Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Pós-Graduação em Ciência da Computação

Contribuições para reconfiguração parcial, remota e dinâmica de FPGAs

Daniel Gomes Mesquita

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação

Orientador: Fernando Gehm Moraes

Porto Alegre, março de 2002

Ao meu filho, Gabriel, que ainda no ventre de sua mãe já é fonte de inspiração para minha vida.
À minha mulher, Renata, pela paciência, carinho e apoio.
Aos meus pais, Sady e Ecilda e irmão, Davi, pela torcida e incentivo.
Ao meu orientador, Fernando, porque mais que orientador, é um exemplo a ser seguido.

"*Eloquence is a painting of the thoughts*"

Blaise Pascal, filósofo e matemático (1623-1662)

Agradecimentos

Um trabalho extenso como uma dissertação de mestrado jamais é construído sozinho. E foram tantos os colaboradores nesta empreitada, que nomeá-los acabaria gerando um volume com número de páginas superior ao da própria dissertação, e ainda sim poderia incorrer na injustiça de esquecer de alguém. Por isto, ao invés de um agradecimento nominal, agradeço a todos que de alguma forma auxiliaram nesta empreitada, através de uma oração:

Pelo apoio do lar;
pelo amparo da escola;
pela proteção do trabalho;
pela alegria de servir;
pelo aviso da experiência;
pelo exercício da tolerância;
pela capacidade de ser útil;
pelo dom de discernir;
pela força da paciência;
pelo amigo que me socorre;
pelo adversário que me instrui;
pelos estímulos com que me conduzes;
pelas provações com que me esclareces;
pelas dificuldades com que me controlas;
pela energia da esperança e
por todas as bençãos de amor que me proporcionas,
através dos entes queridos que me confias...
...Obrigado meu DEUS!

Resumo

Este trabalho disserta a respeito de Sistemas Digitais Reconfiguráveis (SDRs). Aborda as tendências acadêmicas e industriais dessa tecnologia. Em função destas, é proposta uma classificação de SDRs conforme seus objetivos e tecnologia empregada. O estudo de SDRs evidencia duas carências: ferramentas e dispositivos que permitam reconfiguração parcial. Entre os FPGAs disponíveis comercialmente é identificado um que contempla características arquiteturais para implementação de SDRs. Em função de um estudo minucioso da organização interna de FPGAs da família Virtex, são desenvolvidas algumas ferramentas para o projeto e implementação de SDRs. Estas ferramentas almejam prover a reconfiguração remota, parcial e dinâmica. As ferramentas foram validadas em plataformas de prototipação com dispositivo Virtex. O trabalho também reflete sobre problemas para implementação de SDR por causa de limitações impostas pelas características dos atuais FPGAs comerciais. Para diminuir esse problema, é proposto um barramento para interconexão de diferentes módulos de hardware em um FPGA.

Palavras-chave: FPGAs Virtex, Ferramentas para reconfiguração, reconfiguração parcial, reconfiguração remota, barramento para interconexão de cores, Sistemas digitais Reconfiguráveis.

Abstract

This work addresses academic and industrial Digital Reconfigurable Systems (DRS). A chronological classification of DRS is considered, summarizing the characteristics of each DRS generation, and the new features of succeeding generations. Also, it is observed the current lack of devices allowing partial reconfiguration and CAD tools supporting DRS. Among the available commercial devices, the Virtex FPGA is selected for implement DRS. The internal structure of this component is detailed. A set of tools were developed using the knowledge of the Virtex architecture. These tools allow remote and partial reconfiguration, as well as cores manipulation. All tools were validated on prototyping boards. The final part of this work discusses how partial reconfiguration can be used to implement virtual hardware.

Keywords: Virtex FPGAs, reconfiguration tools, partial reconfiguration, remote reconfiguration, cores interconnection bus, Digital Reconfigurable Systems.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

FPGA	*Field-Programmable Gate Array*	1
SDR	Sistemas Digitais Reconfiguráveis	2
ULA	Unidade Lógica e Aritmética	2
LUT	*Look-Up Table*	3
DPGA	*Dinamically Programmable Gate Array*	4
RTR	*Run-Time Reconfiguration*	4
CLB	*Configurable Logic Block*	5
IOB	*Input/Output Block*	5
CAD	*Computer Aided Design*	7
ASIC	*Application Specific Integrated Circuit*	8
μP	Microprocessador	11
GPP	*General Purpouse Processor*	12
ASIP	*Application Specific Instruction-set Processor*	14
SoC	*System-on-a-Chip*	16
A/D	Analógico para Digital	19
D/A	Digital para Analógico	19
CAB	*Configurable Analog Blocks*	19

DMC	*Digital Macro Cells*	19
E/S	Entrada e Saída	22
RRANN	*Run-time Reconfigurable Artificial Neural Network*	24
CDFG	*Control Data Flow Graph*	31
CM	*Clock Morphing*	32
DCS	*Dynamic Circuit Switching*	32
EDIF	*Eletronic Design Interchange Format*	32
VHDL	*VHSIC Hardware Description Language*	32
VHSIC	*Very High Speed Integrated Circuit*	32
SCORE	*Stream Computations for Reconfigurable Execution*	34
UCP	Unidade Central de Processamento	39
API	*Application Programming Interface*	47
CCM	*Custom Computing Machine*	48
XBI	*Xilinx Bitstream Interface*	49
JRTR	*Java Run-Time Reconfiguration*	53
MCU	*Memory Control Unit*	58
CRC	*Cyclic Redundancy Check*	65
UCF	*User Constraints File*	67

Capítulo 1

Introdução

Este Capítulo define o escopo do trabalho e o contextualiza, bem como introduz o que seja reconfiguração dinâmica, além de outras definições necessárias para o bom entendimento do mesmo.

1.1 Contextualização

Muitas aplicações emergentes em telecomunicações e multimídia necessitam que suas funcionalidades permaneçam flexíveis mesmo depois do sistema ter sido manufaturado [HAD95]. Tal flexibilidade é fundamental, uma vez que requisitos dos usuários, características dos sistemas, padrões e protocolos podem mudar durante a vida do produto. Essa maleabilidade também pode prover novas abordagens de implementação voltadas para ganhos de desempenho, redução dos custos do sistema ou redução do consumo geral de energia.

A flexibilidade funcional é comumente obtida através de atualizações de software, mas desta forma a mudança é limitada somente à parte programável dos sistemas. Desenvolvimentos recentes na tecnologia de matrizes de elementos lógicos programáveis no campo (*Field-Programmable Gate Arrays*, ou FPGAs) têm introduzido suporte para modificações rápidas e em tempo de execução do hardware do sistema [XIL00].

Essas modificações referem-se a mudanças em circuitos digitais via reconfiguração com ou sem a interrupção da operação do circuito. A implementação de sistemas que exigem flexibilidade, alto desempenho, alta taxa de transferência de dados e eficiência no consumo de energia são possibilitadas por essas tecnologias. Isto inclui aplicações de televisão digital, comunicação sem fio reconfigurável, sistemas de computação de alto desempenho, processamento de imagens em tempo real, produtos para consumo atualizáveis remotamente, entre outros.

Além das características citadas acima, a reconfigurabilidade também contribui para a economia de recursos: quando uma dada tarefa pode ser realizada em várias fases, uma diferente configuração pode ser carregada para cada fase seqüencialmente [VIL97], [DEH00]. Desta forma o tamanho do sistema pode ser menor, o que implica na redução de preço. Reconfigurabilidade também faz do desenvolvimento e teste de hardware tarefas mais rápidas e mais baratas. E pode ainda ser usada como tecnologia para construção de sistemas tolerantes a falhas: tais sistemas podem realizar auto-verificação e reconfigurar a si mesmos, substituindo elementos defeituosos por elementos reserva disponíveis [SAN99].

Este trabalho insere-se no contexto de que há necessidade de se identificar dispositivos que permitam a implementação de sistemas digitais reconfiguráveis (SDR), e documentar a seu respeito.

1.2 Definições

Por ser uma área relativamente nova, a computação reconfigurável introduz alguns neologismos e altera o significado de algumas expressões. Assim sendo, a seguir são apresentados conceitos, palavras e expressões relevantes ao entendimento deste trabalho.

FPGA: Dispositivo que consiste de uma matriz de blocos lógicos, cercada de blocos de entrada e saída, e conectada por fios de interconexão, onde todos esses itens são configuráveis.

Configuração: Programação dos elementos internos do FPGA, através de bits de memória ou tecnologia do tipo anti-fusível.

Granularidade: Característica do dispositivo ou sistema relacionada com o grão; sendo que entende-se por grão a menor unidade configurável da qual é composta um dispositivo (FPGA), ou um SDR que pode conter diversos FPGAs.

Grão-grande: Os FPGAs de grão grande podem possuir como grão unidades lógicas e aritméticas (ULAs) e/ou pequenos microprocessadores e memórias. Como exemplos desse tipo de arquitetura podem ser citadas as máquinas RAW [WAI97] e a arquitetura GARP [CAL00].

Grão-médio: Os blocos lógicos dos FPGAs que têm grão médio freqüentemente contêm duas ou mais tabelas-verdade (*look-up tables* ou LUTs) e dois ou

mais *flip-flops*. A maioria das arquiteturas de FPGAs implementa a lógica em LUTs de quatro entradas. Como exemplos de dispositivos que atualmente possuem grão médio podem ser citados as famílias Spartan e Virtex, da Xilinx; Flex e Apex, da Altera; e AT40K, da Atmel.

Grão-pequeno: Nos dispositivos com grão pequeno há um grande número de blocos lógicos simples. Os blocos lógicos normalmente contêm uma função lógica de duas entradas ou um multiplexador 4 para 1 e um *flip-flop*. As famílias SPGA (Actel) e AT6000 (Atmel) são exemplos de dispositivos de grão pequeno.

Reconfiguração total: É a forma de configuração onde o dispositivo reconfigurável é inteiramente alterado. Também tratada apenas como configuração.

Reconfiguração parcial: É a forma de configuração que permite que somente uma porção do sistema reconfigurável seja reconfigurada. Uma reconfiguração parcial pode ser não-disruptiva - onde as porções do sistema que não estão sendo reconfiguradas permanecem completamente funcionais durante o ciclo de reconfiguração; ou disruptiva - onde a reconfiguração parcial afeta outras partes do sistema, tipicamente necessitando de uma parada no sistema inteiro.

Reconfiguração dinâmica: Também chamada de *run-time reconfiguration* (RTR), *on-the-fly reconfiguration* ou *in-circuit reconfiguration*. Todas essas expressões podem ser traduzidas também como reconfiguração em tempo de execução. Não há necessidade de reiniciar o circuito ou remover elementos reconfiguráveis para programação.

Reconfiguração extrínseca: O sistema pode ser reconfigurado parcialmente, mas somente considerando cada FPGA que o compõe como unidade atômica de reconfiguração. O sistema FireFly [SAN99] pode ser citado como exemplo.

Reconfiguração intrínseca: Cada FPGA que compõe o sistema pode ser reconfigurado parcialmente. Por exemplo podem ser citados os FPGAs das famílias Virtex [XIL00a] (Xilinx) e AT40K [ATM00] (da Atmel).

DPGA: *Dynamically Programmable Gate Array* [DEH94]. Dispositivo que utiliza a técnica de implementação do circuito de controle de configuração

que permite a existência de múltiplas configurações simultaneamente carregadas, usando uma delas ativa em em um determinado instante.

Cache-Logic: É um termo usado para indicar hardware reconfigurável dinamicamente através de chaveamento de contexto, como o DPGA. É uma marca registrada da empresa Atmel [ATM00].

Core: Um *core* é um módulo de hardware, digital ou analógico, podendo ser descrito em diferentes níveis de abstração. Estes *cores* são pré-projetados, pré-verificados (por simulação funcional e *back annotation*) e prototipados em hardware pelo menos uma vez. Os cores são usados para construir aplicações maiores e mais complexas em um dado circuito integrado [VAH01].

1.3 Motivação para a pesquisa em RTR

Sistemas digiais baseados em FPGAs têm demonstrado ser uma boa alternativa entre implementações em software e em ASICS, mas ainda apresentam alguns pontos fracos. Esses pontos pontos, ou desvantagens, podem ser minimizados em função de novas pesquisas nessa área. Dentre as possibilidades de evolução dos sistemas digiais baseados em FPGAs, destaca-se a reconfiguração dinâmica de FPGAs.

As Seções seguintes mostram as atuais desvantagens dessa classe de sistemas digitais e apresentam a idéia de reconfiguração dinâmica - assuntos que motivaram esta dissertação.

1.3.1 Desvantagens de sistemas baseados em FPGAs

Os resultados iniciais dos sistemas baseados em FPGAs foram impressionantes, pela flexibilidade que possibilitam, mas principalmente pelo ganho de desempenho sobre GPPs. Contudo esses sistemas apresentam algumas limitações implícitas, tais como:

- Tempo de configuração: O tempo para carregar uma configuração no FPGA é um fator que depõe contra sua utilização em sistemas dinamicamente reconfiguráveis. Para FPGAs demostrarem ganhos sobre processadores de propósito geral, precisam amortizar o tempo de reconfiguração sobre enormes quantidades de dados, o que limita sua aplicabilidade.

- ➢ Compatibilidade em relação a novos dispositivos: Para obter benefícios de uma nova família de FPGAs, os sistemas desenvolvidos devem ser resintetizados e muitas vezes até mesmo reescritos.
- ➢ Restrições de tamanho: Atualmente FPGAs comerciais (bem como as ferramentas de CAD disponíveis) permitem que sejam implementados *cores* de tamanho limitado pelo tamanho dos recursos disponíveis no dispositivo.
- ➢ Tempo de compilação: Tarefas de síntese, roteamento e posicionamento consomem centenas de vezes mais tempo que a compilação de software para GPPs.

Outras observações podem ser feitas a partir da abstração do funcionamento de um FPGA, que pode ser visto como possuidor de duas camadas, do ponto de vista da implementação de algoritmos:

1. Camada operacional: onde tomam lugar blocos lógicos configuráveis (ou CLBs - *Configurable Logic Blocs*), blocos de entrada/saída (*Input/Output Blocks - IOBs*) e roteamento. Este tipo de arquitetura baseada em CLB é projetada para manipular dados em nível de bits.
2. Camada de configuração: pode ser vista como uma grande SRAM. A configuração é carregada nessa memória antes da fase operacional, o que caracteriza uma arquitetura estaticamente configurável.

Em vista disso, pesquisas vêm sendo realizadas no sentido de virtualizar a lógica referente à aplicação [GOL00], [DEH00], além de revisões quanto ao tamanho do grão dos dispositivos que servirão como base para tais aplicações [GOL00], [WAI97] e [SAS01]. Nesse sentido, a reconfiguração dinâmica exerce importante papel, no intuito de incrementar a flexibilidade e a agilidade na reconfiguração dos dispositivos.

1.3.2 *Run-Time-Reconfiguration*

O campo da computação reconfigurável avançou amplamente na década passada, utilizando FPGAs como a base para sistemas reprogramáveis de alto desempenho [GUC00]. Muitos desses sistemas alcançaram altos níveis de desempenho e demostraram sua aplicabilidade à resolução de uma grande variedade de problemas. Contudo, apesar dos autores desses sistemas os classificarem como reconfiguráveis, eles são tipicamente configurados uma vez antes de iniciarem a execução da aplicação.

Sistemas RTR são distintos de sistemas digitais estaticamente configuráveis por permitirem especialização da lógica e/ou do roteamento em tempo de execução. A Figura 1.1 ilustra o que seja

RTR através de um gráfico com três eixos. Os eixos x e y indicam o plano espacial, no qual ocorre o roteamento e o posicionamento da lógica a ser implementada em um FPGA. Sistemas digitais baseados em FPGAs, mas que não são dinamicamente reconfiguráveis podem ser vistos apenas neste plano. Contudo, para SDRs, há a necessidade de uma análise temporal das reconfigurações. Em vista disso foi introduzido o eixo z, que indica as diferentes implementações de módulos funcionais no dispositivo, em relação ao tempo que ocorreram.

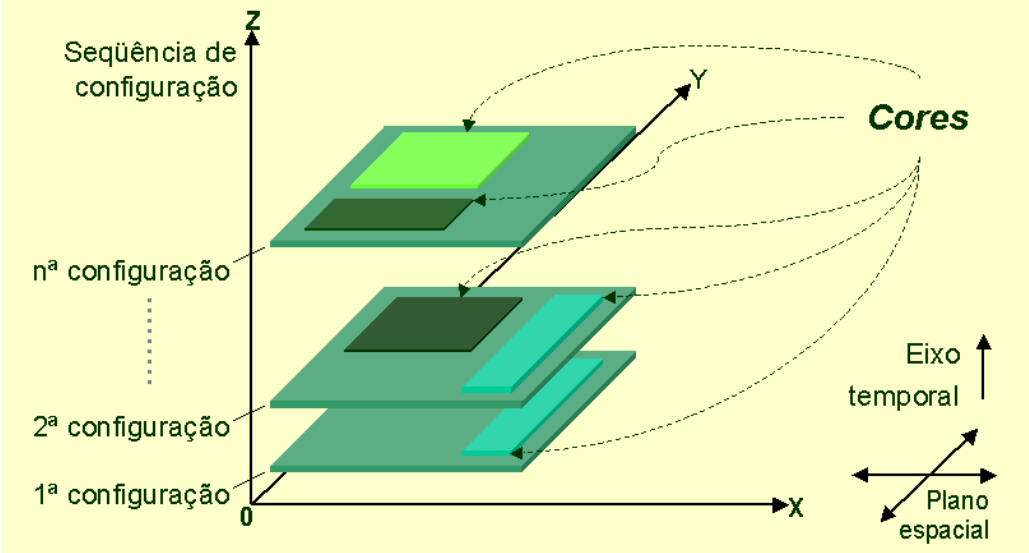


Figura 1.1: Gráfico indicativo de reconfiguração parcial não-disruptiva.

As áreas de um programa que podem ser aceleradas através do uso de hardware reconfigurável freqüentemente são muito numerosas ou complexas para serem carregadas simultaneamente no FPGA disponível. Para esses casos, é interessante que seja possível trocar entre diferentes configurações no mesmo dispositivo.

RTR cria a possibilidade de se pensar em hardware de forma semelhante ao conceito de memória virtual. Conforme essa idéia, o hardware físico pode ser muito menor que do que o somatório dos recursos requeridos para cada uma das configurações. Então, ao invés de reduzir o número de configurações que são mapeadas, apenas ocorre uma troca entre o hardware necessário e o hardware implementado fisicamente [DEH00].

Apesar da aparente vasta aplicabilidade para sistemas RTR, existem poucos sistemas que de fato implementam esta característica. Isto se dá por dois fatores principais:

1. Falta de software para projeto, depuração e teste;

2. Falta de hardware (comercialmente disponível) especialmente projetado para permitir RTR .

Essas lacunas motivam a pesquisa no desenvolvimento de técnicas de projeto para sistemas digitais parcialmente reconfiguráveis.

1.4 Objetivos

Conforme a Seção anterior, existem dois grandes problemas relacionados com o projeto e a implementação de sistemas que utilizem RTR. Este trabalho pretende contribuir para a redução de ambos.

A respeito da falta de hardware, são estudadas arquiteturas de FPGAs disponíveis comercialmente. É analisada a possibilidade de implementar um sistema digital parcialmente reconfigurável nesses dispositivos. Além disto, é proposta a criação de uma interface (lógica) para conexão entre *cores*. Interface esta que viabilizaria a virtualização do hardware, pois que facilitaria a inserção e a remoção de *cores* em um FPGA.

Relativamente à falta software, é realizado um estudo minucioso da organização interna de uma família de FPGAs. Em vista deste estudo são definidas (em linhas gerais) as necessidades para projeto de CAD para RTR nesses dispositivos. A conseqüência deste trabalho é o desenvolvimento de uma ferramenta que possibilita a reconfiguração parcial e remota de *cores*.

1.5 Organização

Para que os objetivos mencionados na Seção 1.4 sejam atingidos, este trabalho organiza-se da seguinte forma:

- ➢ O estado-da-arte em sistemas digitais reconfiguráveis é mostrado no Capítulo 2. A análise do estado-da-arte inicia-se pela definição de critérios para comparação entre as diversas arquiteturas reconfiguráveis; passa pelas propostas de hardware reconfigurável e por métodos de projeto, avaliação e implementação; e termina com a descrição de alguns sistemas digitais reconfiguráveis.

- ➢ Duas iniciativas encontradas na literatura para preenchimento da lacuna de ferramentas para desenvolvimento de sistemas digitais dinamicamente reconfiguráveis são abordadas no Capítulo 3, sendo que a segunda foi adotada para a implementação de um estudo de caso sobre reconfiguração parcial [XIL99] neste trabalho.

- ➢ O Capítulo 4 apresenta FPGAs disponíveis comercialmente que permitem reconfiguração parcial. Além disto, como este trabalho propõe a criação de uma interface para conexão de *cores*

em dispositivos da família Virtex, um estudo minucioso de sua arquitetura interna foi imprescindível. Este estudo é também mostrado no Capítulo 4.

➢ O Capítulo 5 explicita a contribuição deste trabalho para a comunidade científica. Apresenta um reconfigurador parcial e remoto de FPGAs, bem como implementa um estudo de caso em reconfiguração parcial. Sua última Seção sugere características para uma interface para conexão de *cores* FPGAs, nos moldes do que já existe para ASICs.

Este trabalho foi organizado de forma que há possibilidade de uma leitura não seqüencial, ou seja, o leitor pode iniciar por onde melhor lhe aprouver. Contudo, o Capítulo 5 é exceção, pois contém elementos compreensíveis apenas através do entendimento do Capítulo 4.

A conclusão do trabalho é feita no Capítulo 6, onde também há indicações para trabalhos futuros.

Capítulo 2

Estado-da-arte em sistemas digitais reconfiguráveis

Este Capítulo objetiva situar o leitor no que se refere ao estado-da-arte em computação reconfigurável.

Primeiramente explicita critérios de classificação de sistemas digitais reconfiguráveis, e em seguida exibe uma tabela classificando diversos SDRs conforme esses critérios. Ainda no que tange à classificação, neste Capítulo é feita análise de SDRs em relação ao tempo, à tecnologia empregada e aos tipos de problemas que procuravam endereçar.

O Capítulo 2 trata também de propostas de hardware e métodos para implementação de sistemas digitais reconfiguraveis.

2.1 Taxonomia de Arquiteturas Reconfiguráveis

Nos próximos itens são apresentados os critérios de algumas tentativas de classificação de SDRs. A taxonomia destes sistemas é complementada com a inclusão dos quesitos reconfigurabilidade e abrangência da reconfigurabilidade. A escolha destes critérios busca a elaboração de um conjunto de características de classificação cujos elementos sejam ortogonais entre si [CAL98].

2.1.1 Olimpo

Olimpo é uma proposta de taxonomia para arquiteturas reconfiguráveis baseada nos critérios de objetivo da arquitetura, granularidade, integração e reconfigurabilidade da rede de interconexão

[RAD98]. A Figura 2.1 exibe essa proposta no formato de árvore.

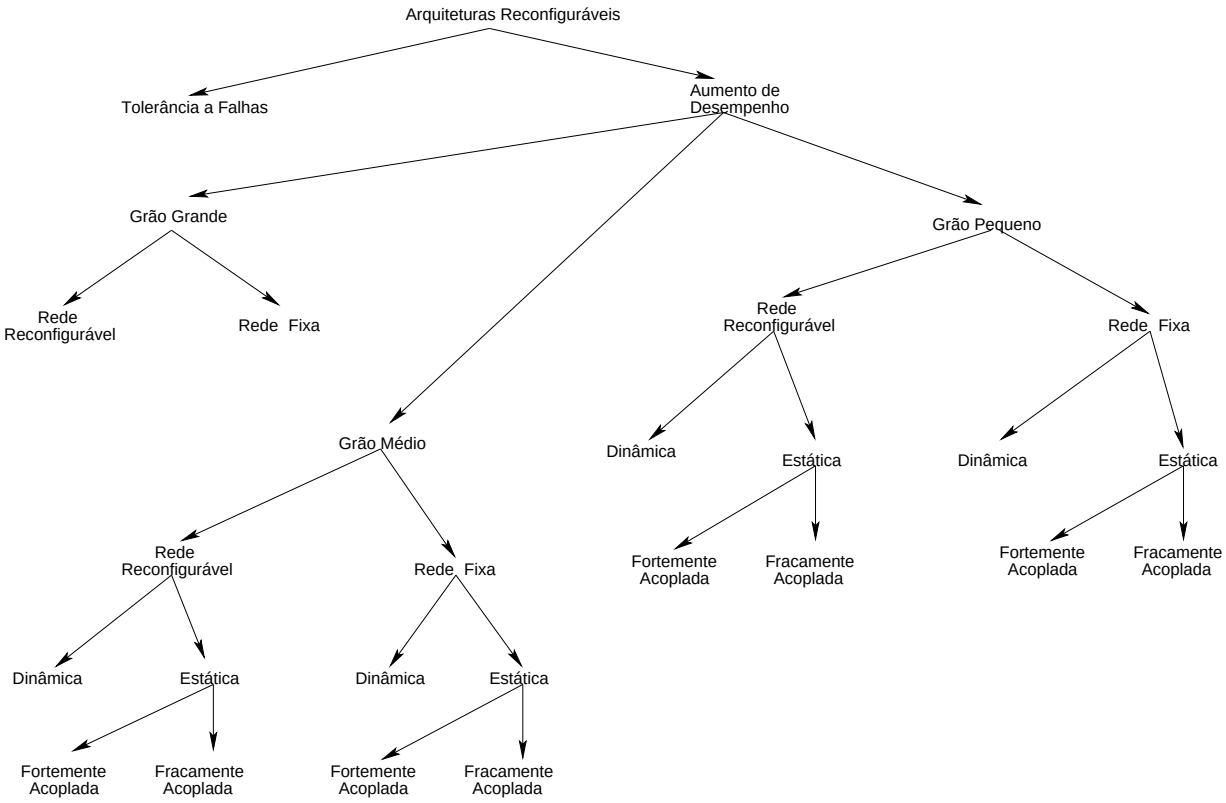


Figura 2.1: Taxonomia Olimpo.

A seguir serão descritos os critérios de classificação do Olimpo.

2.1.1.1 Objetivo da Arquitetura

Uma arquitetura reconfigurável pode ter como meta uma das duas características abaixo:

- ➢ Tolerância a falhas: É um dos primeiros campos onde a computação reconfigurável foi implementada. Durante a fabricação e a utilização, há uma certa possibilidade de que uma parte do circuito integrado torne-se defeituosa. Em arquiteturas clássicas uma parte defeituosa implica que o circuito integrado inteiro está inutilizado. Contudo, numa arquitetura reconfigurável tolerante a falhas, o sistema ainda poderia continuar operando, pois é capaz de detectar e corrigir um certo conjunto de falhas.

- ➢ Acréscimo de velocidade: O uso de arquiteturas reconfiguráveis para incrementar o desempenho de sistemas é uma tecnologia que emergiu rapidamente a partir da década de 80, principalmente pela possibilidade de implementação de certos algoritmos lentos nos atuais μPs diretamente em hardware.

2.1.1.2 Granularidade

Granularidade pode ser definida de várias maneiras, relativas a um dispositivo (e.g. circuito integrado), como por exemplo o número de funções booleanas que um bloco lógico pode implementar, o número de portas NAND de duas entradas equivalentes, o número total de transistores, a área normalizada total, ou pelo número de entradas e saídas [ROS93]. A classificação do Olimpo trata granularidade, contudo, conforme definido na Seção 1.2.

2.1.1.3 Integração

Quanto ao quesito forma de integração a classificação pode ser a seguinte:

- ➢ Sistemas Autônomos: Dada a carga inicial, o sistema adapta-se dinamicamente (e sem interferência externa) até que o resultado parcial convirja na direção de um resultado esperado. O melhor exemplo para essa característica é a simulação de sistemas vivos, por exemplo, implementação de algoritmos genéticos [SAN99].
- ➢ Sistemas fortemente acoplados: Unidades de reconfiguração atuam como unidades de execução do processador, e podem manipular dados diretamente de seus registradores, ou de um barramento comum.
- ➢ Sistemas fracamente acoplados: Unidades de reconfiguração são anexadas ao sistema como co-processadores, comumente em uma placa em separado.

2.1.1.4 Reconfigurabilidade da rede externa de interconexão

Este quesito trata da rede de interconexão entre unidades reconfiguráveis, que pode ser fixa ou reconfigurável.

- ➢ Arquitetura com rede externa reconfigurável: Uma rede reconfigurável conecta todas as unidades reconfiguráveis, criando uma grande unidade reconfigurável virtual, garantindo uma maior escalabilidade para o sistema.
- ➢ Arquitetura com rede fixa: É uma arquitetura mais barata e mais simples. Pode não ser adequada para aplicações com computações intensivas, mas é suficiente para a maioria das aplicações.

2.1.2 Critérios de Page

Hardware dinamicamente reconfigurável é a tecnologia que impulsiona o desenvolvimento de sistemas que unam as características de processadores de propósito geral (GPPs) e a flexibilidade de FPGAs. Sob este prisma, SDRs podem ser classificados quanto a sua atuação como coprocessadores, com relação à arquitetura da memória, e com relação à execução dos programas pelo sistema reconfigurável [PAG96].

2.1.2.1 Coprocessamento

Levando-se em consideração um FPGA e um GPP como descritos acima, existem diversas formas de usá-los em um sistema. Uma forma de classificar esses sistemas é através dos diferentes níveis de interação entre o FPGA e o GPP:

- ➢ Coprocessamento: O microprocessador lança uma instrução, ou seqüência de instruções que pode ser detectada e interpretada pelo FPGA. Na realidade, o FPGA atuaria como um coprocessador.

- ➢ Chamada remota de funções: O microprocessador lança uma instrução, ou seqüência de instruções que é interpretada pelo FPGA como uma chamada remota de função (*Remote Procedure Call* - RPC). É semelhante ao coprocessamento, exceto pelo fato que é uma interface mais poderosa que usa sincronização explícita sempre que necessário.

- ➢ Modelo Cliente-Servidor: O algoritmo implementado no FPGA é um processo servidor que atua de forma semelhante ao mecanismo do RPC, mas onde comunicações podem chegar de qualquer processo que esteja sendo executado pelo microprocessador.

- ➢ Processos paralelos: Os processos que são executados pelo FPGA são independentes daqueles que são executados pelo GPP. A comunicação entre processos pode acontecer a qualquer momento, via troca de mensagens.

2.1.2.2 Arquitetura de memória

O programa que está sendo executado no coprocessador FPGA normalmente precisa de certa quantidade de variáveis e de armazenamento temporário para sua operação. Assim, pode haver três diferentes modelos de memória:

- ➢ FPGA sem acesso a memória externa: Em alguns casos o algoritmo executado no FPGA pode operar sem memória externa. É uma situação aceitável somente quando o FPGA necessita de poucos estados para sua operação.

- ➢ FPGA compartilha memória com o microprocessador: O FPGA pode usar qualquer memória associada ao barramento por ele compartilhado. Contudo isto gera sobrecarga de tempo e espaço para que seja negociada a arbitragem de acesso a memória. Pode ser necessária a existência de um controlador de endereçamento, o que tenderá a reduzir a velocidade de acesso a memória para processamento intensivo de dados.

- ➢ FPGA com memória local própria: O algoritmo da parte reconfigurável é organizado com memória privada, consumindo o mínimo de ciclos de relógio para cada acesso, aumentando o desempenho durante um processamento. Contudo pode haver duplicação de dados com relação ao GPP. Por este motivo é também necessário um controle de coerência e um meio para troca de mensagens.

2.1.2.3 Forma de operação da memória local

Os modelos de operação de memória não são necessariamente mutuamente exclusivos, isto é, podem não ser ortogonais [PAG96].

- ➢ Memória de bloco: A memória local é suficientemente grande para conter um conjunto de dados completo para o processamento. Por exemplo este poderia ser um quadro de vídeo completo a ser processado por um algoritmo de compressão de imagens.

- ➢ Memória de fila: A memória age como uma fila enquanto há troca de dados e resultados com o microprocessador. Como exemplo tem-se o armazenamento de uma região completa de suporte para um filtro de tempo real.

- ➢ *Cache*: A memória local pode funcionar como uma *cache* sobre uma estrutura de dados maior controlada pelo GPP.

2.1.2.4 Execução dos programas

Existem também diversas formas de os módulos funcionais serem configurados no FPGA. Estes modelos podem explorar diferentes partes da relação custo/desempenho das implementações para suportar execução de programas no FPGA.

- ➢ Hardware puro: O algoritmo é convertido (por síntese para hardware) em uma descrição de hardware que é carregada no FPGA.

- ➢ Microprocessador de aplicações específicas: O algoritmo é compilado em um código de máquina abstrato, para um processador abstrato (*Application Specific Instruction-Set Processor* - ASIP). Os dois são então co-otimizados para produzir a descrição de um processador de aplicação específica e o código de máquina para ele. A descrição do microprocessador pode então ser compilada em um FPGA.

- ➢ Reutilização seqüencial: O algoritmo pode ser muito grande para ser implementado no FPGA, ou por razões de engenharia ou econômicas divide-se o algoritmo em partes, de tal forma que ele seja executado parcialmente, usando a capacidade de reconfiguração dinâmica do FPGA. Os ganhos relacionados com a reutilização do hardware devem ser balanceados com o tempo que é gasto com a reconfiguração.

- ➢ Uso múltiplo simultâneo: Se os recursos do FPGA são grandes o bastante, é possível haver um número de algoritmos co-residentes, e cada um deles pode interagir separadamente com o processador.

- ➢ Uso sob demanda: Existe a possibilidade de sistemas computacionais serem construídos onde o hardware não existe todo ao mesmo tempo, mas cuja demanda de tempo-real do sistema dita qual parte do hardware deve ser construída e qual parte deve ser destruída. Há uma analogia razoável com sistemas de memória virtual, e por isto esse esquema pode ser chamado de "hardware virtual".

2.1.3 Critérios de Sanchez

Um SDR pode ser configurado de duas formas: estática ou dinamicamente [SAN99]. Na primeira o arquivo de configuração de um FPGA é carregado somente uma vez antes da execução de uma tarefa, e o FPGA será reconfigurado somente depois desta execução. Na segunda, dinâmica, o arquivo de configuração do FPGA pode mudar parte do dispositivo a qualquer momento. Portanto, pode-se adotar os seguintes critérios de classificação para SDRs:

- ➢ Sistema estático: Possui pouca flexibilidade. A configuração do sistema ocorre antes do início da execução da tarefa a que se propõe executar. Durante toda execução o FPGA permanece inalterado. Após, pode ser reconfigurado totalmente. É o mais comumente utilizado.

