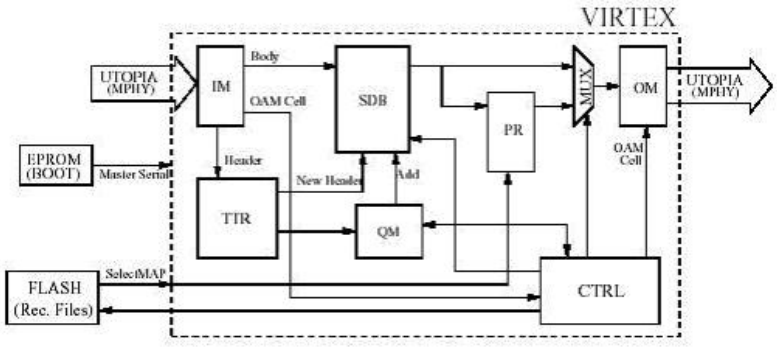

Figura 3.23: Arquitetura RECATS

- Geração de um *bitstream* completo, incluindo as colunas para PR, mas sem lógica implementada nessa área.
- Geração de um *bitstream* completo apenas com o módulo PR na mesma área reservada anteriormente.
- Geração de um *bitstream* parcial com o módulo PR.
- Armazenamento e download para o dispositivo dos *bitstream* parciais de acordo com a necessidade do Recats.

Os *bitstream* completos podem ser gerados com ferramentas comerciais. Os *bistreams* parciais, por sua vez, são gerados no projeto Recats a partir da ferramenta Parbit descrita, a qual permite a criação de um arquivo de configuração parcial que modifique apenas a lógica existente dentro de uma região previamente definida. No Recats, o módulo CTRL é o responsável pelo download das configurações.

3.7.3 Controladores de Reconfiguração Parcial

A troca de funções e configurações é uma tarefa extremamente importante no sistema reconfigurável. Se o mecanismo não for adequado para a estrutura da arquitetura os ganhos produzidos pela especialização podem ser desperdiçados. Serão mostrados em seguida, dois modelos de controladores de reconfiguração que apresentam os elementos básicos e sua interação no contexto do processo de reconfiguração parcial.

3.7.3.1 Reconfigurador de Shirazi

Em (SHIRAZI; LUK; CHEUNG, 1998) é apresentado um framework genérico para controle de reconfigurações. O gerenciador de configurações possui três componentes principais: um carregador de configurações, um componente monitor e um repositório de configurações (figura 3.24).

O monitor mantém informações sobre o estado atual de configuração, também é de sua responsabilidade acionar o carregador e ativar o processo de carga de uma nova configuração em função de um evento, como uma requisição por parte da aplicação em execução.

Como o monitor deve responder rapidamente a diversas condições é interessante que sua implementação seja realizada em hardware. Existem, basicamente, três situações distintas a serem enfrentadas pelo monitor:

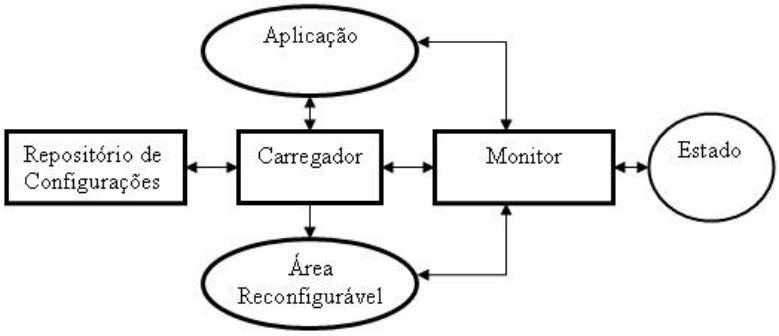

Figura 3.24: Modelo de Reconfiguração

- A duração da tarefa sendo executada atualmente é conhecida em tempo de compilação e a próxima configuração também. Este caso acontece com frequência em processamento de vídeo, uma vez que um dado algoritmo irá processar, normalmente, um conjunto conhecido de frames. A implementação do monitor passa a ser um mecanismo de tempo como um contador.

- Não é possível determinar anteriormente por quanto tempo a tarefa atual irá executar, contudo conhece-se a próxima configuração. Neste caso, são necessárias informaçõe geradas em tempo de execução para determinar quando a próxima configuração deve ser ativada.

- Não é possível saber com antecedência a duração da tarefa corrente e nem qual a próxima configuração. O monitor, nesta situação, depende totalmente de informações geradas dinamicamente é o caso mais complexo. Todas as próximas configurações possíveis devem estar presentes no sistema para uma eventual carga.

O carregador é o responsável por localizar e obter a configuração requisitada na área de armazenamento e trazê-la para a região de execução. Ao final de sua tarefa, o carregador, pode determinar novas condições iniciais para o circuito como um novo *clock*, além de notificar o monitor de que a execução pode ser continuada.

O repositório de configurações é a parte da arquitetura na qual as configurações possíveis são armazenadas. Se necessário, uma série de mecanismos como compactação, *wildcarding* e outros podem ser utilizados para minimizar o espaço necessário para as configurações. No modelo descrito por Shiraze e colaboradores o repositório é subdividido em áreas específicas com indexação das configurações para economia de espaço.

3.7.3.2 Sistema RAGE

Este gerenciador de configurações (BURNS et al., 1997) foi desenvolvido no grupo de pesquisa da Universidade de Glasgow. O primeiro ponto interessante sobre este trabalho reside na maneira como ele foi desenvolvido. Ao invés de definir primeiramente os requisitos e funções necessários ao sistema operacional (de reconfiguração), o caminho tomado foi a criação de aplicações reconfiguráveis demonstrativas, com a finalidade de estudar e extrair as características comuns entre elas e identificar os aspectos importantes para um sistema de gerenciamento eficiente.

Entre as aplicações desenvolvidas estão um renderizador de PostScript e filtros FIR. A partir desses e outros problemas foram definidos os requisitos necessários para um sistema operacional reconfigurável, pode-se citar como exemplo de funções necessárias derivadas

do estudo rotinas para reservar áreas do dispositivo reconfigurável e a capacidade de transformar o circuito, no sentido de modificar sua orientação e realizar translações dentro do espaço de células reconfiguráveis.

A figura 3.25 apresenta os componentes do sistema e suas interações. O gerenciador de hardware virtual é o principal controlador do sistema, ele recebe requisições de novas configurações e tem como tarefa realizar a liberação de áreas ocupadas. A manutenção do status de utilização dos recursos também é responsabilidade deste módulo que possui e atualiza em resposta a cada evento um mapa completo do uso dos operadores.

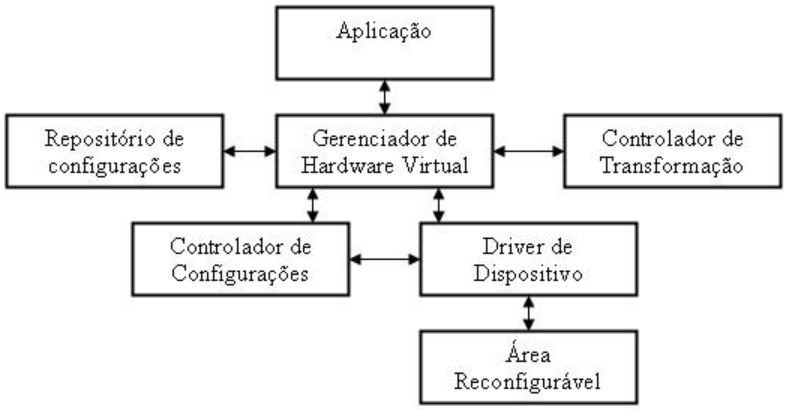

Figura 3.25: Modelo do sistema RAGE

O controlador de transformações age como um relocador. Cabe a ele atuar para resolver conflitos como reposicionar uma dada tarefa se a localização estipulada para ela estiver ocupada ou não for suficiente para receber o mapeamento.

Como pode ser observado na figura, o driver de dispositivos comunica-se com a área reconfigurável. Dessa maneira, ele pode fornecer os comandos e dados que possibilitam a interação da área com o restante do sistema. A vantagem de possuí-lo como uma unidade razoavelmente desacoplada do controle central é que o sistema torna-se facilmente adaptável para qualquer dispositivo configurável. No OS4RS analisado posteriormente, o sistema operacional está mais fortemente ligado à estrutura da malha reconfigurável. A vantagem do caso do OS4RS está na possibilidade de, através do conhecimento e vinculação com o hardware, executar tarefas de gerenciamento de forma mais eficiente.

3.7.4 Sistemas Operacionais para Computação Reconfigurável

Diversos CSoCs e sistemas reconfiguráveis embarcados possuem juntamente com a área reconfigurável um processador com conjunto de intruções RISC (ISP). Isso acontece, por exemplo, na arquitetura MorphoSys como foi visto anteriormente. A utilização de um ISP poderoso pode trazer grandes benefícios para o sistema, pois:

- Pode tratar de maneira mais adequada tarefas irregulares.
- Pode atuar muito próximo à área reconfigurável nas tarefas de reconfiguração.
- Pode gerenciar o funcionamento de módulos auxiliares do sistema (como DMA) de maneira eficiente.

Dessa maneira, as novas aplicações reconfiguráveis podem utilizar tanto a área reconfigurável quanto o ISP para realizar a computação e oferecer ao usuário a máxima performance possível.

Para explorar de maneira adequada esse paradigma é necessário a existência de uma infraestrutura que permita o gerenciamento das tarefas em um ambiente reconfigurável em tempo de execução. É importante também que essa infraestrutura facilite o desenvolvimento de aplicações para uma dada arquitetura oferecendo uma API que esconda o máximo possível do programador a complexidade da arquitetura reconfigurável. Algumas das tarefas que um sistema operacional deve realizar são:

- Proporcionar um ambiente no qual diversas tarefas podem executar concorrentemente.
- Permitir o interfaceamento entre as tarefas através de alguma estratégia de troca de mensagens.
- Gerenciar os recursos disponíveis no sistema de maneira consistente e eficiente.

Na próxima seção será analisada uma proposta para a criação de um sistema operacional para computação reconfigurável que visa preencher os requisitos mencionados.

3.7.5 OS4RS

O OS4RS (NOLLET et al., 2002) é um sistema operacional proposto no IMEC da Bélgica. A idéia é oferecer um sistema capaz de atuar de maneira eficiente sobre um SoC reconfigurável no qual as tarefas a serem executadas podem ser bastante heterogêneas, ou seja, algoritmos de natureza distintas passíveis de execução tanto em um ISP quanto na região reconfigurável de uma arquitetura.

O OS4RS foi contruído através da extensão de um RTOS linux chamado RTAI. O desenvolvimento, a partir de um sistema já existente é viável em virtude da disponibilidade de código fonte, permitindo aos projetistas focarem na implementação da parte reconfigurável. Além disso, o RTAI é um sistema operacional de tamanho pequeno e bastante modularizado, contando com a existência de diversos serviços já criados (USB, UART, etc.).

3.7.5.1 Estratégia de Comunicação

A eventual relocação de uma tarefa de hardware para software (e vice-versa) é uma possibilidade durante a execução de múltiplos algoritmos simultâneos. A estratégia seguida pelos desenvolvedores do OS4RS foi utilizar uma estrutura de comunicação uniforme para hardware e software e assim esconder essa complexidade.

A comunicação baseia-se em troca de mensagens. As mensagens são transmitidas em um formato comum para software e hardware. Cada tarefa recebe um endereço lógico e sempre que o OS4RS escalona uma tarefa em hardware uma tabela de tradução de endereços é atualizada. Essa tabela permite ao sistema operacional realizar a transformação de um endereço lógico em endereço físico e vice-versa. O endereço físico é baseado na localização da tarefa dentro da rede de interconexão denominada pelos autores de ICN.

O OS4RS oferece uma API de troca de mensagens, a qual utiliza os endereços lógicos e físicos para rotear as mensagens. Existem definidos 3 tipos de troca de mensagens:

- Mensagens trocadas entre tarefas escalonadas no processador (P1 e P2 na figura 3.26) são roteadas utilizando somente os endereços lógicos e não passam pela camada de abstração de hardware (HAL).

- A comunicação entre uma tarefa no ISP e uma no FPGA (P3 e Pa) passa pelo HAL. Durante o processo, a API de comunicação executa a tradução entre o endereço lógico e o endereço físico. O endereço físico permite à camada de abstração determinar em qual parte (*tile*) da ICN a tarefa que está enviando/recebendo a mensagem está localizada.

- Embora não ilustrado na figura, tarefas dentro da parte reconfigurável também podem se comunicar. Neste caso, as mensagens não passam pela HAL, trafegando diretamente pela rede de interconexões existente. Contudo, uma vez que o sistema operacional controla o posicionamento das tarefas cabe a ele também determinar as rotas de transporte das mensagens dos algoritmos em hardware através das tabelas de roteamento.

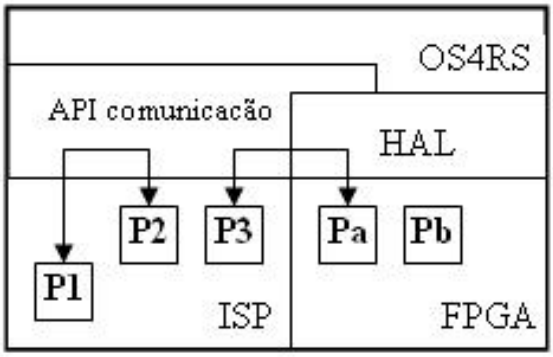

Figura 3.26: Diferentes tipos de mensagens

3.7.5.2 Posicionamento das Tarefas

O problema do posicionamento consiste em definir em qual parte do hardware reconfigurável a tarefa será mapeada. Em tempo de execução o software de gerenciamento do sistema operacional é o responsável por essa atividade. O problema existente nessa situação é que quando a tarefa se traduz em um circuito irregular, o algoritmo de alocação executa rotinas extremamente complexas que introduzem um grande *overhead* ao sistema.

A solução adotada no OS4RS consiste em forçar as rotinas de posicionamento e roteamento a respeitarem limites de área (subdivisões) pré-definidas da área reconfigurável. Essa ”simplificação“ embora introduza limitações ao tamanho das tarefas, permite que o processo de posicionamento seja bastante facilitado. O *overhead* torna-se quase inexistente, já que o posicionamento de um novo algoritmo passa a ser a simples substituição do circuito existente em uma subdivisão do FPGA por outro. Essas modificações são possíveis em virtude da reconfiguração parcial permitida pelo FPGA. Os autores do OS4RS optaram por um desperdício de área (fragmentação) , uma vez que a tarefa pode não utilizar todos os recursos da subdivisão em favor da drástica redução do *overhead* de realocação, o qual, efetivamente, representava o fator mais limitante da performance do sistema.

Para o roteamento de uma nova tarefa mapeada, de maneira semelhante, não existe a necessidade da execução de algoritmos complexos. A malha de interconexões é fixa e basta ao OS4RS executar a atualização das tabelas de roteamento.

3.7.5.3 Chaveamento de Contexto

Uma tarefa em execução pode, de acordo com a necessidade e devido ao número limitado de subdivisões no hardware ter seu processamento interrompido e ser realocada para outa região (ISP ou parte reconfigurável). Dessa maneira, o OS4RS possui suporte para a preempção e remanejamento de tarefas.

Todo o processo de preempção necessita de salvamento de contexto. Para tarefas sendo executadas no ISP, os registradores do processador e a memória da tarefa são o conjunto total de informações que representam o estado corrente do algoritmo. Assim, o processo de salvamento consiste na cópia de todos os valores dos registradores para uma pilha referente à tarefa. Quando a tarefa é reescalonada e volta a executar, o processo inverso ocorre com os valores sendo restaurados nos registradores a partir da pilha.

O mesmo não se aplica ao hardware, uma vez que a informação de estado fica distribuída entre registradores, *latches* e memória interna da tarefa. Para resolver esse interessante e importante problema os autores do OS4RS apresentam uma técnica de alto nível desenvolvida e descrita através do exemplo apresentado a seguir.

O modelo utilizado no OS4RS utiliza o conceito de estados de mudança (pontos de chaveamento). O sistema operacional pode notificar a tarefa de que ela será realocada em qualquer momento (1 na figura 3.27). A tarefa continua em execução até atingir um ponto de chaveamento, quando então vai para o estado de interrupção (2). No estado de interrupção todas as informações necessárias para restauração são transferidas para o OS4RS (3), o qual irá reinicializar a tarefa em outra área de processamento.

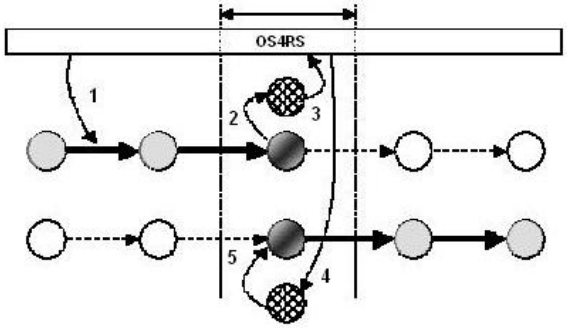

Figura 3.27: Chaveamento de contexto

O O4RS é um reconfigurador prático. Nas próximas seções serão apresentados sistemas que estão no universo de modelos para a construção de controles de reconfiguração.

3.8 Tendências em Computação Reconfigurável

Este capítulo analisa tendências no campo de arquiteturas reconfiguráveis. Neste contexto, destacam-se as potencialidades de arquiteturas multigranulares, ou seja, arquiteturas que reúnem operadores elementares reconfiguráveis de diferentes tamanhos no mesmo *chip*, tornando-se assim poderosas e flexíveis para uma ampla classe de aplicações.

3.8.1 Arquiteturas Multigranulares e Polimorfismo

A noção de polimorfismo, neste contexto, refere-se à capacidade de configurar o hardware para o processamento eficiente de não apenas uma classe de problemas, mas sim uma ampla gama de aplicações. Essas aplicações podem ser distintas como DSP, multimídia, computação científica em geral, telecomunicações, etc.