➢ Sistema dinâmico: Envolve um a arquivo de configuração que pode mudar durante a execução de uma tarefa pelo FPGA. Tem como objetivos adaptação dinâmica a mudanças de especificação, bem como manipulação de especificações incompletas.

2.1.4 Principais SDRs classificados pelos critérios vistos

O propósito desta Subseção é comparar, através da Tabela 2.1, algumas das mais importantes arquiteturas na área de computação reconfigurável. Foram utilizados os critérios analisados previamente para prover a classificação, por serem aqueles ortogonais o bastante para permitir clareza na comparação entre os sistemas.

2.2 Revisão de alguns sistemas digitais reconfiguráveis

As arquiteturas reconfiguráveis podem ser analizadas temporalmente, em função dos problemas a que se dispuseram resolver. A partir do amadurecimento da tecnologia habilitadora para esses sistemas (FPGAs), alguns centros de pesquisa criaram as primeiras arquiteturas reconfiguráveis, com o intuito principal de aumentar o desempenho de algoritmos que até então eram executados em software. Dentro desta primeira geração estão projetos como DECPeRLe [BER96], PRISM [ATH93] e Splash [GOK90]. Alguns sistemas mais modernos ainda utilizam essa abordagem, como o Transmogrifier-2 [LEW98], o RPM-2 [DUB98] e o SPYDER [SAN99]. Tais sistemas podem ser vistos como a primeira geração dos sistemas digitais reconfiguráveis, conforme a Figura 2.2 à esquerda.

Já neste primeiro momento verificou-se a eficiência da utilização de FPGAs em domínios de aplicação específicos, tanto em termos de desempenho com relação a abordagens em software quanto no que tange ao critério econômico, quando comparada a soluções ASIC. Contudo também alguns problemas foram detectados. Em geral esses sistemas possuíam um gargalo de comunicação entre μP e FPGAs e apresentavam um tempo de reconfiguração muito alto, além de poderem ser reconfigurados apenas totalmente. Essa última desvantagem significa que o sistema precisava necessariamente ser parado para que pudesse ser reconfigurado.

Em função disso novas propostas de arquiteturas surgiram. O problema de comunicação entre μP e FPGAs contou com o avanço da tecnologia habilitadora para ser resolvido, formando uma segunda geração de SDRs (Figura 2.2, centro). Com o aumento do número de transistores por circuito integrado (CI) tornou-se possível o desenvolvimento de um sistema composto por μP, FPGA(s) e memória

Tabela 2.1: Classificação de sistemas reconfiguráveis

Critério / Sistema	Grão	Integração	Rede externa	Coprocessamento	Memória / Arquitetura	Memória / Operação	Execução dos programas	Reconfiguração
DECPeRLe [BER96]	pequeno	fracamente acoplado	fixa	coprocessador	compartilhada	n/a	hardware puro	estática
DISC [WIR95]	pequeno	fortemente acoplado	fixa	processador	compartilhada	cache	reutilização	dinâmica
DPGA [DEH94]	médio	fortemente acoplado	reconfigurável	coprocessador	própria / local	cache	uso múltiplo sim.	dinâmica
FIPSOC [SID99]	pequeno	fortemente acoplado	fixa	processo paralelo	compartilhada	cache	μP de aplicação	dinâmica
FireFly[SAN99]	pequeno	autônomo	fixa	processador	própria / local	bloco	hardware puro	dinâmica
Garp [CAL00]	médio	fortemente acoplado	fixa	coprocessador	compartilhada	fila	μP de aplicação	dinâmica
PRISM [ATH93]	pequeno	fracamente acoplado	fixa	cliente / servidor	compartilhada	fila	μP de aplicação	estática
RAW [WAI97]	grande	fracamente acoplado	reconfigurável	coprocessador	própria / local	bloco	μP de aplicação	estática
RPM-2 [DUB98]	pequeno	fracamente acoplado	fixa	processador	própria / local	bloco	hardware puro	estática
Splash [GOK90]	pequeno	fracamente acoplado	fixa	processador	compartilhada	fila	μP de aplicação	estática
Splash2 [ARN92]	fino	fracamente acoplado	reconfigurável	RPC	própria / local	bloco	μP de aplicação	estática
SPYDER[SID99]	fino	fortemente acoplado	fixa	coprocessador	compartilhada	n/a	μP de aplicação	estática
Transmogrifier [LEW98]	pequeno	fracamente acoplado	fixa	cliente / servidor	própria / local	bloco	μP de aplicação	estática
Trumpet [PER99]	pequeno	fortemente acoplado	fixa	processo paralelo	própria / local	cache	μP de aplicação	dinâmica

em um único CI (*System-on-a-Chip*, ou simplesmente SoC). Foram desenvolvidos SoCs de granularidade baixa (FIPSoC [SID99], TRUMPET [PER99]) e com granularidade média (Garp [CAL00], RAW [WAI97]).

Outro avanço tecnológico ocorrido foi a possibilidade de reconfiguração dinâmica. Isto permitiu que as arquiteturas em questão pudessem ser reconfiguradas sem que precisassem parar totalmente de desempenhar suas funções. Essa reconfiguração dinâmica pode ser realizada por chaveamento de contexto, como o que ocorre com os sistemas derivados do DPGA [DEH94]. Este também é o caso do DISC, FireFly, FIPSoC e Garp. Tais sistemas encontram-se na segunda geração de SDRs, conforme a Figura 2.2. Splash2 apresenta uma reconfiguração alternativa, na sua rede de interconexão externa [ARN92] .

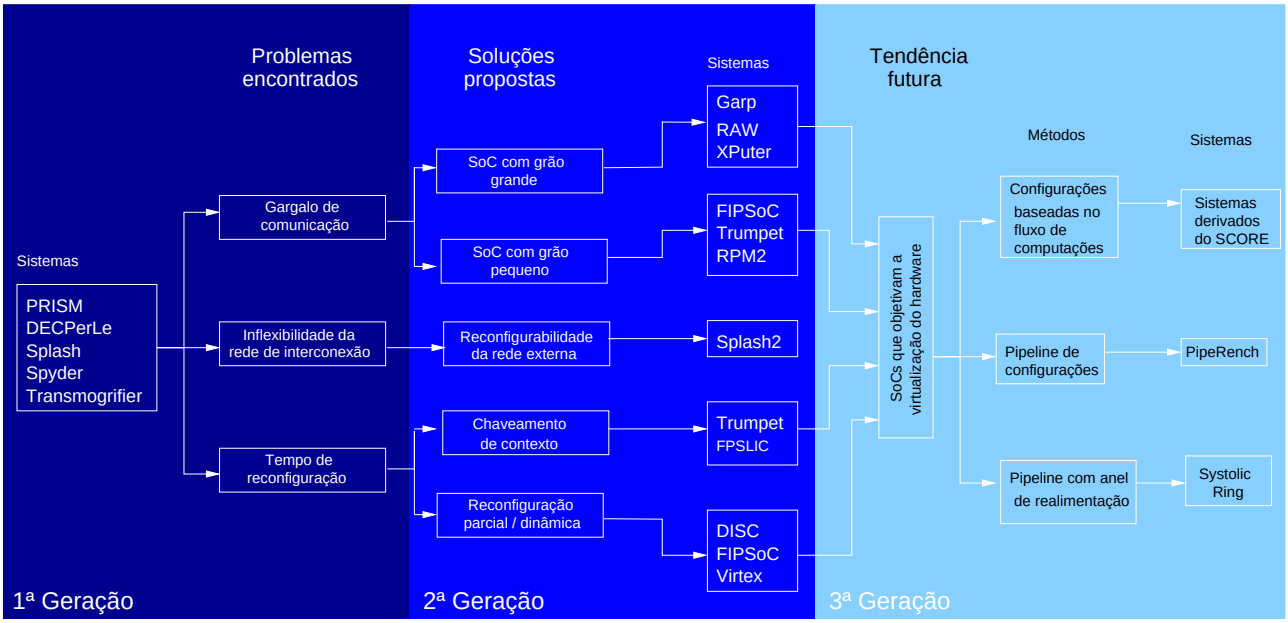


Figura 2.2: Evolução das arquiteturas reconfiguráveis.

Além das abordagens derivadas de avanços tecnológicos e resolução de antigos problemas, arquiteturas reconfiguráveis têm sido utilizadas para implementação de algoritmos genéticos. FireFly é um exemplo de "hardware evolutivo" criado a partir desse enfoque. Demonstra uma utilização exótica de reconfiguração parcial e dinâmica, embora ambas ocorram de forma extrínseca.

Contudo, analisando as características da segunda geração, e comparando-as a trabalhos em desenvolvimento, percebe-se a tendência de especialização das aplicações-alvo (*workload*) para algoritmos baseados em fluxo de dados. Uma necessidade emergente para computação reconfigurável são aplicações multimídia, que demandam certos tipos de operações que não são facilmente implementadas (ou não são executadas eficientemente) em FPGAs convencionais (multiplicadores, por exemplo). Além disso há necessidade de preservação da aplicação, a fim de torná-la menos dependente do dispositivo para o qual foi projetada. A isto denomina-se virtualização do hardware e é detalhada no Item 2.1.2.4. Essa virtualização pode ocorrer através de métodos diferentes, tais como:

- ➢ Configurações baseadas no fluxo de computações [DEH00];
- ➢ Por um *pipeline* de configurações [GOL00];
- ➢ Reconfigurações em função de um anel de retroalimentação [SAS01].

A Figura 2.2-direita mostra essa tendência e cita exemplos de dispositivos que implementam as idéias citadas acima, evidenciando uma terceira geração de sistemas digitais reconfiguráveis.

Informações adicionais a respeito de arquiteturas reconfiguráveis totalmente funcionais podem ser encontradas no site do pesquisador Steve Guccione [GUC00].

É importante ressaltar que esta classificação temporal para SDRs é uma contribuição do presente trabalho.

2.3 Propostas de hardware para reconfiguração parcial

As Seções a seguir abordam as características que um dispositivo reconfigurável deve ter para permitir reconfiguração dinâmica. O primeiro item aborda reconfiguração por chaveamento de contexto, o segundo traz o conceito de SoC dinamicamente reconfigurável e o terceiro trata da questão da regularidade do dispositivo e de características como relocação e desfragmentação.

2.3.1 DPGA

A organização interna do DPGA é particularmente bem adaptada para computação reconfigurável. Diferentemente de FPGAs normais, onde a função de cada elemento da matriz é determinada por seqüências relativamente lentas de reconfiguração, os elementos da matriz DPGA podem chavear rapidamente[1] entre várias configurações pré-programadas [DEH94].

Semelhantemente aos FPGAs, DPGAs são compostos de uma cadeia de elementos computacionalmente simples. Muitos FPGAs modernos são construídos como LUTs programáveis. A LUT também é o elemento configurável do DPGA. Em um FPGA comum, após a configuração, o roteamento permanece fixo durante o funcionamento do sistema. No DPGA, ao contrário, há uma rede de interconexão também programável, que permite aos elementos serem ligados conforme a aplicação necessita. Cada elemento do DPGA usa uma segunda LUT para armazenar um diferente contexto. Uma mensagem de configuração global informa a cada elemento do DPGA qual a função que irá desempenhar no próximo ciclo do relógio. A interconexão configurável em um DPGA tem uma tabela de configurações carregadas, e seleciona entre as configurações baseada no identificador de contexto corrente.

Ao custo de um tamanho maior de elementos e interconexão, o DPGA serve como um FPGA de múltiplo contexto. Com o espaço dos contextos pré-carregados, o DPGA pode chavear entre funcionalidades completamente diferentes, de um ciclo de relógio para outro. A tabela de configuração do DPGA efetivamente atua como uma (memória) *cache* de configurações de elementos da matriz. Dan-

[1] Rapidamente, neste contexto, significa em apenas um ciclo de relógio.

do a cada elemento uma pequena cache de configuração estreitamente acoplada a ele, consegue-se efetivamente uma taxa de reconfiguração alta.

Os múltiplos contextos carregados permitem a utilização mais eficiente da matriz de elementos. Em aplicações mais pesadas, a rápida reconfiguração permite a uma única matriz DPGA ser carregada com múltiplas configurações simultaneamente. O DPGA pode chavear entre configurações para acelerar diferentes partes de uma aplicação. Essa tecnologia foi utilizada para implementação do Trumpet [PER99].

2.3.2 FIPSoC

FIPSoC é um dispositivo do tipo *Sistem-on-a-Chip* criado pela empresa SIDSA, e tem por objetivo o desenvolvimento rápido de aplicações analógicas e digitais integradas [SID99]. A Figura 2.3 mostra que o FIPSoC possui um microcontrolador 8051 embutido, um FPGA, e um conjunto de células analógicas otimizadas para aquisição de sinais e conversões A/D e D/A. A SIDSA disponibiliza várias ferramentas de CAD integradas, que possibilitam ao usuário especificar, simular e mapear todo o projeto em apenas um circuito integrado. A lógica programável do FIPSoC é uma matriz de macro células digitais (*Digital Macro Cells* - DMCs) que podem ser configuradas para funções específicas. As DMCs são de grão pequeno, baseadas em células de RAM estática programáveis, que incluem LUTs de 4 entradas e 4 flip-flops para cada uma. O subsistema analógico consiste de blocos analógicos configuráveis (*Configurable Analog Blocks*) de grão grande[2]. Os CABs permitem configurar diferentes funções analógicas, tais como amplificação diferencial e conversão de dados. Existe uma parte digital das células analógicas que pode ser independentemente controlada pelo μP ou pelo FPGA.

A configuração do circuito integrado é armazenada em bits de RAM estática. Pode ser feito o *download* de uma nova configuração para a memória de configuração enquanto a célula está em operação (semelhante ao cache de configurações do DPGA). Não há a necessidade de parar o circuito integrado para reconfigurá-lo: configurações extras podem ser carregadas em tempo-real, dentro da idéia de reconfiguração parcial.

Por exemplo, enquanto uma parte do FIPSoC realiza uma recuperação de contexto em uma determinada área, outra parte do dispositivo pode estar fazendo uma computação de propósito geral. Esta reconfiguração dinâmica, parcial ou total, pode também ser colocada em funcionamento pelo próprio hardware reconfigurável, sem a intervenção do μP. A reconfiguração dinâmica pode ser aplicada sobre uma única célula digital configurável, sobre um conjunto selecionado delas, ou sobre toda a lógica reconfigurável.

[2]Em relação aos blocos analógicos, grão grande significa um amplificador operacional, por exemplo.

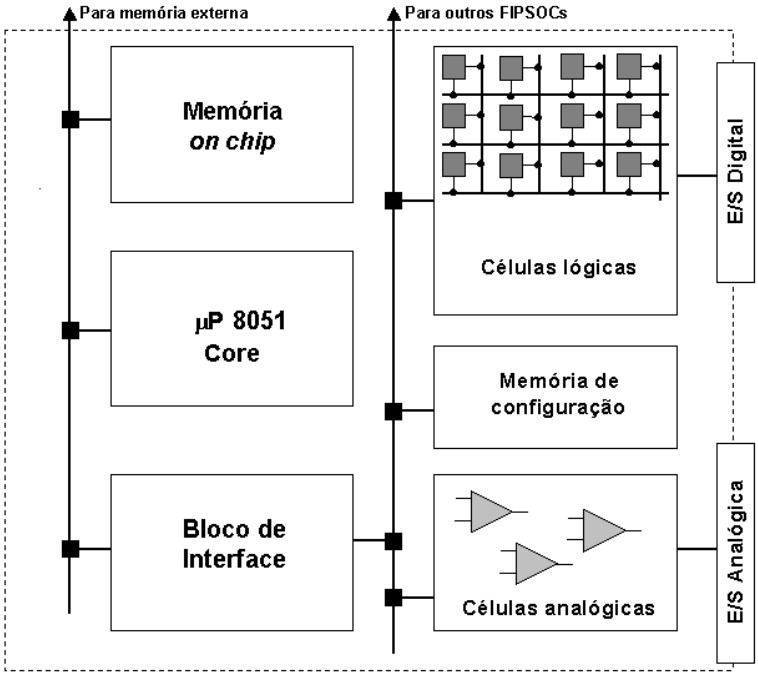


Figura 2.3: Esquema interno em alto nível do FIPSoC.

2.3.3 Modelo para suporte a RTR de Compton

Este modelo trata de uma proposta de FPGA que permite reconfiguração em tempo de execução, com ênfase na otimização de relocação e desfragmentação dos recursos do dispositivo [COM99].

Um dos problemas dos sistemas que permitem RTR ocorre quando duas configurações parciais são sobrepostas durante a compilação na mesma posição física no FPGA. Uma solução para este problema é permitir que o posicionamento final das configurações ocorra em tempo de execução, de forma que aconteça a relocação dinâmica dessas configurações. Utilizando relocação, uma nova configuração pode ser posicionada no FPGA onde ela causar menos conflitos com outras configurações já presentes no dispositivo. A Figura 2.4 ilustra uma situação onde relocação e desfragmentação são utilizadas. O primeiro desenho (Figura 2.4-1) mostra o estado inicial do FPGA, onde os diferentes tons de cinza indicam diferentes módulos funcionais, e a parte em branco significa recursos de lógica não utilizados (livres). Observe-se que há uma área branca entre as áreas hachuradas. O segundo desenho mostra o início da desfragmentação (Figura 2.4-2). A lógica que encontrava-se na quinta linha do dispositivo é transferida para uma memória auxiliar, para, em seguida (Figura 2.4-3), ser escrita na linha quatro. Ação semelhante ocorre com a lógica que encontra-se na linha 6 (Figura 2.4-4). Então, a Figura 2.4-5 denota um dispositivo cuja lógica foi relocada, a fim de prover a desfragmentação.

Mesmo com a relocação, dispositivos que permitam RTR podem ainda sofrer com alguns conflitos de posicionamento. Mas estes podem ser evitados usando uma otimização de hardware adicional

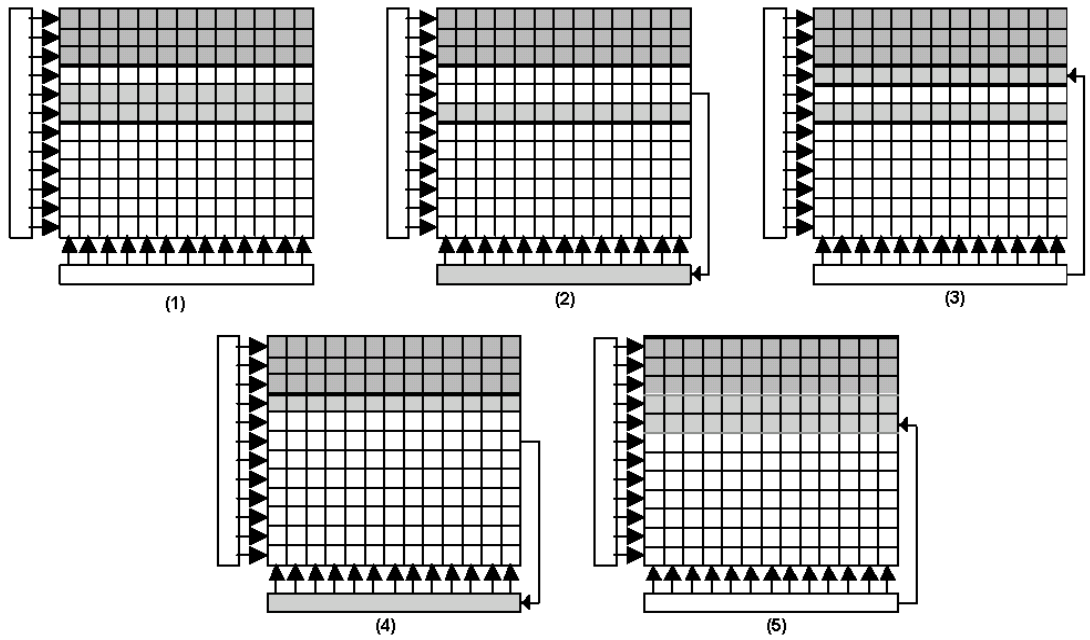


Figura 2.4: Exemplo de relocação e desfragmentação.

[COM99]. Como dispositivos que permitem RTR carregam e descarregam configurações no decorrer do tempo, a localização da área desocupada no FPGA é fragmentada, de forma semelhante ao que ocorre com a memória de programas e dados de um sistema computacional genérico. Mesmo que haja área vazia suficiente no FPGA para receber uma determinada configuração, pode acontecer que esta esteja distribuída pelo dispositivo. Uma configuração normalmente requer uma área contígua, então acabará tendo que sobrescrever uma área do CI que contém uma configuração válida. Um dispositivo que pretenda permitir RTR deve incorporar a possibilidade de realizar desfragmentação, consolidando áreas não-utilizadas movendo configurações válidas para novas áreas, disponibilizando assim as áreas vazias.

Uma nova arquitetura, baseada nos princípios de relocação e desfragmentação foi proposta, e chamada de R/D FPGA [COM99].

Usando alguns poucos e simples conceitos na fase de projeto de um FPGA, pode ser assegurado que ele permita relocação e desfragmentação. Tais conceitos são:

1. O primeiro é o conceito de configuração parcial, já citado na Seção 1.2. A proposta do R/D FPGA é baseada num *core* genérico parcialmente configurável. A Figura 2.5 (a) mostra um dispositivo cuja lógica programável é composta por elementos configuráveis individualmente, o que permite a reconfiguração parcial.

2. A segunda idéia é a de homogeneidade. Se cada célula na estrutura for idêntica, não haverá obstáculos funcionais para mover uma configuração de uma posição para outra dentro dos limi-

tes do dispositivo. Da mesma forma, a estrutura de roteamento deve ser homogênea para evitar limitações de posicionamento.

3. O terceiro conceito é o de Entrada/Saída (E/S) virtualizada. Uma estrutura de E/S baseada em barramento fornece um método independente de posição para ler e escrever dados de configurações individuais. As configurações então não são limitadas por restrições de E/S evitando a preocupação com o posicionamento dos pinos externos do FPGA. Com isto o roteamento de E/S permanece inalterado quando a configuração é mapeada para uma nova localização.

4. A quarta idéia é a de unidimensionalidade. FPGAs comerciais atuais são baseados em uma estrutura bidimensional. O movimento de configurações em duas dimensões para relocação e desfragmentação pode ser muito difícil, pois há muitas possibilidades diferentes de posicionamento a serem consideradas. Uma estrutura em linhas, onde cada linha seria uma estrutura atômica para configuração do FPGA, removeria a complexidade da relocação e desfragmentação. Isto se dá porque reduz as configurações em objetos unidimensionais, onde a única variação da área de configuração permitida é no número de linhas utilizadas. Operações de rotação horizontal ou vertical, ou deslocamento horizontal não são mais necessárias. A única operação requerida para relocação de uma configuração é a mudança no deslocamento vertical.

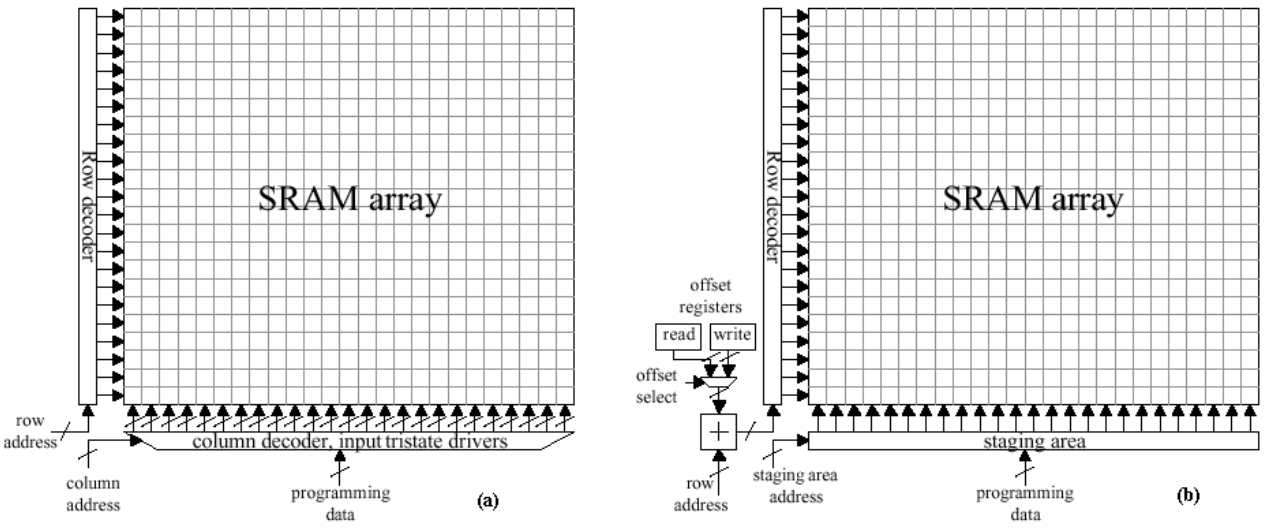


Figura 2.5: Um FPGA com arquitetura básica para permitir RTR (a), e a arquitetura do R/D FPGA (b).

Por causa da unidimensionalidade, a E/S virtualizada também é simplificada. Ao invés de incluir fios de E/S ao longo de cada coluna e de cada linha do FPGA, esses fios são necessários somente para cada coluna. Apesar da abordagem 2D proporcionar maior flexibilidade de roteamento, acaba por

complicar os programas de roteamento e posicionamento. Numa estrutura de roteamento unidimensional, o posicionamento dos *cores* é restrito ao longo de apenas um eixo, o que faz com este ocorra muito mais rápido.

2.3.3.1 Arquitetura do R/D FPGA

A lógica programável do R/D FPGA é composta por uma matriz de bits de SRAM. Esses bits são habilitados para leitura/escrita através da decodificação do endereço de linha pelos dados de programação. Contudo, a coluna do decodificador, multiplexador e *drivers tri-state* de entrada foi substituída por uma estrutura denominada área de preparação (*staging*), conforme Figura 2.5 (b).

Essa área de preparação é uma pequena área de armazenamento temporário (*buffer*), o qual é essencialmente um conjunto de células de memória igual ao número de uma linha de programação do FPGA. Cada linha, e por conseguinte a área de preparação, contém palavras de dados para configuração. Uma vez que a informação para a linha esteja completa na área de preparação, a área inteira é escrita em uma operação na memória de configuração do FPGA, conforme o endereço contido na linha de endereços. Essa estrutura é semelhante à proposta implementada nos FPGAs da família Virtex da Xilinx.

Na área de preparação do R/D FPGA há um pequeno decodificador que habilita leituras/escritas endereçáveis. A coluna de decodificação determina qual das palavras na área de preparação está sendo referenciada num dado momento.

O decodificador de linha inclui uma sutil modificação, qual seja, a adição de dois registradores, um multiplexador 2 para 1 para escolher entre esses registradores, e um somador. Todas essas estruturas são iguais em largura ao endereço da linha. Isto permite que um deslocamento vertical seja carregado em um ou mais registradores para ser adicionado ao novo endereço de linha, o que resulta num novo endereço de linha relocado. Um dos registradores é o registrador de escrita de deslocamento, o qual armazena o deslocamento de relocação enquanto uma configuração é escrita. O outro é o registrador de leitura, que é usado durante a desfragmentação para ler do FPGA uma configuração relocada.

O ponto principal da arquitetura proposta é a possibilidade de posicionar a lógica em qualquer parte do FPGA, independentemente dos pinos de E/S que são necessários para comunicação com o mundo externo. Somado a isto, a relocação e a desfragmentação ocorrem de forma simplificada, criando o ambiente ideal para implementação de sistemas RTR.

2.4 Métodos para projeto / implementação de SDR

Esta Seção aborda métodos utilizados no projeto e/ou na implementação de sistemas digitais reconfiguráveis. O Item 2.4.1 propõe uma métrica para avaliação de sistemas RTR, a fim de analisar a viabilidade de implementação de um sistema assim perante um estático. Já o Item 2.4.2 traz um *framework* de uma ferramenta para projeto de planta-baixa temporal, que viabilizaria o projeto de sistemas RTR.

Os itens 2.4.3 e 2.4.4 apresentam diferentes enfoques relativos a métodos para projeto de sistemas com características de reconfiguração dinâmica.

2.4.1 Método para projeto de sistemas RTR

Pesquisadores do departamento de engenharia elétrica da Brigham Young University desenvolveram o protótipo de um sistema de rede neural baseada em RTR parcial [HAD95]. O sistema foi chamado de RRANN-2 (*Run-time Rreconfigurable Artificial Neural Network*). Durante o projeto e implementação do RRANN-2, foi desenvolvido um método de projeto para sistemas RTR.

O ponto mais importante deste método é maximizar a lógica que permanecerá inalterada (estática) e minimizar a lógica que precisa ser modificada durante a execução da aplicação (dinâmica). Esta maximização dá-se pelo particionamento da aplicação em blocos funcionais que são, em sua maior parte, comuns a todas configurações utilizadas na implementação da aplicação. Esses blocos representam as partes da configuração que não mudam, e, portanto, podem ser implementadas como um circuito estático. O projeto de recursos para reconfiguração dinâmica só ocorrerá quando não houver unidade funcional correspondente entre a configuração implementada e a que será utilizada. Adicionalmente, a parte estática do circuito é utilizada para reter estados intermediários do dispositivo reconfigurável. Isto elimina circuitos de controle e roteamento que seriam necessários para armazenar valores.

O segundo passo no desenvolvimento de um projeto de RTR parcial é mapear fisicamente os blocos no dispositivo. Além das restrições de implementação e localização necessárias, um bloco lógico também é restringido pelo contexto físico que o cerca. Todos os assuntos referentes a interconexão global, localização global, e posicionamento e interconexão com os blocos próximos têm que ser resolvidos. O problema é que muitas dessas restrições não são conhecidas em tempo de projeto, o que vem a requerer um difícil processo interativo.

Os autores deste método ressaltam a necessidade de pesquisa contínua nessa área, para cristalizar conhecimento a respeito de uma metodologia de projeto para sistemas RTR. Também citam a falta de CAD eficiente para suportar definições de restrições, posicionamento e roteamento.

2.4.1.1 Métrica para avaliação de sistemas RTR

Os aumentos na eficiência providos através de RTR não são obtidos sem custo [WIR98]. Tempo e largura de banda de memória adicionais são necessários para transferir os bits de configuração do circuito de uma unidade de armazenamento fora do FPGA para dentro da memória de configuração do dispositivo. Em alguns casos esse tempo extra elimina as vantagens de uma especialização em tempo de execução. Para outros casos, contudo, a relação entre custo de reconfiguração e ganhos em desempenho e economia de recursos compensa em muito o tempo de configuração.

Foi proposta uma forma de avaliar a validade da utilização de RTR através de um método que calcula o equilíbrio entre ganho de eficiência e custo de configuração. A base da análise que realizaram é a densidade funcional [WIR98].

A métrica de densidade funcional é definida nos termos de custo da implementação de uma lógica no hardware. Esse custo (C) é tradicionalmente medido como o produto entre a área total requerida para implementar a lógica em hardware (A) e tempo total de operação (T):

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

E aplica-se àqueles sistemas onde a vazão (número de operações por segundo) é mais importante que a latência. T inclui o tempo necessário para execução, controle, inicialização e transferência de dados. Densidade Funcional (*D*) mede a vazão computacional de uma unidade de hardware, e é definida como o inverso de *C:*

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

A métrica densidade funcional em (2.2) será utilizada para comparar circuitos modificados estaticamente contra as alternativas em reconfiguração dinâmica.

O incremento (I) na densidade funcional de alguns sistemas RTR (D_r) sobre as alternativas estaticamente reconfiguráveis (D_s) é computada como a diferença normalizada entre D_r e Ds, como segue:

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

A porcentagem de incremento é medida multiplicando (2.3) por 100.

A maior diferença entre sistemas RTR e os estáticos é o custo adicional de reconfiguração. Os dispositivos atuais exigem que a configuração do circuito e a execução ocorram separadamente. Isto força a adição do tempo de configuração ao tempo total de operação de um sistema RTR. Para estes sistemas, o tempo total de operação T inclui o tempo total de execução (T_e) e o tempo total de configuração (T_c), ou:

$$T = T_c + T_e \tag{2.4}$$

Substituindo T em (2.2) tem-se a densidade funcional acrescida do custo de configuração:

$$D_r = \frac{1}{A(T_c + T_e)} \tag{2.5}$$

A partir de (2.5) fica claro que o tempo de configuração reduz a densidade funcional.

Contudo, apesar do tempo de configuração absoluto ser um importante parâmetro, o tempo de configuração relativamente ao tempo de execução é muito mais informativo. A taxa de configuração:

$$f = \frac{T_c}{T_e} \tag{2.6}$$

define este importante parâmetro. O tempo total de operação pode ser expresso como:

$$T = T_e(1 + f) \tag{2.7}$$

Substituindo este tempo em (2.2) obtem-se a métrica de densidade funcional:

$$D_r = \frac{1}{AT_e(1 + f)} \tag{2.8}$$

Como sugerido em (2.8), tempos longos de configuração podem ser tolerados se houver um tempo longo de execução correspondente (f pequeno). No limite, se $f \rightarrow 0$, a sobrecarga imposta pela configuração é negligenciável. Tais sistemas aproximam-se da máxima densidade funcional provida por RTR. Essa valor máximo (D_{max}) é calculado ignorando-se o tempo de configuração:

$$D_{max} = Lim_{f \rightarrow 0} D_r = \frac{1}{AT_e} \tag{2.9}$$

Usando-se D_{max}, o limite superior do incremento (I_{max}) sobre um sistema estático pode ser encontrado. Este parâmetro sugere o benefício máximo de um sistema RTR, e é uma boa indicação da pertinência da utilização de RTR para uma dada aplicação:

$$I_{max} = \frac{D_{max}}{D_s} - 1 \tag{2.10}$$

Além de propor essas métricas, os autores do artigo sugerem que ao projetar sistemas RTR procure-se aumentar ao máximo o tempo de execução do sistema, para que se justifique o uso de RTR.

2.4.1.2 Exemplo de comparação entre um sistema RTR com reconfiguração extrínseca e um sistema estático

A seguir será justificado o tempo de configuração de uma aplicação RTR, segundo a abordagem apresentada acima. A aplicação analisada é a implementação de uma rede neural que usa o algoritmo de aprendizagem por retro-propagação em FPGAs (denominada RRANN-*Runtime Reconfigured Neural Network*) [ELD94]. Como mostrado na Figura 2.6, o algoritmo é particionado em três estágios: pós-alimentação (*feed-forward*), retro-propagação (*back-propagation*) e atualização. Esse processo de três estágios é repetido até que o algoritmo de treinamento convirja.

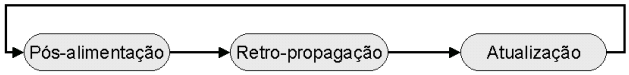


Figura 2.6: Três estágios do algoritmo de treinamento por retro-propagação.

Como sugerido na Figura2.6, o algoritmo é executado da seguinte forma:

1. Configuração dos recursos do FPGA com o processador neural de pós-alimentação, e computação dos dados;
2. O circuito de retro-propagação é configurado e entra em execução;
3. O circuito de atualização é implementado e executa, completando uma iteração do algoritmo.

Este processo de reconfiguração e execução continua até os resultados do algoritmo de treinamento convergirem. Uma vez que a abordagem RTR requer todo hardware apenas por um estágio por vez, o algoritmo inteiro pode ser executado com menos recursos do que em uma abordagem estática.

Este processo é uma reconfiguração em tempo de execução extrínseca, isto é, o sistema como um todo é reconfigurado dinamicamente, mas a reconfiguração do dispositivo é estática, uma vez que é interrompida a execução de uma determinada lógica para que outra seja configurada e executada. Ainda assim é uma abordagem dinâmica, e a possibilidade de substituir controle e aritmética associados a um algoritmo inativo por outros estágios incrementa a eficiência do hardware.

Na Tabela 2.2 são mostrados os parâmetros de comparação entre o sistema estático e o sistema dinâmico, para uma rede de 60 neurônios. Como mostrado, o circuito RTR completa o algoritmo com um terço do hardware necessário para o sistema estático. Essa redução em hardware fornece um incremento máximo na densidade funcional (I_{max}) de 168% sobre a alternativa estática.

Para calcular a densidade funcional de cada sistema, o tempo total de operação ($T_e + T_c$) e a área (A) devem ser determinados. O tempo de execução e área de uma rede neural executando RRANN

Tabela 2.2: Parâmetros de circuito para uma rede neural com 60 neurônios no sistema RRANN.

	ESTÁTICO	*R T R*	
		Máximo	Real
CONEXÕES	10800	10800	10800
A (células)	14250	2702	2702
T_e (ms)	1,93	1,93	1,93
T_c (ms)	N/A	0	19,8
f	N/A	0	9,9
D	394	2079	189
I	0	4,28	-0,52

depende do número de neurônios implementados nesta rede. Com o aumento de neurônios, tanto a área quanto o tempo de execução crescem linearmente. A Tabela 2.2 sumariza os parâmetros de um sistema com 60 neurônios (10800 conexões), para um sistema estático e outro RTR (com reconfiguração extrínseca).

A abordagem RTR tem uma evidente vantagem em termos de área sobre o sistema estático. Se o tempo de configuração (T_c) for ignorado, o sistema RTR provê um ganho de 435% em densidade funcional ($I_{max} = 4,35$). Esse resultado sugere que um sistema reconfigurável em tempo de execução pode tolerar um tempo de configuração de até 4 vezes o tempo de execução ($Se\ [T_c <= (4 * T_e)]...$) e ainda assim prover maior densidade funcional que um sistema estático.

Contudo, para que seja conhecida a densidade funcional real de um sistema RTR, o tempo de configuração deve ser considerado. Assumindo que o dispositivo (no caso, um FPGA XC3090, da Xilinx) seja configurado à taxa máxima indicada pelo fabricante, os três passos de configuração consumiriam, somados 19,8ms. Com um tempo de execução relativamente baixo (1,93ms) a taxa de configuração (f) ficaria em 9,9. Como esse resultado é superior a I_{max}, a versão RTR provê menos densidade funcional que a versão estática. Como mostrado na Tabela 2.2, a abordagem RTR reduz em 52% a densidade funcional da aplicação. Apesar disto, para redes com mais neurônios e com mais tempo de execução, a tendência é que f diminua, tornando a abordagem RTR viável.

2.4.1.3 Exemplo de comparação entre um sistema RTR com reconfiguração intrínseca e um sistema estático

Por causa do elevado tempo de reconfiguração de RRANN, e sua alta sensibilidade ao tempo de configuração, uma arquitetura nova, chamada de RRANN-II foi desenvolvida.

A estrutura básica do sistema RRANN-2 foi projetada para explorar a característica de reconfigurabilidade parcial do FPGA Clay, da *National Semiconductor* [NAT98]. Essa particularidade é usada para redução do tempo de reconfiguração, pois não há mais necessidade de configurar o sistema inteiro. RRANN-2 consiste de um processador de propósito geral e um conjunto de processadores de

propósito específico para execução do algoritmo de retro-propagação. Os três processadores neurais foram projetados para compartilhar o máximo de hardware, com a finalidade de reduzir informação de configuração para cada passo de reconfiguração.

Assim como no caso do RRANN, o RRANN-II envolveu o projeto de um processador neural de propósito geral e um conjunto de processadores de propósito específico para executar o algoritmo de retro-propagação. RRANN-II difere de RRANN por explorar as similaridades entre os processadores neurais. Os 3 processadores neurais são projetados para compartilhar o máximo de hardware possível, no intuito de reduzir os dados de configuração, em cada passo de reconfiguração [HAD95]. Quanto mais recursos compartilhados por cada processador, menos dados de configuração são necessários para converter um processador neural para o próximo.

Para o projeto de reconfiguração dinâmica intrínseca de RRANN-II foi necessário muito esforço de projeto e *layout* manual. Funções comuns foram identificadas manualmente e também manualmente mapeadas no hardware. Funções únicas a cada estágio foram identificadas e projetadas para operar corretamente com os subcircuitos remanescentes no dispositivo durante a configuração.

Os parâmetros de 60 neurônios para um sistema estático e outro dinâmico são listados na Tabela 2.3. Como mostrado, o circuito de propósito específico RTR completa o algoritmo com um terço do hardware necessário para o sistema estático. Esta redução de hardware acarreta um incremento na densidade funcional de 168% sobre a alternativa estática.

Tabela 2.3: Parâmetros de circuito para uma rede neural com 60 neurônios no sistema RRANN-II.

	ESTÁTICO	*RTR*		
		Máximo	*Global*	*Parcial*
CONEXÕES	10980	10980	10980	10980
A (células)	50764	18094	18904	18904
T_e (ms)	1,49	1,49	1,49	1,49
T_c (ms)	N/A	0	2,42	1,16
f	N/A	0	1,62	0,78
D	146	391	148	219
I	0	1,68	0,014	0,50

Para calcular a densidade funcional real do sistema RTR o tempo de configuração deve ser conhecido. Os três passos de reconfiguração requeridos pelo RRANN-II consomem 2,42ms quando a reconfiguração global é utilizada. Apesar de isto justificar o sistema com uma taxa de reconfiguração de 1,62ms, o incremento total obtido pela utilização de RTR é de apenas 1,4%. Reconfigurando somente as partes do circuito que precisam ser mudadas, o tempo de reconfiguração é reduzido para 1,16ms. Esta redução no tempo de configuração tem impacto significante na densidade funcional total. Como resumido na Tabela 2.3, a taxa de configuração é reduzida para 0,78 e um incremento na densidade funcional é de 50%.

2.4.2 Dynasty

Dynasty é uma iniciativa que visa implementar uma biblioteca para projeto de sistemas RTR. A implementação é baseada na abstração de RTR no sentido da utilização de uma planta-baixa temporal, que permite manipulação do projeto nas dimensões espacial e temporal [VAS99].

2.4.2.1 Projeto de planta-baixa temporal

Projeto de planta-baixa (*floorplanning*) é uma técnica comum em fluxos de projetos para FPGAs ou ASICs, a qual é utilizada para definir ou modificar posições espaciais de módulos de projeto, no sentido de incrementar desempenho ou eficiência na implementação do mesmo. Para fins de diferenciação, este tipo de projeto de planta-baixa será doravante denominado planta-baixa espacial. Uma planta-baixa espacial de um projeto estático (que não sofrerá reconfiguração) permanece inalterado durante todo o tempo de vida do projeto. Em sistemas RTR, contudo, a presença e a posição espacial de cada módulo de projeto reconfigurado podem mudar com o tempo. Em cada instante durante o tempo de execução, o projeto de planta-baixa espacial é determinado pela sua coordenada num eixo de projeto temporal. Um plano definido pela coordenada temporal corresponde ao projeto de planta-baixa temporal para uma configuração de projeto. Este conceito é ilustrado na Figura 1.1.

O processo de transformação do projeto de alto nível na forma comportamental para sua implementação em projetos de planta-baixa espacial posicionados num espaço temporal foi chamado de projeto de planta-baixa temporal. Este processo combina dois problemas de fluxo de projeto: (i) particionamento e seqüenciamento de módulos de projeto em configurações de projeto (também chamadas de particionamento temporal), e (ii) posicionamento temporal de módulos de projeto e conexão com a área reconfigurável de cada configuração.

No sentido de permitir completa exploração de projeto espacial das porções RTR do projeto, o projeto de planta-baixa temporal precisa resolver outros problemas relacionados: (iii) escalonamento (particionamento do passo de controle), (iv) alocação de unidades funcionais, e (v) alocação de registradores.

O projeto de planta-baixa temporal fornece ao projetista uma melhor abstração de projeto de espaço para sistemas RTR. O fluxo de projeto convencional não provê visualização do espaço em relação ao tempo, além de que torna sua exploração uma tarefa difícil. Por outro lado, projeto de planta-baixa temporal é bem ajustado para visualização de projeto para dimensões espacial ou temporal, em 2 ou 3D.

2.4.2.2 O *framework* Dynasty

Dynasty é um *framework* extensível e portável, projetado para pesquisa de técnicas e métodos para projeto de sistemas RTR.

No sentido de suportar a geração de plantas-baixas espaciais tardiamente no fluxo de projeto (necessário no caso de sistemas RTR), a representação interna do projeto disponibiliza visões de projeto em vários níveis de abstração: (i) comportamental, (ii) estrutural, (iii) projeto de planta-baixa temporal, e (iv) disposição (layout). Uma representação robusta de projeto implementada como uma coleção de classes C++ é usada para capturar soluções de projeto durante todo o fluxo do projeto. Outro conjunto de classes em C++ é utilizado para implementar o gerenciamento de biblioteca baseado num servidor comum de bibliotecas.

A representação interna do projeto permite combinação de visões de projeto durante o desenvolvimento do sistema RTR. Por exemplo, durante o projeto de planta-baixa temporal várias porções do projeto podem estar disponíveis nas visões arquiteturais e comportamentais. Representações incompletas do projeto são também suportadas, com o intuito de permitir inserção posterior de controladores, ou outros circuitos estáticos. A representação interna do projeto também facilita a análise de métricas de projeto e armazenamento de resultados analisados.

Abaixo são citadas as ferramentas do Dynasty que permitem a exploração do espaço de soluções de um projeto de sistema RTR:

- ➢ Visualizador do grafo de dados e controle (CDFG): provê informação sobre comportamento do sistema, o qual será implementado durante o fluxo de projeto.

- ➢ Navegador de estrutura e hierarquia do projeto: fornece detalhes de todos os elementos do projeto e suas relações hierárquicas. Isto permite alocação de elementos/módulos para operadores CDFG.

- ➢ Editor de controle de iterações e escalonador de configurações: é uma ferramenta de análise que permite ajuste, controle e escalonamento de configurações através do espaço de soluções do projeto.

- ➢ Gerador de planta-baixa 3D: permite visualização e manipulação de relacionamentos entre módulos, com o projeto de planta-baixa espacial e entre duas ou mais configurações.

- ➢ Editor de disposição detalhada: permite manipulação de elementos de disposição detalhados, tais como chaves lógicas e roteamento.

Os parâmetros e estruturas de uma solução em RTR podem ser diretamente manipulados via interface gráfica do Dynasty. Uma mudança em qualquer visão será propagada para representações de projeto relevantes em outras visões. Por exemplo, quando um módulo é colocado em uma configuração cujos resultados acarretam uma violação na dependência de dados do CDFG, o passo de controle e o escalonador de configurações são automaticamente recalculados para refletir a latência de projeto e a sobrecarga de configuração resultantes.

A ferramenta central do *framework* é o 3D Floorplanner, que é um gerador de plantas-baixas. Ele utiliza visualização de projetos 2D para mostrar projetos de plantas-baixas espaciais para cada configuração, as quais podem ser inseridas no tempo (terceira dimensão) usando-se controles de navegação entre configurações. O 3D Floorplanner usa um número de características que facilitam a manipulação de múltiplas configurações parciais :

1. Amostragem simultânea de múltiplos projetos de planta-baixa espaciais, que pode também ser utilizada para verificar conflitos de compartilhamento de recursos entre configurações.

2. Estimador de tempo de configuração, que calcula o número de bits de configuração necessários para mudar entre duas configurações ou para o projeto inteiro.

3. É possível demarcar regiões como "não utilizáveis", o que é útil para posterior inserção de módulos, ou controle do posicionamento e do roteamento do projeto.

Com relação à simulação, o *framework* Dynasty suporta dois níveis de abstração:

1. O 3D Floorplanner pode gerar um modelo de simulação em VHDL para o projeto RTR. O projeto finalizado pode ser importado para a ferramenta de roteamento e posicionamento da Xilinx (XACT6000), onde um modelo de temporização detalhado pode ser gerado e simulado por ferramentas de terceiros.

2. Chaveamento dinâmico de circuito (*Dynamic Circuit Switching* - DCS): A implementação de DCS no *framework* Dynasty usa multiplexadores virtuais, chaves e demultiplexadores que são implementados como procedimentos VHDL parametrizáveis, em um pacote VHDL separado.

O *framework* Dynasty fornece interfaces para terceiros através das linguagens EDIF e VHDL.

A síntese automática de um controlador de configurações não é diretamente suportada pelo *framework* Dynasty. Contudo, o 3D Floorplanner pode gerar um escalonador de configurações em um

arquivo texto (nos formatos VHDL e pseudo-código), a partir do qual tal controlador pode ser construído, através da utilização de ferramentas de projeto e compilação padrões.

Módulos de projeto que não são reconfigurados durante o tempo de execução podem ser sintetizados fora do *framework* Dynasty, e então importados para dentro de posições reservadas na planta-baixa, e conectados com as partes remanescentes do projeto através de rótulos únicos de rede.

2.4.3 Reconfiguração baseada no fluxo de computações

Apesar de os FPGAS atuais possuírem características de hardware interessantes, como dispositivos com mais de um milhão de portas lógicas equivalentes, altas taxas de relógio, reconfiguração rápida e grande largura de banda, seu uso para reconfiguração dinâmica ainda é limitado. Essa limitada aplicabilidade deve-se a exígua oferta de ferramentas para automação de projeto e implementação, e também pela ausência de um modelo computacional que abstraia os recursos fixos dos dispositivos da descrição do hardware a ser projetado [DEH94].

A Figura 2.7 ilustra três situações relativas a uma determinada aplicação que se prentende implementar em um FPGA. No primeiro caso, o FPGA oferece mais recursos do que seria necesário para esta aplicação. Portanto é possível realizar a configuração do dispositivo. No caso seguinte, o *download* do arquivo de configuração ainda é possível, mas o FPGA ficaria sem nenhum recurso adicional disponível. Em contrapartida, no último caso, para um fluxo de projeto e implementação convencionais, seria impossível implementar a aplicação, já que ela demandaria mais recursos do que os existentes no dispositivo de destino.

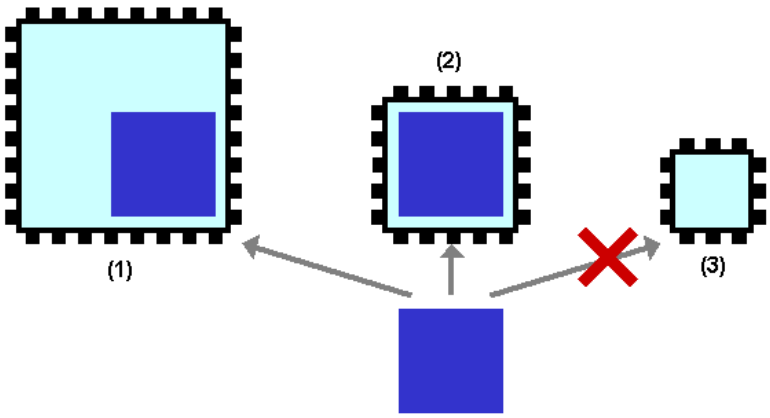


Figura 2.7: Lógica projetada é dependente do dispositivo.

Portanto, a limitação de recursos de hardware tende a prejudicar a aplicabilidade desses sistemas. Em tais modelos de projeto de hardware, a escolha de um algoritmo para aplicação está restrita ao

tamanho do hardware disponível. Além disso, uma estrutura de computação tem que ser fixa em tempo de execução, sem a possibilidade de alocação dinâmica de recursos.

Além disso, programas desenvolvidos para um dispositivo em particular (ou conjunto de dispositivos) possuem compatibilidade de código-fonte bastante limitada, e nenhuma compatibilidade do arquivo de configuração entre dispositivos de diferentes fabricantes. Organizar um programa para uma nova geração de dispositivos, ou para um dispositivo menor e mais barato, ou ainda com consumo mais baixo, tipicamente requer esforço humano substancial.

Como solução a estes problemas, é apresentada a computação organizada em fluxos para execução reconfigurável (*Stream Computations for Reconfigurable Execution* - SCORE) [DEH00]. O modelo computacional SCORE tenta resolver a limitação de recursos físicos através da virtualização de recursos computacionais, de comunicação e de memória do hardware reconfigurável. Configurações do FPGA são particionadas em páginas de tamanho fixo que se comunicam: em analogia à memória virtual, o hardware é carregado (*paged in*) sob demanda.

A comunicação por fluxo entre as páginas que não estão simultaneamente no dispositivo pode ser transparentemente buferizada[3] através da memória. O fluxo é uma ligação unidirecional página-à-página (Figura 2.8). Este esquema permite uma aplicação particionada explorar mais as páginas disponíveis fisicamente, sem necessidade de recompilação.

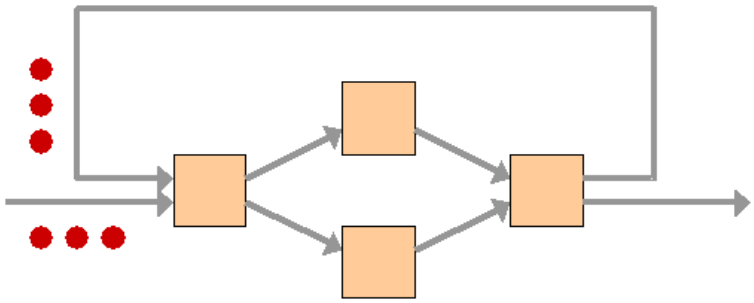


Figura 2.8: Ligação unidirecional página-à-página.

Em função dessa abordagem, SCORE permite que as reconfigurações sejam menos freqüentes, o que acarreta a amortização do custo de reconfiguração sobre um grande conjunto de dados.

Com um projeto adequado de hardware, este esquema permite compatibilidade binária e escalabilidade entre diferentes famílias de dispositivos através da compatibilidade das páginas.

Para as aplicações beneficiarem-se dos recursos físicos adicionais (páginas), o modelo de programação é uma abstração natural da comunicação que ocorre espacialmente entre blocos de hardware. Isto é, o grafo do fluxo de dados captura os blocos de computação (operadores) e a comunicação (fluxo) entre eles. Uma vez capturado, é possível explorar técnicas para mapeamento desses grafos em

[3]Neologismo baseado no termo em inglês *buffer*, que é um recurso de armazenamento temporário na memória

hardware de tamanho arbitrário. Além do mais, a composição em tempo de execução dos grafos é suportada, possibilitando uma estrutura de aplicação orientada a dados, alocação dinâmica de recursos e integração entre módulos de hardware (*cores*) desenvolvidos ou compilados separadamente.

2.4.4 *Pipeline* de Configurações

Analisando-se a natureza estática de uma configuração, verifica-se a existência de dois problemas significativos:

a) Uma computação pode requerer mais hardware do que há disponível;

b) Um projeto de hardware não explora recursos adicionais de hardware que serão disponibilizados em futuras gerações desse hardware.

Uma técnica chamada de reconfiguração por *pipeline* de configurações pode implementar uma lógica maior que a capacidade do dispositivo alvo, através de rápidas e sucessivas reconfigurações desse hardware [GOL00].

Pipeline de configurações envolve a virtualização do hardware pela divisão da lógica estática em pequenos pedaços que corresponderão a estágios do *pipeline* na aplicação. A Figura 2.9 ilustra o processo de virtualização de uma lógica com cinco estágios em um dispositivo capaz de implementar apenas três desses estágios. A Figura 2.9-a mostra a aplicação com 5 estágios, e cada estágio do *pipeline* em sete ciclos consecutivos de relógio. A Figura 2.9-b ilustra o estado dos estágios implementados no dispositivo, bem como a forma que a aplicação é executada. Neste exemplo, o primeiro estágio do *pipe* virtual é configurado no ciclo 1, e está pronto para ser executado no ciclo seguinte, e é executado por dois ciclos consecutivos. Embora não haja 4 estágios físicos no *pipe*, no ciclo 4, o quarto estágio do *pipe* virtual é configurado no estágio físico 1, substituindo o primeiro estágio virtual.

Em geral, quando uma aplicação é virtualizada através de um dispositivo (ver Item 2.5.2.1) com v estágios virtuais, em um dispositivo com capacidade para implementar p estágios, com $p < v$, a vazão da implementação é proporcional a $\frac{(p-1)}{v}$. A vazão é uma função linear da capacidade do dispositivo. Portanto, o aumento de desempenho não ocorre apenas quando novas gerações de dispositivos aumentam sua freqüência de relógio; mas também acontece quando diminui o tamanho dos transistores, pelo acréscimo de recursos em um mesmo circuito integrado, até que $p = v$.

Como alguns estágios são configurados enquanto outros estão sendo executados, a reconfiguração não diminui a performance. Uma reconfiguração por *pipeline* de configurações é considerada bem sucedida quando tem-se a configuração de um estágio do *pipe* por ciclo de relógio. Para obter-se isto conectou-se ao dispositivo um *buffer* de configuração *on-chip*, enquanto um pequeno controlador gerencia o processo de configuração.

Contudo, a reconfiguração por *pipeline* impõe algumas restrições quanto aos tipos de aplicações

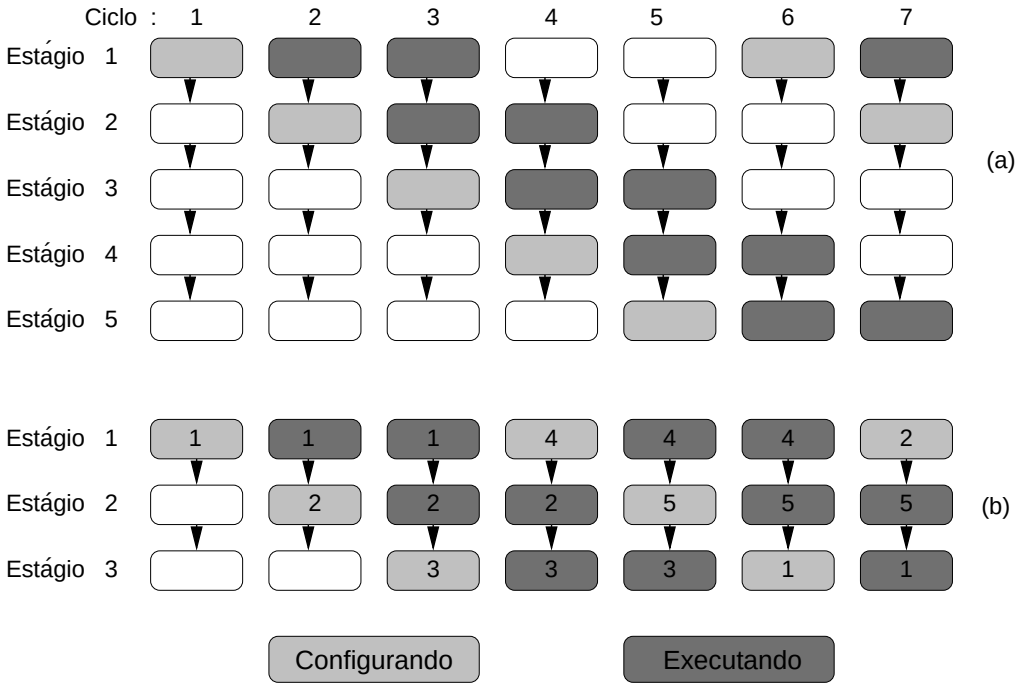


Figura 2.9: Exemplo de aplicação do PipeRench: virtualização de um *pipeline* de 5 estágios em um dispositivo que suporta apenas 3 estágios

que podem ser implementadas. A maior delas é que o estado em qualquer estágio do *pipeline* deve ser uma função do estágio seguinte e do estágio anterior a ele. Portanto, só são permitidas conexões físicas entre estágios consecutivos.

2.5 Sistemas Digitais Reconfiguráveis

Esta Seção tem por objetivo mostrar dispositivos que têm características compatíveis com alguma tendência citada na Seção 2.2, ou então que implementam métodos citados na Seção 2.4. Pode-se notar que são dispositivos com grão grande em sua maioria, todos suportam reconfiguração dinâmica, embora uns por *pipeline* de configurações e outros por chaveamento de contexto.

2.5.1 Garp

Garp é um processador baseado no conjunto de instruções MIPS-II. Foi projetado com o intuito de acelerar laços de programas de propósito geral [CAL00]. Garp atua como um coprocessador junto ao MIPS, e consiste em uma matriz bidimensional de CLBs interconectados por recursos de roteamento programáveis. A célula lógica básica do vetor reconfigurável é uma unidade de 2 bits com quatro entradas de 2 bits cada. A matriz tem pelo menos 24 blocos lógicos em uma coluna, dos quais 16 blocos a partir da primeira posição são conectados ao barramento do processador. A configuração da matriz é carregada da memória. Uma vez que um tempo significativo é necessário para carregar a configuração inteira, muitas abordagens têm sido tentadas para encurtar a latência da reconfiguração. Uma dessas é a habilidade de reconfigurar parcialmente o vetor. Uma unidade de memória *cache* é embutida junto ao sistema, e armazena as configurações usadas recentemente para um rápido recarregamento. Na Figura 2.10 pode ser observado um esquema da matriz Garp, onde é mostrado o bloco de dados reconfigurável. Barramentos de memória provêem uma reconfiguração com alta largura de banda e alta taxa de transferência de dados entre a matriz e a memória.

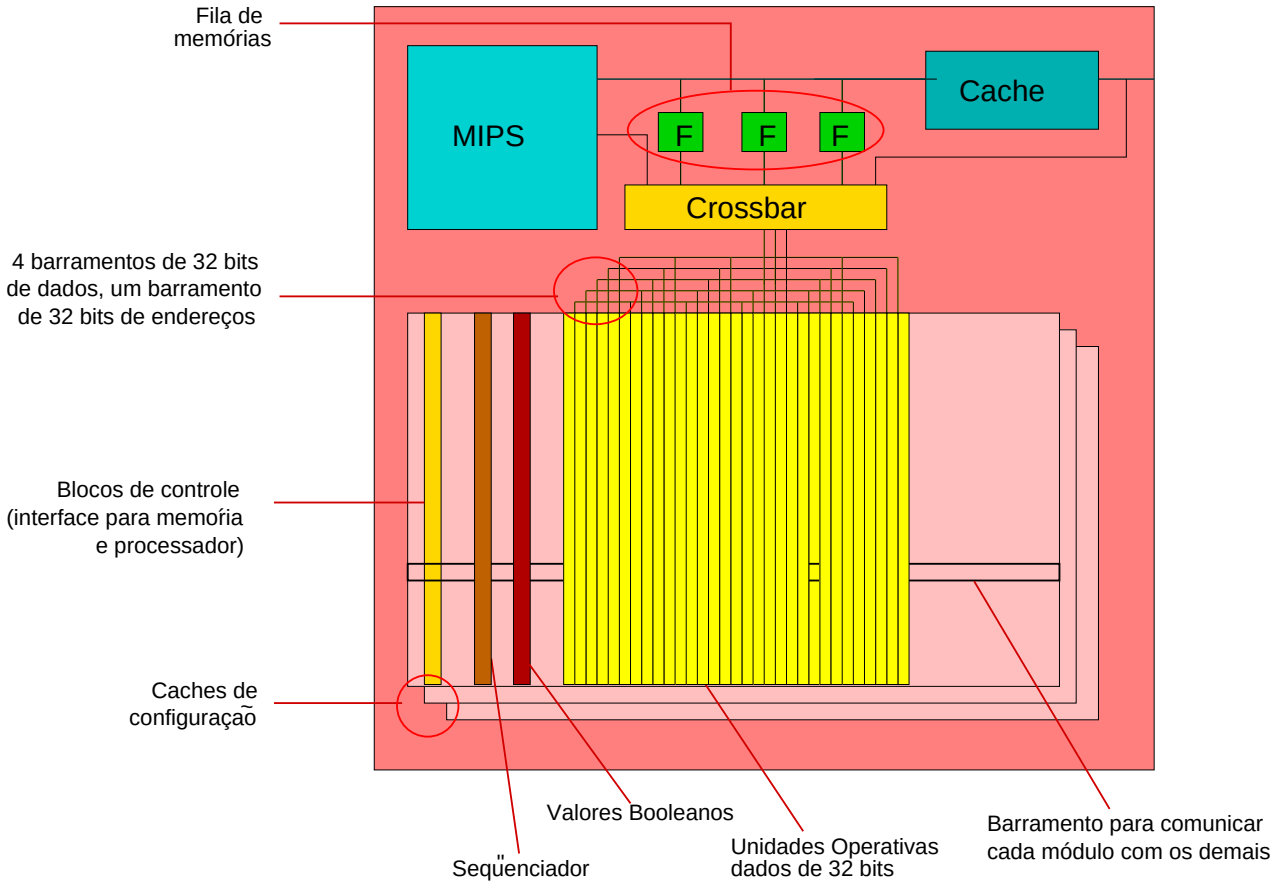


Figura 2.10: Arquitetura Garp.

Um suporte de software em baixo nível foi introduzido através de um conjunto de instruções de controle. Como diferentes configurações da matriz exigem diferentes tempos de execução, uma instrução que inicia a execução contém o parâmetro de contagem de relógio, que especifica o com-

primento da execução no processador em ciclos de relógio. A plataforma de desenvolvimento de software consiste de um compilador C e uma linguagem de configuração da matriz, que é utilizada para descrever a configuração a ser carregada na matriz do Garp.

Seguindo a idéia de unidades reconfiguráveis fortemente acopladas a um processador, o projeto Garp coloca um microprocessador completo com uma unidade reconfigurável num mesmo circuito integrado, e demonstra o incremento no desempenho se comparado a um microprocessador convencional. Seu problema é o custo em termos de tempo, envolvido na programação do sistema, que ainda é muito alto.

2.5.2 PipeRench

PipeRench é um dispositivo composto de uma rede de elementos lógicos e de armazenamento interconectados. Difere dos FPGAs convencionais porque propõe a virtualização do hardware, utilizando um *pipeline* de configurações. É particularmente interessante para aplicações baseadas em fluxos uniformes (como processamento de imagens ou sons), ou qualquer computação simples e regular sobre grandes conjuntos de pequenos elementos de dados [GOL00].

Com este dispositivo há um ganho considerável de desempenho sobre processadores convencionais. Enquanto estes forçam uma serialização de operações intrinsecamente paralelas, PipeRench explora o paralelismo dessas operações, além de aumentar a flexibilidade e diminuir o tempo de projeto do sistema.

PipeRench também demostra ser uma alternativa interessante sobre FPGAs convencionais. As limitações citadas na Seção 1.4 são relativizadas pela proposta de virtualização do hardware, implementado através de um *pipeline* de configurações (conforme analisado no Item 2.4.4).

2.5.2.1 A arquitetura do PipeRench

Na implementação atual, PipeRench atua como um processador acoplado. A Figura 2.11 é uma visão abstrata de um estágio do PipeRench, e a Figura 2.12 é uma visão mais detalhada de um elemento de processamento (EP). PipeRench contém um conjunto de estágios físicos de *pipeline*. Cada estágio possui uma rede de interconexão e um conjunto de EPs.

Cada EP contém uma unidade lógica e aritmética (ULA) e um conjunto de registradores para comunicação entre estágios (registrador de passagem). Cada ULA contém LUTs e controle extra para propagação de vai-um, detecção de zero, e assim por diante. Lógica combinacional pode ser implementada usando um conjunto N de ULAs de B bits. Funções combinacionais complexas podem ser obtidas com o cascateamento das linhas de propagação de vai-um, através da rede de interconexão. Disto decorrem ULAs mais complexas.

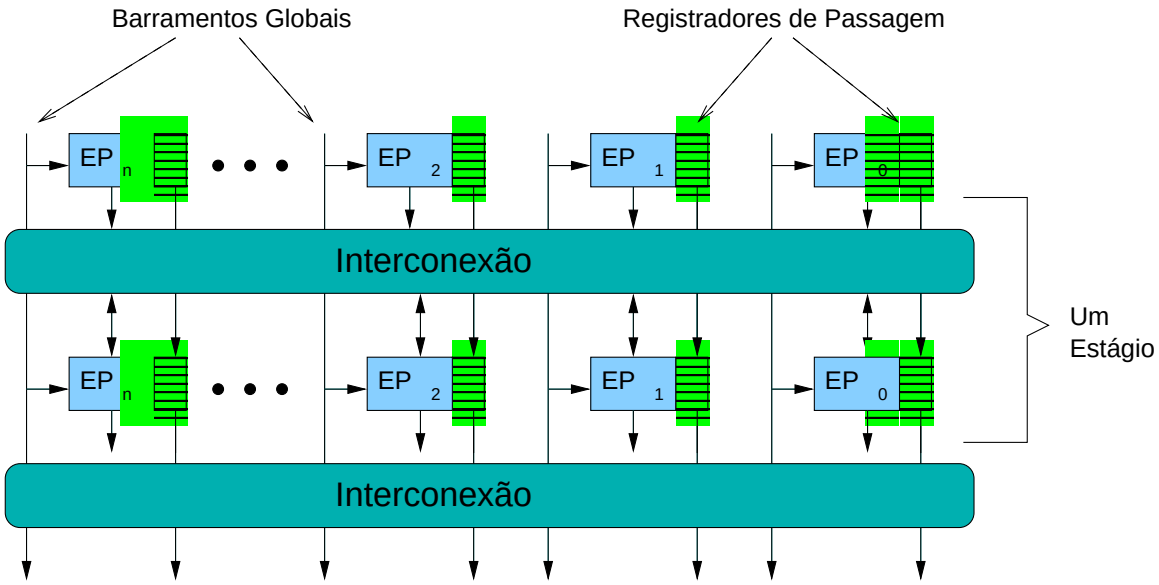


Figura 2.11: Abstração da arquitetura PipeRench.