Uma primeira questão a ser pesquisada é a granularidade dos processadores dos futuros chips com 1 bilhão de transistores. A granularidade tem impacto direto no desempenho da arquitetura, assim, o mesmo problema mapeado em uma área de silício com milhões de processadores elementares de granularidade fina e nessa mesma região com poucos processadores complexos de granularidade grossa traz resultados bastante diferentes.

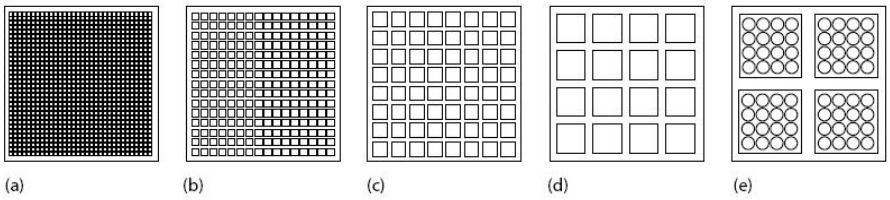

Figura 3.28: Múltiplas Granularidades

A figura 3.28 mostra 5 diferentes granularidades:

- a) Granularidade super fina (FPGAs)
- b) Matriz de processadores elementares. Exemplos: RaPiD (EBELING; CRONQUIST; FRANKLIN, 1997), Piperench (SCHMIT et al., 2002), PACT (PACT, 2004)
- c) Múltiplos processadores mais complexos. Exemplo RAW (WAINGOLD et al., 1997)
- d) Processadores de granularidades bastante grossa, já explorando o paralelismo em nível de thread.
- e) Processadores chamados de ”ultra-grandes“ (NAGARAJAN et al., 2001). Cada processador, possui diversas ULAs.

Um ponto importante a ser mencionado é que uma infra-estrutura de comunicação eficiente entre esses *cores* (como c), d), e e) na figura 3.28) é também uma característica imprescíndível. A customização do hardware para uma aplicação não fica restrita apenas aos operadores do *datapath*. Sistemas de comunicação flexíveis e adaptáveis para diferentes classes de problemas começam a tornar-se fundamentais. Além disso, em um tempo no qual os projetos estão cada vez mais complexos o reuso de módulos é muito importante. As NoCs (*Network-on-Chip*) permitem que o reuso de *IP-Cores* aconteça de modo escalável e sem a necessidade de modificações no hardware dos blocos.

Os modernos sistemas reconfiguráveis contam com diferentes elementos de hardware. O conceito de tarefa e relocação de tarefas já está presente em diversas plataformas reconfiguráveis nas quais um algoritmo pode estar mapeado em um processador de instruções RISC ou em uma área reconfigurável dentro do chip. Nesse contexto, o estabelecimento de uma estrutura de comunicação ponto-a-ponto para permitir a troca de dados entre tarefas passa a ser praticamente inviável de ser realizado de maneira eficiente, uma vez que seriam necessários algoritmos complexos de roteamento operando em tempo real. As NoCs viabilizam a construção dessa infra-estrutura. Através de sistemas de roteadores *on-chip* permitem o desenvolvimento de sistemas multitarefa reconfiguráveis.

Voltando ao exemplo dos sistemas com diferentes tamanhos de operadores. As arquiteturas de menor granularidade terão um bom desempenho para aplicações com grande

paralelismo de dados, contudo não são adequadas para processar tarefas de maior generalidade ou com palavras de dados maiores (compressão e compilação são exemplos). As arquiteturas de granularidade grossa, por sua vez, serão prejudicadas ao computar aplicações altamente concorrentes em nível de dados de pequeno tamanho, como filtragem de imagens.

A idéia de construir arquiteturas polimorfas é evitar que o hardware sofra penalidades de performance significativas, conforme as aplicações distanciam-se do ponto de granularidade ideal. Existe, no projeto, dois caminhos básicos a serem seguidos (SANKARALINGAM et al., 2003):

- Abordagem de síntese: Nesta abordagem tem-se diversos processadores elementares de granularidade fina. De acordo com a aplicação, parte desses processadores elementares podem combinar-se e formar um processador lógico de granularidade maior. Assim, as tarefas mais irregulares e com menor paralelismo seriam executados por uma região reconfigurável mais bem preparada.

- Abordagem de particionamento: O hardware implementa processadores de granularidade grossa que, de acordo com a tarefa, podem ser particionados em processadores menores e mais preparados para explorar o paralelismo de dados.

Um aspecto muito interessante sobre polimorfismo arquitetural é que ele vem a introduzir mais generalismo na computação reconfigurável. Obviamente, a arquitetura não conseguirá atingir o desempenho de um circuito específico para a tarefa, mas poderá atingir taxas de processamento próximas para diversas classes de problemas.

A arquitetura TRIPS (SANKARALINGAM et al., 2003) constitui um arquitetura voltada ao polimorfismo. Utiliza a abordagem do particionamento sendo composta por *cores* processadores de granularidade grossa, a fim de obter alta performance para aplicações com paralelismo em nível de instrução. Possui também capacidades de processar tarefas adequadas para hardware de granularidade menor, uma vez que os processadores subdividem-se em estruturas processadoras menores.

A figura 3.29 apresenta uma visão geral da arquitetura em três níveis: *chip*, *core* e nodo de execução. A arquitetura é formada por 4 *cores* processadores, os quais possuem um *array* de nodos de execução. Em cada nodo de execução encontra-se uma ULA, uma unidade de ponto flutuante, unidades de armazenamento e conexões especializadas de roteamento na entrada e na saída.

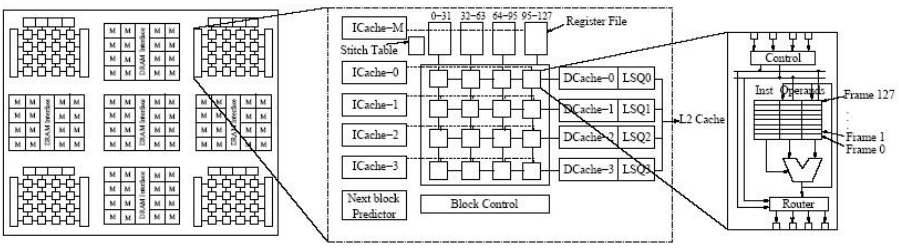

Figura 3.29: Arquitetura TRIPS

Os elementos de armazenamento do nodo podem guardar dois operandos e uma instrução. Quando os dados estão prontos e a instrução é válida o nodo executa a operação e o resultado pode ser encaminhado para a área de armazenamento de operandos de qualquer nodo do *array*.

Observando a Figura 3.29, é posível ver que TRIPS conta com um banco de *caches* de instrução (parte esquerda da figura), sendo um módulo de *cache* reponsável por uma linha. Existe ainda, uma *cache* de instruções adicional para realizar a busca de valores nos registradores e inserir diretamente no *array*. Ao lado do *array* existe uma série de *caches* de dados, que podem ser acessadas pelas ULAs através da rede de roteamento.

O autor acredita que cada vez mais surgirão implementações multigranulares nos meios acadêmico e industrial. Esses novos projetos aumentarão a generalidade da computação reconfigurável que estará presente de maneira crescente nos diferentes dispositivos de computação pessoal e científica.

4 DESENVOLVIMENTO DE UM PROCESSADOR DE IMAGENS RECONFIGURÁVEL

Este capítulo apresenta o desenvolvimento e evolução da arquitetura reconfigurável DRIP. A arquitetura seguiu um fluxo de construção e aperfeiçoamento amplo, começando como um projeto estático (processador NP9) até uma arquitetura flexível e reconfigurável em tempo de execução (DRIP-RTR).

4.1 Domínio da Aplicação

Conforme já mencionado, o DRIP foi projetado para executar algoritmos de processamento de imagem de baixo nível. Utilizando apenas duas funções (soma e máximo, detalhadas nas próximas seções) não possui, por exemplo, operadores de multiplicação. Dessa maneira, ele é adequado principalmente para filtragem morfológica e não para aplicações de compressão e problemas de nível mais alto. São exemplos de algoritmos de baixo nível:

- Convolução linear, não linear e filtros híbridos.
- Operações morfológicas binárias e em níveis de cinza (dilatação, erosão, detecção morfológica de contorno, gradiente).
- Operações geodésicas binárias e em nível de cinza.

Um conjunto completo de informações sobre esses diversos algoritmos e técnicas pode ser encontrado em (GONZALEZ; WOODS, 2002).

4.2 Processador NP9

Com o processador NP9 (ADÁRIO, 1997) surgiu a estrutura base de desenvolvimento do processador DRIP. O NP9 é um processador de vizinhança, ou seja, o pixel de saída é função de seu próprio valor e de seus vizinhos mais próximos na imagem de entrada. A organização do processador está baseada nos conceitos de programação funcional (BACKUS, 1978).

A idéia da programação funcional é combinar operações simples para criar operações mais complexas. Neste contexto, um programa pode ser visto como um grafo de fluxo de dados (LEITE; BARROS, 1994). O primeiro passo para a definição da arquitetura é, portanto, definir os nodos do grafo, ou seja, os processadores elementares da arquitetura.

4.2.1 Processador Elementar (PE)

O processador elementar é o bloco de programação básico do NP9/DRIP. Apesar de ser uma célula simples, a interconexão de processadores elementares na arquitetura proporciona a flexibilidade necessária para implementar uma grande variedade de algoritmos de processamento digital de imagens, como os mencionados na seção 4.1.

O processador elementar (figura 4.1) executa apenas duas operações básicas: MAX (máximo), representando a classe de operações não-lineares e ADD (soma) representando a classe de algoritmos lineares. Cada processador elementar recebe dois pixels de entrada (X1 e X2) de 8 bits. Para aumentar as capacidades lógicas do PE um peso inteiro (-1, 0 ou 1) é associado a cada uma das entradas (pesos respresentados por W1 e W2 na figura 4.1).

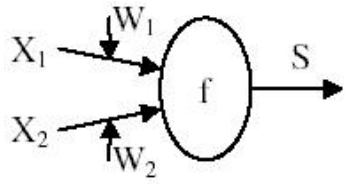

Figura 4.1: Representação lógica do processador elementar

Dessa maneira a saída (S) de um processador elementar depende da multiplicação dos pixels de entrada pelos pesos associados e da função configurada. A equação 4.1 ilustra essa relação:

$$S = f(\varepsilon, X1.W1, X2.W2) \tag{4.1}$$

Onde X1 e X2 representam os pixels de entrada, W1 e W2 os pesos associados e ε a função configurada. Dessa maneira, como exemplos, f(MAX,8.-1,9.0) vale 0 e f(ADD,8.-1,9.0) é igual a -8. Conforme será visto nas seções seguintes, a tradução em hardware do PE faz com que seja necessário para a sua configuração apenas 5 bits. Dois bits para cada multiplexador existente na aplicação dos pesos e um bit para a função.

4.2.2 Parte Operativa e Grafo de Fluxo de Dados

Para definir o grafo de fluxo de dados do processador foi considerada uma classe de filtros não lineares largamente utilizada em processamento de imagem de baixo nível. Esses filtros operam em uma vizinhança pequena e atuam de maneira uniforme na imagem, de modo que cada pixel é substituído por uma função dele e de seus vizinhos mais próximos. Esses filtros podem ser representados pela equação 4.2:

$$y = f(\sum_{i=1}^{n} a_i (b_i g(x_i))_{(i)}) \tag{4.2}$$

A tradução para hardware dessa estrutura foi baseada em um algoritmo de classificação. Esse algoritmo é o algoritmo de transposição par-ímpar (KNUTH, 1973), o qual representa um bom compromisso entre área e tempo de computação, além de oferecer uma construção altamente regular e paralela (figura 4.2).

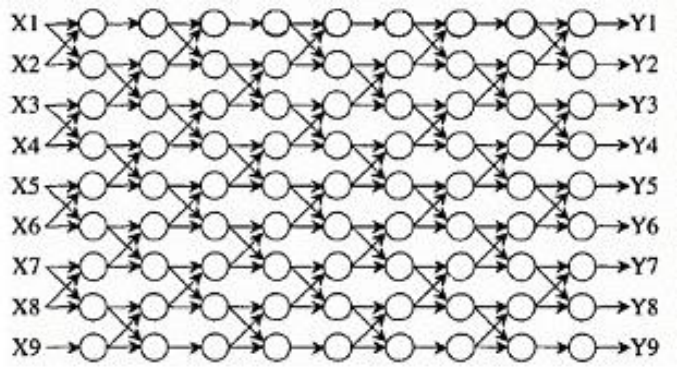

Figura 4.2: Estrutura lógica da parte operativa NP9/DRIP

Tabela 4.1: Configurações possíveis do PE

Configuração	**Função**
Add(0,0); Max(0,0)	0
Add(0,X); Add(X,0)	X
Add(-X,0); Add(0,-X)	-X
Add(X1,X2)	addition
Add(-X1,X2); Add(X1,-X2)	subtraction
Add(-X1,-X2)	-X1 - X2
Max(0,X2); Max(X1,0)	If X1(2)> 0 then X1(2) else 0
Max(0,-X2); Max(-X1,0)	If X < 0 then X else 0
Max(X1,X2)	Max(X1,X2)
Max(-X1,X2);Max(X1,-X2)	If X1(2) > X2(1) then X1(2)
Max(-X1,-X2)	-Min(X1,X2)

A parte operativa do NP9 (e do DRIP) reflete essa estrutura. A vizinhança utilizada é 3x3, resultando na matriz bidimensional de 81 processadores elementares.

4.3 Processador Reconfigurável DRIP

O NP9 possui uma parte operativa fixa para todos os algoritmos. Dessa maneira, muitas vezes alguns processadores elementares acabam por não serem utilizados no processamento, embora estejam mapeados e, portanto, consumindo recursos lógicos do dispositivo programável. Além disso, apesar de o PE ser uma célula funcionalmente simples, a matriz de PEs pode ser configurada para computar diversos algoritmos de processamento digital de imagens, resultando em uma grande flexibilidade do sistema.

Com base nos fatores descritos, foi desenvolvido o processador reconfigurável DRIP. Ao contrário do NP9 apenas os recursos necessários para a tarefa corrente são mapeados para o FPGA. Como resultado, tem-se uma ocupação de área muito mais eficiente e uma taxa de processamento muito mais alta.

Com 2 pesos inteiros para cada entrada e duas funções, existem, no máximo, 18 configurações possíveis para um PE. Entretanto, algumas delas executam exatamente a mesma função. As configurações ADD(0,0) e MAX(0,0), por exemplo, representam uma única funcionalidade. A tabela 4.1 mostra as configurações não redundantes.

4.3.1 Fluxo de Projeto

O fluxo de projeto do DRIP está baseado no que pode-se chamar de reconfigurabilidade arquitetural. Nesta metodologia de desenvolvimento uma arquitetura já existente, no caso o NP9, é decomposta em seus elementos básicos, resultando em uma biblioteca de componentes.

Esses componentes são instanciados de acordo com o algoritmo a ser implementado, produzindo as diferentes configurações que podem ser invocadas durante a execução.

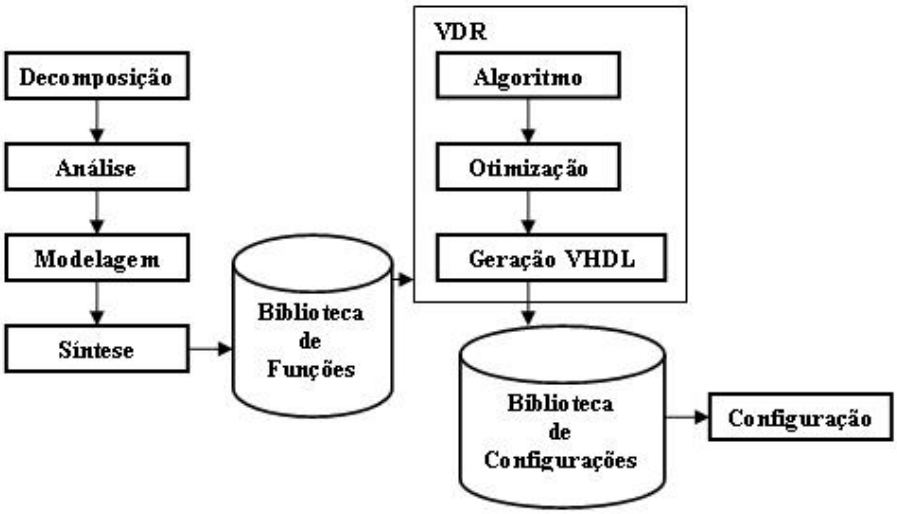

Figura 4.3: Fluxo de projeto DRIP

A figura 4.3 ilustra o fluxo. No processo de decomposição todas as possíveis configurações da arquitetura são identificadas. No caso do DRIP são as 18 configurações possíveis de um PE. O passo seguinte é a análise desses dados. O resultado da análise são as configurações não simétricas/redundantes apresentadas anteriormente na tabela 4.1, o que leva à definição da biblioteca de funções.

A biblioteca é composta por implementações de células, cada célula corresponde a uma das entradas da tabela 4.1. Esses processadores elementares não possuem capacidades de programação, executam uma única função e, por isso, são chamados de células super-especializadas. Vale ressaltar que a parte operativa do processador é um *pipeline* (ilustrado na figura 4.4 com os registradores intermediários) onde o conjunto de PEs está incluído.