A partir das redes de interconexão os EPs podem acessar operandos dos estágios anteriores que estão armazenados nos registradores de saída, bem como saídas (buferizadas ou não) de outros EPs do mesmo estágio. Em função da virtualização do hardware, nenhum barramento dos EPs pode conectar estágios consecutivos. Contudo, os EPs podem acessar barramentos de E/S globais. Esses barramentos são necessários porque os estágios do *pipeline* de uma dada aplicação podem estar fisicamente em qualquer uma das classes. Para alcançar seus destinos, entradas e saídas da aplicação devem usar um barramento global.

O registrador de passagem provê a interconexão de EP de uma classe para o EP correspondente na classe subseqüente. A ULA pode escrever suas saídas em quaisquer dos registradores P. Se a ULA não escreve em algum registrador do EP atual, seu valor é recebido do registrador equivalente da classe anterior.

Enquanto o registrador de passagem perfaz as conexões interclasses, a ligação entre EPs da mesma classe é realizada horizontalmente através da rede de interconexão (Figura 2.11). Em cada classe, a rede de interconexão aceita entradas a partir de cada EP daquela classe, além da entrada de valores de cada registrador de passagem da classe anterior. Além do mais, um registrador de deslocamento em cada EP desloca $B - 1$ bits para a esquerda (Figura 2.12). Assim, PipeRench pode manipular os alinhamentos de dados necessários para aritmética baseada em palavras.

Como na implementação atual o PipeRench é um co-processador, a largura de banda entre ele, a memória principal e o processador é limitada. Isto inviabiliza alguns tipos de aplicações. Mas pretende-se que, em fases posteriores o PipeRench torne-se parte integrante da UCP, e que haja uma incorporação dos conceitos de computação reconfigurável ao conjunto de instruções do processador.

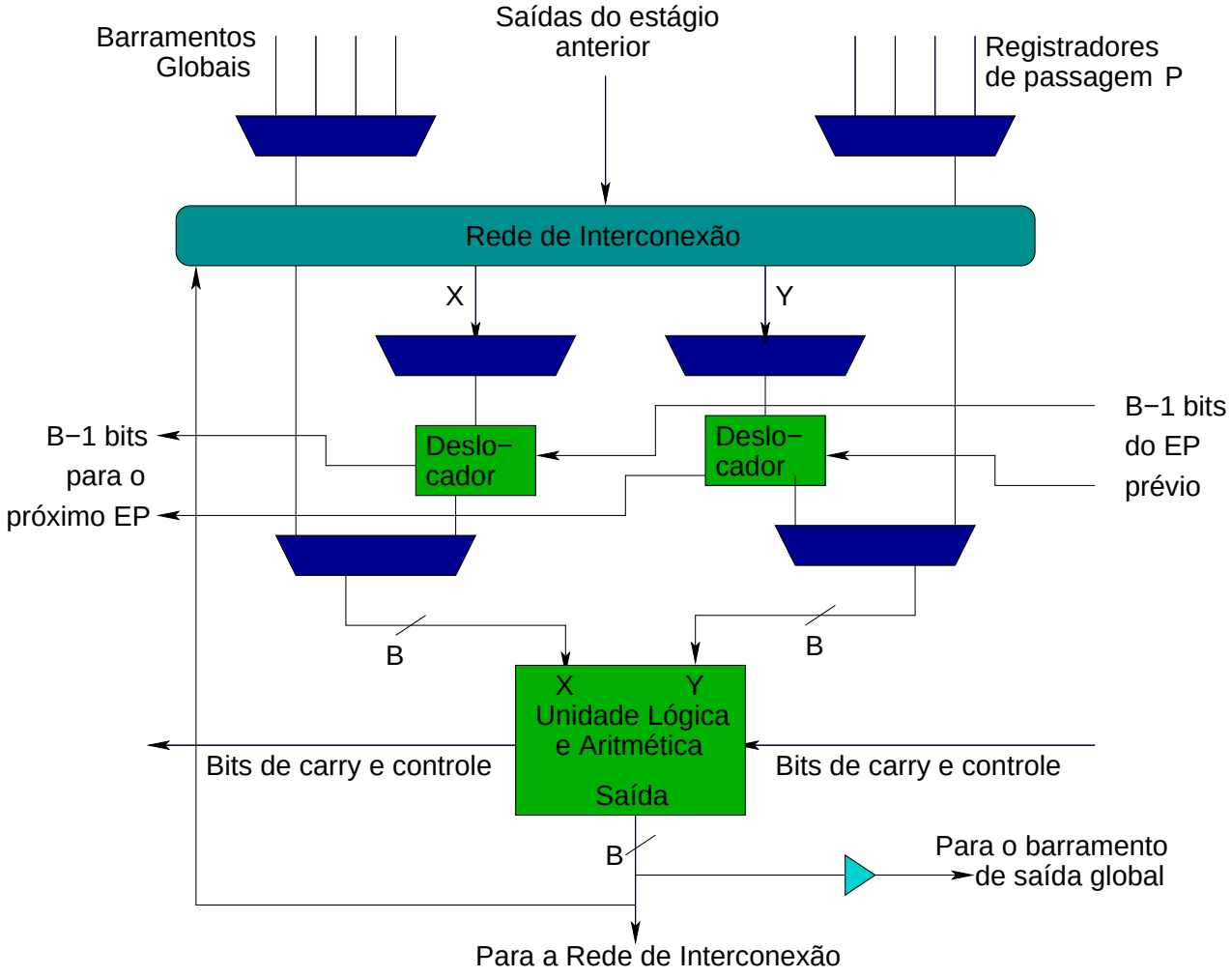


Figura 2.12: Esquema de um Elemento de Processamento (EP).

2.5.3 RAW

O microprocessador RAW consiste de um conjunto de unidades de processamento replicadas e altamente interconectadas, conforme pode ser observado na Figura 2.13 :

Cada uma dessas unidades contém um processador simples, semelhante a processadores RISC, uma porção de memória para instruções e dados, lógica configurável, e chaves programáveis.

A arquitetura RAW tem duas partes reconfiguráveis. Na primeira camada (mais baixa) existe lógica reconfigurável de grão pequeno em cada unidade. A outra parte é a rede que interconecta essas unidades. RAW usa software para implementar operações como renomear registradores, escalonamento de instruções e verificação de dependências. Esta abordagem diminui o suporte de hardware para estas operações, o que disponibiliza mais área do circuito integrado para memória e lógica, resultando num clock mais rápido, e reduz a verificação da complexidade do circuito integrado. A granularidade do RAW é mais grossa que a maioria dos sistemas baseados em FPGAs, pois o nó cor-

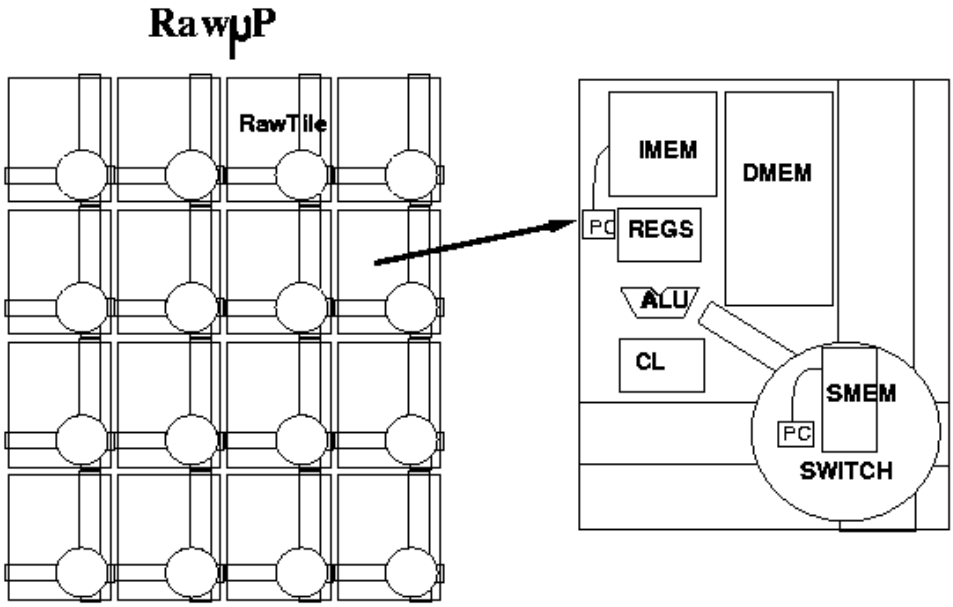


Figura 2.13: Esquema da estrutura interna do RAW.

responde a um μP, e não a LUTs. Ainda comparando, RAW tem um tempo de configuração menor, atrasos de propagação melhores para operações comuns, requisitos de roteamento mais baixos, e menor área por operação semelhante a operações de ULAs. Porém, RAW pode ainda manter o mesmo nível de paralelismo de granularidade fina que um sistema FPGA.

O sistema inteiro é distribuído sem a necessidade de uma unidade de controle central. Cada unidade é independente com suas próprias memórias de dados, intruções e chaves - sem um gargalo potencial de acesso à memória. A maior parte dos recursos do RAW estão sob o controle de um compilador. Ao invés de construir uma lógica especializada para cada operação dinâmica, tal como renomear registradores, ou prever saltos, a arquitetura RAW introduz um enfoque muito mais simples. Conseqüentemente o compilador é muito mais complexo, e tende a eliminar intervenções do usuário durante o tempo de compilação. Contudo, ainda há a necessidade de implementação de um compilador que manipule uma quantidade maior de programas, e explore agressivamente a capacidade de paralelismo de RAW.

2.5.4 Trumpet

Trumpet é um circuito integrado de teste que foi projetado para uso no estudo das vantagens envolvidas no projeto de matrizes reconfiguráveis acopladas a bancos de memória DRAM de alta capacidade [PER99]. O ponto alto desta arquitetura consiste em uma rede de " páginas computacionais" (submatrizes configuráveis) e páginas de memória (*Configurable Memory Blocks - CMBs*). Páginas computacionais são baseadas em LUTs de 5 entradas. Páginas de memória e computacionais

são interconectadas por uma rede (em forma de árvore) que se estende desde cada submatriz (folhas), alcançando os blocos lógicos individualmente, até uma conexão com um microprocessador (raiz). A Figura 2.14 ilustra essa arquitetura.

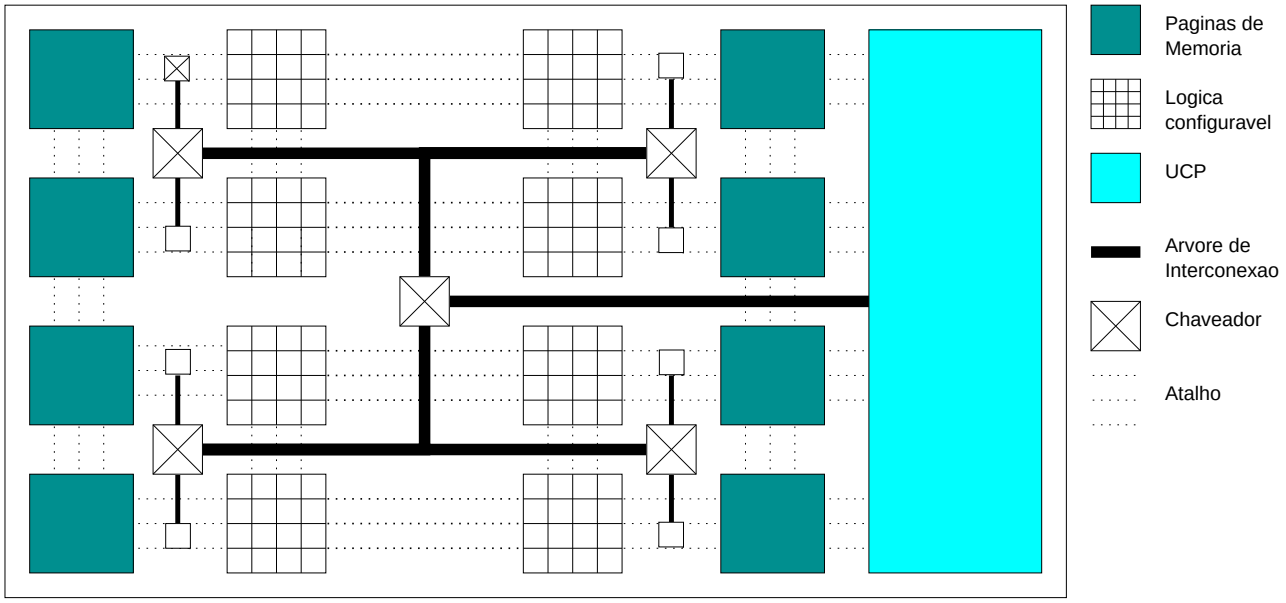


Figura 2.14: Arquitetura Trumpet.

Cada submatriz, CMB e chaveador necessitam um ou mais conjuntos de bits de configuração, dependentes da aplicação. Tais conjuntos de bits podem ser pré-carregados na DRAM, e carregados sob demanda em seus respectivos destinos. Submatrizes e CMBs são arranjados em pares para propósitos de reconfiguração, enquanto a configuração de cada chaveador pode ser assinalada ao CMB mais próximo. Desta forma é possível conseguir reconfiguração total ou parcial no tempo em que se leva para configurar um subarray, um CMB e um ou dois chaveadores. Além disso, conjuntos de bits de estados para CMBs e submatrizes podem ser carregados para inicialização, diagnóstico, ou chaveamento de contexto.

Trumpet é importante por ser uma das primeiras abordagens a valorizar a união de processador, memória e lógica reconfigurável. Grandes bancos de memória tornam possível armazenar várias configurações num mesmo circuito integrado, possibilitando uma rápida reconfiguração em tempo de execução. Também vale ressaltar que o dispositivo Trumpet serve como base para a abordagem proposta para o SCORE (Seção 2.4.3).

Apesar de não haver comprovação de que Trumpet represente o melhor equilíbrio na utilização de recursos de memória e processamento, ainda assim é um passo importante nessa direção.

2.6 Interfaces para conexão entre *cores*

Com o aumento da densidade ocasionado pelos avanços na tecnologia de semicondutores, tornou-se possível a integração de numerosas funções em um único circuito integrado. Com isto, alguns periféricos que antes comunicavam-se com o processador através de barramentos externos, agora são integrados na mesma pastilha. Conseqüentemente, os projetistas passaram a ter que se preocupar com problemas tais como ter desenvolver barramentos intra-dispositivo que provejam comunicação entre os *cores* embutidos, e de cada *core* com o mundo externo.

Em vista dessa necessidade várias iniciativas surgiram. A maioria é voltada para ASICs, muito embora seus princípios possam ser aproveitados para o desenvolvimento de barramentos de comunicação para SoCs ou mesmo para FPGAs convencionais. A seguir, serão mostradas as principais idéias de dois desses barramentos, que inspiram a proposta apresentada na Seção 5.6.

2.6.1 *CoreConnect*

CoreConnect é uma iniciativa da IBM para prover integração e reutilização de processadores e *cores* de aplicações específicas em um só dispositivo (SoC) [IBM99]. A arquitetura do barramento *CoreConnect* permite aos projetistas de hardware montar projetos de SoCs facilmente, desde que o façam utilizando *cores* projetados com as especificações do barramento. Isto faz com que o tempo de projeto e, conseqüentemente o *time-to-market*, do produto sejam bastante reduzidos.

Conforme pode ser visto na Figura 2.15, essa arquitetura é composta por um barramento de processador local (PLB - *Processor Local Bus*), barramento periférico interno ao circuito integrado (OPB - *On-chip Peripheral Bus*), um circuito que une esses dois barramentos (*bus bridge*), e um barramento de controle de dispositivo (DCR - *Device Control Register*). Periféricos podem ser conectados ao PLB com uma grande largura de banda e baixa latência. *Cores* com freqüência mais baixa são conectados ao OPB, reduzindo o tráfego no PLB, o que aumenta o desempenho geral do sistema. O princípio de funcionamento baseia-se em um árbitro, que decide qual *core* de sistema (*System core*) vai fazer acesso o barramento de cada vez.

O PLB soluciona questões de alto desempenho e baixa latência para *cores* integrados ao SoC. Ele é totalmente síncrono, e suporta a conexão de até 8 *cores* atuando como mestres e múltiplos *cores* escravos, cujo número vaira de acordo com a implementação do árbitro. Existem 3 diferentes versões de arquitetura, com 32, 64 ou 128 bits. Possui ainda barramentos de dados diferentes para leitura e escrita, o que torna possível transferências sobrepostas, a altas taxas. A especificação do PLB descreve uma arquitetura de sistema com detalhamento de sinais e transações.

O OPB é um barramento de entrada e saída, criado para aliviar gargalos de desempenho entre

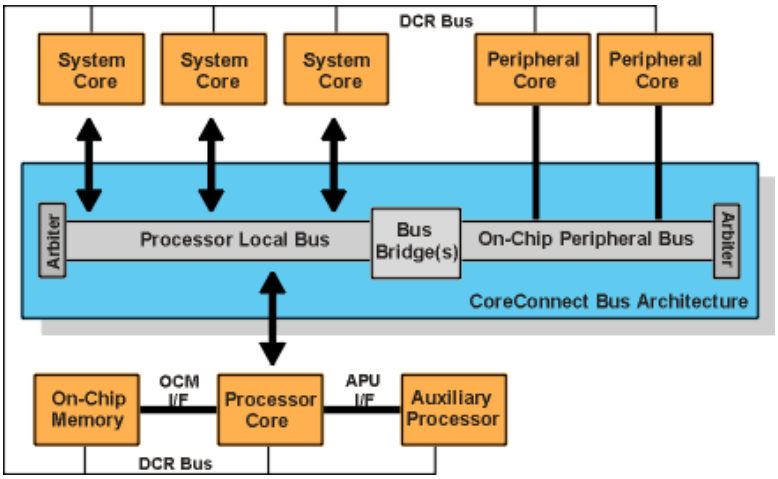


Figura 2.15: Arquitetura do barramento *CoreConnect*.

periféricos e o PLB. Exemplos de periféricos são portas paralelas, portas seriais, UARTs e outros dispositivos de banda estreita. Este barramento permite aos projetistas de sistema integrar facilmente periféricos ao ASIC. O OPB possui as seguintes características:

- ➢ Protocolo totalmente síncrono, com barramentos para dados e endereços separados (cada qual com 32 bits);
- ➢ Dimensionamento de barramento que permite transferências de bytes, palavras, e meias-palavras;
- ➢ Protocolo de endereçamento seqüencial (para modo rajada);
- ➢ Inserção de ciclos de espera, para transferências de reduzida latência.

A IBM disponibiliza, além das especificações do *CoreConnect*, um conjunto de ferramentas e *testbenchs* em VHDL ou Verilog.

2.6.2 *WishBone*

Wishbone é uma interface portável para uso com *cores*, como um barramento interno para aplicações em SoCs [SIL01]. Propõe a reutilização de projetos, diminuindo o problema de integração de *cores*. Assim como o *CoreConnect*, o *Wishbone* também contribui para a diminuição do *time-to-market* de produtos. A arquitetura *Wishbone* permite conectar *cores* de uma forma simples, flexível e portável. Geralmente esses *cores* são desenvolvidos independentemente, e são integrados por terceiros. O barramento *Wishbone* auxilia o integrador de sistemas através de uma tentativa de padronização da interface entre *cores*.

Wishbone utiliza uma arquitetura mestre/escravo. Isto significa que módulos com interfaces mestre iniciam transações de dados, e interfaces escravo participam passivamente dessas transações. Mestres e escravos comunicam-se através de uma interface de interconexão chamada de *Intercon*. A rede de interconexão do *Wishbone* pode ser modificada pelo integrador do sistema, conforme sua necessidade. A isto chama-se *interconexão variável*.

A interconexão variável permite ao integrador do sistema mudar a forma com que as interfaces mestre e escravo comunicam-se entre si. Por exemplo, um par de interfaces mestre e escravo podem comunicar-se através de topologia ponto-a-ponto, barramento compartilhado, ou por um chaveador *crossbar*.

O esquema de interconexão variável é muito diferente do esquema utilizado em barramentos de computadores, como PCI, ISA, etc. Esses sistemas usam placas de circuito impresso e conectores fixos (*hardwired*). As interfaces nesses barramentos não podem ser mudadas, o que limita seriamente a forma como as placas comunicam-se umas com as outras. *Wishbone* elimina essa limitação permitindo que o integrador mude a interconexão do sistema.

Isto é possível por que os circuitos integrados possuem caminhos que podem ser ajustados (roteamento e portas lógicas configuráveis). Isto pode ser configurado no dispositivo utilizando-se uma descrição em VHDL (ou Verilog), e com o auxílio de uma variedade de ferramentas que sintetizam essa descrição.Os autores do Wishbone (*Silicore Corporation*) afirmam que geralmente definem as interconexões utilizando VHDL, o que permite que as interconexões sejam definidas de forma a melhor adaptar-se à aplicação.

A descrição completa das formas de interconexão e dos sinais do Wishbone podem ser encontrados em: http://www.silicore.net/wishfaq.htm.

Capítulo 3

Software para apoio ao projeto e/ou implementação de SDRs

Conforme mencionado anteriormente, uma das grandes lacunas a serem preenchidas para o bom desenvolvimento da tecnologia de reconfiguração parcial de dispositivos FPGAs é a falta de ferramentas de CAD que permitam o projeto e a implementação de sistemas dinamicamente reconfiguráveis.

Duas iniciativas destacam-se no intuito de preencher parcialmente esse vazio. A primeira delas é uma pesquisa acadêmica realizada na Universidade de York, que pretende o desenvolvimento de um conjunto de ferramentas de CAD baseado em Java que permita RTR. Uma visão geral desta pesquisa é mostrada na Seção 3.1.

A segunda iniciativa no sentido de prover ferramentas para reconfiguração parcial parte da indústria. A equipe de desenvolvimento da empresa Xilinx (fabricante de FPGAs) criou um conjunto de classes Java, denominado JBits, com a intenção de servir como uma API (*Application Programming Interface*) para manipular o arquivo de configuração de seus FPGAs. Esta API é descrita na Seção 3.2.

3.1 JHDL

Durante o desenvolvimento de aplicações para computadores configuráveis (*Custom Computing Machines* - CCMs), projetistas devem realizar duas tarefas gerais. Primeiro, devem projetar o circuito que implementa a funcionalidade necessária para a aplicação. Isto é tipicamente feito usando ferramentas comerciais de CAD, tais como as que provêem síntese lógica e física, em conjunto com ferramentas *back-end* obtidas junto a fabricantes de FPGAs. Segundo, projetistas devem escrever um programa para supervisionar a operação da aplicação. Nos casos de aplicações RTR de maior com-

plexidade, este programa de controle pode ser igualmente complexo, carregando uma variedade de configurações e dados, sob demanda, conforme a necessidade da aplicação. Atualmente, o programa de controle e a descrição do circuito devem ser desenvolvidos simultânea e independentemente; o projetista é responsável por garantir que esses dois pedaços de programas irão cooperar corretamente, tipicamente através de ciclos de carga, execução e compilação repetidos.

Esta divisão entre descrição do circuito e programa de controle é na realidade a divisão da aplicação entre partes estática e dinâmica: a estática representada pela biblioteca de circuitos, e a dinâmica constituída pelo programa de controle, que escolhe configurações de hardware de uma biblioteca, configura o dispositivo e executa a aplicação. Contudo, devido às mudanças que ocorrem no campo da computação reconfigurável, tratar separadamente as partes estáticas e dinâmicas da aplicação é inadequado e limitante. O que é necessário é uma única e integrada descrição que permita ao projetista naturalmente expressar as partes dinâmicas e estáticas da aplicação simultânea e conjuntamente.

Nesse sentido, Brad Hutchings e Peter Bellows [BEL98] propuseram uma abordagem de projeto e uma ferramenta de CAD com o objetivo de criar uma descrição integrada. Seu projeto foi desenvolvido baseado nos seguintes requisitos:

1. A ferramenta deve usar uma liguagem de programação existente, sem extensões. Isto possibilita que um grande número de programadores possam usar a ferramenta.

2. O paradigma de controle da CCM deve ser o de CCM independente. Detalhes de controle da CCM devem ser elevados a um nível mais alto de abstração, com o objetivo de facilitar o processo de redirecionamento das aplicações para vários dispositivos-alvo.

3. O método de descrição deve suportar RTR total e parcial. Espera-se que esse seja o tipo de CCMs que mais necessita ferramentas de CAD.

4. A descrição integrada deve servir para simulação e execução, sem modificações.

O sistema JHDL é implementado como um conjunto de bibliotecas de classes Java, com funcionalidade dividida em duas áreas básicas: simulação de circuito e suporte à execução da CCM. As classes referentes ao suporte da execução provêem acesso transparente às funções de controle da CCM via mecanismos de construção/destruição.

Projetistas desenvolvem circuitos em JHDL selecionando um conjunto de elementos síncronos ou combinacionais, e ligando-os de modo a formar um circuito síncrono arbitrário. Existem três classes diferentes que podem ser utilizadas para implementar um circuito: *CL* (combinacional), *Synchronous* (síncrono) e *Structural* (interconexão entre elementos síncronos e combinatórios). No uso de cada classe, o projetista define uma nova classe que herda características da classe apropriada e implementa a funcionalidade desejada no construtor e em outros métodos. Circuitos individuais são

interconectados instanciando objetos *Wire* e passando esses objetos como argumentos para os construtores.

Os autores deste projeto relatam que foi feita uma comparação de produtividade com alunos de sua instituição, entre VHDL e JHDL. Segundo eles, para usuários novatos, JHDL originou um ganho significativo no tempo de projeto. Contudo JHDL, em função de ser baseada em Java, acarreta um tempo de simulação bastante elevado.

Maiores informações a respeito do status atual do projeto JHDL podem ser obtidas no endereço http://www.jhdl.org/

3.2 JBits

Apesar de FPGAs baseados em SRAM poderem sofrer inúmeras configurações, existem muito poucas ferramentas, bem como poucas informações disponíveis comercialmente para realização de RTR [MCM99]. Este cenário começou a mudar com a apresentação da família de FPGAs XC6200 [XIL99a] da Xilinx, que permitia reconfiguração parcial. Porém, tal família não logrou sucesso por não ter surgido nenhuma ferramenta comercial que explorasse a característica de RTR da XC6200.

No sentido de criar um conjunto de ferramentas para reconfiguração parcial, a Xilinx iniciou o projeto JERC6K, vinculado à família 6200. Com a descontinuação dessa família, o projeto foi transferido para suportar a família XC4000, e foi renomeado para *Xilinx Bitstream Interface* (ou XBI). Esse programa foi mais tarde renomeado para JBits. Contudo, a família XC4000 não continha suporte para reconfiguração parcial. Quaisquer mudanças na configuração do circuito exigiam a parada do dispositivo, e sua reconfiguração era lenta ao ponto de ser inaceitável para determinadas aplicações. Mais recentemente, o conjunto de classes JBits foi portado para a família de dispositivos Virtex (ver Capítulo 4).

JBits é um conjunto de classes Java que fornecem uma API que permite manipular o arquivo de configuração da família de FPGAs Virtex da Xilinx. Esta interface opera tanto em arquivos de configuração gerados pelas ferramentas de projeto da Xilinx quanto em arquivos de configuração lidos do hardware [XIL01].

A motivação original do JBits foi suportar reconfiguração dinâmica em FPGAs da família Virtex [GUC99]. Para atender a este requisito, a ferramenta deve ser rápida e prover informações físicas a respeito do circuito a ser reconfigurado. A solução encontrada foi criar uma biblioteca de classes que dá acesso a todos os elementos arquiteturais reconfiguráveis do dispositivo. Isto permite que o circuito seja reconfigurado sem a necessidade de se refazer o posicionamento e roteamento com as ferramentas convencionais.

O fluxo de projeto com o uso do JBits é ilustrado na Figura 3.1. A Figura 3.1-a mostra uma

aplicação Java escrita pelo usuário. Este programa acessa a *Interface do JBits* (Figura 3.1-b1) para manipular os recursos configuráveis do FPGA. Cada chamada de função no nível de interface do JBits realiza uma ou mais chamadas de funções à *Interface em Nível de Bit* (Figura 3.1-b2). Neste nível, é possível modificar até um único bit no arquivo de configuração. Mesmo que seja possível configurar bits individualmente, não é provável que o usuário do circuito deseje lidar com um nível tão baixo de abstração. Por isto, a *Interface JBits* disponibiliza uma abstração que permite a modificação de conjuntos de bits.

Outro nível de detalhe que deve ser escondido do usuário diz respeito às especificidades relativas a dispositivos diferentes de uma mesma família de FPGAs. A *Interface em Nível de Bit* pode ser vista como uma camada que provê suporte necessário para que detalhes como a localização de um bit num FPGA XCV300 ou num FGPA XCV150 seja transparente para o usuário.

Finalmente, a *Interface em Nível de Bit* interage com a classe *Bitstream* (Figura 3.1-b3). Esta classe gerencia o arquivo de configuração do dispositivo e provê suporte para leitura e escrita das configurações dos arquivos e para eles (Figura 3.1-c) .

Ademais, a classe *Bitstream* pode receber dados lidos diretamente do dispositivo (*readback*) e mapeá-los para o formato de *bitstream*. Esta habilidade de gerenciar dados obtidos por *readback* é essencial para reconfiguração dinâmica (Seção 5.2). A API JBits utiliza o software XHWIF (*Xilinx HardWare InterFace*) para configurar o dispositivo e realizar o *readback* (Figura 3.1-d).

Além do que foi exposto, ainda há outra parte relacionada na Figura3.1: a biblioteca de *Cores* (Figura 3.1-e). Essa biblioteca é um conjunto de classes em java que define macrocélulas, ou módulos parametrizáveis de hardware, que podem ser adicionados ou removidos do dispositivo.

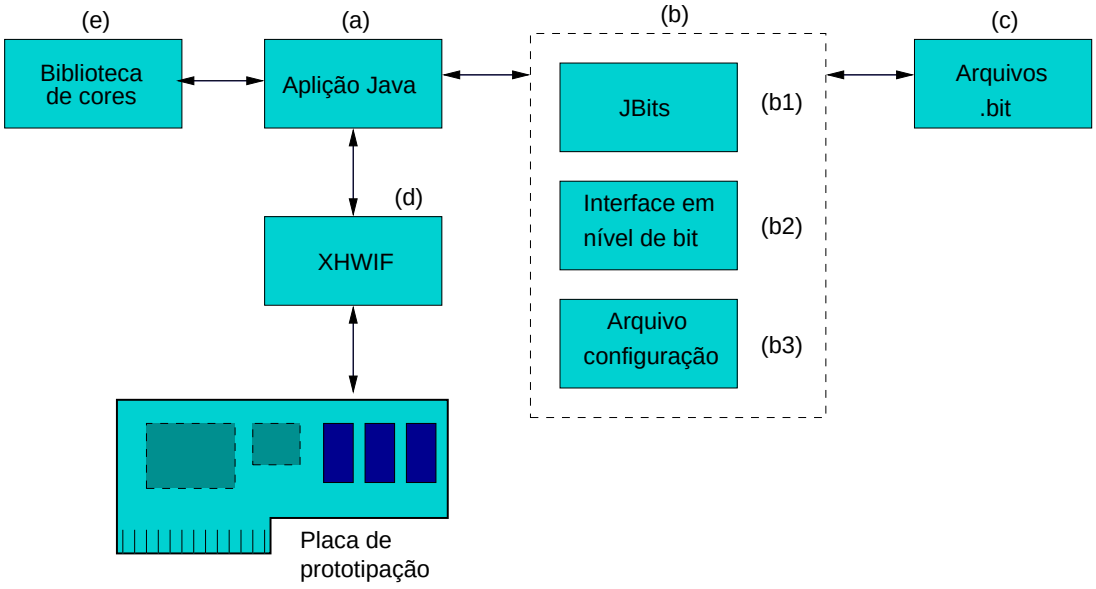


Figura 3.1: Fluxo de projeto com JBits e XHWIF.

3.2.1 Modelo de programação do JBits

O modelo de programação utilizado pelo JBits é baseado em uma matriz bidimensional de CLBs. Cada CLB é referenciada por uma linha e uma coluna. Assim, todos os recursos configuráveis na CLB selecionada podem ser configurados ou analisados. Além disso, o controle de todo o roteamento adjacente à CLB selecionada torna-se disponível. Pelo fato de o controle ser ao nível de CLB, os arquivos de configuração podem ser modificados ou gerados rapidamente.

Esta API tem sido utilizada para construir circuitos completos ou para modificar circuitos existentes. Esta API pode ser utilizada como base para a construção de outras ferramentas. Isto inclui ferramentas de projeto tradicionais para executar tarefas como posicionamento e roteamento do circuito, bem como ferramentas de aplicação específica, como por exemplo um configurador de *cores*.

O JBits fornece uma abordagem de linguagem de alto nível para desenvolvimento de sistemas reconfiguráveis incluindo reconfiguração em tempo de execução. É necessário que o projetista tenha conhecimento do seu circuito e dos detalhes de configuração do dispositivo, pois, caso contrário, o JBits pode gerar dados que danifiquem o dispositivo.

3.2.2 Limitações do JBits

A maior desvantagem da API JBits é sua natureza manual. Tudo deve ser explicitamente relacionado no código-fonte. Esta característica, contudo, pode ser amenizada com a utilização de *cores* ou macrocélulas. Devido a esta necessidade de especificação explícita de todos recursos, a interface JBits favorece circuitos mais estruturados. Circuitos não-estruturados (lógica aleatória, por exemplo) não são bem adaptadas para uso com o JBits.

Outra limitação importante é que a API Jbits requer que o usuário seja familiarizado com a arquitetura do dispositivo. Conforme antes citado, a utilização de aplicações desenvolvidas com o auxílio das classes JBits, sem o devido conhecimento de detalhes arquiteturais do dispositivo, pode causar danos físicos ao FPGA.

Além disto, como a interface JBits necessariamente atua no nível de configuração do *bitstream*, a criação ou modificação de um arquivo de configuração com o JBits elimina qualquer possibilidade de utilização de ferramentas de análise disponíveis para depuração futura do circuito. Especificamente, a habilidade de realizar quaisquer tipos de análises de temporização está ausente no JBits.

3.2.3 XHWIF

XHWIF, a interface de hardware Xilinx (*Xilinx Hardware InterFace*), é uma interface escrita em Java para comunicação entre um computador hopedeiro (*host*) e uma placa baseada em FPGA. Ela é composta por métodos para leitura e escrita de *bitstreams* no FPGA, e métodos para descrição do tipo e do número de FPGAs na placa em questão. Também inclui métodos para leitura e escrita nas memórias na placa, e possibilita ligá-la e desligá-la.

A interface padroniza a forma de comunicação com o hardware, de tal forma que, utilizando a mesma interface, diversas aplicações podem comunicar-se com uma variedade de placas. Toda informação específica do hardware é encapsulada dentro de uma classe que implementa a interface XHWIF. Utilizando a interface nativa Java (*JNI - Java Native Interface*), que permite que programas Java interajam com programas C, chamadas à interface são convertidas em chamadas aos *drivers* da placa. Esta metodologia possibilita que as aplicações comuniquem-se diretamente com os *drivers* ou com o barramento. Escondendo informações específicas do barramento e dos *drivers* na classe que implementa XHWIF é possível que aplicações comuniquem-se com placas conectadas através de qualquer barramento, que pode ser PCI, ISA, ou qualquer outro padrão.

XHWIF possui uma aplicação, denominada XHWIF *Server* que habilita que outros programas comuniquem-se com o dispositivo reconfigurável localizado em qualquer *host* através da Internet. Esta possibilidade admite a depuração do projeto mesmo sem o acesso direto ao hardware, além de permitir que diversos usuários testem suas aplicações no mesmo hardware (obviamente, através de um escalonamento no tempo).

3.2.4 Reconfiguração dinâmica com JBits

Na documentação do JBits encontra-se afirmação de que reconfiguração dinâmica pode ser obtida com esta API, em conjunto com XHWIF [GUC99]. Apesar de não ter sido encontrada na literatura nenhuma referência a respeito de uma reconfiguração dinâmica realizada com a utilização do JBits, teoricamente isto pode ser possível, segundo o método a seguir descrito.

Uma aplicação RTR pode realizar chamadas à interface JBits para modificar os dados do arquivo de configuração. Então, esta pode fazer uma chamada ao XHWIF para interagir com o dispositivo. Por exemplo, a aplicação utilizaria o método $setConfiguration(dispositivo, informacoes)$ e em seguida faria chamadas $getConfiguration()$ para realizar *download* e *readback* dos dados de configuração.

Contudo, em testes realizados no laboratório do GAPH (Grupo de Apoio ao Projeto de Hardware[1]), observou-se que a geração do arquivo de configuração parcial utilizando JBits deu-se com erro. Primeiramente foi criado o *bitstream* completo, depois, com a ferramenta que é descrita na Seção 5.1, foi gerado o *bitstream* parcial. Quando tentou-se realizar o *download* parcial, o software

[1]Laboratório vinculado à Faculdade de Informática da PUCRS.

utilizado para isto (*JTagProgrammer*) retornou mensagem de erro. Posteriormente, analisando-se o arquivo de configuração parcial, percebeu-se que o protocolo de configuração gerado não condizia com o que é descrito na documentação do dispositivo [XIL00a].

A última versão do JBits (2.7), no entanto, gera um *bitstream* parcial correto. Utilizando-se o *JTagProgrammer* é possível carregar o *bitstream* parcial no dispositivo. Contudo, o FPGA não é configurado corretamente, em função de limitações do *JTagProgrammer*. Tais limitações são comentadas na Seção 5.5.

3.2.5 JRTR

No sentido de obter maior vantagem do suporte da arquitetura Virtex à reconfiguração parcial, a API JBits foi estendida com a API JRTR. Esta interface provê um modelo de cache onde as mudanças dos dados de configuração são ajustadas, e somente os dados realmente necessários são escritos no dispositivo, ou lidos dele.

A recente adição do programa JRTR (*Java Run-Time Reconfiguration*) à nova versão do conjunto de classes JBits resultou no suporte direto à reconfiguração parcial. Este suporte utiliza uma combinação de técnicas de hardware e programas para permitir que pequenas alterações sejam feitas diretamente no arquivo de configuração da Virtex, de forma rápida e sem a interrupção da operação.

A interface do JBits existente é ainda utilizada para ler e escrever arquivos de configurações do disco, ou de outro dispositivo externo. O JRTR *Bitstream Parser/Generator* é utilizado para analisar o arquivo de configuração, e para manter a imagem dos dados e as informações de acesso.

A API atual provê controle simples, mas completo, da cache de configurações. O usuário pode produzir configurações parciais em qualquer tempo, e então carregá-las no hardware.

Capítulo 4

Hardware que habilita reconfiguração parcial

Os primeiros dispositivos que suportaram reconfiguração parcial foram criados pelas empresas National, Algotronix e Xilinx. Os resultados foram as famílias de FPGAS Clay [NAT98], Cal1024 [ALG89] e XC6200 [XIL99a], respectivamente. Tais FPGAs não lograram grande sucesso comercial principalmente pelo fato de não terem sido produzidas ferramentas eficientes de projeto, de roteamento e de posicionamento.

Outro fabricante de FPGAs, a Altera, alega que a partir da família APEX permitiu reconfiguração parcial [ALT01]. Contudo isto ocorre de forma muito limitada. A reconfiguração parcial dessa família dá-se através do projeto de lógica em RAM, criando uma LUT[1] onde podem ser implementadas funções com 7 entradas e 16 saídas. Depois dessa lógica ser implementada no bloco de RAM o sistema pode reescrevê-la em qualquer tempo, mudando a configuração de parte do sistema. A grande limitação desta abordagem é que em algum lugar do circuito deve-se armazenar todas as configurações possíveis que irão modificar a RAM, isto porque não há como fazer a carga externa de novas configurações. Portanto, na visão deste autor, a família APEX não suporta reconfiguração parcial, e a alegação do fabricante é incorreta, constituindo mais uma peça de publicidade que uma informação técnica.

Duas empresas - Atmel e Xilinx - comercializam famílias FPGAs que permitem reconfiguração parcial. Na próxima Seção (4.1) será feita uma breve descrição da família AT40k da Atmel. A Seção 4.2 trará um detalhamento da arquitetura interna dos FPGAs da família Virtex, bem como analisará a disposição dos elementos internos de sua arquitetura no arquivo de configuração.

[1] Ver subseção 1.2.1

4.1 Atmel

Os FPGAs da família AT40K foram especialmente projetados para suportarem *Cache Logic*, que é uma técnica para construir sistemas e lógica adaptáveis, permitindo reconfigurar parte do dispositivo sem interromper a operação da lógica remanescente. Num sistema de *Cache Logic* somente as porções da aplicação que estão ativas em um dado momento realmente estão implementadas no FPGA, enquanto funções inativas são armazenadas externamente numa memória de configuração. Se novas funções se fazem necessárias, as antigas são sobrescritas, como mostrado no diagrama da Figura 4.1. Este procedimento aproveita-se da latência funcional inerente a muitas aplicações - em qualquer tempo dado, somente uma pequena proporção da lógica está de fato ativa. Ou seja, funções podem ser substituídas em tempo de execução no FPGA, enquanto o sistema continua a operar [ATM00].

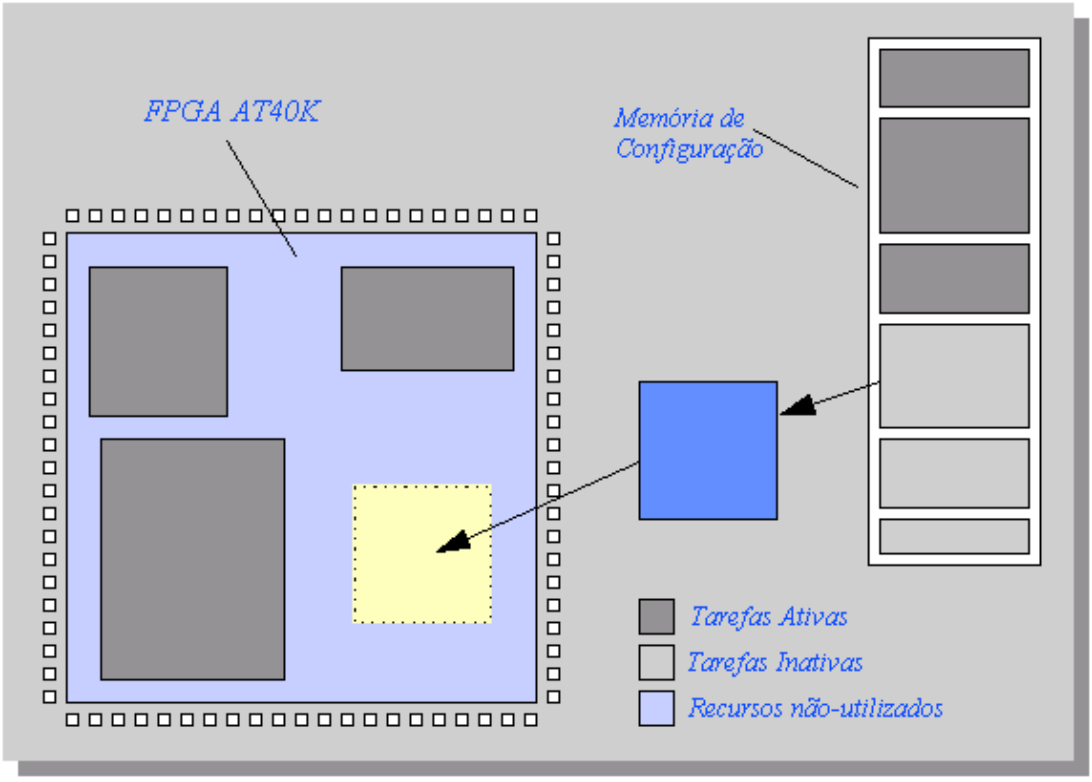


Figura 4.1: Diagrama da *Cache Logic*, onde *cores* armazenados em memória configuram o FPGA em tempos diferentes.

4.1.1 Implementação da *Cache Logic*

A implementação do *Cache Logic* é conceitualmente semelhante à organização de caches em subsistemas de memória [ATM00a]. No cache de memória uma memória de velocidade mais rápida (normalmente uma SRAM) é utilizada para armazenar os dados ativos, enquanto um maior volume de dados reside numa memória de menor custo, tais como DRAM, EPROM ou disco magnético, etc. *Cache Logic* trabalha de forma semelhante. Na *Cache Logic* somente uma pequena porção do circuito - aquelas funções que são nela carregadas - estão ativas no sistema num dado momento, enquanto as funções não utilizadas permanecem numa memória de custo mais baixo. À medida que novas funções são requeridas, elas podem ser carregadas na *Cache Logic*, substituindo ou complementando a lógica já presente.

A habilidade de implementar *Cache Logic* requer do FPGA a capacidade de ser reconfigurado dinamicamente. Outro requisito é a simetria da arquitetura. Isto é necessário para tornar possível o posicionamento arbitrário de blocos genéricos, em uma localização que esteja disponível no momento necessário.

Existem dois tipos de *Cache Logic:* predeterminada e dinâmica. A primeira envolve o uso de funções predeterminadas, gravadas numa memória externa não volátil (EPROM, por exemplo). Essas funções são previamente roteadas e posicionadas, e o arquivo de configuração correspondente a elas foi previamente gerado. A implementação dessas funções é previamente controlada por um gerenciador residente na *Cache Logic*. Novas funções devem ser carregadas na *Cache Logic* em *background*, sem promover uma parada na operação da cache.

O segundo tipo de *Cache Logic*, dinâmica, é a base para construção de um hardware evolutivo. Cache dinâmico envolve a determinação da lógica, posicionamento e roteamento, geração do arquivo de configuração, e configuração da *Cache Logic* em tempo de execução. Os principais aspectos abrangidos no desenvolvimento dessa capacidade incluem o escalonamenteo e alocação de funções, coleta de lógica aleatória e detecção de colisão.

4.1.2 FPSLIC

De acordo com a tendência em desenvolver sistemas que integrem lógica programável, memória e processador genérico em um único dispositivo (SoC), com a possibilidade de desenvolvimento de soluções integradas de hardware e software, a ATMEL produziu o FPSLIC (*Field Programmable System Level Integrated Circuits*).

O FPSLIC é um dispositivo que integra em um único *chip* o *core* de um processador AVR de 8 bits (RISC) que atua como controlador capaz de processar até 30 MIPS (Figura 4.2-a), um FPGA AT40K para o bloco de dados (Figura 4.2-b), um multiplicador em hardware (Figura 4.2-c), controladores de memória e de configurações, e 32K de memória SRAM (Figura 4.2-d) - tudo no mesmo CI. A Figura 4.2 ainda mostra detalhes do FPGA, como a presença de memória a cada setor de células programáveis

(Figura 4.2-e) e os blocos lógicos com conexões em 8 lados, para implementar funções complexas sem causar impacto nos recursos de barramento (Figura 4.2-f). Esta arquitetura possibilita a prototipação rápida de sistemas complexos com um baixo consumo de potência, e é passível de ser reconfigurada dinamicamente. Foi desenvolvido para o FPSLIC um conjunto de ferramentas de suporte para criação de sistemas integrados GPP-FPGAs, utilizando síntese, simuladores e co-simulação.

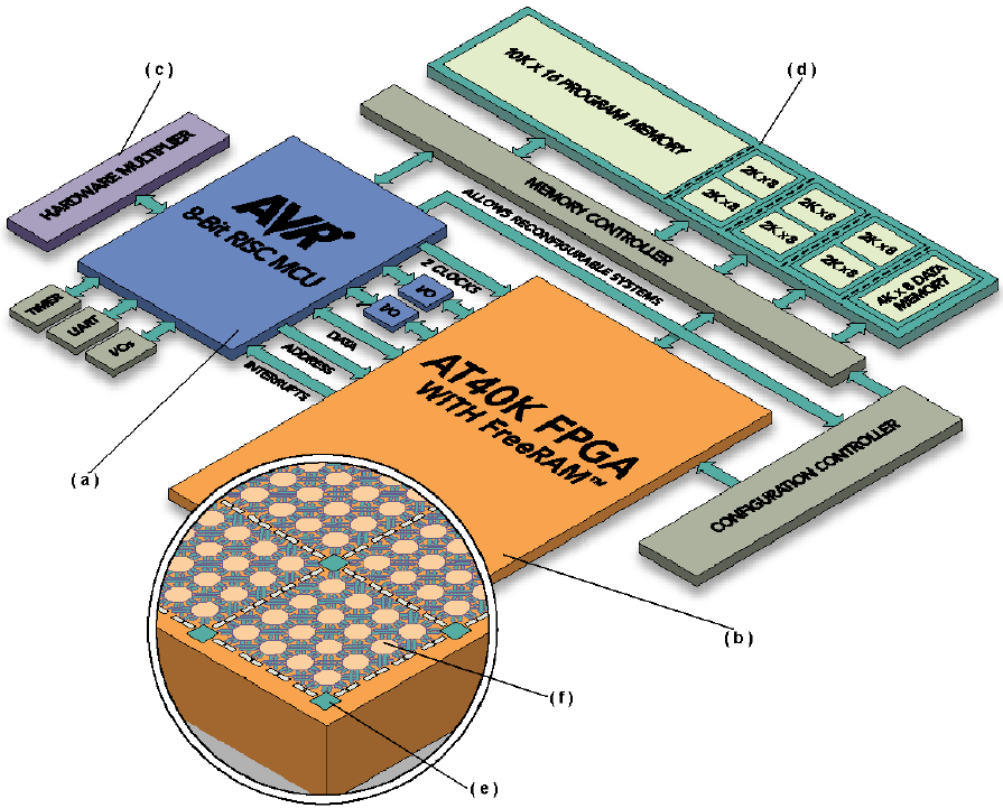


Figura 4.2: Organização do SoC FPSLIC, da Atmel.

Devido a esta integração entre componentes de hardware e software, problemas de implementação podem ser detectados em estágios preliminares de projeto, diminuindo o impacto desses problemas na estimativa de tempo do projeto. Esta integração permite também a redução no consumo de potência executando funções de algoritmos complexos em DSP ou FPGAs, ao invés de usar componentes em software. Além disto, o microcontrolador pode ser usado para controlar a *Cache Logic*, auxiliando no processo de reconfiguração dinâmica do sistema.

4.2 Xilinx

4.2.1 Características gerais de um FPGA Virtex

Cada FPGA da família Virtex possui CLBs, IOBs, blocos de RAM, recursos de relógio, roteamento programável e configuração do circuito elétrico. Essas funcionalidades lógicas são determinadas através de um arquivo de configuração. Arquivos de configuração contêm uma mescla de comandos e dados. Eles podem ser lidos e escritos através de uma das interfaces de configuração da Virtex.

A memória de configuração da Virtex pode ser vista como uma matriz bidimensional de bits. Estes bits são agrupados em quadros verticais com um bit de largura, e se estendem do topo à base da matriz. Um quadro é a unidade atômica de configuração: é a menor porção de memória de configuração que pode ser lida ou escrita.

Quadros são lidos e escritos seqüencialmente, com endereços crescentes para cada operação. Múltiplos quadros consecutivos podem ser lidos ou escritos com um único comando de configuração. A matriz de CLBs inteira, mais o bloco de interconexão de *SelectRAM* podem ser lidos ou escritos com apenas um comando. Cada bloco de conteúdo de *SelectRAM* deve ser lido ou escrito separadamente.

Como os quadros podem ser lidos ou escritos individualmente, é possível reconfigurar parcialmente esses dispositivos através da modificação desses quadros no arquivo de configuração. Além disto, a disposição regular de elementos permite ações de relocação e desfragmentação, que são importantes para reconfiguração parcial [COM99].

Os elementos configuráveis são CLBs. A Figura 4.3 mostra um CLB do FPGA Virtex XCV300.

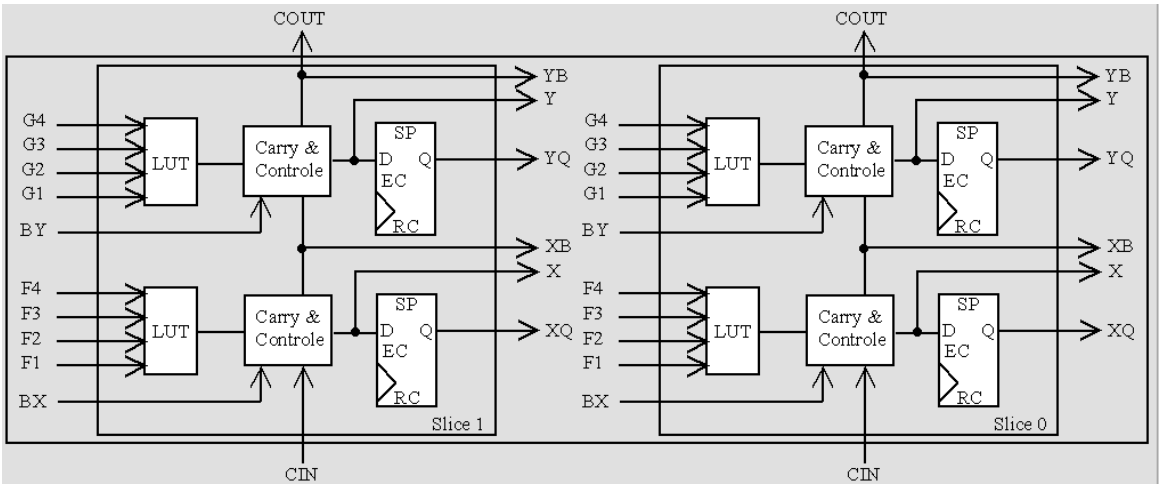


Figura 4.3: Esquema de um CLB do FPGA XCV300

Nesta Figura nota-se que cada CLB contém duas fatias (*Slices*), que por sua vez contém duas LUTs cada, além de recursos de *Carry*[2] e dois *flip-flops*.

Os elementos internos ao FPGA (CLBs, IOBs, BlockRAMs, recursos de relógio e roteamento programável) são distribuídos pelo dispositivo de forma regular. Tal regularidade pode ser denotada através da disposição desses elementos em colunas, conforme a Figura 4.4.

[2]Propagação rápida de "Vai-um", para acelerar operações aritméticas.

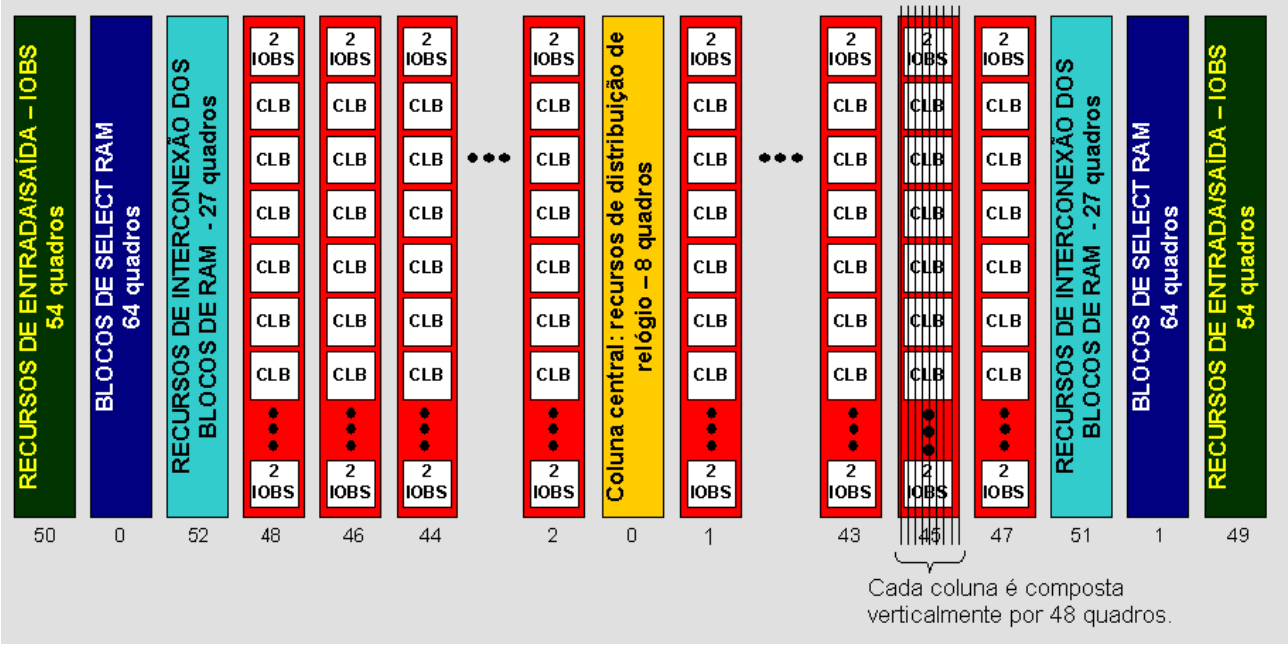


Figura 4.4: Disposição em colunas dos elementos do FPGA Virtex XCV300.

Na Figura 4.4 ainda podem ser vistos os quadros em relação à uma coluna. Cada coluna CLB é cortada verticalmente por 48 quadros sucessivos. Como a disposição das CLBs é em colunas, a modificação de uma CLB implica na alteração de todas as CLBs da coluna a que pertence. Note-se ainda na mesma Figura que as colunas são numeradas a partir do 0 (zero) - atribuído à coluna central. As demais colunas são numeradas em ordem crescente, com valores pares à esquerda da coluna central, e ímpares à sua direita. Tal numeração será importante para localização de elementos, conforme o item 4.2.3.

Conforme mencionado anteriormente, a memória de configuração da Virtex pode ser vista como uma matriz regular de bits. Esses bits são agrupados em quadros verticais com um bit de largura, que se estendem do topo à base da matriz.

Cada quadro contém informações de cada elemento de uma coluna. Cada CLB é cortado por 48 quadros, utilizando 18 bits de cada um. Desta forma, um CLB é completamente configurado por 864 bits. A Figura 4.5 mostra uma abstração da composição de um quadro de uma coluna de CLBs. Os 18 primeiros bits de um quadro correspondem aos 2 IOBs do topo da coluna, e os 18 últimos, aos 2 IOBs da base da coluna (Figura 4.4). Entre esses grupos, há conjuntos de 18 bits correspondentes aos CLBs do dispositivo. Para o dispositivo XCV300, há 32 desses conjuntos.

No arquivo de configuração dos FPGAs da família Virtex as informações são gravadas em conjuntos de 32 bits, chamados *palavras*. Portanto, o quadro deve conter um número de bits que seja múltiplo de 32. Por exemplo, para quadro ilustrado na Figura 4.5 há 612 bits (576 para CLBs e 36 para os IOBs). Como 612 não é divisível por 32 (o resto desta divisão seria 0,125 palavras), são

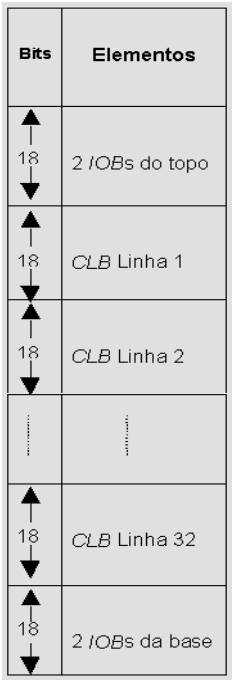


Figura 4.5: Abstração de um quadro do FPGA XCV 300.

adicionados 28 bits, de forma que o quadro passa a ter 640 bits - o que equivale a 20 palavras. Uma vez que a organização dos quadros da Virtex exige uma palavra de preenchimento[3] ao final de cada quadro [XIL00a], uma coluna de CLBs do XCV300 possui 21 palavras, ou 672 bits de altura.

As características dos FPGAs membros da família Virtex podem ser vistas na Tabela 4.1. Nela constam o número de colunas e linhas de cada dispositivo, bem como o número de bits e palavras por quadro. Estas informações são relevantes para as equações que serão analisadas nas Seções adiante.

Tabela 4.1: Diferenças entre dispositivos da família Virtex.

Dispositivo	Linhas	Colunas	Bits por quadro	Palavras por quadro
XCV50	16	24	384	12
XCV100	20	30	448	14
XCV150	24	36	512	16
XCV200	28	42	576	18
XCV300	32	48	672	21
XCV400	40	60	800	25
XCV600	48	72	960	30
XCV800	56	84	1088	34
XCV1000	64	96	1248	39

Calculando-se o total de quadros, pode-se chegar ao número de bits necessários à configuração de um FPGA Virtex. Este total é dado pela Equação 4.1 :

[3]Palavra contendo 32 zeros

$$T_{quadros} = (IOB + RAM + RAMConnect) \times 2 + (Chip_{cols} \times 48) + relogio + 3 \qquad (4.1)$$

Onde:

- IOB é o número de quadros por coluna de IOBs (54);
- RAM é o número de quadros por coluna de RAM (64);
- RAMConnect é o número de quadros por coluna de RAMConnect (27);
- A constante 2 significa que há dois conjuntos de IOB+RAM+*RAMConnect* (um à esquerda e outro à direita do FPGA);
- $CHIP_{Cols}$ é o número de colunas de CLBs do dispositivo (48)[4];
- Relógio é a coluna central da Virtex, que possui 8 quadros;
- A constante 3 representa que há um quadro de preenchimento depois de cada coluna de RAM, e outro para complementar as colunas de CLBs + *RAMConnect*.

Por exemplo, substituindo as constantes na equação, tem-se que o total de quadros do FPGA XCV300 é de 2605. Como cada quadro possui 21 palavras de 32 bits, o XCV300 possui **1.750.560** bits de configuração, excetuando-se palavras de comando e sincronização, que serão discutidas na próxima Seção.

4.2.2 Formato do arquivo de configuração

Os bits de configuração de um FPGA da família Virtex são organizados em um arquivo de configuração chamado de *bitstream*. Uma ferramenta para síntese de alto nível pode gerar dois formatos de *bitstream*. O primeiro é o formato binário, mais compacto, utilizado normalmente para configurar FPGAs (possui a extensão ".bit"). O segundo tem o formato ASCII, utilizado para fins didáticos (possui a extensão ".rbt"). É possível realizar o *download* de ambas formas, mas através da segunda é possível analisar a estrutura do arquivo de configuração.

[4]Os números entre parênteses correspondem aos quadros de um XCV300

O *bitstream* é composto por um fluxo de palavras que segue o protocolo de configuração determinado pelo fabricante para o dispositivo [XIL00a]. Este protocolo de configuração é controlado por uma coleção de registradores de 32 bits. A lógica de configuração é controlada e acessada através desses registradores, que são exibidos na Tabela 4.2.

Tabela 4.2: Endereços dos registradores de configuração.

Registrador	Mnemônico	Endereço
CRC	CRC	0000
Frame Address	FAR	0001
Frame Data Intput	FDRI	0010
Frame Data Output	FDRO	0011
Command	CMD	0100
Control	CTL	0101
Control Mask	MASK	0110
Status	STAT	0111
Legacy Output	LOUT	1000
Configuration Option	COR	1001
Reservado	-	1010
Frame Length	FLR	1011
Reservado	-	1100
Reservado	-	1101
Reservado	-	1110
Reservado	-	1111

O formato do registrador no arquivo de configuração é exibido na Figura 4.6. Segundo o fabricante do dispositivo, os campos que estão preenchidos com zero (0) ou um (1) devem permanecer desta forma. Nesta figura, um "X" indica um campo onde o bit correspondente é variável, e deve ser escrito. As linhas verticais são usadas para facilitar a leitura, separando a palavra em *nibbles*.

Tipo				OP				Endereço do Registrador												Reserv	Campo de dados											
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	x	x	0	0	0	0	0	0	0	0	0	0	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x

Figura 4.6: Formato de um registrador de comando.

A Figura 4.7 mostra a estrutura do arquivo de configuração para um dispositivo XCV300, composto por 54.744 palavras (51975 palavras com informações correspondentes aos dados dos quadros, e o restante são palavras referentes aos registradores de configuração e palavras de enchimento). Há uma seqüência de inicialização composta por uma palavra de enchimento, uma palavra de sincronização e 16 palavras que correspondem aos comandos de escrita de valores nos registradores internos de configuração do FPGA (Figura 4.7-I). A última palavra da inicialização é uma escrita no registrador FDRI e indica quantas palavras de dados (quadros com informações de configuração da lógica

a ser implementada) seguir-se-ão (Figura 4.7-II). Nestas palavras é que está escrita a lógica que será implementada no FPGA.

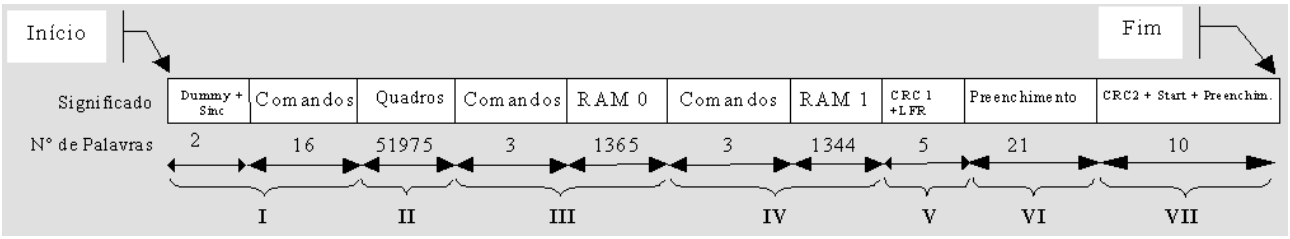


Figura 4.7: Estrutura do arquivo de configuração do XCV300.

Seguem-se dados do primeiro bloco de RAM (Figura 4.7-III) que é precedido por comandos que informam que serão gravados dados em RAM. Essa estrutura repete-se para o segundo bloco de RAM (Figura 4.7-IV). O número de palavras dos blocos de RAM diferem porque o primeiro bloco é seguido de um quadro de enchimento.

O arquivo de configuração possui um método para verificação de erro, através do registrador de verificação de redundância cíclica (*Cyclic Redundancy Check* - CRC). Quando uma informação é escrita em algum registrador (exceto o LOUT) um valor de CRC de 16 bits é calculado, usando tanto os dados como o endereço do registrador. Este valor é armazenado no registrador de CRC. Ao final de uma série de escritas, um valor pré calculado é escrito no registrador. Se o resultado for um valor diferente de zero, isto indica que algum erro ocorreu, e o bit *CRC_ERROR*, acessível através do registrador de estado, é setado. A Figura 4.8 ilustra este processo. O CRC é calculado em dois pontos do arquivo de configuração. O primeiro ocorre logo após a escrita do segundo bloco de RAM (porção V da Figura 4.7) e o segundo aconte depois da escrita do último quadro de preenchimento (parte VII da Figura 4.7)

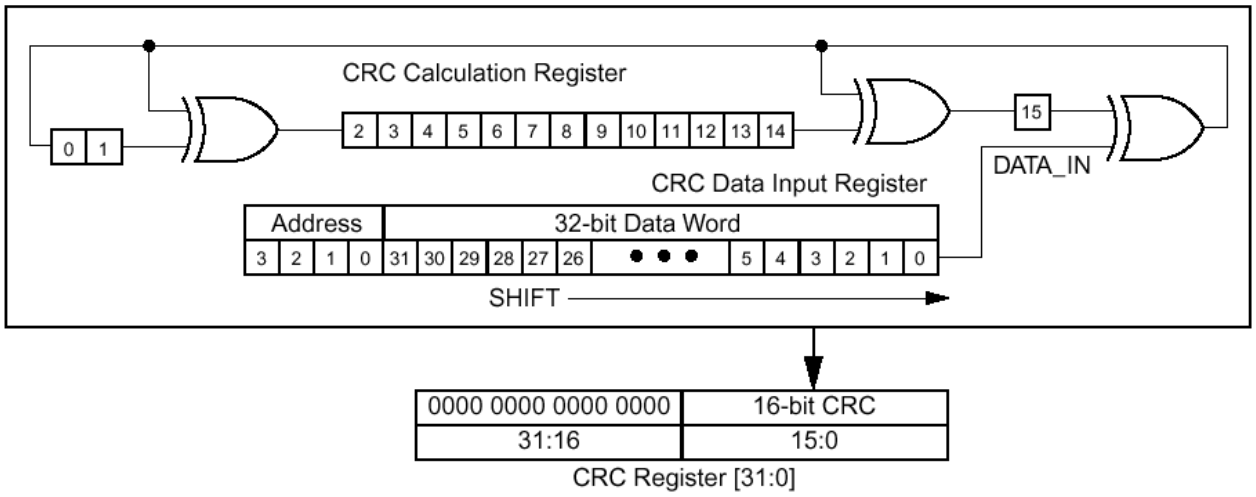


Figura 4.8: Cálculo do CRC.