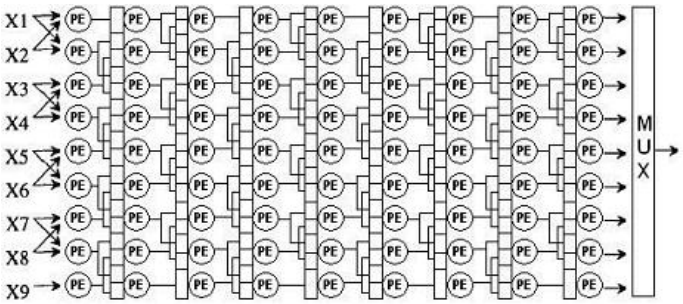

Figura 4.4: Pipeline DRIP

Na sequência do fluxo de projeto, os componentes da biblioteca são instanciados de acordo com o algoritmo e passam por um processo de otimização e geração de código

VHDL. Esse modelo VHDL representa um algoritmo de processamento digital de imagens pronto para ser mapeado para o FPGA. O processo de definição do algoritmo, otimização e geração VHDL são automatizados através de ferramentas auxiliares desenvolvidas especialmente para a arquitetura DRIP e descritas na seção 4.3.2.

Como pode ser observado, o fluxo descrito reflete um projeto estaticamente reconfigurável. Um novo algoritmo deve esperar o final da tarefa corrente para ser mapeado. Essa característica tende a incluir atrasos consideráveis, uma vez que o processo de reconfiguração dos FPGAs Altera APEX e similares ocorre na ordem de dezenas a centenas de milisegundos. Dessa forma, surge a necessidade de incluir mecanismos de reconfiguração dinâmica, o que torna necessário os estudos de modificação do processador elementar e da parte de controle apresentados a seguir.

4.3.2 Ferramenta VDR

Uma questão importante existente no desenvolvimento de arquiteturas reconfiguráveis é a necessidade constante de criação de ferramentas de CAD específicas. As diferentes arquiteturas possuem características distintas que precisam ser exploradas, o caminho seguido pelos diferentes grupos de pesquisa resulta na implementação de um conjunto de softwares com duas finalidades básicas: agilizar/facilitar o processo de síntese, mapeamento e desenvolvimento de aplicações e otimização do desempenho do sistema reconfigurável.

Dentro desse contexto e levando-se em consideração o fluxo de projeto do DRIP reconfigurável descrito na seção anterior foi desenvolvida a ferramenta VDR (*Visual interface for Dynamic Reconfiguration*). Esta ferramenta compreende três estágios do fluxo de projeto apresentado na figura 4.3: definição do algoritmo, otimização e geração de modelo VHDL.

Um software de projeto deve oferecer ao desenvolvedor o máximo de facilidades possível. Assim, a interface da ferramenta (figura 4.5) apresenta uma representação direta dos 81 processadores elementares que compõem o *pipeline* do DRIP. Interagindo sobre essa interface o usuário/projetista pode configurar totalmente a matriz de PEs definindo visualmente um algoritmo de processamento de imagens.

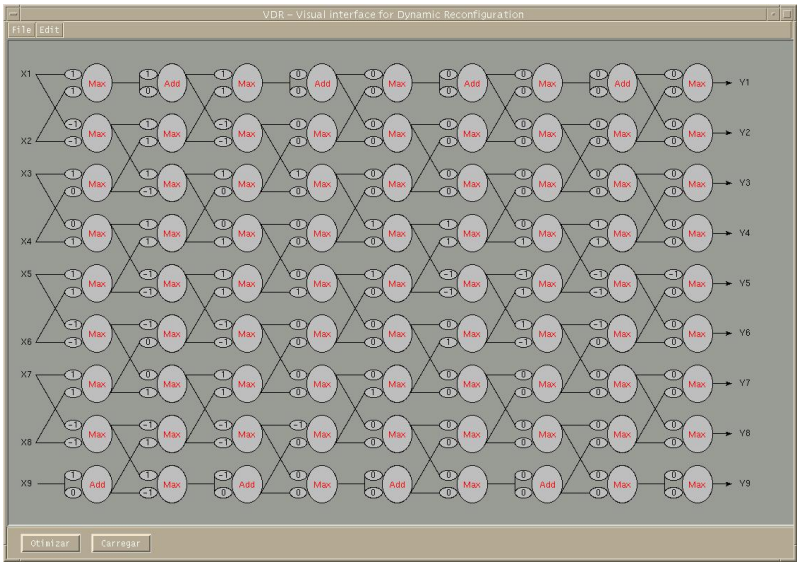

Figura 4.5: Interface VDR

4.3.2.1 Otimizações

Uma vez definido o algoritmo, o módulo responsável pelas otimizações entra em ação. Esse módulo contém rotinas implementadas em C que mapeiam o algorimo para um grafo orientado acíclico e realizam um conjunto de eliminações e transformações sobre a matriz de PEs. O Objetivo é encontrar uma representação do algoritmo que, embora equivalente, possibilite uma execução mais rápida e com menor utilização dos recursos do FPGA.

Existem três diferentes tipos de otimizações: otimização de avanço, otimização de retorno e eliminação de nodos nulos. A figura 4.6 ilustra essas três classes.

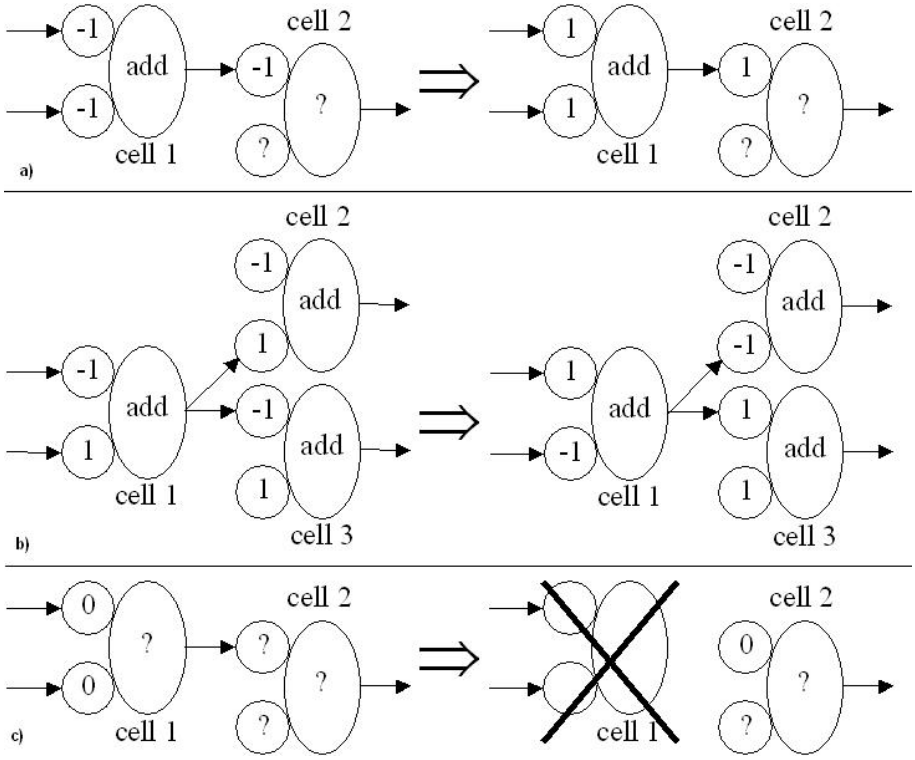

Figura 4.6: Otimizações

A primeira linha (a) da figura 4.6 mostra uma transformação de avanço. Essa otimização possui como objetivo substituir uma célula complexa por outra mais simples, em especial com relação à eliminação de circuitos de geração de complemento de 2. A idéia utilizada é que se um processador elementar tem seus pesos de entrada em -1, podemos mudar ambos para 1 desde que seja alterado também o peso de entrada da célula seguinte do *pipeline*. O conceito, portanto, é similar a uma multiplicação por -1 nos dois lados de uma igualdade matemática.

A segunda linha ((b) na figura 4.6) demonstra uma transformação de retorno. O algoritmo é exatamente o mesmo da otimização de avanço a única diferença é que o *pipeline* é analisado da direita para a esquerda.

Finalmente, tem-se a eliminação de nodos nulos ((c) na figura 4.6). Neste caso, a ferramenta procura identificar PEs desnecessários e evitar seu mapeamento para o dispositivo programável. Nodos que, por exemplo, possuem ambos os pesos de entrada em zero irão apenas propagar o valor 0 e, portanto, não precisam estar no circuito consumindo recursos lógicos.

A ferramenta, após a otimização, acessa as células existentes na biblioteca de funções básicas e cria um modelo VHDL otimizado do algoritmo, o qual pode ser diretamente

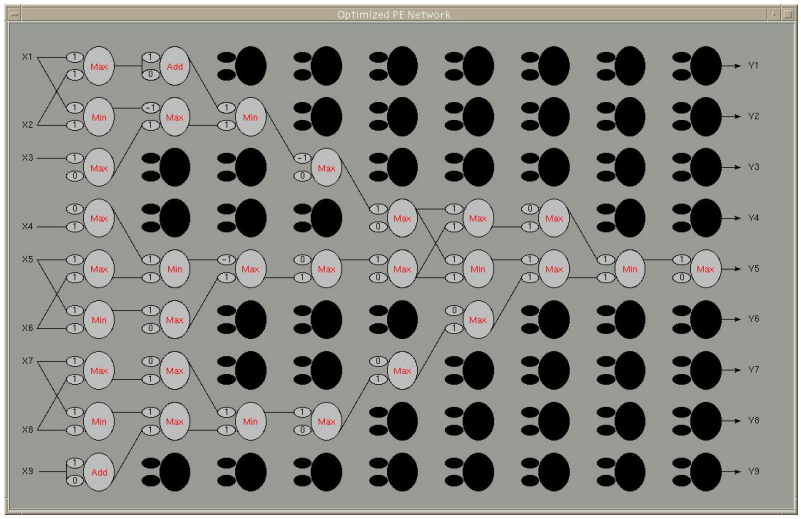

Figura 4.7: Algoritmo filtro de mediana separável otimizado

configurado no DRIP.

Para o algoritmo de filtro de mediana separável, por exemplo, a otimização do tipo eliminação de nulos removeu cerca de 60% dos PEs (em cor preta na figura 4.7). Sobre os PEs remanescentes houve ainda um ganho de área de aproximadamente 20% devido às transformações realizadas.

Dessa forma, o sistema de CAD permite que o projetista, automaticamente, possua uma representação VHDL sintetizável de um determinado algoritmo de procesamento digital de imagem. Repetindo-se esse processo algumas vezes, pode-se construir um ampla biblioteca de configurações para o DRIP.

4.4 DRIP-RTR

Reconfiguração dinâmica permite ao desenvolvedor explorar ao máximo a flexibilidade das arquiteturas reconfiguráveis e evitar o *overhead* existente quando ocorre a mudança de tarefa na abordagem estaticamente reconfigurável.

No contexto do DRIP, para permitir reconfiguração em tempo de execução foi necessário um estudo preliminar da estrutura dos processsadores elementares da parte operativa. A modificação necessária consiste na transformação do processador elementar em uma estrutura de granularidade maior, aumentando o nível de programabilidade da célula.

No entanto, para determinar a melhor maneira de realizar modificações no PE é necessário uma metodologia. E o caminho seguido neste projeto é o que podemos chamar de análise de similaridades.

4.4.1 Análise de Similaridades: Definindo PEs Programáveis

Considerando-se tanto FPGAs parcialmente programáveis quanto FPGAs comuns, é intuitivo que para minimizar atrasos na substituição da tarefa corrrente é melhor a ocorrência de poucas modificações entre as duas configurações sucessivas. Dessa maneira, é importante identificar uma metodologia que ajude a definir o melhor conjunto de transformações existentes.

Um caso interessante para análise é a arquitetura RRANN (ELDREDGE; HUTCHINGS, 1994). O RRANN implementa um algoritmo de treinamento de redes neurais

particionado em três estágios. Ele utiliza FPGAs parcialmente reconfiguráveis, assim, grande parte do esforço de projeto está na redução da quantidade de hardware a ser reconfigurado. Nessa análise, três tipos diferentes de blocos ou subcircuitos são definidos:

- Blocos estáticos: Não se alteram entre configurações sucessivas.
- Blocos semi-estáticos: Possuem poucas diferenças estruturais entre uma configuração e outra.
- Blocos Dinâmicos: Modificam-se completamente ou possuem diversas diferenças entre uma configuração e outra.

O objetivo principal da análise de similaridades é preservar blocos estáticos e identificar os níveis de semelhança entre blocos semi-estáticos. Com esse estudo, é possível obter informações importantes sobre como realizar a fusão de funcionalidades dos PEs em células programáveis maiores.

Uma vez que a estrutura do DRIP admite um grande número de algoritmos de processamento digital de imagens, foram escolhidos 5 algoritmos representativos para realização da análise: detector morfológico de contornos, filtro de mediana separável, mediana, dilatação e erosão.

Conforme pode ser visto na figura 4.8, no projeto DRIP a análise de similaridades ocorre após as fases de decomposição e análise inicial. Através da comparação da configuração de um dado processador elementar em um algoritmo com o correspondente PE em um segundo algoritmo, foi possível perceber a existência de uma similaridade significativa entre configurações do *pipeline*, inclusive com um grande número de coincidências exatas, a isso podemos chamar de similaridade em nível de algoritmo.

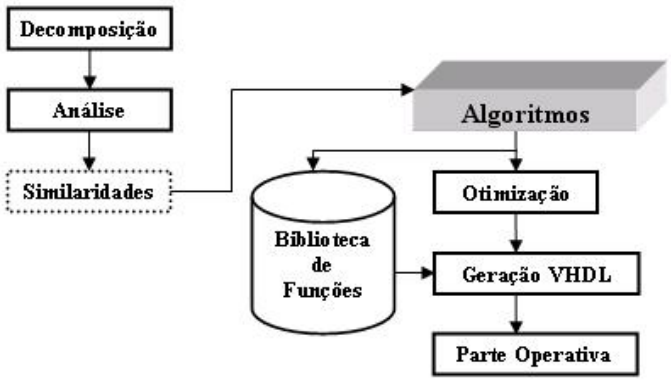

Figura 4.8: Novo fluxo de projeto

Contudo, a análise não fica restrita a esse domínio, analisando-se a estrutura lógica dos PEs percebe-se que muitos possuem poucas diferenças, podendo ser classificados como blocos semi-estáticos (ex: Max(1,1) e Max(1,0)). Esses PEs possuem uma significativa similaridade em nível de célula.

Devido aos fatores mencionados, os processadores elementares podem ser remodelados como células de granularidade maior. Dessa forma, torna-se possível aumentar o nível de programabilidade do *pipeline* trazendo a possibilidade de ativar um novo algoritmo em tempo de execução e, assim, evitar a latência de reconfiguração.

O resultado da análise de similaridade pode ser visto como um mapa de transformações. É, portanto, preciso remodelar os PEs através da fusão de funcionalidades,

tornando-os sensíveis a sinais de controle. As modelagens descritas a seguir foram realizadas em VHDL. As linguagens de descrição de hardware, devido a sua capacidade de abstração, oferecem diversas possibilidades de descrição. Para encontrar a melhor relação área-performance duas bibliotecas de células diferentes para os PEs foram desenvolvidas.

4.4.2 Modelagem Comportamental

Este nível de modelagem explora a similaridade em nível de algoritmo. O estudo realizado com os 5 algoritmos mostra que um PE de maior granularidade, em geral, pode ser formado agrupando-se 3 células super-especializadas DRIP. A idéia pode ser resumida pelo pseudo-código abaixo:

```
INPUT(X1,X2,CLOCK,CONTROL)
    OUTPUT(Y)
    CASE CONTROL
    Sinal0: Executa código comportamental da célula 0
    Sinal1: Executa código comportamental da célula 1
    Sinal2: Executa código comportamental da célula 2
    END CASE
```

O controle de qual função deve ser executada pela célula é feita pelo sinal CONTROL. Essa biblioteca quando sintetizada apresentou custo médio de 53 elementos lógicos Altera.

4.4.3 Modelagem Estrutural

A segunda abordagem segue um modelo estrutural de definição dos processdores elementares. Pode-se dizer que esta biblioteca explora a similaridade em nível de algoritmo e em nível de célula. Cada PE pode ser visto como um sistema baseado em dois submódulos: um módulo quantificador, responsável pela aplicação do peso (-1,0 ou 1) ao pixel de entrada e um módulo qualificador cuja obrigação é executar a função de máximo ou soma.

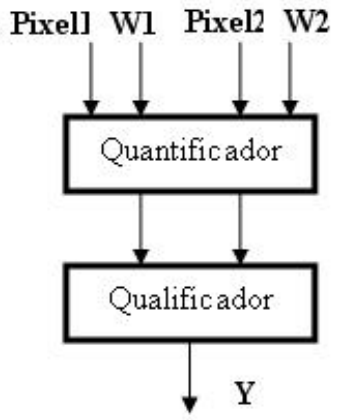

Figura 4.9: Modelo quantificador/qualificador

Como exemplo da similaridade em nível de célula, podemos tomar as funções MAX(1,1) e MAX(1,0). Em ambas, o pixel associado à primeira entrada deve ser multiplicado por 1 (ou seja, é o próprio valor de entrada). A diferença ocorre apenas na segunda

entrada (multiplicação por 1 e 0, respectivamente).

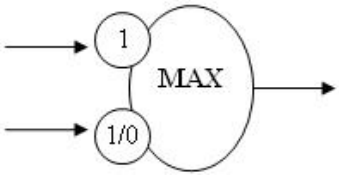

Figura 4.10: Visão lógica do PE resultante

Dessa forma (conforme ilustrado na figura 4.10), existe apenas uma pequena diferença no módulo quantificador, o restante permanece exatamente o mesmo. É possível, então, na modelagem da célula instanciar apenas os subcomponentes necessários e relevantes.

No exemplo, para a primeira entrada basta transferir diretamente o valor do pixel para o módulo qualificador. Para a segunda entrada, os pesos possíveis são 1 e 0, realiza-se, nesta caso, uma multiplexação entre o valor da entrada e o valor 0.

O grande mérito da similaridade em nível de célula é que cada PE está customizado para suportar os requisitos mínimos de sua funcionalidade. Mesmo que o modelo genérico e mais completo possível de um procesador elementar suporte, por exemplo, a multiplicação por -1, o circuito de geração de complemento de 2 apenas será instanciado quando necessário.

Esta biblioteca apresenta um custo médio menor, de cerca de 43 elementos lógicos e também um reuso maior de modelos em relação à biblioteca comportamental. Para suportar os 5 algoritmos, foi necessário descrever 24 células na primeira biblioteca. Para esta segunda, bastou a criação de 12 modelos VHDL.

A explicação passa pela similaridade em nível de célula existente no modelo quantificador/qualificador. Considere como exemplo três funções: MAX(1,0), MAX(0,-1) e MAX(1,-1). Conforme mencionado, na abordagem comportamental um PE é formado pela unificação em um mesmo modelo das funcionalidades das células superespecializadas do DRIP. Sendo assim, para formar a célula correspondente a essas três funções na abordagem comportamental agruparíamos o código VHDL das três células correspondentes. Entretanto, suponha que é necessária a criação de um novo processador elementar que suporte apenas as duas primeiras funções. Nesse caso, para economia de recursos, não devemos incluir o código referente à terceira célula. O resultado, portanto, seria um novo modelo, resultando na criação de duas células para a biblioteca, uma capaz de executar as três funções da figura 4.11-a e a outra capaz de executar as duas da figura 4.11-b.

Para a abordagem estrutural, as duas funcionalidades são representadas por um único modelo VHDL. uma vez que a parte qualificadora é idêntica e os pesos a serem aplicados são iguais em ambas as situações (figura 4.11-c).

4.4.4 O Novo Modelo de Processador Elementar

Analisando-se os resultados obtidos em relação aos estilos de modelagem o modelo quantificador/qualificador de estruturação do PE de granularidade maior oferece grandes vantagens, uma vez que explora a similaridade em nível de algoritmo, de célula e apresenta maior reuso e melhores resultados de síntese.

Um fator importante é que a expressão aumento de granularidade, normalmente, está associada ao aumento do tamanho das palavras processadas pelos operadores (de 8 para 16 bits por exemplo). Entretanto, neste trabalho o aumento da granularidade refere-se a um

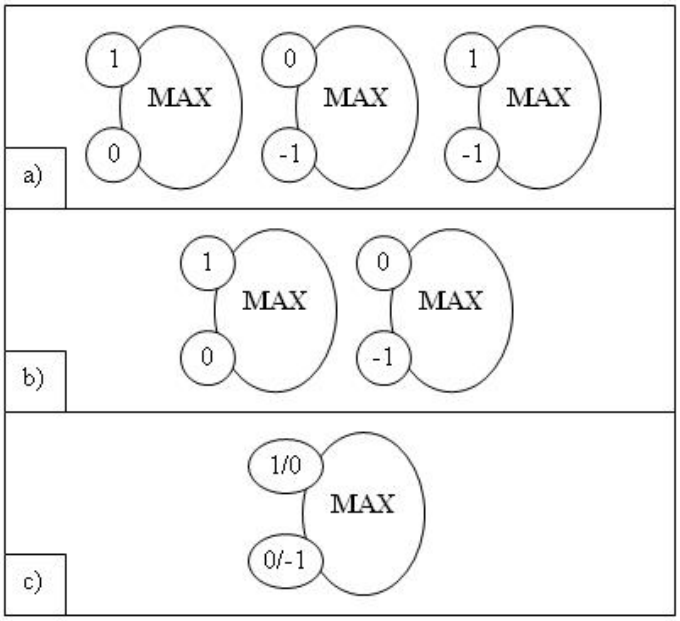

Figura 4.11: Similaridade em nível de célula e reuso

maior número de recursos lógicos em um único PE, permitindo que a sua funcionalidade seja alterada de acordo com bits de controle.

O novo processador elementar permite que o hardware tenha suporte para a alteração em tempo de execução do algoritmo sendo processado. Além disso, é possível realizar (via ferramenta JVDR desenvolvida) a geração automática do processador com a quantidade mínima de recursos necessários para a computação de um dado conjunto de algoritmos.

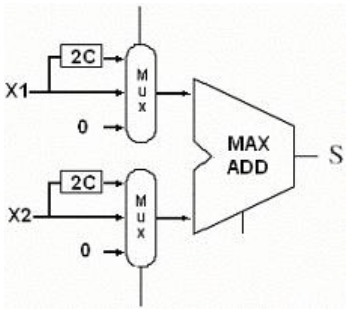

Figura 4.12: Processador elementar geral

A figura 4.12 mostra um processador elementar geral. A parte quantificadora e qualificadora são capazes de processar qualquer função configurável em um PE.

Contudo, na maior parte dos casos, as funções a serem computadas pelo processador elementar são bem mais restritas. A figura 4.13 apresenta um PE que suporta MAX(X1,X2) e MAX(X1,0). Neste caso, a ferramenta de CAD sucessora da VDR e descrita na seção 4.4.9, identificou que o multiplexador da primeira entrada e o circuito de geração de complemento de 2 não eram necessários, além disso o multiplexador da segunda entrada pode ser 2:1. O PE resultante é muito menor que o modelo geral, economizando recursos de hardware e criando condições para um maior desempenho.

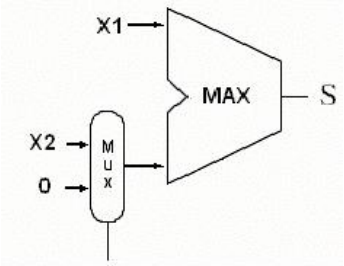

Figura 4.13: Processador elementar otimizado

4.4.5 Visão Geral da Arquitetura

Um processador de vizinhança é similar a um processador matricial. Ele processa uma imagem de entrada gerando outra onde cada pixel de saída é função direta do pixel de entrada e de seus vizinhos mais próximos. O DRIP utiliza uma vizinhança de 3x3, varrendo a imagem linha por linha, entretanto, a estrutura pode ser adaptada para qualquer tamanho de janela. A figura 4.14 mostra uma visão completa da arquitetura desenvolvida.

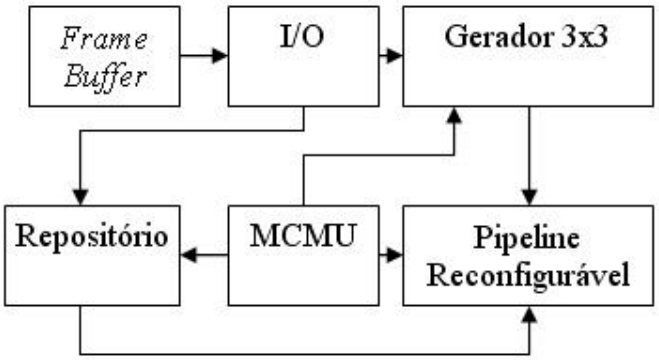

Figura 4.14: Visão geral da arquitetura

O processador de I/O é responsável pela comunicação com o *Frame Buffer* externo que armazena a imagem. Além disso, ele é capaz de receber novas configurações e armazená-las na na área de contextos. A MCMU gerencia o processo de reconfiguração. O processador de I/O contém um módulo especial chamado de gerador de vizinhança descrito em maiores detalhes na próxima seção.

4.4.6 O Gerador de Vizinhança

O gerador fornece a vizinhança 3x3 a ser computada. O gerador de vizinhança deve ser rápido para não se tornar um gargalo na arquitetura DRIP. O caminho seguido para a obtenção de um bom desempenho foi utilizar uma estratégia de bufferização *on-chip*, ou seja, ao invés de realizar sucessivas leituras no *frame buffer* externo, armazenar a parte da imagem a ser computada em uma área próxima ao *pipeline*. Um primeiro detalhe a ser observado é que a matriz de vizinhança (3x3) a ser gerada possui pixels em 3 diferentes linhas da imagem, como ilustra a figura 4.15.

Torna-se importante, portanto, analisar o problema e identificar, por exemplo, qual a quantidade de pixels que deve ser armazenada no *chip* tendo em vista os recursos limitados de memória e também como estruturar o mecanismo de geração de vizinhança.

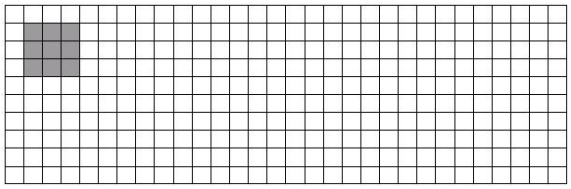

Figura 4.15: Janela de vizinhança 3x3

Os estudos realizados levaram a importante consideração de que, em qualquer situação de processamento, o número máximo de pixels necessários residentes no buffer interno é:

$$TotPixels = 2n + 3 \tag{4.3}$$

Onde n é a quantidade de pixels de uma linha. Para exemplificar de maneira intuitiva a equação 4.3 considere que cada letra no *frame* apresentado na figura 4.16 corresponde a um pixel.

a	b	c	d	e	f
g	h	i	j	k	l
m	n	o	p	q	r
s	t	u	v	w	x

Figura 4.16: Exemplo de pixels em uma imagem

Vamos supor agora o movimento de uma janela 3x3 realizando a convolução com essa imagem. Temos a seguinte sequência de pixels que necessitam estar bufferizados a cada instante:

to = (a,b,c,d,e,f,g,h)
t1 = (a,b,c,d,e,f,g,h,i)
t2 = (a,b,c,d,e,f,g,h,i,j)
t3 = (a,b,c,d,e,f,g,h,i,j,k)
t4 = (a,b,c,d,e,f,g,h,i,j,k,l)
t5 = (a,b,c,d,e,f,g,h,i,j,k,l,m,n)
t6 = (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o)
t7 = (b,c,d,e,f,g,h,i,j,k,l,m,n,o,p)
t8 = (c,d,e,f,g,h,i,j,k,l,m,n,o,p,q)

Como pode ser observado, a partir do momento t6 um pixel pode ser descartado, enquanto que um novo pixel é armazenado. Como a imagem do exemplo possui 6 pixels por linha, precisamos guardar 15 pixels (equação 4.3).

A arquitetura projetada (figura 4.17) para o gerador de vizinhança, utiliza duas FIFOs e nove registradores, todas essas estuturas foram modeladas em VHDL. Note que o tamanho das duas FIFOs está relacionada ao tamanho (N) da linha.

Para mostrar de maneira objetiva o funcionamento do gerador considere o seguinte exemplo. A imagem é a mesma da figura anterior com a diferença de que o valor zero está distribuído por todo o contorno da imagem. Em situações nas quais a janela se encontra em uma das extremidades da imagem alguns pixels da janela acabam por ficar além dos limites. Nessa situação o valor zero é utilizado. Na figura 4.18 a janela está posicionada

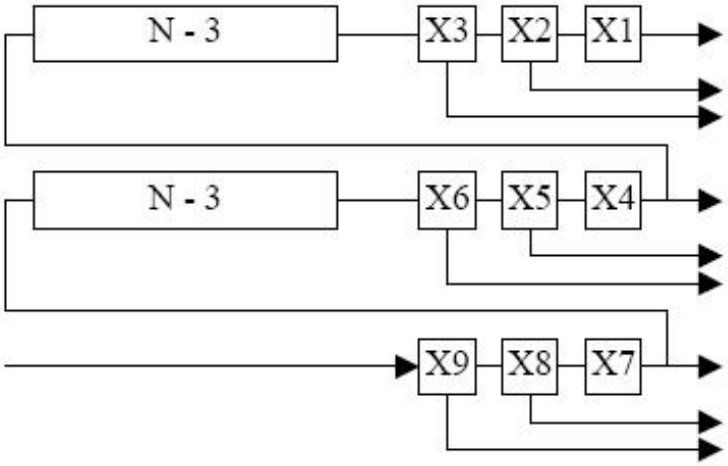

Figura 4.17: Arquitetura do gerador de vizinhança

de forma a computar o novo valor do pixel ”a“. A situação dos pixels na arquitetura em nosso exemplo pode ser vista na figura 4.19.

0	0	0	0	0	0	0	0
0	**a**	**b**	c	d	e	f	0
0	**g**	**h**	i	j	k	l	0
0	m	n	o	p	q	r	0
0	s	t	u	v	w	x	0
0	0	0	0	0	0	0	0

Figura 4.18: Exemplo de funcionamento do gerador

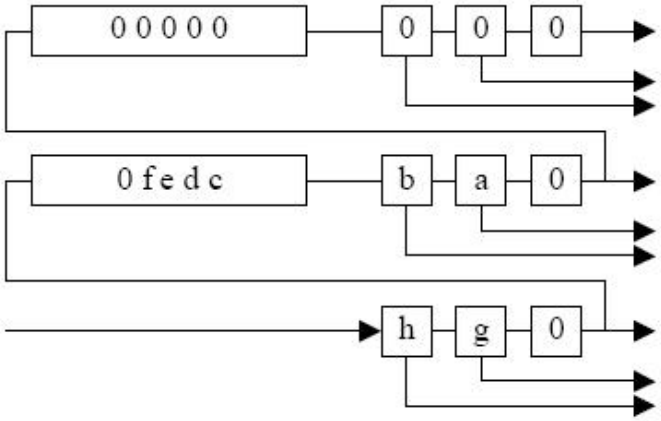

Figura 4.19: Pixels na arquitetura do gerador

A estrutura foi modelada em VHDL de duas maneiras diferentes. Na primeira, as FIFOs foram construídas como registradores de deslocamento parametrizáveis (*ShiftReg*) de tamanho N-3, enquanto que os locais de armazenamento dos pixels de saída como registradores simples de 8 bits.

Na segunda versão, foram usados os mesmos registradores para os pixels da janela de processamento, contudo, para as FIFOs empregou-se o componente LPM-FIFO da biblioteca de modelos da Altera, os resultados obtidos são mostrados na tabela 4.2.

Tabela 4.2: Implementações do gerador de vizinhança

Implementação	Elementos Lógicos	Bits de memória	Frequência (MHZ)
ShiftReg	10827	0	149,32
LPM-FIFO	120	18432 bits	239,12

Em ambos os casos o resultado obtido garante que o circuito gerador possui uma taxa de processamento suficiente para não limitar a atuação do *pipeline*. Entretanto, o sintetizador Quartus da Altera aproveita os recursos do FPGA de maneira mais adequada quando da utilização do LPM-FIFO. No primeiro caso, os registradores são construídos a partir da descrição de flip-flops sensíveis à borda positiva, o software de síntese aloca os elementos de armazenamento existente dentro das células lógicas para a construção desses *buffers*. Entretanto, quando ele reconhece o LPM, o mapeamento do circuito faz uso das estruturas de memória especializadas.

As ações e decisões realizadas pelo software que posiciona, aloca e mapeia o circuito para o hardware traz sempre impactos diretos e significativos nos resultados. Em (BOMAR, 2002) é apresentada a implementação de estratégias de controle microprogramadas adequadas para FPGAs. O trabalho, de maneira não surpreendente, procura explorar as estruturas especiais do dispositivo programável alvo e assim, obter resultados superiores de desempenho.

4.4.7 Repositório de Configurações

O repositório ou banco de configurações corresponde à região do DRIP-RTR na qual os diferentes contextos são armazenados. Nessa área, a unidade de gerenciamento de contextos pode escrever e recuperar a configuração dos diversos algoritmos.

Os contextos, portanto, ficam totalmente residentes *on-chip*. Esse tipo de armazenamento é viável em virtude de o DRIP-RTR necessitar de apenas 405 bits para uma configuração completa. Esses 405 bits estão divididos em 9 porções de 45 bits, cada uma dessas partes representando a configuração de uma coluna indivídual.

Essa separação entre as colunas permite que duas estratégias diferentes de configuração sejam aplicadas: reconfiguração estilo *pipeline* e modo paralelo. Ambas descritas em maior detalhe na seção seguinte.

O funcionamento e descrição em linguagem VHDL do repositório, no entanto, é completamente independente da maneira utilizada pela unidade de reconfiguração. A base de operação é o fornecimento de um número de contexto, de acordo com o número, um dos 8 bancos existentes será selecionado para reconfigurar a área ativa do *pipeline* de processamento.

Para configurar totalmente um processador elementar são necessários 5 bits. A primeira abordagem seguida para descrever as unidades de armazenamento do repositório parte de um elemento básico que é um registrador de 5 bits sensível à borda. Esse registrador é instanciado de maneira a formar os diferentes *buffers* de armazenamento, os quais podem ser vistos como registradores de 45 bits. Para cada coluna do *pipeline* considerando-se uma capacidade total de 8 contextos, são necessários, naturalmente, 8 *buffers* de 45 bits.

A saída de cada um dos registradores de 5 bits é entrada para um multiplexador 8:1, o qual define o valor requisitado de acordo com o número de contexto fornecido. A figura 4.20 ilustra essa estrutura, no entanto, apenas um multiplexador foi representado no desenho para fins de simplicidade, na implementação real existe 1 multiplexador para

cada conjunto de registradores de 5 bits (os quadrados na figura).

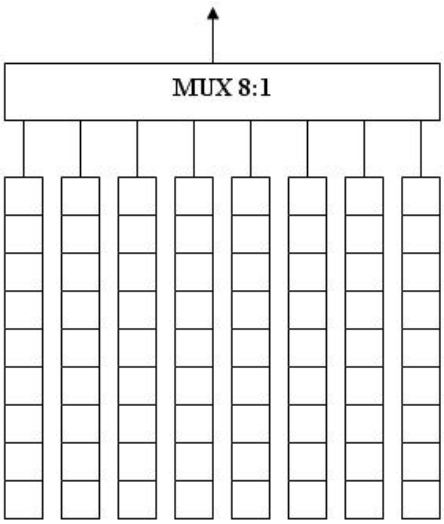

Figura 4.20: Primeira abordagem de implementação do repositório

O resultado da síntese de 8 *buffers* de contexto nesta implementação resultou na utilização de 119 elementos lógicos do FPGA Altera. Os 8 *buffers* mencionados (como na figura 4.20) são suficientes para guardar a configuração de uma única coluna do *pipeline*, de maneira que a síntese para as 9 colunas gera uma ocupação de área bastante alta. Esse comportamento ocorre porque o sintetizador da ferramenta Quartus não utilizou as estruturas dedicadas para memória existentes no dispositivo, ao invés disso foram alocados os flip-flops dos elementos lógicos.