A Figura 4.7-V inclui a escrita de 3 palavras que indicam que será escrito o último quadro (Figura 4.7-VI). A primeira seleciona o registrador CMD. A seguinte escreve no CMD a indicação de último quadro (LFRM - *last frame*). A terceira é a seleção do registrador FDRI, que precede a escrita de um ou mais quadros de configuração.

Ainda antes de completar *bitstream*, há comandos informando a inicialização do dispositivo. Isto se dá através de 4 palavras (Figura4.7 - VII), que são:

- ➢ seleção do registrador CMD (1^a palavra);
- ➢ escrita no registrador CMD (2^a palavra). Essa escrita é a indicação da inicialização do dispositivo, através do comando "*START*";
- ➢ seleção do registrador CTL (*control* - 3^a palavra);
- ➢ escrita no registrador CTL, com a indicação do modo de leitura ou escrita, a utilização de *buffers tri-state*, além de informar se a interface da configuração atual permanecerá após a reconfiguração (4^a palavra).

Depois ocorre a seleção do registrador de CRC, e a conseqüente escrita de seu valor final. Ainda conforme a Figura 4.10, as últimas 4 palavras são um preenchimento utilizado para indicar o fechamento do arquivo (Figura 4.7-VII).

A identificação dos registradores de comandos e do formato da palavra de dados para cada um deles é necessária para que seja possível a modificação dos parâmetros para uma reconfiguração parcial. Por exemplo, uma reconfiguração pode ser disruptiva ou não, dependendo da seleção de um bit no registrador COR. Logo após a seleção deste registrador, o bit 15 da palavra de dados indicará se o modo de inicialização do FPGA será *StartUp* (dinâmico) ou *ShutDown* (estático) - na Figura 4.9 a inicialização é estática.

A Figura 4.9 mostra as 18 primeiras palavras de um arquivo de configuração para o dispositivo XCV300. Para fins didáticos, o arquivo foi comentado. Ao lado das palavras de configuração há uma breve descrição de seu significado.

Já a descrição das últimas 36 palavras do arquivo de configuração pode ser acompanhada através da Figura 4.10.

A próxima Subseção vai explicar outro aspecto de grande relevância para obtenção de uma reconfiguração parcial. Trata-se da localização de elementos (LUTs, e determinados bits de uma LUT) dentro do FPGA.

```
Xilinx ASCII Bitstream
Createdby Bitstream D.26                ; Comentado por Daniel Mesquita  em Ter 01 de maio de 2001
Designname: semaforo.ncd
Architecture: virtex
Part:         v300bg352
Date:         Wed Mar 28 20:49:48 2001
Bits:         1751808
111 11 1111111111 1111 1111111111111  ; Palavra de preenchimento
101 01 0101001100 1010 1010101100110  ; Palavra de sineronização

;Op                    reg
001 10 0000000000 0100 0000000000001  ; Seleciona registrador CMD (haverá um comando)
000 00 0000000000 0000 0000000000111; escreve dados no CMD (Reset CRC)

001 10 0000000000 1011 0000000000001; Seleciona registrador FLR (comprimento do quadro)
000 00 0000000000 0000 0000000010100; 20 palavras (em decimal). Correto para a XCV300

001 10 0000000000 1001 0000000000001; Seleciona registrador COR
000 00 0001010000 0001 1111100101101; seleciona várias opções de configuração(entre elas o relógio   4,3MHz)

001 10 0000000000 0110 0000000000001; Seleciona registrador MASK
000 00 0000000000 0000 0000000000000; Escreve máscara padrão

001 10 0000000000 0100 0000000000001; Seleciona registrador CMD
000 00 0000000000 0000 0000000001001; Muda freqüência do relógio para o indicado pelo COR

001 10 0000000000 0001 0000000000001; Seleciona registrador FAR (Endereço do quadro)
000 00 0000000000 0000 0000000000000; Quadro inicial = 0

001 10 0000000000 0100 0000000000001; Seleciona registrador CMD
000 00 0000000000 0000 0000000000001; WCFG - escrita de dados de configuração conforme FDRI

001 10 0000000000 0010 0000000000000; Seleciona registrador FDRI, para gravação de dados no FPGA
010 10 0000000000 0110 0101100000111; indica o número de palavras:  51975 para XCV300
.
.
.
```

Figura 4.9: Início do arquivo de configuração para a XCV300.

4.2.3 Endereçamento de elementos

Para que seja factível uma reconfiguração parcial, além do domínio da escrita nos registradores de um FPGA, faz-se necessária a possibilidade de localizar determinados bits no arquivo de configuração. A tarefa a princípio é árdua, pois que somadas todas as palavras de um *bitstream* (conforme a Figura4.7) e multiplicadas por sua largura em bits (32), tem-se que localizar um bit dentre 1.750.560 bits!

Analisando as equações apresentadas em [XIL00a], é possível estabelecer um roteiro para localização de um determinado bit de uma CLB no arquivo de configuração. Como conseqüência disto, torna-se possível a leitura de um conjunto de elementos. Este trabalho está direcionado para realização da reconfiguração parcial de LUTs configuradas como memória (*LUTSelectRAM*) [XIL99]. Assim, os exemplos a seguir referem-se à localização do bit 14 de uma F-LUT.

Conforme a Figura 4.11, esse bit está dentro da fatia 0 (S0) da CLB situada na intersecção entre a

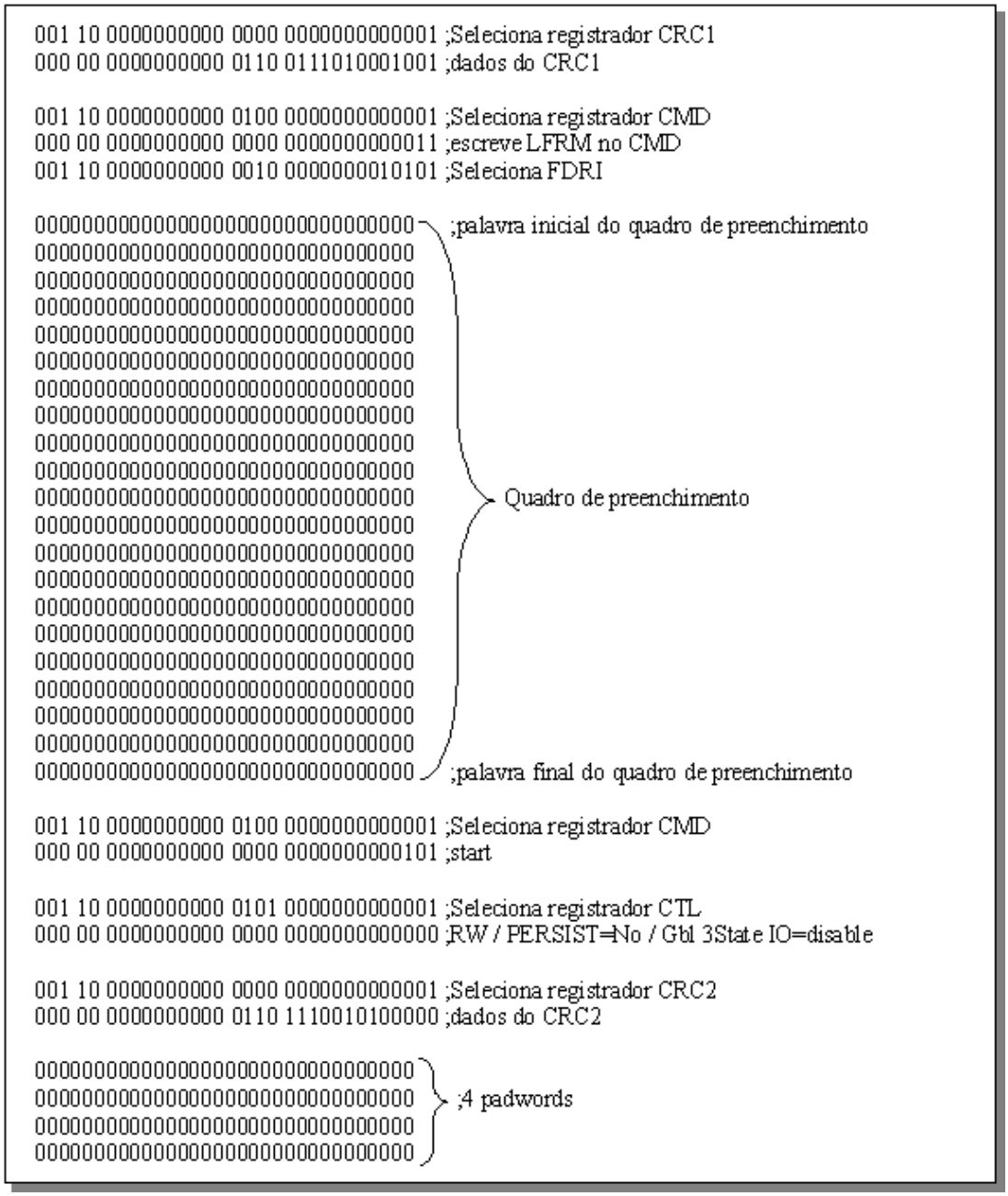


Figura 4.10: Final de um arquivo de configuração de um XCV300.

primeira linha (R1) e a primeira coluna (C1) de um FPGA XCV100. O posicionamento de uma CLB em coordenadas pré-definidas pode ser obtido através da edição do arquivo de restrições de usuário (UCF, do inglês *User Constraints File*). As coordenadas de uma CLB são dadas, pois, por Linha (*Row*), Coluna (*Column*) e Fatia (*Slice*), no seguinte formato: R1C1.S0.

O primeiro passo para a localização desse bit no arquivo de configuração é o cálculo do *Major Address* (MJA), ou seja, a coluna onde está a CLB que contém o bit em questão. Para tanto se fazem necessárias algumas informações (será usada a notação e as equações sugeridas em [XIL00a]):

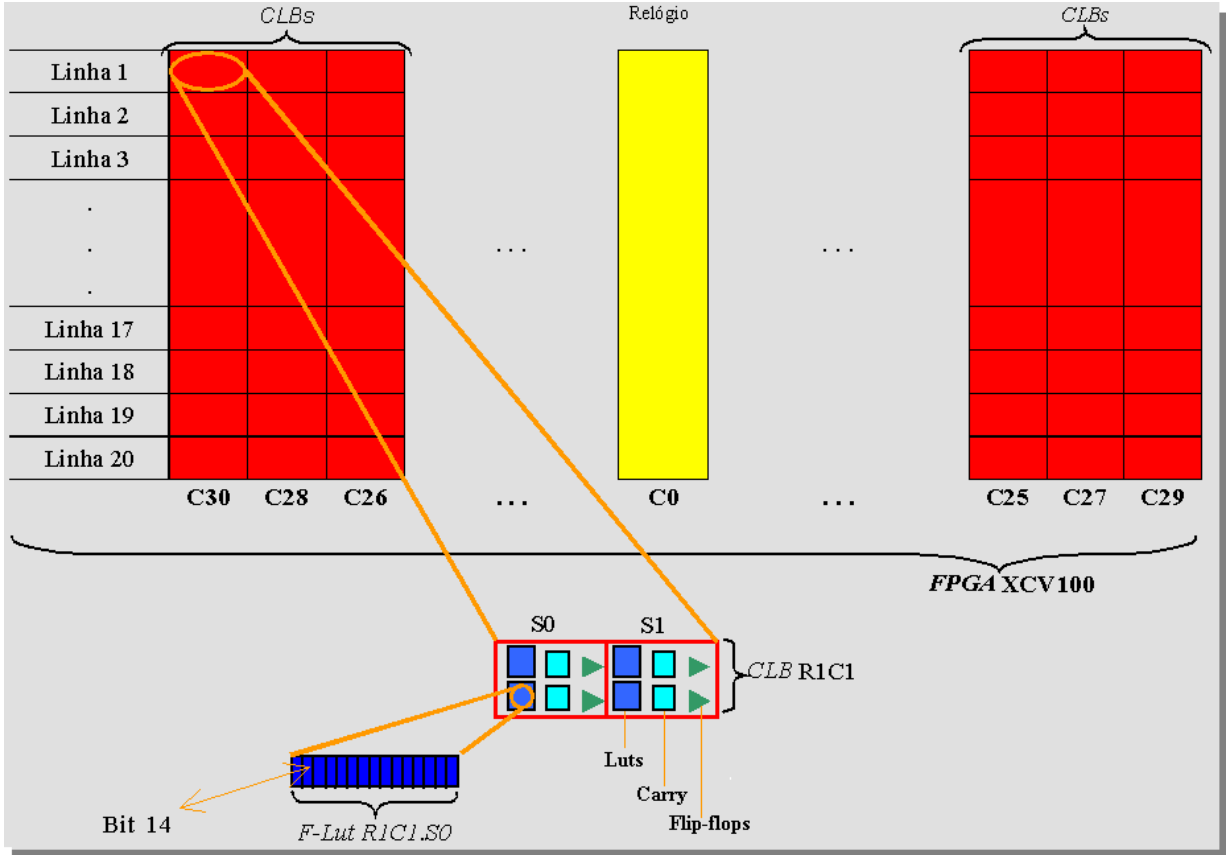


Figura 4.11: Localizando o bit 14 de uma F-LUT para uma dada CLB.

➢ Linha onde encontra-se a CLB desejada: CLB_{row};

➢ Coluna onde encontra-se a CLB desejada: CLB_{col};

➢ Número de colunas do dispositivo: $Chip_{cols}$;

Além disto, há que se aplicar o seguinte algoritmo abaixo:

$$\text{se } (CLB_{col} \leq \tfrac{Chip_{cols}}{2}) \text{ então}$$
$$MJA \leftarrow Chip_{cols} - (CLB_{col} \times 2) + 2$$
$$\text{senão}$$
$$MJA \leftarrow (CLB_{col} \times 2) - Chip_{cols} - 1$$

Considerando o dispositivo XCV100, e aplicando as informações no algoritmo acima, para o caso dado, o *Major Address* será:

$$MJA = 30 - (1 \times 2) + 2 = 30$$

Assim, como mostrado na Figura 4.11, a CLB denotada por R1C1.S0 encontra-se na coluna 30 do FPGA.

O passo seguinte é identificar em qual quadro desta coluna encontra-se o bit desejado. Tal informação é chamada *Minor Address*, ou MNA, e é dada pela Equação 4.2:

$$MNA = lut\_bit + wd - slice \times (2 \times lut\_bit + 17) \tag{4.2}$$

Onde:

- *wd* equivale ao número de bits por palavra (32)
- a diferença entre *lut_bit* + *wd* e o restante da equação deve-se ao deslocamento horizontal dependente da fatia onde se encontra o bit desejado. Na Figura 4.12 pode ser visto que há uma distância em quadros considerável entre as LUTs da fatia 0 e as LUTs da fatia 1. Se *Slice* = 0, *MNA* = *lut_bit* + 32. Se *Slice* = 1, *MNA* = 15 - *lut_bit.*

Utilizando a Equação 4.2 obtém-se:

$MNA = 14 + 32 - 0 \times (2 \times 14 + 17)$, portanto, MNA = 46.

A Figura 4.12 mostra um ilustração do que foi obtido até agora. Através do MJA encontrou-se a coluna desejada. O MNA indicou em qual quadro encontra-se o bit procurado. Contudo, até o momento não é sabida a altura (em bits) onde se localiza o bit 14 da F-Lut contida em R1C1.S0.

A posição vertical do bit (relativa ao quadro) é obtida através do cálculo do *Frame Bit Index*, ou *fm_bit_idx*, pela Equação 4.3:

$$fm\_bit\_idx = 3 + 18 \times CLB_{row} - FG + RW \times 32 \tag{4.3}$$

Onde:

- *FG* vale 1 caso o bit desejado encontre-se na G-Lut, e 0 para F-Lut;
- CLB_{row} é multiplicado por 18 porque esta é a altura em bits de um CLB, logo esta multiplicação fornece o início de um CLB, conforme a linha onde se encontra;
- O valor 3 adicionado indica que o bit inicial de uma LUT fica 3 bits abaixo do início de um CLB;

➢ Conforme [XIL00a], quando escreve-se um arquivo de configuração no FPGA, uma palavra de preenchimento é colocada ao final de cada quadro. Quando se obtém um arquivo de configuração por *ReadBack*[5], uma palavra de preenchimento precede as palavras de informações válidas. Logo, para *ReadBack*, *RW*=1. Como neste caso é uma operação de escrita, *RW*=0.

Substituindo os valores deste estudo de caso na equação, obtém-se:

$fm\_bit\_idx = 3 + 18 \times 1 - 0 + 0 \times 32 = 21$

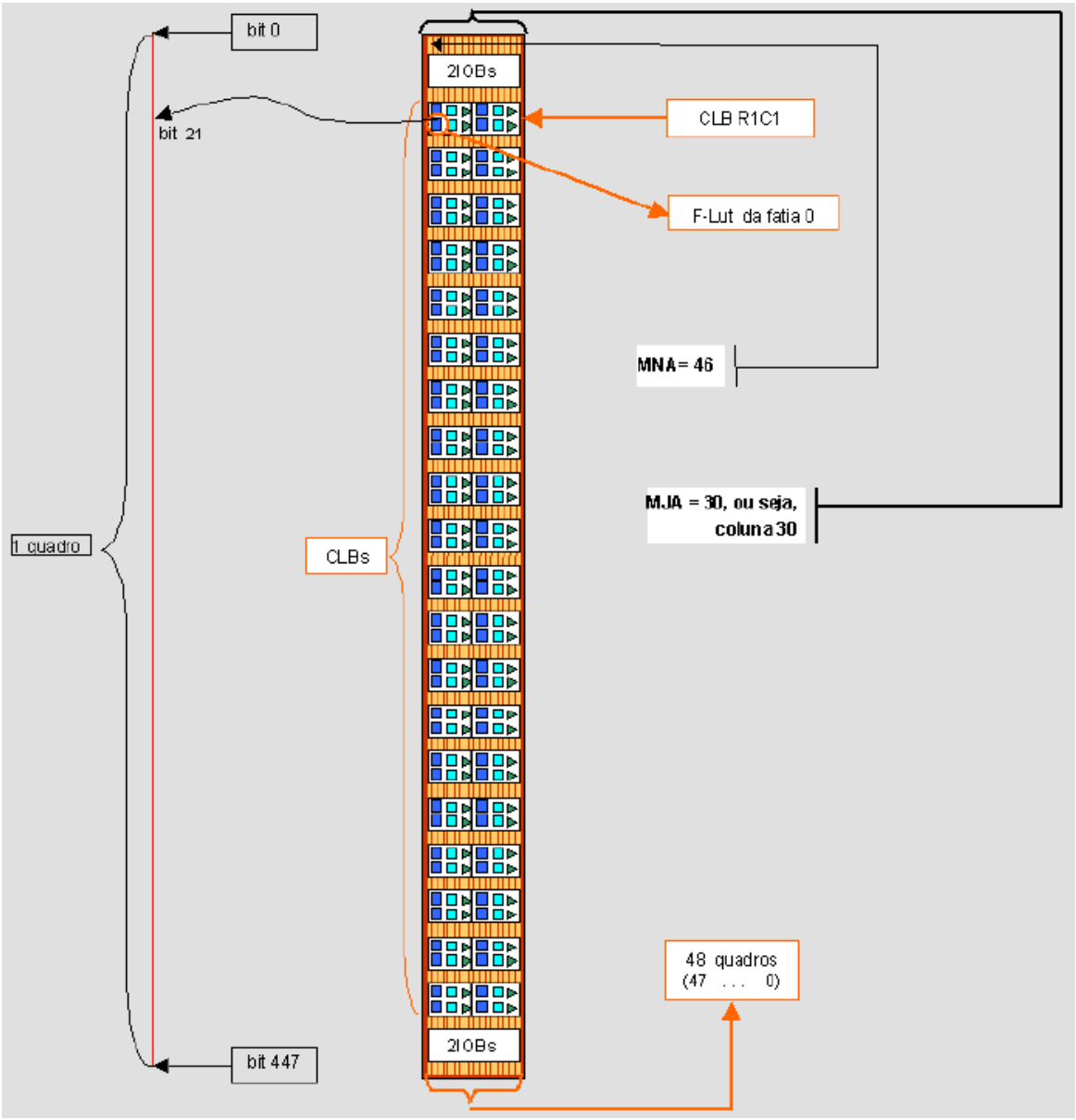


Figura 4.12: Coluna 30 de um XCV100.

[5]Operação de leitura do estado de um dispositivo físico.

Portanto, para este caso, o bit 14 da F-Lut em questão está na posição 21 do quadro 46, tudo na coluna 30 do FPGA XCV100, como também pode ser observado na Figura 4.12.

Contudo, não basta saber onde o bit está fisicamente. Para que seja procedida a leitura deste bit, é imprescindível sua localização no arquivo de configuração. Como visto na Tabela 4.1, um quadro da XCV100 é composto por 14 palavras de 32 bits. Então, a primeira informação necessária para localização do bit no arquivo de configuração é a identificação da palavra inicial do quadro (*Frame Start Word*, ou *fm_st_wd*) que contém esse bit.

Para este cálculo, há que se levar em consideração os seguintes tópicos:

1. O número de quadros do relógio (8) deve ser "saltado", já que está no início do arquivo de configuração (8 quadros após o primeiro comando de FDRI);

2. Deve-se somar ao valor 8 o deslocamento relativo ao número de colunas ($(MJA-1)*48$), e o número de quadros (MNA);

3. O número de palavras por quadro do dispositivo (*FL*=14) deve multiplicar a resultante do cálculo acima, para que se obtenha a palavra inicial do quadro;

4. Como o caso que está sendo analisado trata-se da escrita de um bit, adiciona-se *RW x FL*, com *RW* = 0.

Então chega-se à Equação 4.4:

$$fm\_st\_wd = FL \times (8 + (MJA - 1) \times 48 + MNA) + RW \times FL \tag{4.4}$$

Aplicando-se os valores deste exemplo, tem-se:

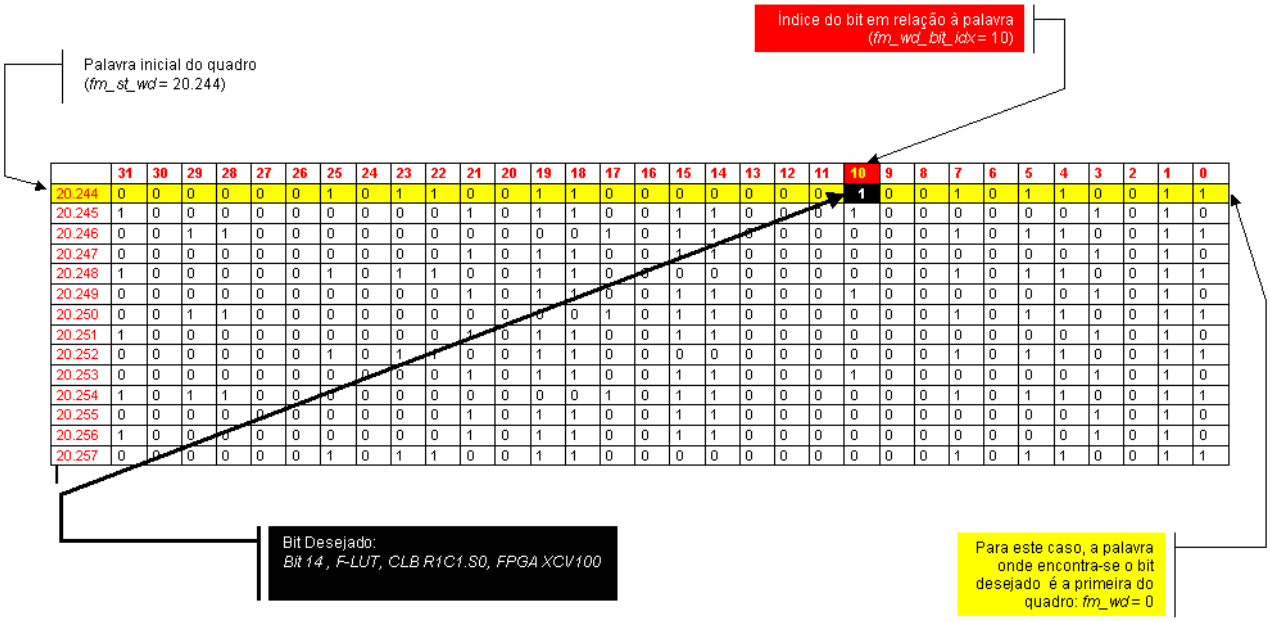
$fm\_st\_wd = 14 \times (8 + (30 - 1) \times 48 + 46) + 1 \times 14 = 20.244$

Resta, pois, computar dois valores para que seja localizado o bit exato através do arquivo de configuração: o primeiro é a palavra em que esse bit se encontra, e o segundo é a posição desse bit na palavra. A Figura 4.13 ilustra o quadro que se inicia na palavra 20.244 do arquivo de configuração.

Dada a palavra em que o quadro é iniciado, a palavra em que se encontra o bit desejado é obtida pela divisão do índice do bit em relação ao quadro (*fm_bit_idx*) pelo número de bits da palavra, sendo que o resultado deve ser truncado[6] (Equação 4.5).

$$fm\_wd = trunc(\frac{fm\_bit\_idx}{32}) \tag{4.5}$$

Para o exemplo em questão, *fm_ wd* = 0, portanto, o bit desejado está na palavra 20.244. Ou seja, na Figura 4.13 corresponde à linha 0.



	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
20.244	0	0	0	0	0	0	1	0	1	1	0	0	1	1	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	0	1	1
20.245	1	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	0	0	0	1	0	0	0	0	0	0	1	0	1	0
20.246	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	1	0	1	1	0	0	1	1
20.247	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0
20.248	1	0	0	0	0	0	1	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1
20.249	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	1	0	1	0
20.250	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	1	0	1	1	0	0	1	1
20.251	1	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0
20.252	0	0	0	0	0	0	1	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1
20.253	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	0	0	0	1	0	0	0	0	0	0	1	0	1	0
20.254	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	1	0	1	1	0	0	1	1
20.255	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0
20.256	1	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0
20.257	0	0	0	0	0	0	1	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1

Figura 4.13: Um quadro de um XCV100.

O último passo, então, é encontrar o bit na palavra. Este cálculo é dado pela equação 4.6:

$$fm\_wd\_bit\_idx = 31 + 32 \times fm\_wd - fm\_bit\_idx \tag{4.6}$$

Observa-se na Figura 4.13 que desta forma foi localizado o bit 14 da F-Lut da CLB R1C1.S0.

Recapitulando o problema (Figura 4.11) e a seqüência das equações, tem-se que MJA (Figura 4.4 e Figura 4.12) indica a coluna que contêm a CLB R1C1.S0. O MNA informa o quadro onde está o bit que se quer encontrar (Figura 4.12). *Fm_bit_idx* mostra a posição desse bit em relação ao quadro, conforme a Figura 4.12. *fm_st_wd* indica, no arquivo de configuração, a palavra onde o quadro informado pelo MNA é iniciado (Figura 4.13). *fm_wd* é o deslocamento, em termos de palavras, do início do quadro até a palavra que contém o bit procurado (Figura 4.13). Finalmente, *fm_wd_bit_idx* localiza o bit desejado dentro de uma palavra.

Desta forma está compreendida a localização de um determinado bit dentro de um FPGA Virtex, o que é essencial para que se proceda a uma reconfiguração parcial. No próximo capítulo serão descritas ferramentas para manipulação de *bitstreams* desenvolvidas a partir do estudo apresentado nesta Seção.

[6] *trunc(x)* é uma função que retorna o maior inteiro não maior que x. Por exemplo, trunc(2,7) = 2

Capítulo 5

Ferramentas para reconfiguração remota e parcial

Como uma das lacunas para o desenvolvimento da tecnologia de reconfiguração dinâmica de sistemas digitais é a falta de ferramentas de CAD, durante a pesquisa que originou esta dissertação, buscou-se desenvolver um conjunto de ferramentas que contribuísse à diminuição dessa carência.

Este Capítulo mostra dois conjuntos de aplicativos desenvolvidos para manipular arquivos de configuração, desde um nível mais baixo (bit) até a reconfiguração de módulos completos de hardware. O primeiro conjunto compreende aplicativos baseados nas classes JBits para prover reconfiguração remota de FPGAs, além de possibilitar que essa reconfiguração ocorra sem que o usuário tenha conhecimento de detalhes arquiteturais do dispositivo com o qual está trabalhando.

O segundo conjunto é composto por outros três aplicativos desenvolvidos sem o auxílio do JBits, baseados apenas nos estudos da estrutura e organização interna de FPGAs da família Virtex, descritos na Seção 4.2. O primeiro aplicativo deste conjunto permite manipular um arquivo de configuração em nível de bit, oferecendo grande flexibilidade e permitindo a geração de um *bitstream* parcial. O segundo aplicativo trata apenas de esconder alguns detalhes referentes aos protocolos de configuração, permitindo ao usuário manipular sua aplicação no nível de LUTs. O terceiro programa deste conjunto é uma ferramenta de alto nível que permite a unificação de *cores* através de uma interface gráfica, gerando um novo arquivo de configuração a partir de dois *cores* iniciais.

A necessidade de uma certa compatibilização entre os *cores* que serão inseridos ou removidos de determinado sistema induz ao desenvolvimento de uma interface de comunicação entre esses módulos, e deles com o mundo externo. Esta proposta é apresentada na Seção 5.6.

5.1 Configurador de *bitstream*

A primeira ferramenta desenvolvida tem o objetivo de manipular dados contidos em LUTs, fazendo uso das classes oferecidas pelo JBits. Todo o trabalho de reconfiguração é feito sobre o *bitstream*, ou seja, com *hard cores* e não *soft cores* [PAL01]. A aplicação criada permite as seguintes funcionalidades:

1. Acessar arquivos de configuração (parciais ou completos);
2. Manipular dados contidos em LUTs (localizar e alterar);
3. Salvar as alterações e realizar *download* no dispositivo, local ou remotamente.

As duas primeiras funcionalidades implicam na possibilidade de modificar o comportamento do circuito pela alteração dos seus parâmetros de configuração do circuito. A terceira funcionalidade pode significar uma grande redução de custos para o projetista do sistema. Por exemplo, um determinado fabricante detecta um erro em seu circuito. Se reconfiguração remota é utilizada, ele pode atualizar o hardware de todos os seus clientes à distância, sem que estes percebam que o sistema foi modificado. Outros exemplos, como atualização do hardware em função de evolução na tecnologia (por exemplo, um novo algoritmo de compressão de imagem para *set-top box*) também são válidos.

A ferramenta em questão foi desenvolvida em Java. Ela permite acessar remotamente os *bitstreams* e visualizar suas configurações. Possibilita modificação de LUTs, procura de uma LUT com uma configuração específica e visualização da listagem de todas as LUTs com configurações diferentes da configuração padrão. Após modificar o *bitstream* é possível salvá-lo e baixá-lo remotamente para a placa sem a necessidade de nenhum software extra.

A utilização do JBits é bastante simples. A Tabela 5.1 ilustra os principais métodos disponíveis no JBits para manipular um *bitstream*. Já a Figura 5.1 mostra um trecho de código onde um desses métodos é utilizado. No caso, trata-se de uma leitura de valores de uma LUT. As linhas 339 a 342 da Figura 5.1 são responsáveis por receber do usuário as coordenadas da LUT (linha e coluna da CLB, fatia e LUT F ou G). Já a linha entre duas setas, na mesma Figura, mostra como o método $jbits.get$ é utilizado. Esse método retorna 16 valores que correspondem aos bits de uma LUT. No caso do exemplo em questão, esses valores são armazenados em um vetor (val) para que sejam posteriormente modificados.

Uma observação importante é quanto ao endereçamento das CLBs por parte das classes JBits e da documentação da Xilinx. Por exemplo, no FPGA XCV300 as CLBs são numeradas de (1,1) a (32,48) [XIL00a]. Entretanto, no JBits as CLBs são numeradas de (31,0) a (0,47). Tais discrepâncias

Tabela 5.1: Alguns comandos do JBits.

Comando	**Descrição**
JBits jbits	Criar instância de JBits para posterior uso dos métodos
JBits jbits.read("entrada.bit")	Abrir um *bitstream*
jbits.get(row, col, LUT.SLICE0_F)	Capturar os valores de uma LUT
jbits.set(row, col, LUT.SLICE0_F, val)	Gravar novos valores em uma LUT
jbits.write("saida.bit")	Salvar um *bitstream*

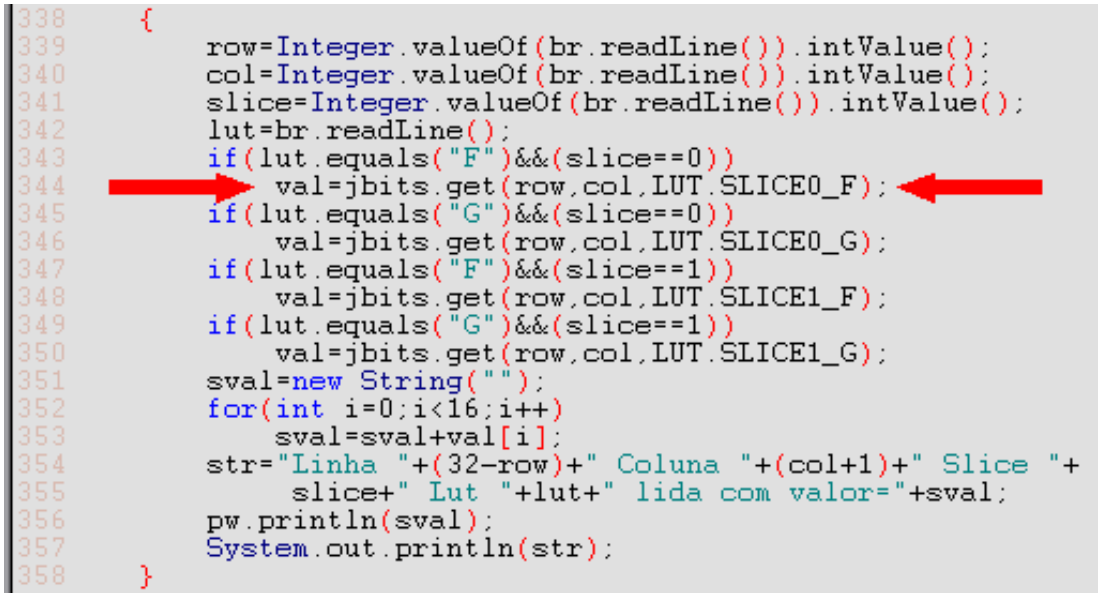


Figura 5.1: Trecho de código mostrando a utilização de um método da classe JBits.

atrasaram um pouco o desenvolvimento, pois não estavam documentadas, mas não impediram que os objetivos propostos para essa ferramenta fossem plenamente alcançados.