A solução para obtenção de melhores resultados passa a ser, novamente, a exploração das primitivas da biblioteca de módulos parametrizáveis da Altera. Assim, o componente de memória RAM lpm-ram-dq foi adaptado para ser uma memória de 45 bits, representando uma coluna inteira. A criação de 8 *buffers* para cada coluna, formando os 8 contextos resultou na alocação de 232 elementos lógicos (inclui-se nesse número também a lógica para os multiplexadores) no total além de 20.160 bits de memória dedicados. Essa estrutura possui condições de operar com um ciclo de relógio máximo de 263,71 MHz.

De acordo com esses números, torna-se clara a vantagem obtida explorando-se o segundo estilo de desenvolvimento. Os 119 elementos lógicos para formar os 8 contextos de uma única coluna na primeira situação equivalem a cerca de 50% de todos os elementos lógicos necessários para criação dos 8 contextos para todas as 9 colunas na segunda implementação.

4.4.8 Mecanismo de Controle e Reconfiguração

O processo de reconfiguração do DRIP-RTR é comandado por um módulo reconfigurador existente dentro do próprio FPGA. Entretanto, admite-se que o comando de troca de algoritmo venha de uma fonte externa quando o DRIP está acoplado a um sistema hospedeiro.

Foi desenvolvido um sistema de controle interno que realiza o sequenciamento de configurações a partir de uma memória de programa. A figura 4.21 apresenta em maior profundidade a arquitetura DRIP-RTR mostrando as estruturas de controle

O principal responsável pelo sequenciamento das configurações e controle é a unidade de gerenciamento de múltiplos contextos (MCMU, *Multicontext Management Unit*). O

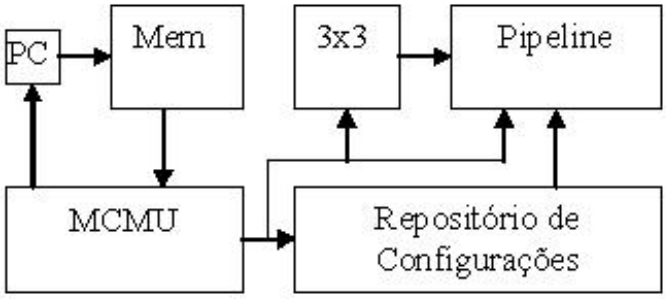

Figura 4.21: Visão do DRIP-RTR detalhando estruturas de controle

DRIP-RTR possui nessa implementação um conjunto de instruções reduzido porém suficiente para exercitar a execução de diversos algoritmos de processamento digital de imagens, através de carga e descarga de contextos na área ativa do *pipeline*.

Tendo como base o modelo de reconfiguração de Shirazi apresentado na seção 3.7.3.1, o carregador de configurações é a própria MCMU, a qual busca os contextos no repositório quando sinalizada pelo monitor. O monitor é também uma parte da lógica da MCMU e cabe a ele manter o status de quantos quadros foram processados e quantos ainda restam até que o próximo algoritmo seja configurado.

O Reconfigurador do DRIP-RTR foi construído com um conjunto de apenas três intruções. No entanto, elas são suficientes para realizar a troca do algoritmo sendo executado por outro existente no repositório de configurações e também para proporcionar a especificação de uma sequência de algoritmos a serem executados. As três instruções são:

- DEFLC [L],[C]: Esta instrução é na verdade uma facilidade para o programador e é avaliada em tempo de compilação. Através dela, é possível especificar o número de linhas [L] e colunas [C] nas imagens a serem processadas. Essa informação é necessária, pois com ela o gerador automático da arquitetura define o tamanho de estruturas como os *buffers* do gerador de vizinhança. Essa informação também é fundamental para o sistema de controle, pois sabendo o tamanho da imagem é possível determinar quando o processamento de um quadro foi finalizado.

- JMP [END]: A intrução de desvio incondicional é usada para construir LOOPs no qual diferentes algoritmos são executados sucessivamente.

- LDCTX [X], [Y]: A instrução de carga de contexto requisita que o contexto número [X] do repositório seja carregado e que mantenha-se na área de processamento por pelo menos [Y] quadros.

O fluxograma da figura 4.22, mostra a sequência de ações tomadas pelo controle de reconfiguração para as instruções de JMP e LDCTX.

Na instrução JMP, o contador de programa (PC) recebe o endereço da próxima instrução a ser buscada na memória. Na instrução LDCTX, o contador de *frames* é ajustado para o número de quadros para o qual o algoritmo atual deve ficar ativo. O modo de execução permanece até que o monitor, implementado como lógica da MCMU, detecte que o algoritmo processou o último quadro para o qual estava escalonado. A partir desse momento, o contador de programa é incrementado e a próxima instrução na memória é lida.

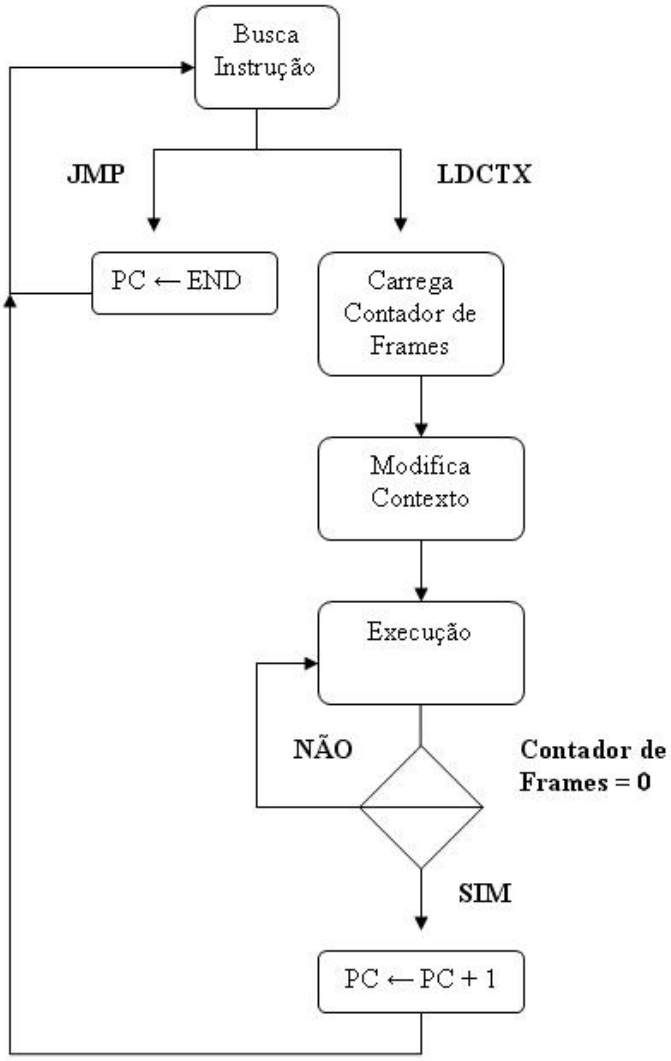

Figura 4.22: Fluxograma de execução das instruções

Conforme mencionado na seção 4.4.7 existem dois modos para executar a reconfiguração. O primeiro pode-se chamar de reconfiguração estilo *pipeline*. Nesta modalidade, de acordo com o controle da MCMU, primeiro muda-se a configuração da coluna 1, no ciclo de relógio seguinte altera-se a coluna 2 e assim sucessivamente.

A outra maneira, a qual podemos chamar de reconfiguração em modo paralelo realiza a mudança de todas as colunas da parte operativa de uma só vez.

4.4.9 Ferramenta JVDR

A Ferramenta JVDR é a sucessora da ferramenta de auxílio ao projeto da versão estaticamente reconfigurável do DRIP. Ela está fortemente ligada ao fluxo de projeto do DRIP-RTR e vem a introduzir uma série de facilidades que visam o desenvolvimento e exploração do espaço de projeto de maneira mais rápida e eficiente.

A principal função da ferramenta é gerar automaticamente um modelo VHDL do DRIP que contemple 1 ou diversos algoritmos de processamento digital de imagens. A figura 4.23 apresenta a interface da JVDR.

A interface apresenta os 81 processadores elementares. Clicando sobre eles é possível customizar os valores dos pesos de entrada e a função a ser computada. É possível

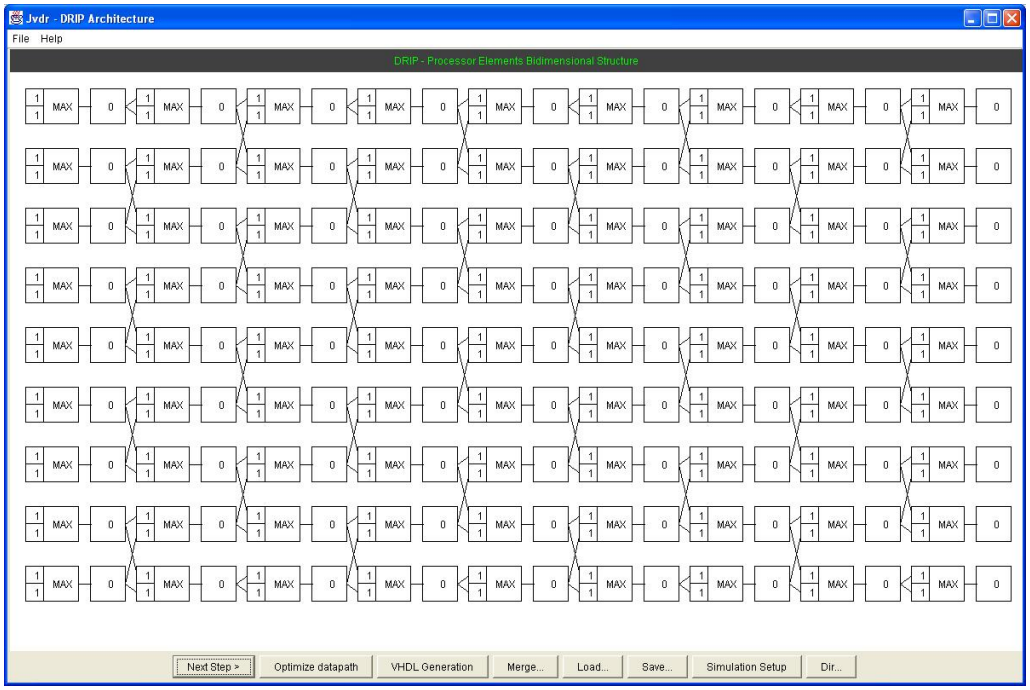

Figura 4.23: Interface da ferramenta JVDR

perceber que ao contrário da versão anterior da ferramenta, na JVDR estão representados os registradores intermediários do pipeline. Esses registradores são utilizados juntamante com a função de simulação existente na ferramenta.

A simulação permite que a validação funcional de um algoritmo ocorra internamente à JVDR. Assim, uma vez que a configuração dos PEs tenha sido definida é possível especificar um arquivo com os pixels a serem processados. Esse arquivo segue o formato dos arquivos de memória utilizados na simulação VHDL pelos componentes da biblioteca de módulos parametrizáveis da Altera (arquivo MIF). Em modo de simulação a JVDR possui a capacidade de exibir passo a passo o que está acontecendo, os valores dos pixels a cada ciclo de relógio podem ser vistos nos registradores intermediários do pipeline (figura 4.24).

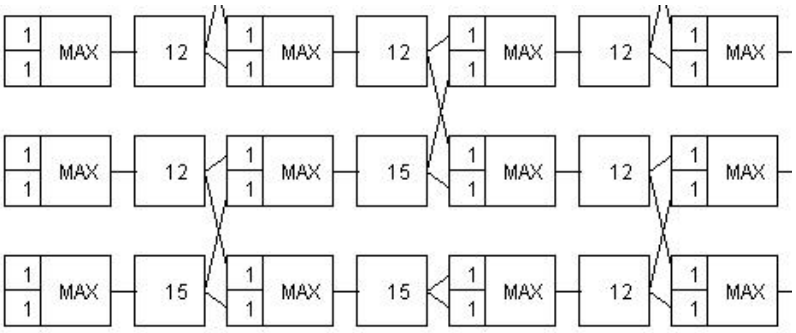

Figura 4.24: Função JVDR de simulação: os valores dos pixels em cada PE são mostrados na interface.

A JVDR conta com uma reimplementação agora em Java de todas as otimizações apresentadas na seção 4.3.2.1. De modo semelhante à VDR o resultado de qualquer otimização é apresentado graficamente (figura 4.25).

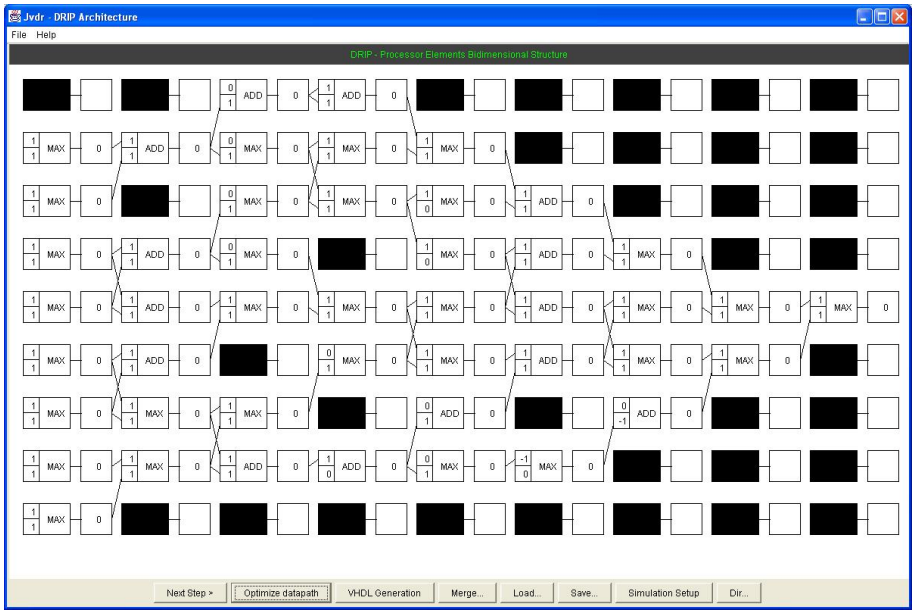

Figura 4.25: Interface JVDR apresentando algoritmo otimizado

Após a otimização o último passo para a geração de um modelo VHDL é instanciar os componentes da biblioteca de funções e formar o *pipeline*. Entretanto, no DRIP estaticamente reconfigurável tínhamos 11 células na biblioteca de funções. Para o DRIP reconfigurável em tempo de execução as possibilidades são maiores, pois podemos ter diferentes tipos de células para uma mesma posição. Suponha que para a primeira célula do *pipeline* em determinado algoritmo esteja configurada a função MAX(1,1) e em outro MAX(1,0). A célula necessária para aquela posição é semelhante à apresentada na figura 4.13. Essa célula é agora criada pela própria ferramenta, não havendo mais a necessidade do projetista gerar uma biblioteca de células. Esses processadores elementares são criados utilizando-se como base a metodologia definida no estudo de similaridades e explorando o conceito de similiaridade em nível de célula apresentado na seção 4.4.3.

Os sub-componentes utilizados para a composição dos processadores elementares são: célula de máximo, célula de soma, gerador de complemento de 2, multiplexador 2:1 e multiplexador 3:1. Através desses subcomponentes a parte operativa inteira é criada e todas as célula necessárias geradas.

O VHDL abaixo é um exemplo de célula gerada pela JVDR. O techo de código refere-se à estrutura VHDL de uma célula que realiza Max(1,-1) e ADD(1,1).

```
t1 <= X1;
MULT2 : mult1n1 port map (X => X2,W => W2,CLK => CLK, XOUT => t2);
MAXIMUM: MAX port map (X1 => t1,X2 => t2,CLK => CLK, XOUT => y1);
ADDER: add port map (X1 => t1,X2 => t2,CLK => CLK, XOUT => y2);
MPX: mux2 port map (X1 => y1,X2 => y2,SEL => SEL,CLK => CLK, XOUT => Y);
```

Analisando-se a descrição podemos indentificar alguns subcomponentes:

- mux2: Multiplexador 2:1 utilizado para selecionar se a função será MAX ou ADD.
- MAX: Função de máximo.

Tabela 4.3: Desempenho do DRIP reconfigurável

Algoritmo	**FLEX 10K (MHz)**	**APEX 20K (MHz)**
Filtro de Mediana	32,89	77,05
Detector Morfológico de Contornos	48,78	110,45
Erosão	39,84	78,74
Dilatação	44,12	84,18
Filtro de Mediana Separável	46,30	105,9

Tabela 4.4: Padrões de imagens

Padrão	**Resolução**
CIF	352x288
VGA	640x480
SVGA	800x600
SVGA	1024x768
SXVGA	2048x1536

- ADD: função de soma.
- mult1n1: multiplexador que seleciona na segunda entrada se o valor a ser computado será o pixel (peso 1) ou o pixel negado (peso -1).

Dessa maneira, tem-se a otimização e a geração de uma biblioteca completa de células contemplando os algoritmos desejados pelo projetista.

4.5 Resultados de Área e Desempenho

O DRIP pode executar uma ampla gama de algoritmos de baixo nível de processamento de imagens. Na tabela 4.3, a máxima frequência obtida pelo DRIP reconfigurável, ou seja, com parte operativa capaz de processar apenas 1 algoritmo por vez para duas famílias de FPGAs Altera.