O configurador de *bitstreams* foi implementado seguindo o paradigma cliente-servidor. O lado servidor da aplicação (Figura 5.2) é responsável por atender os clientes e executar todas as funções citadas anteriormente e passar via *sockets* as respostas para os clientes. Na chamada da aplicação deve ser informado qual é o dispositivo alvo e qual a porta que aceitará conexões. Também é responsabilidade do servidor salvar um arquivo de configuração e realizar a chamada ao programa *JTAG-Programmer*, que realiza o *download* da configuração para o dispositivo alvo.

```
C:\Program Files\JBits2.7>servidor

C:\Program Files\JBits2.7>java BITProgrammerServidor XCV300 5000 log.txt c:\prog
ra~1\jbits2.7\bits\ jt.cmd
Servidor iniciado em: Thu Nov 22 11:52:43 BRST 2001
```

Figura 5.2: Lado servidor da aplicação para reconfiguração de *bitstreams*.

O lado cliente (Figura 5.3) provê uma interface que permite facilmente localizar elementos no

bitstream. O cliente é disponibilizado na forma de um *applet*, o que possibilita que remotamente se façam modificações em *bitstreams* previamente colocados na máquina servidora. No *applet* o usuário deve informar o endereço IP da máquina onde está sendo rodado o lado servidor da aplicação, bem como a porta que está aceitando conexões. O usuário também precisa conhecer previamente o caminho e o nome do arquivo de configuração que deseja alterar no servidor. O usuário não precisa saber qual é o dispositivo-alvo da aplicação, pois essa informação é resgatada do próprio *bitstream* lido.

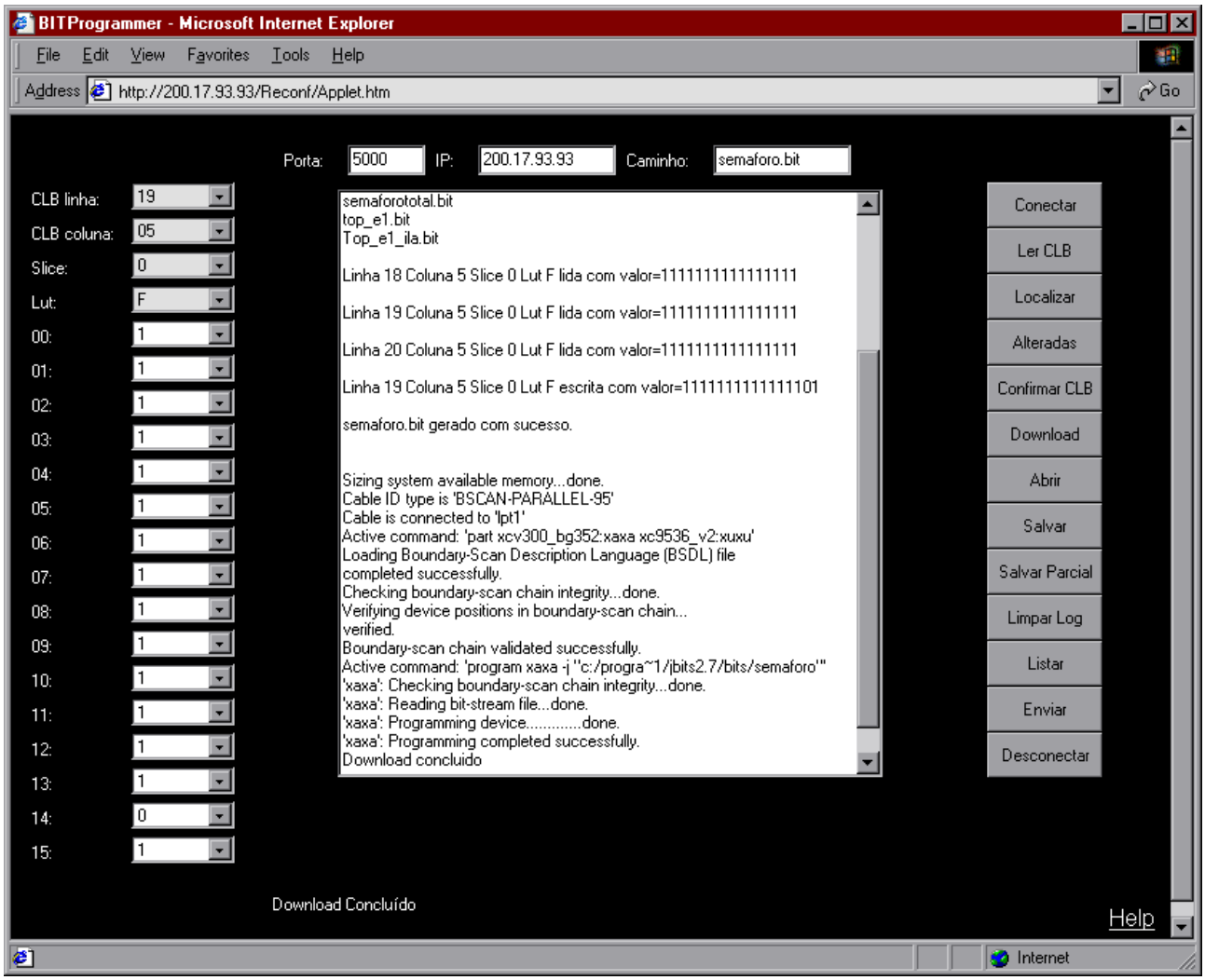


Figura 5.3: Lado servidor da aplicação para reconfiguração de *bitstreams*.

O exemplo da Figura 5.3 é baseado na aplicação descrita no item 5.3.2 deste documento. Esta Figura mostra, no retângulo central (com fundo branco), um relatório das ações realizadas na experiência. Primeiramente foram listados todos *bitstreams* disponíveis na máquina servidora (não aparecem todos porque houve uma rolagem na tela em função do espaço). Em seguida foram lidos os valores armazenados nas LUTs F de coordenadas R18C5.S0, R19C5.S0, e R20C5.S0 (o formato des-

sas coordenadas foi explicado no item 4.2.3). Tais LUTs foram escolhidas dadas as características da aplicação (as mesmas explicitadas na Tabela 5.3 - Na Seção seguinte). Em seguida foi alterado o bit 14 da F-LUT de coordenada R19C5.S0. Logo após o *bitstream* foi salvo e foi realizado o *download* remoto. O dispositivo foi configurado com sucesso, e a aplicação funcionou como esperado.

5.2 Reconfigurador de circuitos

Uma vez obtido sucesso com a reconfiguração remota de *bitstreams* com a ferramenta descrita na Seção 5.1, verificou-se a possibilidade de prover reconfiguração de um circuito em um nível mais alto que a manipulação de LUTs. Uma nova ferramenta foi desenvolvida, com o objetivo de “esconder" do usuário final a arquitetura do FPGA. Devido à complexidade na manipulação de *bitstreams* uma camada de software é acrescentada, de forma que a aplicação final (na forma de uma *applet*) mostre ao usuário apenas os parâmetros que devem ser alterados, sem que ele precise saber como e onde estão armazenados no circuito.

Determinadas aplicações em hardware podem ser flexíveis quanto a alguns parâmetros. Por exemplo, em um projeto realizado no GAPH em parceria com uma indústria de telecomunicações, foi projetado um circuito que substitui canais de uma portadora E1 [CAL01]. Este canal pode trabalhar com dados de E/S codificados de duas formas diferentes (HDB3 ou AMI). Além da codificação, há a definição de quantos e quais canais serão substituídos, o endereço inicial dos canais a substituir, e a transmissão ou não do canal de serviço. Todos esses parâmetros podem ser alterados remotamente, utilizando-se recursos da ferramenta descrita na Seção 5.1.

A Figura 5.4 ilustra os comandos para gerar o *applet* de configuração de uma determinada aplicação. Os quatro primeiros parâmetros dentro da *tag* “<APPLET> </APPLET>” informam o número de sinais passíveis de alteração, o nome do *bitstream*, o endereço IP do servidor e a porta de conexão. Os parâmetros seguintes informam os sinais que podem ser alterados. Por exemplo, o sinal “*code*” definirá se o circuito trabalhará com codificação AMI ou HDB3. Note-se ainda nesta Figura, que o formato dos parâmetros que o usuário enxerga pode ter a base numérica especificada. Neste exemplo existem parâmetros em hexadecimal e binário, mas pode haver sinais exibidos em decimal. Na próxima versão da ferramenta, contudo, algumas informações serão exibidas de forma ainda mais abstrata. Por exemplo, ao invés de escolher entre 0 ou 1 no campo $Code$ para especificar a codificação, será fornecida ao usuário uma *listBox* com as opções textuais AMI e HDB3.

Para cada parâmetro é possível definir as coordenadas da LUT onde se encontra (linha, coluna, fatia e LUT F ou G), se o valor que está sendo trabalhado está em hexadecimal ou binário, o bit inicial e o bit final que se deseja alterar. Excetuando-se estas informações de arquitetura, o projetista do hardware não precisa ter nenhum conhecimento a respeito do endereçamento de elementos do FPGA. E o usuário do sistema, por sua vez, não precisa saber sequer com qual FPGA está trabalhando: basta

conhecer a funcionalidade do seu circuito.

Uma vez editada a página html, ela é utilizada para reconfiguraçao e *download*. A Figura 5.5 mostra a aplicação tal qual o usuário do circuito a enxergará. Na *applet* de reconfiguração tem-se basicamente os parâmetros a inserir, o formato dos parâmetros, a leitura do *bitstream* original e o *download* do *bitstream* modificado. Fica claro que o usuário não precisa saber detalhes estruturais do FPGA com o qual está trabalhando: basta conhecer a funcionalidade do seu circuito.

```
<html>
  <head>
    <title>
      Parks
    </title>
  </head>

  <body bgcolor=blue>
    <center>
      <font color=white size=7 face="Arial Black">
        Parks
      </font>

      <APPLET code="BITGenerico.class" width=400 height=300>
        <PARAM name="nrosinais" value="8">
        <PARAM name="caminho" value="top_e1.bit">
        <PARAM name="ip" value="200.17.93.93">
        <PARAM name="porta" value="5000">
        <PARAM name="l[1]" value="CRCControl     ,bin,32,37,G,0,0,0">
        <PARAM name="l[2]" value="Code           ,bin,32,37,G,0,1,1">
        <PARAM name="l[3]" value="SlotPattern    ,hex,32,37,G,0,2,9">
        <PARAM name="l[4]" value="n64            ,bin,31,37,G,0,4,0">
        <PARAM name="l[5]" value="StartSlot      ,bin,31,37,G,0,9,5">
        <PARAM name="l[6]" value="ServicoIn      ,bin,31,37,G,0,14,10">
        <PARAM name="l[7]" value="InsertServ     ,bin,31,37,G,0,15,15">
        <PARAM name="l[8]" value="DataInsert     ,hex,28,37,G,0,0,15">
      </APPLET>
    </center>
  </body>
</html>
```

Figura 5.4: Parâmetros configuráveis via HTML, para geração da *applet* .

Neste processo evidenciam-se três atores: o desenvolvedor do software, o projetista do circuito, e o usuário do sistema.

O desenvolvedor do programa vai implementar uma camada de software para esconder detalhes arquiteturais do usuário do circuito e do projetista do hardware. O programador deve conhecer o funcionamento das classes JBits - como acessar e alterar informações do *bitstream*, bem como o posicionamento dos parâmetros a serem reconfigurados. Esta etapa realiza-se uma única vez, incorporando o conhecimento adquirido no desenvolvimento da ferramenta da seção 5.1.

O projetista de hardware realiza seu projeto em VHDL conhecendo apenas o posicionamento dos elementos de memória. Então cria uma página html para possibilitar a reconfiguração do circuito. Para criar esta página ele deve saber quais os parâmetros que o *applet* aceita. E é apenas isso - toda a parte de endereçamento e arquitetura é abstraída. A Figura 5.5 mostra a página gerada.

Já o usuário do circuito não precisa sequer saber em qual família de FPGAs seu sistema foi implementado. Ou seja, não há necessidade que ele tenha noção nem de VHDL, nem de JBits. O usuário

Figura 5.5: Interface do reconfigurador de circuitos.

só deve conhecer a funcionalidade do seu circuito e os parâmetros que deve alterar. Através de um *applet* disponibilizado através de uma página na internet ele altera esses parâmetros e reconfigura seu sistema de forma transparente.

5.3 Ferramenta para reconfiguração parcial

A reconfiguração parcial é o primeiro passo para a "virtualização" do hardware. Este termo significa que módulos de hardware podem ser alternadamente inseridos e removidos de um FPGA em operação, exatamente como ocorre com sistemas de memória virtual. Para isto é necessário que o dispositivo FPGA permita reconfiguração parcial e dinâmica.

No sentido de prover um mecanismo para manipulação de arquivos de configuração para FPGAs da família Virtex, desenvolvemos uma aplicação escrita em Java, mas sem a utilização das classes JBits. Baseada no estudo minucioso da arquitetura interna de FPGAs da família Virtex (item 4.2.3),

e de seu arquivo de configuração, essa ferramenta realiza as mesmas operações citadas na Seção 5.1, além de gerar um *bitstream* parcial.

A entrada básica do programa é um arquivo de configuração completo, e um arquivo-texto contendo os parâmetros do protocolo para geração de um *bitstream* parcial, caso seja necessário. Como nesta ferramenta se está trabalhando com *bitstreams* no formato *rawbits* (arquivo no formato ASCII, com extensão ".rbt") ao invés do formato binário (com extensão ".bit"), a abertura do arquivo de configuração completo pode ser um pouco lenta. Isto se dá porque as informações do arquivo de configuração são trazidas todas de uma vez para memória, e são tratadas como uma grande matriz bidimensional. Esta estrutura foi escolhida por representar diretamente a forma com que as informações são tratadas, conforme exposto na Seção 4.2. Ou seja, o programa manipula uma matriz de tantas colunas quantos forem os bits que compõem uma palavra do dispositivo, e tantas linhas quantas forem necessárias para a configuração do dispositivo, levando-se em consideração protocolos de inicialização, sincronização, escrita da lógica, escrita em memória, CRC e comandos de finalização da configuração.

A Figura 5.6 mostra a interface dessa ferramenta. No retângulo maior, o usuário visualiza o conteúdo de um arquivo de configuração. A área retangular na parte inferior direita da Figura 5.6 mostra as opções para localização de elementos dentro do *bitstream*. É possível realizar a busca por um bit específico informando-se LUT, Fatia, Linha e Coluna desejados. Então o bit correspondente é destacado na tela à esquerda, onde este bit pode ser alterado. Também é possível visualizar todos os bits de uma LUT, todos os bits de uma linha e todos os bits de uma coluna.

A ferramenta também permite que se alterne entre blocos de informações, conforme descrito na Figura 4.7, com a finalidade de acessar rapidamente e alterar o conteúdo de algum registrador de comando. Isto pode ser necessário para, por exemplo, escrever no registrador COR se o *bitstream* será carregado com relógio para JTAG ou não. Outro exemplo é a possibilidade de alterar um bit no registrador CTL que define se a configuração prévia do FPGA será mantida - o que é necessário para reconfiguração parcial.

Depois da alteração seja de algum bit de informação, ou de um bit de algum registrador de comando, quando for efetuada a ação de "Salvar" o arquivo, o CRC é recalculado e gravado no registrador apropriado.

5.3.1 Salvando um *bitstream* parcial

Caso seja acionada a opção "Salvar Parcial", um *bitstream* parcial é gerado, conforme os bits que foram alterados e os parâmetros que foram informados no início do processo.

Tais parâmetros foram mantidos flexíveis em função da possível necessidade de alterações que podem ocorrer, uma vez que não foi ainda possível realizar um *download* de *bitstream* parcial que reconfigurasse com sucesso o FPGA.

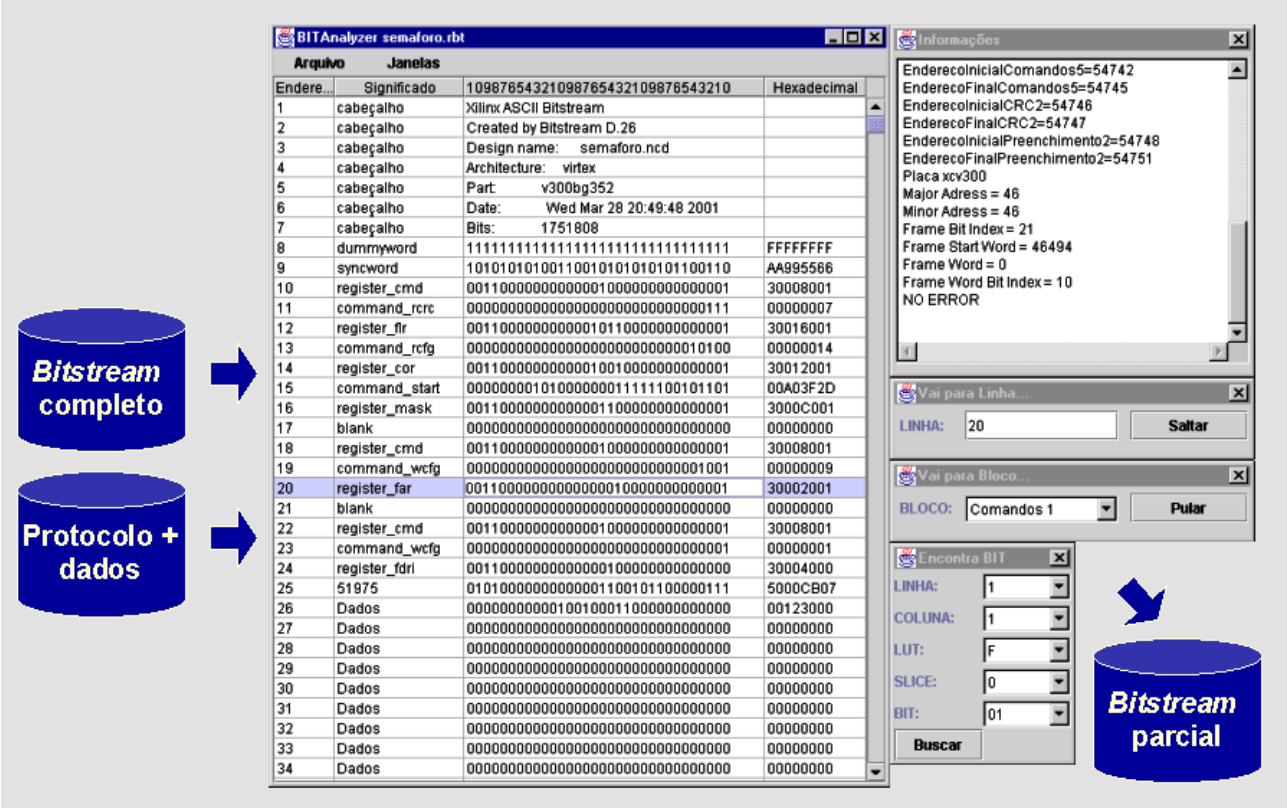


Figura 5.6: Interface do Reconfigurador Parcial.

Por isso foi criado um arquivo que define os parâmetros de inicialização de um *bitstream* parcial, chamado de “prototipoparcial.txt”. Neste arquivo são colocados quatro tipos de informações: blocos (conforme a Figura 4.7), registradores, comandos e parâmetros de registradores. Tais informações podem ser vistas na Tabela 5.2.

Algumas informações do cabeçalho do arquivo de configuração parcial são cópias do *bitstream* total. São as linhas 1, 2 e 4 da Figura 5.7. O campo *design name* (linha 3) indica o nome do arquivo do projeto que gerou este *bitstream*. No caso de o arquivo de configuração ser salvo com outro nome, esta linha é modificada para o nome do arquivo salvo. A família de dispositivos para o qual este *bitstream* foi gerado é indicada na linha 4 (neste exemplo, a família de FPGAs é a Virtex). O campo *part* (linha 5) indica o dispositivo para o qual este *bistream* foi gerado (neste exemplo é o FPGA Virtex XCV300). O campo *date* (linha 6) é captado no momento em que a janela do *bitstream* parcial é aberta, e ao salvar o arquivo ela é atualizada. O campo *bits* (linha 7) refere-se ao número de bits que este arquivo de configuração parcial contém. As linhas seguintes (8 e 9) trazem as palavras de sincronização com o dispositivo (*dummyword* e *syncword*).

A partir da linha 10 do arquivo de configuração é que o protocolo de reconfiguração parcial é realizado.

A lógica deste protocolo reside em manter a configuração pré-existente no dispositivo, permitindo

Tabela 5.2: Parâmetros do arquivo de protótipo para geração do *bitstream* parcial.

Blocos	Registradores	Comandos	Parâmetros
Cabeçalho	FLR	Dummyword	shutdown on (COR)
Comandos 1	COR	Syncword	Shutdown off (COR)
Dados	LOUT	Blank	Done_pipe yes (COR)
Comandos 2	STAT	Switch	Done_pipe no (COR)
RAM 0	MASK	AGHigh	Drive_done open (COR)
Comandos 3	CTL	RCRC	Drive_done high (COR)
RAM 1	CMD	RCap	Single on (COR)
CRC 1	FDRO	Start	Single off (COR)
Comandos 4	FDRI	RCfg	Oscfsel (COR)
Preenchimento 1	FAR	Lfrm	Ssclksrc cclk (COR)
Comando 5	CRC	WCfg	Ssclksrc userclk (COR)
CRC 2			Ssclksrc jtagclk (COR)
Preenchimento 2			Lock_wait (COR)
			Done_cycle (COR)
			Lck_cycle (COR)
			Gts_cycle (COR)
			Gwe_cycle (COR)
			Gsr_cycle (COR)
			Dado (CRC)
			Persist on (CTL)
			Persist off (CTL)
			Gts_usr_b on (CTL)
			Gts_usr_b off (CTL)
			Sbits enable (CTL)
			Sbits disable (CTL)
			Sbits crc (CTL)

Endere...	Significado	10987654321098765432109876543210	Hexadecimal
1	cabeçalho	Xilinx ASCII Bitstream	
2	cabeçalho	Created by Bitstream D.26	
3	cabeçalho	Design name: semaforo.ncd	
4	cabeçalho	Architecture: virtex	
5	cabeçalho	Part: v300bg352	
6	cabeçalho	Date: Sun Sep 02 20:50:17 2001	
7	cabeçalho	Bits: 2859	
8	dummyword	11111111111111111111111111111111	FFFFFFFF
9	syncword	10101010100110010101010101100110	AA995566

Figura 5.7: Detalhamento do cabeçalho do arquivo de configuração parcial.

que apenas os quadros onde ocorreram modificações sejam carregados no dispositivo. Isto se dá através da seqüência de *shutdown* (linhas 10 a 23) [XIL00a], que podem ser vistas na Figura 5.8. A

linha 11 determina um *reset* no registrador de CRC. Em seguida é ativado o registrador COR, e nele são escritas informações que modificam exatamente o bit de *shutdown*. Uma escrita no registrador CMD inicializa o protocolo (*start*, linhas 14 e 15). Um primeiro cálculo de CRC é realizado para garantir que as informações até este ponto estão corretas (linhas 16 e 17), e novamente o registrador de CRC é resetado (linhas 18 e 19). O sinal GHIGH_B é setado, para evitar contenção no momento da escrita dos novos dados. Ou seja, para garantir que não haverá conflito de informações previamente configuradas, todas as saídas de CLBs e sinais que serão afetados pela reconfiguração são colocados em '1' (um).

O registrador seguinte (COR) informa que a seqüência de *shutdown* terminou. Então o registrador FAR informa a partir de que coordenada (linha, coluna, LUT e fatia) será realizada a reconfiguração.

A seqüência então é semelhante ao protocolo de configuração de um *bitstream* completo. Há a escrita de uma palavra de configuração no registrador de comando (*wcfg* - linhas 26 e 27) e então o registrador FDRI informa quantas palavras serão escritas (linha 28). Como neste exemplo foi alterado o bit 14, da linha 1, coluna 2, LUT F da fatia 0, apenas um quadro será escrito. Ainda conforme o exemplo, por tratar-se de um FPGA Virtex XCV300, um quadro é composto por 21 palavras - portanto, o registrador FDRI informa a escrita de 21 palavras.

Endere...	Significado	10987654321098765432109876543210	Hexadecimal
10	cmd	00110000000000001000000000000001	30008001
11	rcrc	00000000000000000000000000000111	00000007
12	cor	00110000000000010010000000000001	30012001
13	shutdown on	01010000000000001100101100000111	5000CB07
14	cmd	00110000000000001000000000000001	30008001
15	start	00000000000000000000000000000101	00000005
16	crc	00110000000000000000000000000001	30000001
17	dado	00000000000000001111111001111010	00007E7A
18	cmd	00110000000000001000000000000001	30008001
19	rcrc	00000000000000000000000000000111	00000007
20	cmd	00110000000000001000000000000001	30008001
21	aghigh	00000000000000000000000000001000	00000008
22	cor	00110000000000010010000000000001	30012001
23	shutdown off	01010000000000001001011100000111	50004B07
24	far	00110000000000000010000000000001	30002001
25	01 02 f 0 14	00000000010111000101110000000000	005C5C00
26	cmd	00110000000000001000000000000001	30008001
27	wcfg	00000000000000000000000000000001	00000001
28	fdri	00110000000000000100000000010101	30004015

Figura 5.8: Visualização do protocolo de configuração de um *bitstream* parcial.

O registrador FAR (Figura 5.8, linha 24) é o responsável por informar as coordenadas do primeiro quadro da seqüência a ser reconfigurada. Na linha 25, coluna "significado" da Figura 5.8, os valores

$01\,02\,f\,0\,14$ indicam que a seqüência de quadros a serem reconfigurados inicia-se na linha 1, coluna 2, e trata-se de uma F LUT na fatia 0, e passa pelo bit 14. Já os 5 bits menos significativos da linha do registrador FDRI (Figura 5.8, linha 28) - 10101- informam que serão escritas 21 palavras (um quadro do FPGA XCV300).

Logo depois da escrita das 21 palavras, há uma verificação de CRC e a escrita de um quadro de preenchimento. A isto segue-se uma escrita no registrador CTL que informa que a interface da configuração deve permanecer após a reconfiguração. Finalmente outra verificação de CRC ocorre e a escrita de 4 palavras de preenchimento termina o arquivo de configuração.

5.3.2 Validação do *bitstream* parcial

O processo de validação da geração do *bitstream* parcial é, a princípio, bastante simples.

Foi criada uma aplicação (descrita em VHDL) que tem seu comportamento alterado conforme alguns dados armazenados em LUTRAM. Consiste em um contador que mostra o resultado da contagem em *leds* de uma placa de prototipação. Algumas LUTs são configuradas para funcionarem como bits de memória RAM dupla-porta [MES00]. Três LUTs são usadas, conforme a lista abaixo:

- ➢ O valor armazenado na F-LUT (fatia 0), da CLB localizada na linha 20, coluna 5 funciona como RESET do circuito;
- ➢ O valor armazenado na F-LUT (fatia 0), da CLB localizada na linha 19, coluna 5 funciona como *Clock Enable* para o contador;
- ➢ O valor armazenado na F-LUT (fatia 0), da CLB localizada na linha 19, coluna 5 define se contador será incremental ou decremental.

Os valores são mostrados nos 3 *leds* menos significativos, enquanto o resultado da contagem é mostrado nos 4 *leds* mais significativos. O *led* 3 deve permanecer apagado. A Tabela 5.3 mostra os efeitos de modificações nos bits referidos acima. O estado inicial (sem modificação dos bits em questão) é o de contagem congelada (todos bits relevantes em 1).

A idéia é realizar as modificações de comportamento do circuito (descritas na coluna “Efeito” da Tabela 5.3) reconfigurando apenas os bits estritamente necessários para tal - ou seja, reconfigurando parcialmente o FPGA.

Com o domínio do endereçamento de elementos internos ao FPGA (LUTs), localizar e modificar tais bits não foi problemático. Em uma experiência inicial foram localizados os bits que interessavam e foram modificados de forma que o contador fosse incremental. Foi gerado um arquivo de configuração total e realizado o *download* na placa. A configuração foi bem sucedida e o a aplicação comportou-se conforme previsto.

Tabela 5.3: Bits configuráveis do contador e seus respectivos efeitos.

CLB:	**18,5**	**19,5**	**20,5**	
Função:	**I/D**	**CKE**	**RST**	**Efeito**
	X	X	0	Reset
	X	1	1	Congela
	1	0	1	Incrementa
	0	0	1	Decrementa

A tentativa seguinte foi gerar um arquivo de configuração parcial. As primeiras experiências foram realizadas com um protocolo de configuração parcial que estava em conformidade com o que está descrito na documentação do fabricante do dispositivo. Contudo, quando das tentativas de realizar *download* para a placa de prototipação, ocorriam erros de INIT e DONE, sinais retornados da placa de prototipação para a aplicação que realiza o *download* (*HardwareDebugger*). Disto denota-se que o protocolo de configuração parcial estava errado.

Estudou-se, então, mais detalhadamente o protocolo de configuração parcial e realizaram-se alguns testes, até que se chegou no protocolo descrito no item 5.3.1. Com este protocolo foi possível gerar um arquivo de configuração parcial de tal forma que o *download* fosse bem sucedido, ou seja, não houveram mais erros de INIT e DONE. Como efeito co-lateral, criou-se a ferramenta para reconfiguração parcial de forma flexível o bastante para que o protocolo de reconfiguração parcial seja modificado facilmente. Notou-se, nessas experiências, que o tempo de *download* do *bitstream* parcial foi bastante baixo, em relação ao *bitstream* completo. Enquanto para este último o tempo ficou em torno de 35 segundos, para o primeiro foi de 0,17 segundo, para um quadro modificado. Isto se dá porque o tempo de configuração é diretamente proporcional à quantidade de bits de configuração que é enviada à placa.

Contudo, apesar do *download* ter ocorrido, o dispositivo não foi reconfigurado. Tal problema deve-se à ferramenta de *download*, que envia sinais à placa de prototipação que coloca o FPGA em um estado de *reset* (colocando todos seus bits em 1) - ou seja, eliminando a configuração anterior antes de configurar com o novo *bitstream*. O fabricante do software em questão informou que nova versão está sendo desenvolvida, e que tal problema será eliminado nesta versão. Infelizmente a nova ferramenta não ficou pronta antes da escrita deste volume.

5.4 Unificador de *cores*

Partindo do domínio do endereçamento de elementos em FPGAs da família Virtex, em conjunto com o conhecimento adquirido no desenvolvimento da ferramenta descrita na Seção 5.3, definiu-se a criação de um software para manipulação de *cores*.

Essa nova ferramenta deveria possuir as seguintes características:

1. Permitir a leitura de informações de configurações diretamente do *bitstream* (*hard cores*);
2. Possibilitar a união entre áreas de dois diferentes *bitstreams* (desde que não sejam sobrepostas), para geração de um terceiro arquivo de configuração;
3. Gerar *bitstreams* parciais.
4. A ferramenta deve ser visual, isto é, possuir uma interface gráfica para seleção de áreas específicas do FPGA;

Assim como a ferramenta anterior, o unificador de *cores* foi programado em Java, sem o auxílio do JBits. Sua interface gráfica é semelhante à da ferramenta *BoardScope* (da Xilinx) a fim de ser facilmente compreendida pelo usuário. Essa interface pode ser vista na Figura 5.9, que também mostra uma funcionalidade interessante da ferramenta. O arquivo de configuração utilizado no exemplo é exibido de forma visual. Nesse caso, o *bitstream* foi gerado para um FPGA XCV300. Por isto, a Figura 5.9 mostra 32 quadrados em cada coluna e 48 quadrados em cada linha. Ou seja, cada quadrado representa um CLB para este dspositivo. Quadrados cinza-claros e cinza-chumbo representam que as respectivas LUTs permanecem com seus valores originais inalterados. Quadrados em contornados com branco mostram onde está implementada a lógica da aplicação.

Outra funcionalidade da ferramenta é a exibição dos valores contidos em cada CLB. Para tanto basta arrastar o *mouse* sobre a CLB que se deseja saber o valor. Então esse valor é mostrado em uma janela auxiliar da aplicação, conforme a Figura 5.10.

Atendendo ao objetivo 2 desta ferramenta, é possível abrir um segundo *bitstream*. Neste caso, o primeiro torna-se o arquivo de configuração que servirá como base para o *bitstream* gerado a partir da junção dos dois. Com esta ferramenta gráfica, a junção dos dois *cores* torna-se fácil. Basta selecionar a área desejada do segundo *bitstream* e em seguida voltar para o primeiro (clicando na aba com nome correspondente ao *bitstream* original. Neste instante, o arquivo correspondente à junção dos dois está pronto para ser salvo. Caso haja algum problema, como por exemplo, a sobreposição de áreas entre os dois *cores*, a ferramenta possui recursos para desfazer. As Figuras 5.9, 5.11 e 5.12 ilustram este exemplo. A Figura 5.9 mostra o primeiro *core* envolvido neste processo, enquanto a Figura 5.11 mostra o segundo *core*. Selecionando este último e retornando ao primeiro, tem-se o mesmo que é exibido na Figura 5.12, que é a representação visual dos dois *cores* em questão reunidos em um único arquivo de configuração.

A ferramenta ainda apresenta a opção de *download*, que chama o software *HardwareDebugger* para realizar a configuração do dispositivo.