Essas taxas de processamento asseguram desempenho de processamento em tempo real mesmo para altas resoluções de imagens. A regularidade e paralelismo da estrutura do *pipeline* aliadas as características dos problemas de processamento de imagem tornam o processador adequado para aplicações multimídia.

O gráfico da figura 4.26 apresenta a taxa de procesamento de quadros (quadros / segundo) para 3 situações distintas. O detector morfológico de contorno e o filtro de mediana representam, respectivamente, o melhor e pior caso na implementação estaticamente reconfigurável. A linha inferior na figura 4.26 (estilo de triângulo) representa o desempenho do DRIP-RTR. O *pipeline* do DRIP-RTR admite a mudança rápida de configuração para qualquer um dos cinco algoritmos vistos anteriormente na tabela 4.3. Os valores dos gráficos foram coletados considerando imagens com as características de padrões de vídeo mostrados na tabela 4.4.

Conforme dito, os valores da tabela 4.3 foram obtidos considerando um algoritmo apenas presente no *pipeline* em um dado momento (e sem capacidades de reconfiguração dinâmica), trata-se, portanto, de uma tabela de performance máxima. Devido aos *overheads* introduzidos no *pipeline*, especialmente para tornar as células responsivas ao controle, a implementação da parte operativa capaz de suportar reconfiguração dinâmica

(DRIP-RTR) entre os 5 algoritmos necessita de 1830 elementos lógicos. A frequência máxima de operação é de 68,2 MHz.

O acréscimo em área em relação ao algoritmo de maior tamanho (1123 elementos lógicos) da versão estaticamente reconfigurável é de 65%. Esse custo a mais deriva do fato de que os processadores elementares são células com maior capacidade de processamento individual e, naturalmente, por causa da existência da infra-estrutura de suporte ao controle como os multiplexadores necessários. Dessa maneira, esse é o preço pago em recursos por um nível maior de programabilidade, contudo, não é uma penalidade proibitiva, uma vez que mesmo com a adição do repositório de configurações e da lógica de gerenciamento de contextos o sistema ainda está adequado em tamanho para a complexidade dos modernos dispositivos programáveis. As sínteses foram relizadas para o dispositivo APEX20K400.

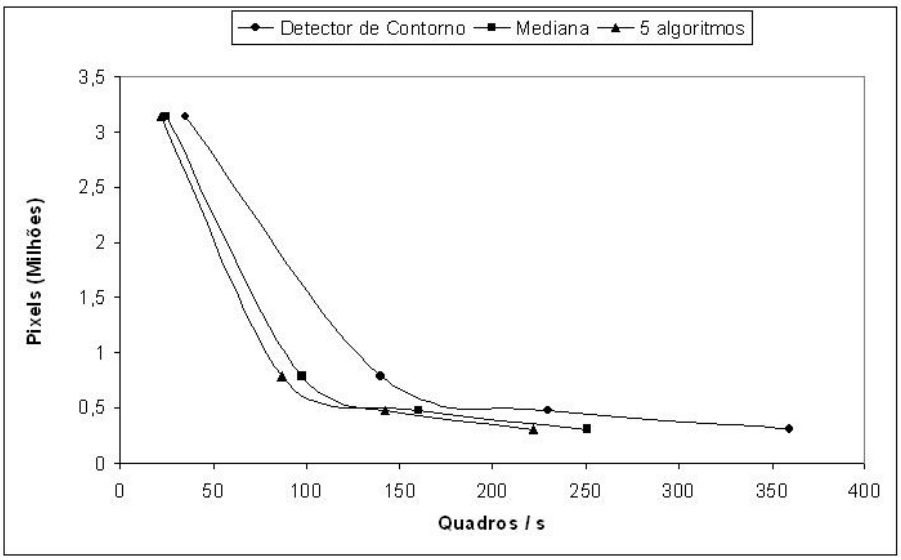

Figura 4.26: Desempenho do melhor e pior caso dos algoritmos no DRIP estaticamente reconfigurável e o desempenho do DRIP-RTR

4.5.1 Tempo de Reconfiguração

Como pode ser visto através da tabela 4.3, da figura 4.26 e da tabela 4.5 da seção seguinte, o *pipeline* de processamento do DRIP-RTR possui um desempenho inferior à versão estaticamente reconfigurável. Entretanto, o grande ganho reside no tempo de reconfiguração. Dispositivos FPGA, mesmo em modo paralelo de configuração demoram dezenas e, por vezes, até centenas de milisegundos para uma reconfiguração completa. Os dispositivos parcialmente configuráveis conseguem efetuar a reconfiguração de maneira ainda mais rápida, contudo, o suporte para diversos módulos dinâmicos ainda é restrito e a latência de reconfiguração continua a introduzir penalidades proibitivas para arquiteturas de alto desempenho.

Em virtude das mudanças introduzidas no DRIP-RTR, ele pode ser completamente reconfigurado em 1 ciclo de *clock* (cerca de 15 nanosegundos). Essa potencialidade representa uma aceleração de 10^7, ou seja, na ordem de milhões de vezes, em relação ao tempo de reconfiguração da versão estaticamente reconfigurável.

O ganho no tempo relativo de configuração (visto na equação 2.7) em relação à versão estaticamente reconfigurável é bastante grande. O resultado é um sistema RTR com

Tabela 4.5: *Speed-up* em relação a um processador de propósito geral

Algoritmo	**DRIP Reconfigurável (Speed-up)**	**DRIP-RTR (Speed-up)**	**Tamanho da Imagem**
Dilatação	0,94	0,76	256x256
Dilatação	3,06	2,4	512x512
Dilatação	17,4	14,15	1024x1024
Erosão	11,55	9,4	256x256
Erosão	45,9	37,2	512x512
Erosão	206,4	167,3	1024x1024

densidade funcional superior.

4.5.2 DRIP versus GPP

Para fornecer uma dimensão da taxa de processamento da arquitetura DRIP em relação a um GPP foi realizada a implementação de alguns algoritmos de processamento digital de imagens em linguagem C. No desenvolvimento desses algoritmos foi utilizada a biblioteca DipLib (VLIET, 2003) (Delft *Image Processing Library*) desenvolvida dentro do grupo de física aplicada e reconhecimento de padrões da universidade de tecnologia Delft da Holanda. Diplib constitui uma biblioteca científica de processamento digital de imagens independente de plataforma.

A biblioteca possui um grande número de funções para processamento e análise de dados multidimensionais de imagens. Dentro da API encontra-se grupos de rotinas disponíveis para transformações, filtragem, inspeção de cenas, medida de objetos e análise estatística. A biblioteca é altamente otimizada, visando máxima performance na execução dos algoritmos.

A tabela 4.5 apresenta a aceleração (*speed-up*) obtida pela arquitetura sobre uma estação Sun SunBlade. Essa estação conta com processador UltraSparc II de 650 MHz e 512 Mb de memória RAM executando sistema operacional Solaris 8. Como é possível observar, conforme o volume de dados aumenta a especialização da arquitetura DRIP começa a gerar um ganho de desempenho cada vez maior. No caso do algoritmo de erosão a aceleração chega a mais de 200 vezes

4.6 Implementação VLSI do Processador Elementar

Ao contrário do que é possível pensar em um primeiro momento, a implementação de uma arquitetura reconfigurável não necessariamente é realizada utilizando-se FPGAs. O custo dos circuitos programáveis, em termos de desempenho, potência e área, pode ser alto e muitas vezes a flexibilidade da configuração em nível de bit (granularidade fina) é mais do que o projetista necessita. Assim, conforme foi mencionado, estruturas com melhor aproveitamento dos recursos e reconfiguração em nível de palavra são frequentemente propostas e desenvolvidas (HARTENSTEIN, 2001). Diversas arquiteturas reconfiguráveis são implementadas sem um dispositivo programável, mas sim com uma arquitetura VLSI reconfigurável em nível de operadores e interconexões, tornando por vezes tênue a linha entre o reconfigurável e o programável.

Nesta seção, é apresentado o projeto elétrico e lógico do DRIP-RTR em tecnologia AMS 0.35μm.

4.6.1 Visão Geral da Célula

Os componentes necessários para a construção do processador elementar são:

- Multiplexadores (3:1 e 2:1)
- Circuito de geração de complemento de 2.
- Circuito que compute a função máximo entre dois números em complemento de 2.
- Circuito somador.

Esses elementos podem ser visualizados na figura 4.27. Para todos os projetos apresentados a seguir, foi utilizada uma largura Wn = 1.5μm para transistores NMOS e Wp = 3μm para transistores PMOS. Em todas as situações empregou-se L = 0.3μm, o mínimo permitido pela tecnologia.

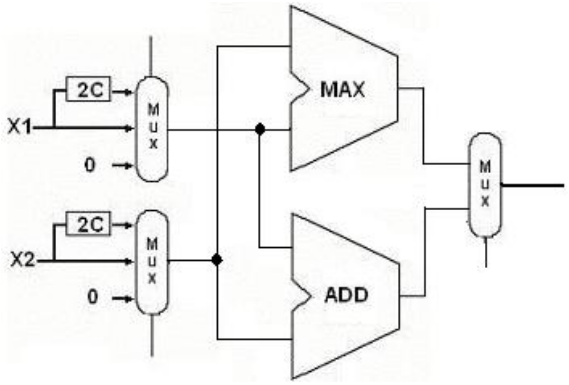

Figura 4.27: Componentes da célula

4.6.2 Projeto do Multiplexador

Os multiplexadores são necessários em dois momentos distintos na célula. Primeiramente, eles atuam no processo de ”multiplicação“ dos pixels de entrada pelos pesos restritos -1, 0 e 1. Dessa forma, existe um multiplexador 3:1 em cada uma das entradas do PE, realizando a seleção entre o valor do pixel direto, valor do pixel negado em complemento de 2 ou o valor zero. Um multiplexador também está posicionado na saída da célula, definindo se o valor resultante será a saída da função MAX ou da função ADD.

O multiplexador 2:1, nada mais é do que um seletor entre duas entradas. Um sinal de controle (S) sinaliza qual entrada terá seu valor transmitido para a saída. Sua função lógica pode ser resumida pela equação

$$Out = A.S + B.not(S) \tag{4.4}$$

A implementação elétrica do multiplexador e de outros circuitos utilizam *transmission gates*. Essa técnica de lógica de passagem contibui para evitar problemas como uma propagação fraca do sinal ”1“. A técnica consiste na utilização de um transistor NMOS e um transistor PMOS com seus terminais de fonte e dreno conectados. Um sinal de controle é aplicado, por exemplo, ao *gate* do transistor NMOS e o inverso do sinal ao *gate* do transistor PMOS (figura 4.28). Dessa forma, combina-se as características dos dispositivos NMOS e PMOS evitando problemas de degradação de sinal.

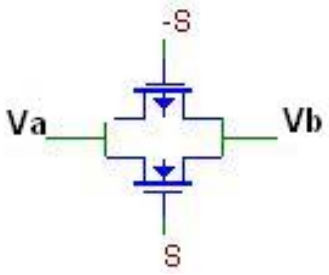

Figura 4.28: Transmission gate

O esquema elétrico completo do multiplexador 2:1 pode ser visto na figura 4.29. O multiplexador é composto por dois *transmission gates* e um inversor, totalizando 6 transistores.

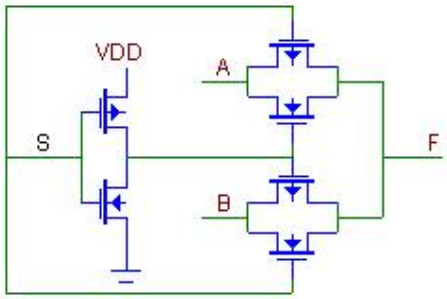

Figura 4.29: Esquema elétrico multiplexador 2:1

O multiplexador 2:1 é utilizado para definir de qual caminho virá o valor de saída da célula entre duas possibilidades. Contudo, em cada entrada da célula existem três fontes de dados possíveis. Para a construção do multiplexador 3:1 utilizou-se uma composição de dois multiplexadores 2:1 (figura 4.30).

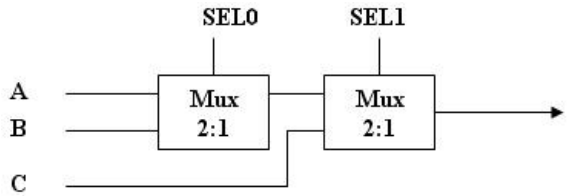

Figura 4.30: Esquema lógico do multiplexador 3:1

Vale ressaltar que o projeto mostrado é para 1 bit. Como a célula opera com pixels de 8 bits. O número de transistores do multiplexador 2:1 é 48 (8 bits x 6), enquanto que para cada multiplexador 3:1 tem-se 96 transistores (8 bits x 12).

4.6.3 Geração de Complemento de 2

Para executar a multiplicação por -1 de um pixel é usado um circuito que gera o complemento de 2 do número. O complemento de 2 é computado através da inversão do valor de entrada e a adição de 1 na sequência. Assim, a estrutura do gerador de complemento de 2 é formada por uma camada de inversores para a negação do operando junto a um somador para a adição do número 1.

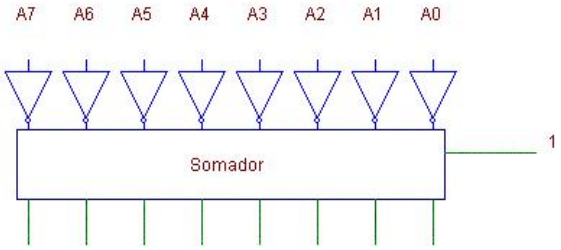

Figura 4.31: Estrutura lógica do gerador de complemento de 2

Tabela 4.6: Exemplo de máximo

A	B	AmBin	BmAin	S	AmBout	BmAout	subcir
0	0	X	X	**0**	0	1	msb
1	1	0	1	**1**	0	1	res
0	1	0	1	**1**	0	1	res

A figura 4.31 mostra a estrutura lógica. O somador utilizado será visto em maiores detalhes adiante. O gerador de complemento de 2 é o circuito que insere o maior atraso combinacional na célula, eliminá-lo com técnicas de otimização como as descritas na seção 4.3.2.1 sobre otimizações tende a trazer, portanto, importantes benefícios.

4.6.4 O Circuito de Máximo

O circuito de máximo retorna o maior entre dois números. É importante observar que ele deve considerar que a representação utilizada é complemento de 2 e não analisar apenas a sequência binária sem sinal. Dessa forma MAX(00000001,11000000) vale 00000001.

A idéia por trás do projeto lógico trabalha gerando três sinais principais S, AmB e BmA. S é o bit de saída de cada estágio do circuito e representa um bit do resultado final. AmB e BmA são sinais auxiliares que propagam-se pela formação *bit-slice* do circuito de máximo. O circuito de máximo é composto por dois subcircuitos chamados aqui de *comp-msb* e *comp-res*. O primeiro faz a análise específica do bit mais significativo, enquanto que o segundo avalia os bits restantes. Com o propósito de melhor exemplificar, a comparação entre um operando de três bits A = 010 e um operando B = 011 é apresentada na tabela 4.6.

As expressões lógicas que definem os circuitos *comp-msb* e *comp-res* em linguagem de descrição de hardware tem a seguinte estrutura:

comp-msb:
S <= A and B
AmB = (A and not(B))
BmA = (B and not(A))

comp-res:
S <= ((not(BmAin) and A) or (B and not(AmBin)))
AmB <= (AmBin or(not(B) and not(BmAin) and A))
BmA <= (BmAin or (not(AmBin) and B and not(A)))

A estrutura lógica do circuito de máximo pode ser visto na figura 4.32. Estão representados na ilustração apenas os 3 primeiros bits para simplificação.

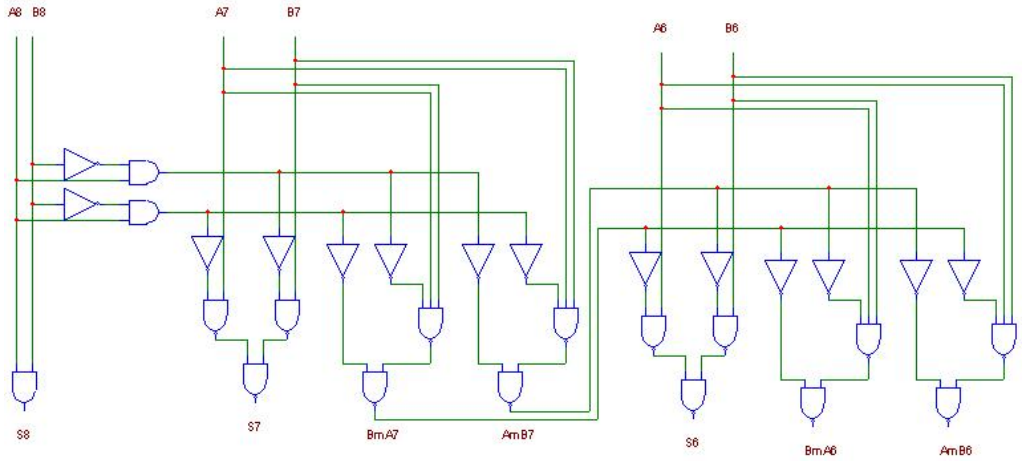

Figura 4.32: Projeto lógico do circuito de máximo

Conforme descrito, o primeiro bit (msb) é computado de maneira diferente do restante. Os sinais AmB e BmA propagam-se pelo circuito como sinais auxiliares.

Com a exceção da célula msb, os demais *gates* são inversores ou NANDS. O circuito original (como descrito em VHDL) utilizava-se de portas OR (ver descrição acima). Foi realizado um passo de otimização lógica para obtenção desta versão final do circuito. Neste caso, de uma implementação inicial de 58 transistores, foi possível reduzir para 44, uma diferença de cerca de 25%. Para os 8 bits na situação original teríamos: msb(20T) + 7 x comp-res (58T) = 426 transistores. Na versão otimizada o resultado é: msb(20T) + 7 X comp-res(44T) = 328 transistores.

O subcircuito que efetua a comparação do bit mais significativo é bem mais simples que o subcircuito que realiza a comparação dos demais. As figuras 4.33 e 4.34 mostram o esquema elétrico de ambos subcircuitos.

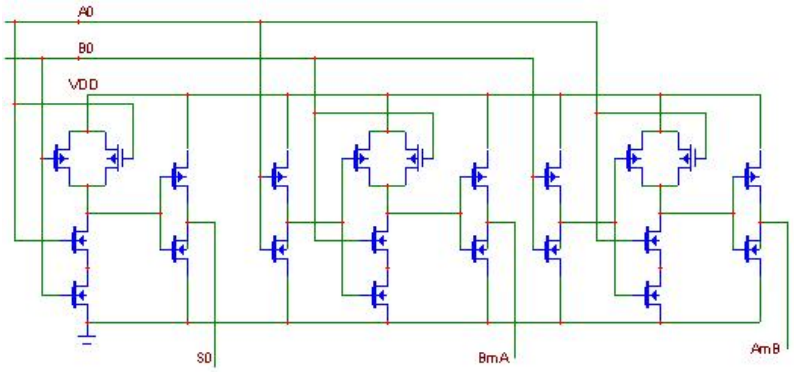

Figura 4.33: Parte elétrica do subcircuito de comparação do bit mais significativo

4.6.5 Circuito Somador

O circuito somador aparece como um bloco do gerador de complemento de 2, além de ser uma das funções da parte qualificadora do processador elementar.

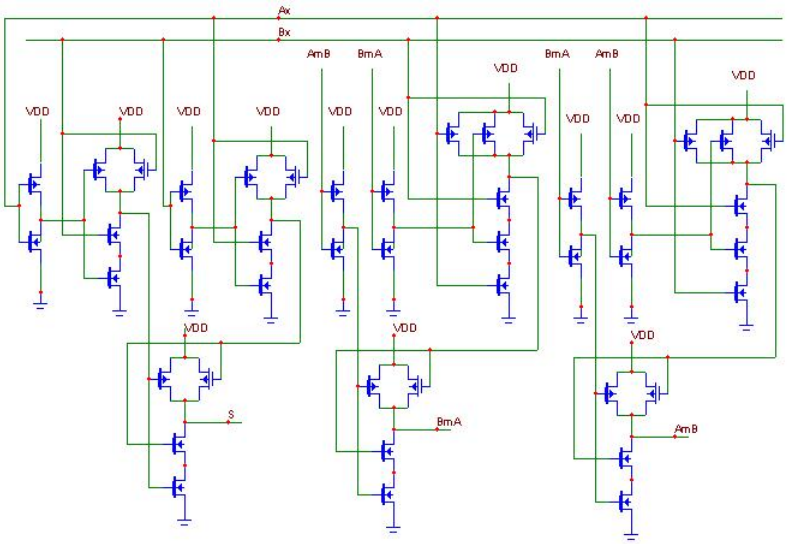

Figura 4.34: Parte elétrica do subcircuito de comparação dos demais bits

O somador inicialmente projetado e utilizado trata-se de um somador em cadeia (*ripple-carry*). Ele é composto por blocos somadores completos (*full-adders*), como os da figura 4.35.

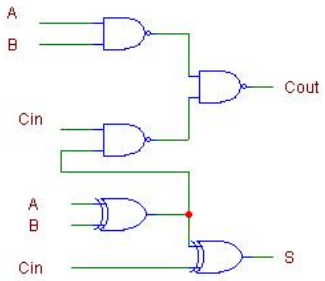

Figura 4.35: Somador completo

Na concepção de seu projeto elétrico (figura 4.36) foi utilizada novamente a técnica de *transmission gates* na implementação dos subcircuitos XOR. Assim, tem-se 6 transistores por XOR, totalizando 24 transistores para o cálculo de cada bit.

4.6.6 Análise de Resultados

Foram contruídas implementações SPICE dos circuitos descritos e uma série de simulações e verificações realizadas. Três configurações distintas apresentam resultados bastante diferentes e demonstram claramente qual o ponto de intervenção para a obtenção de resultados superiores. Vale ressaltar que os atrasos apresentados nesta seção representam o pior caso obtido entre o tempo de subida (Tdlh) e tempo de descida (Tdhl), os quais, no entanto, estão bastante equilibrados em todas as situações não apresentando variações significativas.

Configuração 1: Engloba todas as situações nas quais a função MAX está selecionada e não utilizamos o circuito de geração de complemento de 2. Exemplos: MAX(0,0), MAX(1,1), MAX(1,0) , etc. Nesta situação, foi obtido um atraso do processador elementar de 680ps, o que permite ao DRIP operar em taxas de 1.47GHz.

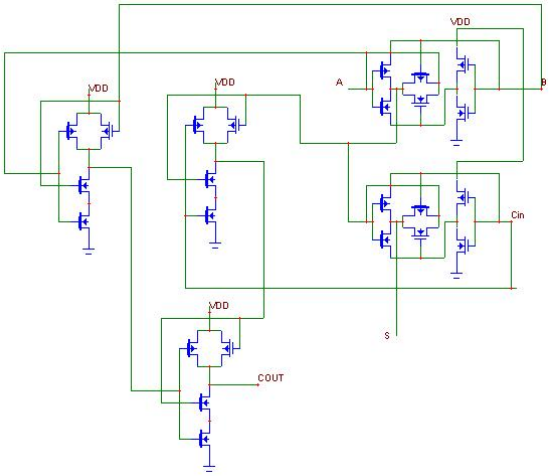

Figura 4.36: Diagrama elétrico do somador completo

Tabela 4.7: Desempenho do processador elementar DRIP para diferentes resoluções

Pixels Linha	**Pixels Coluna**	**Pixels Total**	**Caso 1 (f/s)**	**Caso 2 (f/s)**	**Caso 3 (f/s)**
352	288	101376	14500,47	5800,189	2988,873
640	480	307200	4785,156	1914,06	986,32
800	600	480000	3062,5	1225	631,25
1024	768	786432	1869,20	747,68	385,28
2048	1536	3145728	467,30	186,92	96,32

Configuração 2: São todas as configurações que não utilizam o complemento de 2, mas utilizam o somador. Exemplos: ADD(0,1), ADD(1,1), ADD(0,0), etc. Nesses casos, foi obtido um atraso de 1.7ns, o que permite ao DRIP operar a uma frequência de 588MHz.

Configuração 3: Trata-se do pior caso, compreende todas as configurações que utilizam o complemento de 2 e a função de soma. Exemplos: ADD(-1,0), ADD(-1,-1), etc. Nessa situação, o atraso é de cerca de 3.3ns, possibilitando uma frequência de 303MHz.

Para a obtenção desses dados foram utilizados vetores de entrada que forçam a propagação do *carry* até o bit final. Torna-se evidente que o somador é o ponto que exige otimização, com o emprego de uma estratégia de aceleração de *carry*.

A tabela 4.7 mostra o número de *frames* por segundo que podem ser computados em cada uma das situações. Conforme mostram esses resultados, é possível atingir taxas de processamento em tempo real para todas as resoluções de imagem mostradas.

A tabela 4.8 apresenta os resultados de área obtidos em número de transistores para cada componente do processador elementar. Considerando que a parte operativa completa utiliza 81 processadores elementares o número de transistores total do *pipeline* é de 86184.

Tabela 4.8: Número de transistores utilizados para o PE

ADD	**MAX**	**GC2**	**MUX 2:1**	**MUX 3:1**	**Total**
240T	328T	256T	48T	96T (2)	1064

4.6.7 Circuito Somador Alternativo

Com os resultados obtidos, tornou-se clara a necessidade de modificar o circuito somador. Existe, na literatura, diversas técnicas para essa finalidade como *carry look-ahead*, *mirror adder* e *carry-select*.

A primeira estrutura avaliada é o *mirror adder*. Neste somador as cadeias NMOS e PMOS são perfeitamente simétricas e existem, no máximo, dois transistores em série no caminho de geração do *carry*.

No projeto deste somador, toma-se o cuidado de posicionar os transistores conectados ao Cin (*carry* de entrada) de maneira mais próxima possível da saída. A figura 4.37 apresenta a estrutura elétrica deste somador:

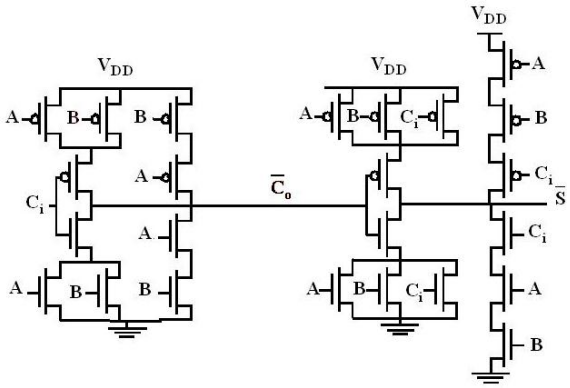

Figura 4.37: Diagrama elétrico do somador *mirror*

Os transistores próximos à saída de *carry out* e de soma foram dimensionados de maneira um pouco diferenciada. Os transistores NMOS possuem um W de 4.5μm e os transistores PMOS um W de 9μm.

Para a implementação do segundo somador avaliado foi escolhida uma estratégia de seleção de *carry* (*carry-select*). Essa opção foi feita com base na experiência anterior do autor com arquiteturas de somadores e com base em resultados comparativos de implementações de algumas arquiteturas para um problema de compressão JPEG (PORTO; AGOSTINI, 2003). Os resultados mostram que embora a ocupação de área seja maior o desempenho tende a superar implementações de *carry look-ahead* e *carry look-ahead* hierárquico.

O somador é composto por duas cadeias *ripple* de 4 bits separadas por um multiplexador. A idéia é que cada cadeia possua duas seções nas quais os elementos somadores são duplicados uma assumindo um valor de *carry* igual a ”1“ e a outra igual a ”0“. Quando o resultado de cada cadeia acaba de ser computado ele é usado simplesmente para selecionar qual saída calculou antecipadamente a soma de forma correta.

Nesta implementação são utilizados dois tipos de somadores completos, um para o estágio par e outro para o estágio ímpar. Esses somadores de 1 bit são diferentes para minimizar a quantidade de portas lógicas no caminho de propagação do *carry* que tende a ser o caminho crítico. A figura 4.38 mostra dois estágios interconectados.

O projeto elétrico do *full adder* de estágio par é composto por três super portas, dois inversores e duas portas NOR. O cuidado tomado na definição da estrutura elétrica foi, novamente, a aproximação de Cin dos nodos de saída dos *gates*. Na figura 4.39 pode-se observar o esquema elétrico do *full adder* de estágio par.

De forma análoga, o *full adder* do estágio ímpar é composto por três portas complexas,

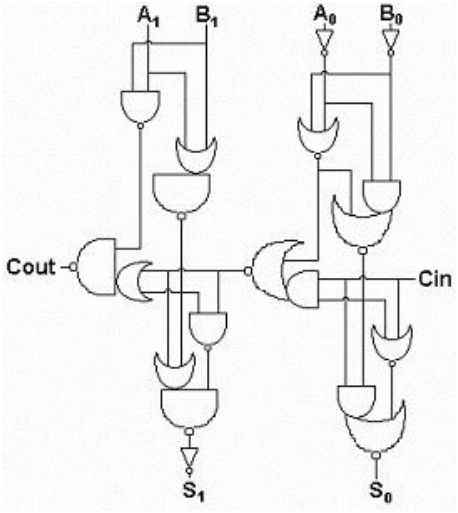

Figura 4.38: Estágios par e ímpar

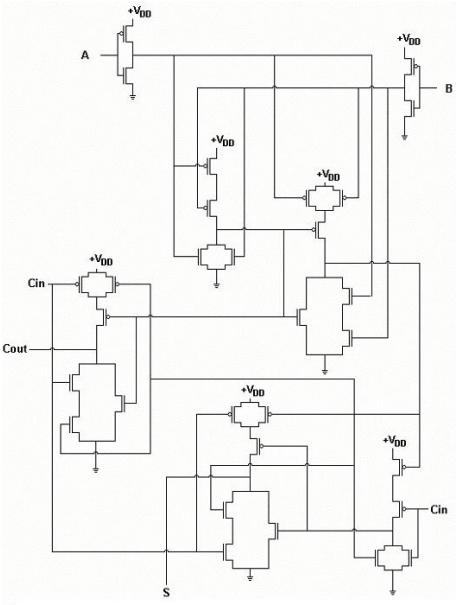

Figura 4.39: Estágio par

duas portas NAND e um inversor, e pode ser observado na figura 4.40. É importante notar que neste caso o sinal da super porta de saída deve ser complementado. Outro ponto interessante é que este estágio consome o sinal de *carry* negado fornecido pelo somador completo anterior.

A figura 4.41, por sua vez, apresenta a planta baixa do somador. Os blocos com a letra M representam os multiplexadores e os blocos com a letra F os somadores completos. Os multiplexadores são utilizados para realizar a seleção correta do valor de cada bit calculado e também para definir qual o valor de *carry* correto gerado por cada cadeia individual de 4 bits.

4.6.8 Análise de Resultados com os Novos Somadores

O desempenho dos somadores no processador elementar para os casos 2 e 3 descritos anteriormente são apresentados na tabela 4.9.

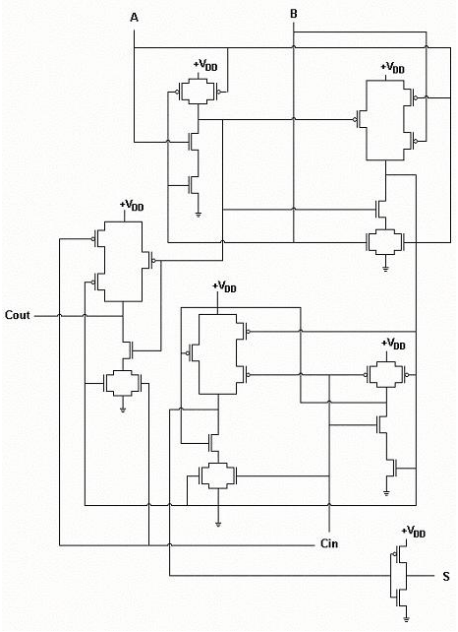

Figura 4.40: Estágio ímpar

Tabela 4.9: Desempenho do processador elementar DRIP para diferentes resoluções

Pixels Linha	**Pixels Coluna**	**Caso 2** Carry-Select (f/s)	**Caso 3** Carry-Select (f/s)	**Caso 2** Mirror Adder (f/s)	**Caso 3** Mirror Adder (f/s)
352	288	12922,19	8216,93	11442,55	7043,08
640	480	4264,32	2711,58	3776,04	2324,21
800	600	2719,16	1735,41	2416,66	1487,5
1024	768	1665,75	1059,21	1475,01	907,89
2048	1536	416,43	264,80	368,75	226,97

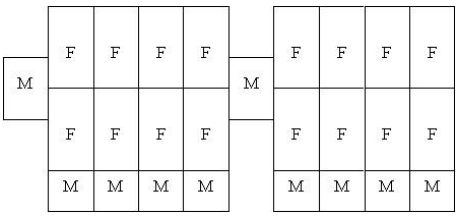

Figura 4.41: Planta baixa do somador carry select

Utilizando esses novos somadores em substituição ao somador *ripple* de 8 bits na célula do processador elementar, atinge-se novos resultados que podem ser vistos na tabela 4.9. O atraso para os casos 2 e 3 mostrados na seção 4.6.6 passam a ser agora, 760ps (1.31GHz) e 1.2ns (833MHz) para o somador tipo *carry-select* e 860ps (1.16GHz) e 1.4ns (714MHz)para o *mirror adder*.

Analisando os dados é possível observar que, priorizando-se o desempenho, o somador escolhido é o *carry-select*. Essa mudança do somador resulta em uma aceleração de cerca de 2,2 entre a configuração 2 na situação atual e a anterior (com *ripple*) e em uma aceleração de 2,7 na configuração 3.

Os dados referentes a área, por sua vez, podem ser vistos na tabela 4.10. Devido a replicação de somadores no *carry-select* existe um acréscimo de aproximadamente 10%

Tabela 4.10: Número de transistores utilizados com novo somador

ADD	MAX	GC2	MUX 2:1	MUX 3:1	Total
292T	328T	308T	48T	96T (2)	1168

Tabela 4.11: Processador elementar equivalente em FPGA (APEX20K400)

Área (em LE)	Frequência (em MHz)
75	157,68

na área. O circuito da parte operativa considerando os 81 processadores elementares totaliza 94608 transistores.

4.6.9 Célula DRIP Equivalente

Os dados analisados referem-se ao projeto VLSI de um processador elementar capaz de computar todos os algoritmos. Podemos, no entanto, realizar a comparação com uma célula DRIP descrita em VHDL e mapeada em FPGA com as mesmas capacidades lógicas, ou seja, capaz de executar qualquer configuração possível de um processador elementar.

Essa célula foi gerada com a ferramenta JVDR e possui as características mostradas na tabela 4.11.

Nessa situação a aceleração projetada para a implementação VLSI seria de cerca de 10 vezes sobre o processador elementar em FPGA no melhor caso e cerca de 6 vezes no pior. Entretanto, esse resultado reflete uma comparação célula a célula, ao considerarmos um circuito completo e os atrasos inseridos pelas capacitâncias e resitências das interconexões a performance obtida para a célula VLSI e para o equivalente em FPGA sofre uma degradação. A degradação de performance tende a ser maior no FPGA (principalmente por causa do roteamento), no entanto, a relação aproximada de uma ordem de grandeza tende a manter-se.

5 CONCLUSÃO E TRABALHOS FUTUROS

As arquiteturas reconfiguráveis constituem um paradigma de desenvolvimento bastante importante atualmente. O fator flexibilidade possibilita a extensão do tempo de vida dos produtos através de manutenção e atualização remota. Essa flexibilidade vem acompanhada por características de arquiteturas de propósito especial como alta performance e, em alguns casos, baixo consumo de energia.

Dentro do contexto do desenvolvimento de arquiteturas reconfiguráveis, este trabalho mostrou os passos realizados de projeto e implementação que possibilitaram a evolução do processador de imagens DRIP, desde uma versão com comportamento estático, até uma implementação reconfigurável em tempo de execução. A versão estaticamente reconfigurável apresenta desempenho superior em relação ao DRIP-RTR, além disso, é possível observar uma utilização de recursos lógicos cerca de 65% maior na versão dinamicamente reconfigurável.

Contudo, um projeto estaticamente reconfigurável necessita o término da tarefa corrente para possibilitar a mudança no programa/algoritmo em execução. Em virtude do tempo necessário para reconfiguração do dispositivo, uma perda de desempenho (muitas vezes alta) é gerada. O DRIP-RTR, embora mais lento, possui uma latência de reconfiguração menor na ordem de milhões de vezes em relação à versão estaticamente reconfigurável.

Para o desenvolvimento do *pipeline* do DRIP-RTR foi realizado um estudo sobre as similaridades entre os algoritmos e as características comuns das células básicas. Essa análise levou ao aumento de granularidade do processador elementar (uma das razões da maior utilização de área). Esse aumento na quantidade de recursos lógicos da célula básica do processador, no entanto, tornou possível que o processo de reconfiguração seja desvinculado do final da tarefa de processamento, ou seja, reconfiguração em tempo de execução.

Em todas as etapas do projeto tem-se a participação de ferramentas de CAD desenvolvidas especificamente para auxiliar na síntese e teste da arquitetura. Conforme discutido anteriormente, uma característica e dificuldade que norteia o desenvolvimento de arquiteturas reconfiguráveis é a constante necessidade de desenvolvimento de ferramentas específicas para o sistema alvo. No projeto do DRIP, a ferramenta VDR e mais tarde a JVDR possuem papéis importantes, contribuindo para a geração de algoritmos de maneira mais ágil, segura e otimizada. As otimizações trazem vantagens importantes, se considerarmos, por exemplo, o algoritmo de filtro de mediana de maneira individual, as otimizações trazem uma economia de recursos de cerca de 70%.

Além da implementação em FPGAs, foi realizado o projeto lógico e elétrico tendo como objetivo a implementação *full custom* do processador elementar. Todos os subcomponentes individuais foram projetados, otimizados, testados e devidamente caracterizados

em função do número de transistores utilizados e desempenho. Tornou-se evidente, analisando os resultados preliminares, a necessidade de refinamento do circuito somador e a substituição do somador *ripple* por um circuito com capacidade de aceleração de *carry*. Essa mudança trouxe um alto ganho de desempenho ao PE. As simulações elétricas feitas mostram que a criação de um *core* DRIP focado na filtragem digital seria bastante útil e adequado, por exemplo, como parte de um CSoC maior de processamento, atingindo um desempenho de cerca de 1.5GHz em tecnologia AMS 0.35μm.

De um maneira geral, este trabalho abordou um fluxo de trabalho completo em arquiteturas reconfiguráveis, passando pelo projeto da arquitetura e posterior criação de mecanismos que viabilizaram a reconfiguração dinâmica. O *overhead* gerado para a implementação do suporte para mudança de contextos foi quantificado.

5.1 Trabalhos Futuros

Pretende-se realizar uma série de trabalhos explorando as potencialidades da arquitetura e pesquisando novas soluções através das possibilidades de exploração do espaço de projeto existentes.

5.1.1 Análise do Comportamento do Processador no Contexto de uma Aplicação de Mais Alto Nível

O desempenho do DRIP-RTR foi analisado e mapeado executando algoritmos de processamento de imagens de baixo nível, em especial algoritmos morfológicos.

O objetivo desta etapa de trabalho será identificar uma aplicação em processamento de imagens em nível mais alto, como, por exemplo, inspeção de cenas e implementar no DRIP a sequência de tarefas necessárias. Dessa maneira, poderemos em uma aplicação bastante utilizada, avaliar questões como a quantidade de reconfigurações necessárias e observar possíveis pontos de aperfeiçoamento do sistema de controle e reconfiguração.

Outro ponto bastante interessante neste contexto é a realização de um estudo voltado para definir alterações na arquitetura com dois propósitos. O primeiro é ampliar o domínio da aplicação, abrangendo problemas de nível maior. Desse estudo espera-se derivar os caminhos para a criação de uma arquitetura CSoC eficiente para processmento de uma ampla gama de algoritmos significativos no processamento de imagens. O segundo propósito está ligado a importante questão da granularidade. Existe uma relação direta entre o desempenho de uma aplicação em uma arquitetura reconfigurável e a granularidade de seus operadores. Pretende-se avaliar ainda mais o impacto de mudanças de granularidade, exercitando-se a capacidade adaptativa do processsador DRIP.

5.1.2 Implementação com Reconfiguração Dinâmica Parcial

Este trabalho pretende avaliar o comportamento da arquitetura através de implementação que utilize FPGAs Virtex-II da Xilinx. Esses FPGAs permitem a adoção de estratégias de reconfiguração dinâmica parcial. Existem ferramentas que viabilizam a manipulação de arquivos de *bitstream* para essas tecnologias, a maioria dessas ferramentas, inclusive, são baseadas em uma API desenvolvida em Java. Esse conjunto de condições abre caminho para expansão da ferramenta JVDR, tendo em vista a geração automática de arquivos parciais representando módulos dinâmicos a serem carregados no FPGA. Este tipo de engenharia cria condições para o estudo de uma série de fatores, entre eles as melhores maneiras de estabelecer um sistema controlador e a interface desse sistema com o mundo exterior.

5.1.3 Finalização da Implementação *Full-Custom*

O Projeto lógico e elétrico foi realizado. Entretanto, dados como as capacitâncias de área e junção dos transistores foram inseridos através de cálculos, estimativas e informações dos parâmetros da tecnologia AMS 0.35μm.

Serão construídos leiautes utilizando um ambiente de projeto como Cadence ou Mentor Graphics. Com a contrução das células, dados de desempenho, área e consumo de energia poderão ser analisados com maior precisão.

5.1.4 Estudo Sobre o sistema de I/O

Ao longo deste trabalho foram realizados experimentos de prototipação com o kit de desenvolvimento PCI da Altera. A intenção é utilizar uma interface PCI para fazer com que os pixels da imagem alcancem o gerador de vizinhança residente dentro do FPGA.

A implementação desta parte do sistema de I/O viabilizará o estudo das alternativas de I/O e também contribuirá para validar de maneira ainda mais profunda e completa a arquitetura e todos os seus mecanismos.

Os tópicos de desenvolvimento citados são linhas de trabalho dentre muitas a serem percorridas. Espera-se com a continuidade dos trabalhos a realização de contribuições importantes para o campo de pesquisa da computação reconfigurável.

REFERÊNCIAS

ADÁRIO, A. M. S. **Uma implementação em FPGA de um Processador de Imagens de Vizinhança**. 1997. Dissertação (Mestrado em Ciência da Computação) — Unicamp, Campinas.

ADÁRIO, A. M. S.; ROEHE, E. L.; BAMPI, S. Dynamically Reconfigurable Architecture for Image Processor Applications. In: DESIGN AUTOMATION CONFERENCE, DAC, 36., 1999, New Orleans, USA. **Proceedings...** New York: ACM Press, 1999. p.623–628.

ALTERA. **Altera Stratix Datasheets**. Disponível em: <http://www.altera.com>. Acesso em: 01 ago. 2003.

BACKUS, J. Can Programming be liberated from the Von Neumann style? A Functional Style and its Algebra of Programs. **Communications Of the ACM**, New York, v.21, n.8, p.613–641, Aug. 1978.

BAUMGARTE, V. et al. PACT XPP: a self-reconfigurable data processing architecture. **The Journal of Supercomputing**, Hingham, v.26, n.2, p.167–184, 2003.

BECKER, J. Configurable Systems-on-Chip, CSoC. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 15., 2002, Porto Alegre, Brasil. **Proceedings...** New York: IEEE Computer Society, 2002. p.9–14.

BECKER, J.; PIONTECK, T.; GLESNER, M. An Application-tailored Dynamically Reconfigurable Hardware Architecture for Digital Baseband Processing. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, SBCCI, 12., 2000, Manaus, Brazil. **Proceedings...** New York: IEEE Computer Society, 2000.

BECKER, J.; PIONTECK, T.; GLESNER, M. An Application-tailored Dynamically Reconfigurable Hardware Architecture for Digital Baseband Processing. In: BRAZILIAN SYMPOSIUM ON INTEGRATED CIRCUIT AND SYSTEM DESIGN, SBCCI, 13., 2000, Manaus, Brasil. **Proceedings...** New York: IEEE Computer Society, 2000. p.18–22.

BOMAR, B. W. Implementation of Microprogrammed Control in FPGAs. **IEEE Transactions on Industrial Electronics**, New York, v.49, n.2, p.15–25, April 2002.

BONDALAPATI, K.; PRASANNA, V. K. Reconfigurable Computing Systems. **Proceedings of the IEEE**, New York, v.90, n.7, p.1201 – 1217, 2002.

BROWN, S. D.; VRANESIC, Z. G. **Fundamentals of Digital Logic With VHDL Design**. New York: McGraw Hill, 2000.

BURNS, J. et al. A Dynamic Reconfiguration Run-time System. In: IEEE SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES, FCCM, 5., 1997, Napa, USA. **Proceedings...** New York: IEEE Computer Society, 1997. p.66–75.

CARDOSO, J. M. P.; WEINHARDT, M. From C Programs to the Configure Execute Model. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2003, Munich, Germany. **Proceedings...** New York: IEEE Computer Society, 2003. p.576–581.

COMPTON, K.; HAUCK, S. Reconfigurable Computing: a survey of systems and software. **ACM Computing Surveys**, New York, v.34, n.2, p.171 – 210, 2002.

DAVID, R. et al. DART: a dynamically reconfigurable architecture dealing with future mobile telecommunications constraints. In: PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, IPDPS, 2002, Fort Lauderdale, USA. **Proceedings...** New York: IEEE Computer Society, 2002.

DEHON, A. DPGA-Coupled Mocroprocessors: commodity ics for the early 21st century. In: IEEE WORKSHOP ON FPGAS FOR CUSTOM COMPUTING MACHINES, 1994. **Proceedings...** New York: IEEE Computer Society, 1994. p.31–33.

DEHON, A. et al. Stream Computations Organized for Reconfigurable Execution (SCORE): introduction and tutorial. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS, FPL, 10., 2000. **Proceedings...** Heidelberg: Springer Verlag, 2000.

DEHON, A.; WAWRZYNEK, J. Reconfigurable Computing: what, why, and implications for design automation. In: DESIGN AUTOMATION CONFERENCE, DAC, 36., 1999, New Orleans, USA. **Proceedings...** New York: ACM Press, 1999. p.610–615.

EBELING, C.; CRONQUIST, D. C.; FRANKLIN, P. Configurable Computing: the catalyst for high-performance architectures. In: INTERNATIONAL CONFERENCE OF APPLICATION-SPECIFIC SYSTEMS, ARCHITECTURES AND PROCESSORS, 1997. **Proceedings...** New York: IEEE Computer Society, 1997. p.364–372.

ELDREDGE, J.; HUTCHINGS, B. L. RRANN: the run-time reconfigurable artificial neural network. In: CUSTOM INTEGRATED CIRCUITS CONFERENCE, 1994, San Diego, USA. **Proceedings...** New York: IEEE Computer Society, 1994. p.77–80.

GONZALEZ, R. C.; WOODS, R. E. **Digital Image Processing**. [S.l.]: Prentice Hall, 2002.

GUCCIONE, S. A.; LEVI, D.; SUNDARARAJAN, P. Java Based Interface for Reconfigurable Computing. In: MILITARY AND AEROSPACE APPLICATIONS FOR PROGRAMMABLE DEVICES AND TECHNOLOGIES CONFERENCE, MAPLD, 2., 1999. **Proceedings...** [S.l.: s.n.], 1999.

GUPTA, R. K.; ZORIAN, Y. Introducing Core-Based System Design. **IEEE Design and Test of Computers**, New York, v.14, n.4, p.15–25, 1997.

HARTENSTEIN, R. A Decade of Reconfigurable Computing: a visionary retrospective. In: INTERNATIONAL CONFERENCE ON DESIGN AUTOMATION AND TESTING

IN EUROPE 2001, DATE, 2001, Munich, Germany. **Proceedings...** New York: IEEE Computer Society, 2001.

HAUSER, J. R.; WAWRZYNEK, J. Garp: A MIPS processor with a reconfigurable coprocessor. In: IEEE SYMPOSIUM ON FPGAS FOR CUSTOM COMPUTING MACHINES, 1997, Los Alamitos, CA. **Proceedings...** New York: IEEE Computer Society, 1997. p.12–21.

HORTA, E.; KOFUGI, S. A Run-Time Reconfigurable ATM Switch. In: RECONFIGURABLE ARCHITECTURES WORKSHOP, RAW, 9., 2002, Fort Lauderdale, USA. **Proceedings...** New York: IEEE Computer Society, 2002.

HORTA, E. L.; LOCKWOOD, J. W.; KOFUJI, S. T. Using Parbit to Implement Partial Run-Time Reconfigurable Systems. In: INTERNATIONAL CONFERENCE ON FIELD PROGRAMMABLE LOGIC AND APPLICATION, FPL, 12., 2002, Montpellier, França. **Proceedings...** New York: IEEE Computer Society, 2002. p.182–191.

KNUTH, D. E. **The Art of Computer Programming**. [S.l.]: Addison-Wesley, 1973.

LEITE, N. J.; BARROS, M. A. A Highly Reconfigurable Neighborhood Processor Based on Functional Programming. In: IEEE INTERNATIONAL CONFERENCE ON IMAGE PROCESSING, 1994. **Proceedings...** New York: IEEE Computer Society, 1994. p.659–663.

MIYAMORI, T.; OLUKOTUN, K. REMARC: reconfigurable multimedia array coprocessor (abstract). In: ACM/SIGDA INTERNATIONAL SYMPOSIUM ON FIELD PROGRAMMABLE GATE ARRAYS, 6., 1998, Monterey, USA. **Proceedings...** New York: ACM Press, 1998. p.261.

NAGARAJAN, R. et al. A design space evaluation of grid processor architectures. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 34., 2001. **Proceedings...** [S.l.: s.n.], 2001. p.40–51.

NOLLET, V. et al. Designing an Operating System for a Heterogeneous Reconfigurable SoC. In: RECONFIGURABLE ARCHITECTURES WORKSHOP, RAW, 10., 2002, Nice, França. **Proceedings...** Berlin: Springer-Verlag, 2002.

PACT. **Smart Media Processing with XPP**. Disponível em: <http://www.pactcorp.com>. Acesso em 01 set. 2003.

PATTERSON, D. A.; HENESSY, J. L. **Computer Organization and Design - The Hardware/Software Interface**. [S.l.]: Morgan Kauffman, 1997.

PORTO, R. E. C.; AGOSTINI, L. V. Adder Architectures Design for JPEG Compression. In: SOUTH SYMPOSIUM ON MICROELETRONICS, 18., 2003, Novo Hamburgo, Brasil. **Proceedings...** [S.l.: s.n.], 2003. p.25–28.

RABAEY, J. Reconfigurable Processing: the solution to low-power programmable dsp. In: INTERNATIONAL CONFERENCE ON ACOUSTICS, SPEECH AND SIGNAL PROCESSING, ICASSP, 1997, Munich, Germany. **Proceedings...** New York: IEEE Computer Society, 1997.

SALEFSKI, B.; CAGLAR, L. Re-configurable computing in wireless. In: DESIGN AUTOMATION CONFERENCE, DAC, 38., 2001, Las Vegas, USA. **Proceedings...** New York: ACM Press, 2001.

SANKARALINGAM, K. et al. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In: INTERNATIONAL ANNUAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 30., 2003, California, USA. **Proceedings...** New York: IEEE Computer Society, 2003.

SCHMIT, H. et al. PipeRench: a virtualized programmable datapath in 0.18 micron technology. In: IEEE CUSTOM INTEGRATED CIRCUITS CONFERENCE, CICC, 2002. **Proceedings...** New York: IEEE Computer Society, 2002. p.63 – 66.

SHIRAZI, N.; LUK, W.; CHEUNG, P. Y. K. Run-time Management of Dynamically Reconfigurable Designs. In: INTERNATIONAL WORKSHOP ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS, FPL, 8., 1998, Tallinn, Estonia. **Proceedings...** Heidelberg: Springer Verlag, 1998. p.59–68.

SINGH, H. et al. MorphoSys: case study of a reconfigurable computing system targeting multimedia applications. In: DESIGN AUTOMATION CONFERENCE, DAC, 37., 2000, Los Angeles,CA, USA. **Proceedings...** New York: ACM Press, 2000. p.573–578.

SKAHILL, K. **VHDL for Programmable Logic**. New York: Prentice Hall, 2002.

TRIMBERGER, S. M. **Field-Programmable Gate Array Technology**. Hingham: Kluwer Academic Plublishers, 1994.

VLIET, V. L. J. **DipLib - The Delft Image Processing library**. Disponível em: <http://www.ph.tn.tudelft.nl/DIPlib/>. Acesso em 01 set. 2003.

WAINGOLD, E. et al. Baring It All to Software: raw machines. **Computer**, New York, v.30, n.9, p.86–93, 1997.

WIRTHLIN, M. J.; HUTCHINGS, B. L. Improving Functional Density Using Run-Time Circuit Reconfiguration. **IEEE Transactions on very large scale integration (VLSI) systems**, New York, n.2, June 1998.

XILINX. **Xilinx Virtex Datasheets**. Disponível em: <http://www.xilinx.com>. Acesso em: 01 ago. 2003.