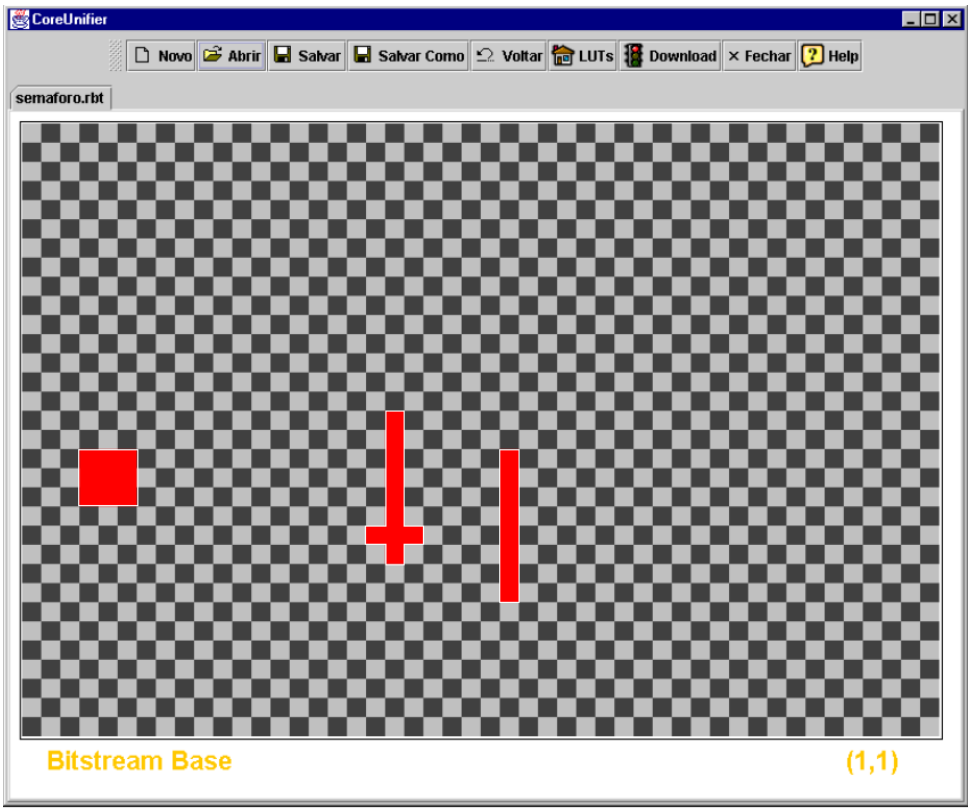


Figura 5.9: Visualização do *core* correspondente ao exemplo do semáforo.

Esta ferramenta é fundamental para validação da idéia de uma interface para conexão de *cores*. Ou seja, será utilizada para realizar a inserção e remoção de *cores* que conectar-se-ão através de uma interface. Esta interface proverá a comunicação entre os *cores* e deles com o mundo externo. A proposta desta interface é feita na Seção 5.5. Já a validação desta ferramenta é mostrada no item a seguir.

5.4.1 Validação do unificador de *cores*

A primeira forma de teste desta aplicação foi realizar a modificação de informações armazenadas em LUTRAM, conforme mostrado nas Figuras 5.9, 5.11 e 5.12. Neste caso a aplicação é a mesma descrita no item 5.3.2. O circuito originalmente estava em um estado de espera (Figuras 5.9), até que fosse excitado por valores armazenados em LUTs. O *core* mostrado na Figura 5.11 continha apenas os parâmetros que definiriam o comportamento do primeiro circuito (neste exemplo, valores que indicavam que o circuito deveria comportar-se como um contador incremental). Foi realizada a união entre os dois primeiros *bitstreams*, e o arquivo de configuração decorrente (Figuras 5.12) foi carregado no FPGA. O *download* e a configuração do dispositivo ocorreram com sucesso.

Uma segunda experiência foi proposta, mas ainda não está operacional. Trata-se de reconfigurar

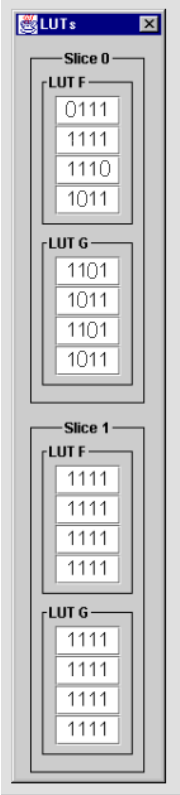


Figura 5.10: Visualização do conteúdo de um CLB.

cores de modo a garantir o roteamento entre eles. Esta é uma necessidade quando se pretende que os *cores* envolvidos comuniquem-se entre si. Se os cores em questão forem disjuntos, ou seja, se possuírem apenas comunicação com o mundo externo e não um com o outro, não há problema de roteamento, nem sequer há necessidade de um barramento para prover a conexão entre eles. Mas também não faz sentido, no contexto deste trabalho, uma aplicação implementada desta forma.

Contudo, pretende-se exatamente que seja possível reconfigurar o FPGA parcialmente para inserir ou remover *cores* que tenham interdependências ou que de alguma forma necessitem comunicar-se. Mesmo que o barramento em questão ainda seja apenas uma proposta, é possível vislumbrar uma forma de conectar *cores* a ele, conforme é explanado no item a seguir.

5.4.2 Interconexão entre um *core* de aplicação e o barramento

Enquanto o protótipo de barramento de interconexão de *cores* para FPGAs está sendo desenvolvido por outro membro do GAPH [PAL01], este trabalho propõe um método para interconexão entre um *core* de aplicação e esse barramento. Da Figura 5.13 pode ser abstraída esta idéia. Na Figura 5.13-b verifica-se que uma área correspondente a um *core* é encontrada e selecionada de um arquivo de configuração gerado pelas ferramentas de CAD da Xilinx. Os bits relativos à esta área são

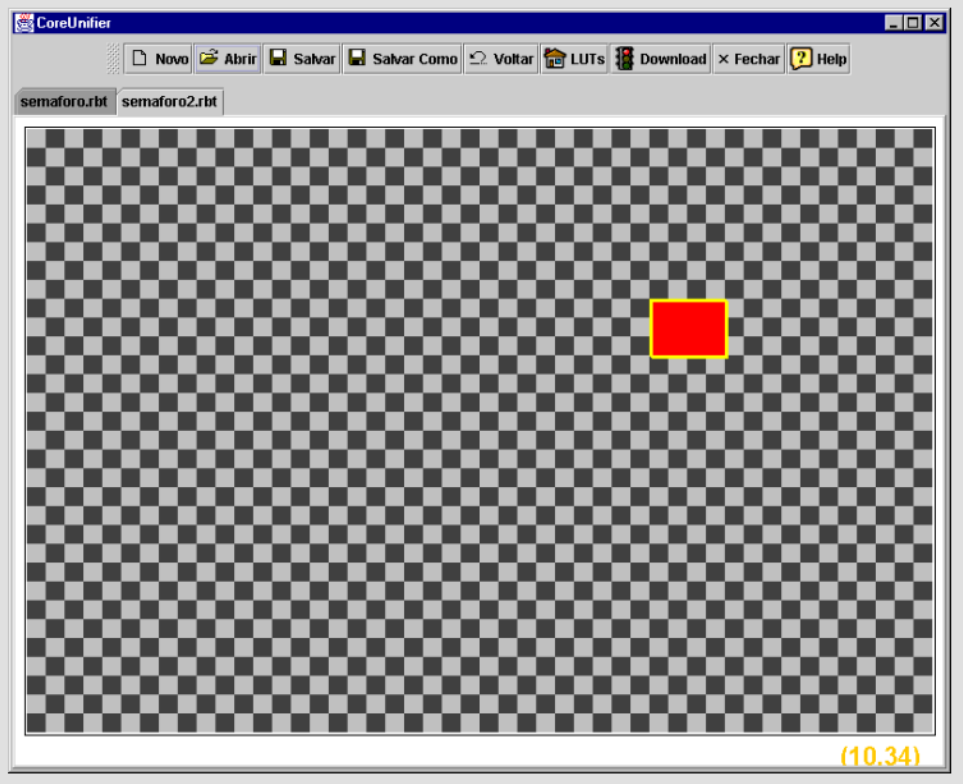


Figura 5.11: Seleção de um *core* no segundo *bitstream*.

armazenados em memória para utilização posterior. Repete-se a operação com um segundo módulo (Figura 5.13-a), correspondente à uma interface para conexão de *cores* (item 5.5.1). É realizada uma concatenação dos dois, gerando um terceiro *bitstram* completo (Figura 5.13-c).

O *download* do *bitstream* gerado foi parcialmente bem sucedido. O *bitstream* configurou a placa, o *core* do barramento apresentou o comportamento esperado, mas o *core* da aplicação não comunicou-se com o barramento. Verificou-se então que a ferramenta para unificação de *cores* não garante que o roteamento seja mantido. Novas experiências estão sendo feitas, com a utilização de *dummy cores* [PAL01a], mas até o fechamento deste volume não havia sido obtido sucesso.

5.5 Proposta de interface para conexão de *cores*

A computação reconfigurável será uma realidade somente quando houver uma certa independência da aplicação em relação ao dispositivo para a qual foi projetada. Tal independência pode ser entendida como a possibilidade de a aplicação ser executada em dispositivos configuráveis compatíveis, mesmo que estes possuam mais (ou menos) recursos computacionais do que o dispositivo para o qual a aplicação foi inicialmente escrita. Ou seja, é preciso que ocorra a virtualização do hardware

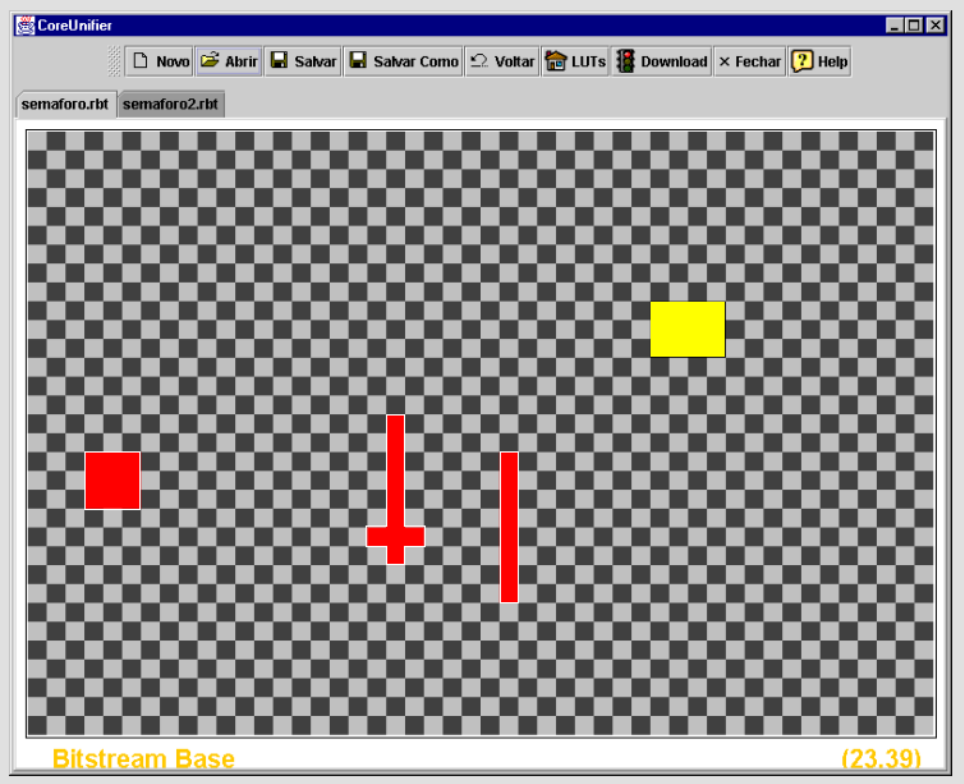


Figura 5.12: *Bitstream* que unifica dois *cores*.

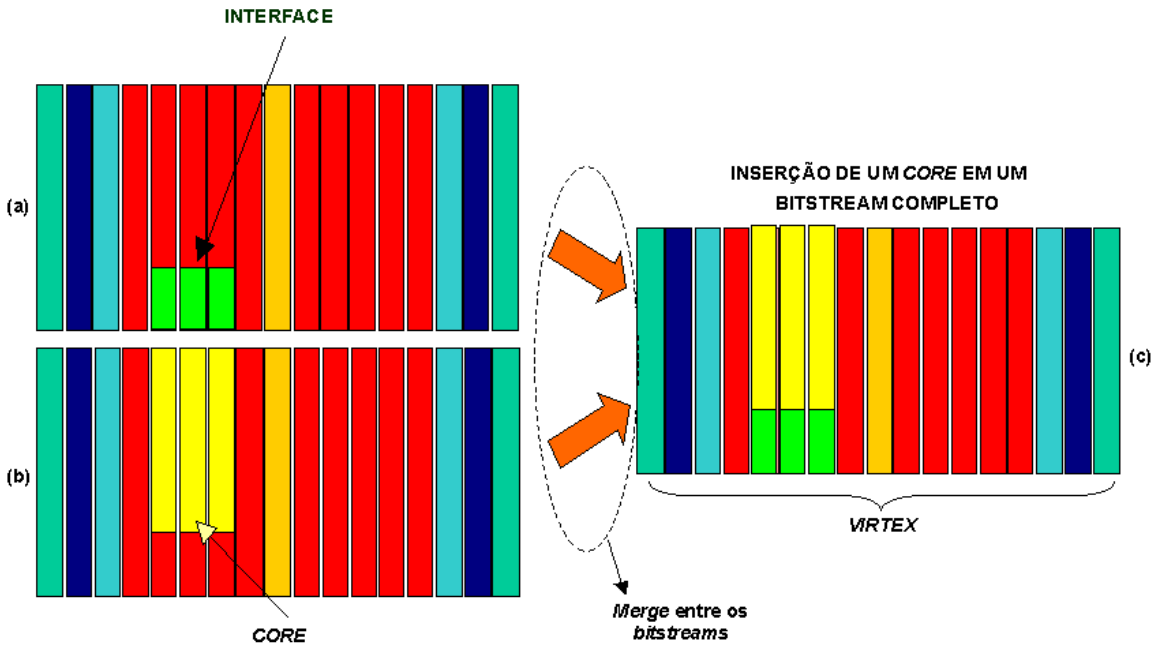


Figura 5.13: Método para validação do barramento para interconexão de *cores*.

para que seja obtida a sobrevivência da aplicação. Passos importantes estão sendo dados com esse objetivo, mas há uma lacuna ainda não preenchida nesse caminho.

A computação reconfigurável pressupõe virtualização do hardware. E esta prescinde de algumas características, tais como:

- ➢ Lógica programável baseada em SRAM;
- ➢ Disposição regular dos recursos de lógica programável;
- ➢ Possibilidade de reconfiguração dinâmica;
- ➢ Microprocessador, memória e lógica programável no mesmo circuito integrado;
- ➢ Um barramento que permita a inserção/remoção de *cores* sem que seja interrompida a comunicação de *cores* pré-existentes entre si e com o mundo externo.

O último item é a lacuna a ser preenchida aqui.

A idéia é prover uma maneira de diminuir os problemas de integração em SoCs, aumentando a portabilidade e disponibilidade de *cores*, o que resulta em uma aceleração no tempo para um determinado produto chegar ao mercado.

O que se encontra até agora é uma dificuldade na integração de *cores* por não haver uma padronização nos esquemas de interconexão. Isto requer a criação de lógica de cola para conectar cada parte dos *cores* envolvidos entre si e com o mundo externo. Fica evidente que a existência de um barramento de interconexão de *cores* possibilitaria sua integração de forma mais simples e rápida. Analogamente, essa interface pode ser comparada a um barramento de um microcomputador, tal como o barramento PCI.

Neste sentido, este trabalho propõe um método para prover esse barramento em um FPGA comercial, cuja arquitetura não previu tal necessidade. Na Seção 2.6 foram analisadas propostas de barramentos semelhantes, mas que foram idealizadas para ASICs. No item 5.5.1 é apresentada nossa proposta.

5.5.1 Proposta de barramento para interconexão de *cores*

O processo de inserção ou remoção de *cores* do FPGA é especialmente prejudicado em função de problemas relacionados com roteamento e conflitos de entrada e saída. Fica claro que os dispositivos da família Virtex não foram projetados com recursos previamente estabelecidos que facilitassem esse processo. Em vista dessa necessidade, caso pretenda-se utilizar o FPGA com essa finalidade, é imprescindível estabelecer uma forma de comunicação entre *cores* e dos *cores* com o mundo externo.

Isto pode ser possível com o projeto de um *core* que atue como um árbitro entre os *cores* de aplicações. Tal módulo, chamado a partir de agora de “barramento” é configurado no dispositivo e

nele permanece de forma estática (Figura 5.14-a). Esse barramento é composto por duas partes. A primeira é o controlador tem por função prover conexão com os pinos de entrada e saída do dispositivo (Figura 5.14-b). Os demais *cores* que forem configurados no dispositivo (Figura 5.14-d) comunicar-se-ão somente com o controlador. Desta forma, tem-se uma entrada e saída virtualizada, tornando possível a inserção e remoção de *cores* como se inserem ou removem páginas em um sistema de memória virtual.

A segunda parte do barramento é um circuito que atua como árbitro (Figura 5.14-c). Tem por objetivos gerenciar o aceite ou rejeição de conexões de novos *cores*, bem como controlar possíveis conflitos relativos aos acessos aos pinos e entrada/saída do dispositivo. Quando um determinado módulo de aplicação deseja comunicar-se com outro, ou com o mundo externo, envia um pedido de requisição ao árbitro. Se a linha de comunicação (ou de acesso ao pino de entrada/saída) requisitada está disponível, o árbitro permite que o *core* utilize o recurso solicitado. Caso contrário, o *core* permanece em um estado de espera até que o árbitro permita que acesse o recurso.

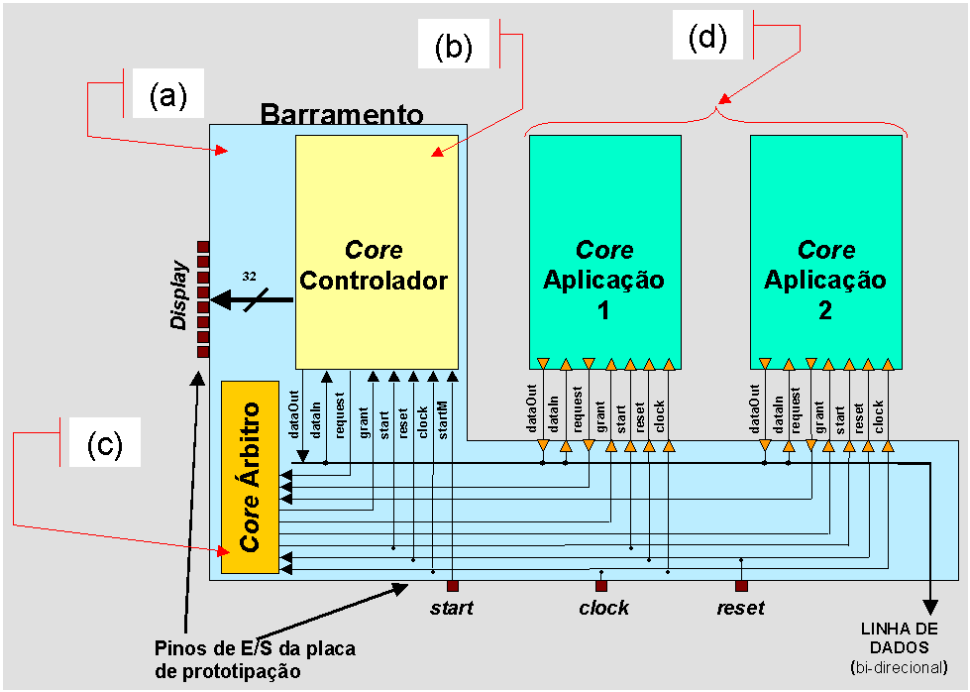


Figura 5.14: Proposta de barramento para interconexão de *cores*.

Ainda na Figura 5.14 percebe-se que tanto o árbitro quanto o controlador são posicionados de forma vertical. Isto deve-se às seguintes considerações:

- FPGAs da família Virtex possuem estrutura interna em colunas de CLBs;
- CLBs são cortadas verticalmente por quadros;
- Um quadro é a unidade atômica de reconfiguração;

➢ Os módulos em questão devem permanecer no dispositivo de forma estática.

Em resumo, se o controlador e/ou o barramento fossem em parte posicionados nas mesmas colunas que os *cores* de aplicações, quando estes fossem reconfigurados, fatalmente todo o FPGA teria que parar sua execução, pois o funcionamento barramento seria interrompido.

Embora não seja escopo deste trabalho o projeto desse circuito, toda pesquisa aqui realizada serviu como base para que experimentos com o barramento fosse viável.

Capítulo 6

Conclusões

Este trabalho propôs-se a investigar reconfiguração parcial, remota e dinâmica de FPGAs comercialmente disponíveis. A contribuição para o estado-da-arte em computação reconfigurável ocorre em três diferentes aspectos:

1. Apresenta uma revisão do estado-da-arte em sistemas digitais reconfiguráveis;
2. Realiza um estudo aprofundado da organização interna dos FPGAs da família Virtex;
3. Mostra um conjunto de ferramentas desenvolvidas durante esta pesquisa que se destinam a auxiliar na reconfiguração parcial e remota de FPGAs.

A primeira contribuição classifica SDRs segundo uma série de critérios, e identifica uma tendências em computação configurável. A segunda contribuição é o estudo aprofundado dos FGPAs Virtex, que possibilita o endereçamento componentes no FPGA, bem como o entendimento de seu protocolo de configuração. Como conseqüência, é possível a terceira contribuição deste trabalho. As ferramentas produzidas no âmbito desta pesquisa permitem tanto manipular individualmente bits armazenados em LUTRAMs, quanto tornam possível a reconfiguração de *cores*.

Em vista do que foi pesquisado e experimentado, a reconfiguração remota não só é factivel de ser implementanta, como tem aplicação industrial imediata. E apesar disto não está sendo ainda comercialmente explorada.

Quando da pesquisa sobre o endereçamento de elementos, trabalhou-se com a modificação do comportamento de um circuito em função de bits armazenados em LUTRAM. Esse tipo de reconfiguração, ainda que completa, local ou remota, ajuda a reduzir o custo do sistema, pois reduz em muito o hardware de controle do circuito.

A reconfiguração de parâmetros, de forma parcial, apresenta sobre a experiência acima a vantagem de tempo, pois é muito mais rápida. Esta reconfiguração parcial não foi totalmente validada, por

causa de problemas já descritos. A despeito deste contratempo, gerou-se o *bitstream* parcial de forma correta.

Outra conclusão a que se chega é que os FPGAs Virtex podem ser parcialmente reconfigurados, porém não existe suporte algum para tal característica. Ainda que disponibilizem uma biblioteca de classes em Java para manipular o *bitstream* de seus dispositivos, essa biblioteca não está suficientemente madura e é mal documentada. Percebeu-se claramente que o fabricante não explorou esta característica devido às dificuldades de desenvolver o CAD para tal suporte à reconfiguração parcial.

Mesmo que academicamente a idéia de reconfiguração dinâmica seja amplamente investigada, nota-se que a indústria investe pouco nesta tecnologia. Isto provavelmente se dê por não terem surgido ainda aplicações comerciais em grande escala que necessitem dessa característica. Outro motivo possível é a dificuldade de desenvolver CAD para reconfiguração dinâmica e para roteamento. Saliente-se que FPGAs mais antigos (Xilinx 6200 e National Clay, por exemplo) contemplavam essa característica, mas foram descontinuados por não terem sido desenvolvidas ferramentas eficientes para auxiliar na sua utilização.

Ainda que a reconfiguração parcial de parâmetros de hardware seja interessante, a reconfiguração de *cores* só será realmente útil se for possível conectar *cores* diretamamente usando barramentos internos aos FPGAs.

Ao conhecimento do autor deste texto, tais barramentos ainda não existem , e seu desenvolvimento é bastante problemático conforme ficou comprovado por trabalho de Dissertação complementar a este [PAL01a].

Em suma:

1. Reconfiguração completa remota funciona e CAD é simples de desenvolver;
2. Reconfiguração de parâmetros completa/parcial, local/remota, é também factível de se implementar;
3. Reconfiguração parcial de parte do circuito é difícil com os atuais FPGAs, pois necessita de CAD para projeto e barramento interno para interconexão de *cores*.

Ainda, com base nas tendências observadas, FPGAs de grão mais grosso (ULAs, processadores) com reconfiguração parcial em nível de instruções (parâmetros) são uma alternativa interessante para aplicações como multimídia e telecomunicações. A idéia é que haja uma rede de elementos de processamento com conexão/funcionalidade definidas por instruções configuradas em RAM.

O conjunto de ferramentas desenvolvidas pode ser integrado de duas formas distintas nos atuais sistemas de CAD. A primeira utilização é na forma de uma ferramenta que é fornecida ao usuário, juntamente com o arquivo de configuração. Esta forma de utilização permite que o usuário personalize seu circuito, sem a necessidade de ferramentas de síntese. A segunda forma de aplicação é sua

integração a ferramentas de projeto conjunto de hardware e software. A idéia é permitir que os módulos de hardware obtidos pelas metodologias de codesign sejam alocados sob demanda no dispositivo programável. Hoje, a capacidade do dispositivo programável deve ser suficiente para conter todos os módulos de hardware. Da forma proposta, teremos uma abordagem de "hardware dinâmico", da mesma forma que temos "memória virtual" para software.

6.1 Trabalhos futuros

Segundo um dos autores do JBits é possível realizar o *download* parcial em uma placa de prototipação não disponível em nossos laboratórios. Portanto, um dos trabalhos futuros seria testar a reconfiguração parcial em uma placa do fabricante Celoxica. Em conseguindo-se a reconfiguração parcial, o trabalho seguinte seria a realização de experimentos com reconfiguração dinâmica.

Outra investigação interessante seria avaliar FPGAs de outros fabricantes que dizem suportar reconfiguração dinâmica. O FIPSoC (da Sidsa) é uma alternativa, mas o FPSLIC (da ATMEL) parece ser uma opção mais madura, já acompanhada de algumas ferramentas de CAD.

Finalmente, a proposta de barramento para interconexão de *cores* deve ser revista e ampliada. Um estudo mais aprofundado de iniciativas de barramentos como o *Wishbone* e o *CoreConnect*, juntamente com análise de características de dispositivos SoCs, poderiam gerar a proposta de um novo tipo de dispositivo reconfigurável que incorporasse esse barramento.

Referências Bibliográficas

[ALG89] ALGOTRONIX. **Cal1024 datasheet**. Disponível por WWW em http://dec.bournemouth.ac.uk/drhw_lib/archive/cal1024.ps.gz (1989).

[ALT01] ALTERA. **Configuration (in-circuit reconfiguration)**. Disponível por WWW em http://www.altera.com/support/devices/programming/sup-configuration.html (2001).

[ARN92] ARNOLD, Jeffrey M. The splash 2. In: SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES, 1992. **Anais...** 1992. p.316–324.

[ATH93] ATHANAS, Peter M.; SILVERMAN, Harvey F. Processor reconfiguration through instruction-set metamorphosis. **IEEE Computer**, p.11–18, 1993.

[ATM00] ATMEL. **At40k series configuration**. Disponível por WWW em http://www.atmel.com/atmel/postscript/doc1009.ps.zip (2000).

[ATM00a] ATMEL. **Implementing cache logic with fpgas**. Disponível por WWW em http://www.atmel.com/atmel/postscript/doc0461.ps.zip (2000).

[BEL98] BELLOWS, Peter; HUTCHINGS, Brad. Jhdl-an hdl for reconfigurable systems. In: PROCEEDINGS OF THE IEEE SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES, 1998. **Anais...** 1998. p.124–133.

[BER96] BERTIN, Patrice; VUILLEMIN, Jean; RONCIN, Didier; SHAND, Mark; SHAND, Hervé; BOUCARD, Philippe. Programmable active memories: reconfigurable systems come of age. **IEEE Transactions on VLSI Systems**, v.4, n.1, p.56–69, 1996.

[CAL01] CALAZANS, Ney Laert Vilar; MORAES, Fernando Gehm; MARCON, César Augusto Missio; VALIATI, Vitor Hugo; MANFROI Édison. Effective industry-academia cooperation in telecom: a method, a case study and some initial results. In: XIX Simpósio Brasileiro de Telecomunicações, 2001, Fortaleza, CE-Brasil. **Anais...** 2001.

[CAL98] CALAZANS, Ney Laert Vilar. **Projeto lógico automatizado de sistemas digitais seqüenciais**. [S.l.]: XI Escola de Computação, 1998. 318p.

[CAL00] CALLAHAN, Timothy J.; HAUSER, John R.; WAWRZINEK, John. The Garp architecture and C compiler. **Computer Magazine**, n.4, p.62–69, 2000.

[COM99] COMPTON, Katherine. **Programming architectures for run-time reconfigurable systems**. Washington, Estados Unidos: University of Washington - Electrical Engineering, 1999. Dissertação de Mestrado.

[DEH00] DEHON, André; CASPI, Eylon; CHU, Michael; HUANG, Randy; YEH, Joseph; MARKOVSKY, Yury; WAWRZYNEK, John. Stream computations organized for reconfigurable execution (SCORE): introduction and tutorial. In: FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS, 2000. **Anais...** 2000.

[DEH94] DEHON, André. **Dpga-coupled microprocessors**: commodity ics for the early 21st century. Disponível por WWW em http://www.ai.mit.edu/projects/transit/tn100/tn100.html (Janeiro 1994).

[DUB98] DUBOIS, Michael; JEONG, Jaeheon; SONG, Yong Ho; MOGA, Adrian. Rapid hardware prototyping on RPM 2. **IEEE Design and Test of Computers**, n.3, p.112–118, 1998.

[ELD94] ELDREDGE, James G.; HUTCHINGS, Brad L. Density enhancement of a neural network using fpgat's and run-time reconfiguration. In: IEEE WORKSHOP IN FPGAS CUSTOM COMPUTING MACHINES, 1994, California, EUA. **Anais...** 1994. p.180–188.

[GOK90] GOKHALE, Maya. Splash, a reconfigurable linear logic array. **International Conference on Parallel Processing**, v.9, p.219–314, 1990.

[GOL00] GOLDSTEIN, Seth Copen; SCHMIT, Herman; CADAMBI, Srihari; MOE, Matt; TAYLOR, R Reed. Piperench: a reconfigurable architecture and compiler. **Computer Magazine**, n.4, p.70–77, 2000.

[GUC00] GUCCIONE, Steve. **List of fpga-based computing machine**. Disponível por WWW em http://www.io.com/ guccione/HW_list.html (Agosto 2000).

[GUC99] GUCCIONE, Steven; LEVI, Delon; SUNDARARAJAN, Prasanna. Jbits: a java-based interface for reconfigurable computing. In: 2ND ANNUAL MILITARY AND AEROSPACE APPLICATIONS OF PROGRAMMABLE DEVICES AND TECHNOLOGIES CONFERENCE (MAPLD), 1999, Laurel, Maryland, USA. **Anais...** 1999.

[HAD95] HADLEY, John D.; HUTCHINGS, Brad L. Design methodologies for partially reconfigured systems. In: IEEE WORKSHOP ON FPGAS FOR CUSTOM COMPUTING MACHINES, 1995, California, Estados Unidos. **Anais...** 1995. p.78–84.

[IBM99] IBM. **The CoreConnect bus architecture**. Disponível por WWW em http://www.chips.ibm.com/products/coreconnect/docs/crcon_wp.pdf (Agosto 1999).

[LEW98] LEWIS, David M.; GALLOWAY, David R.; IERSSEL, Marcus van; ROSE, Johnathan; CHOW, Paul. The Transmogrifier-2: a 1 million gate rapid prototyping system. In: IEEE TRANSACTIONS ON VLSI SYSTEMS, 1998, Nova Iorque, EUA. **Anais...** 1998. p.188–198.

[MCM99] MCMILLAN, Scott; GUCCIONE, Steven A. Partial run-time reconfiguration using JRTR. In: PROCEEDINGS OF FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS, 1999, Glasgow, Escócia. **Anais...** 1999.

[MES00] MESQUITA, Daniel; MORAES, Fernando Gehm. Implementação de sistemas reconfiguráveis parcial, remota e dinamicamente. [S.l.: s.n.], 2000. (Plano de Estudos e Pesquisa artigo pré-requisito para dissertação de mestrado PPGCC-FACIN-PUCRS).

[NAT98] NATIONAL. **Navigator magazine**. Disponível por WWW em http://www.national.com/appinfo/milaero/files/f61.pdf (Julho 1998).

[PAG96] PAGE, Ian. Reconfigurable processor architectures. **Elsevier Microprocessors & Microsystems**, v.20, p.185–196, 1996.

[PAL01] PALMA, José Carlos Sant'Anna; MORAES, Fernando Gehm; CALAZANS, Ney Laert Vilar. Métodos para desenvolvimento e distribuição de IP cores. In: SCR'2001 - Seminário de Computação Reconfigurável, 2001, Belo Horizonte, MG-Brasil. **Anais...** 2001.

[PAL01a] PALMA, José Carlos Sant'Anna. **Métodos para desenvolvimento e distribuição de IP cores**. Porto Alegre, Rio Grande do Sul, Brasil: PPGCC - FACIN - PUCRS, 2001. Dissertação de Mestrado.

[PER99] PERISSAKIS, Stylianos; JOO, Yangsung; AHN, Jinhong; DEHON, Adré; WAWRZYNEK, John. Embedded DRAM for a reconfigurable array. In: PROCEEDINGS OF THE SYMPOSIUM ON VLSI CIRCUITS, 1999, USA. **Anais...** 1999.

[RAD98] RADUNOVIC, Bozidar; MILUTINOVIC, Veljko. A survey of reconfigurable computing architectures. In: FPL'98 WORKSHOP, 1998, Tallin, Estonia. **Anais...** 1998.

[ROS93] ROSE, Jonathan; GAMMAL, Abbas El; SANGIOVANNI-VINCENTELLI, Alberto. Architecture of field programmable gate arrays. **Proceedings of the IEEE**, v.81, n.7, p.1013–1024, July 1993.

[SAN99] SANCHEZ, Eduardo; SIPPER, Moshe; HAENNI, Jacques-Olivier; BEUCHAT, Jean-Luc; STAUFFER, André; PEREZ-URIBE, Andrés. Static and dynamic configurable systems. Nova Iorque, Estados Unidos: [s.n.], 1999. p.556–564.

[SAS01] SASSATELLI, Gilles; TORRES, Lionel; GALY, Jerome; CAMBOM aston; DIOU, Camille. The Systolic Ring: a dynamically reconfigurable architecture for embedded systems. In: PROCEEDINGS OF THE FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS WORKSHOP, 2001, Belfast, Irlanda do Norte. **Anais...** 2001.

[SID99] SIDSA. **Fipsoc mixed signal system-on-chip**. Disponível por WWW em http://www.sidsa.com/FIPSOC/fipsoc1.pdf (Setembro 1999).

[SIL01] SILICORE. **Wishbone frequently asked questions (faq)**. Disponível por WWW em http://www.silicore.net/wishfaq.htm (Outubro 2001).

[VAH01] VAHID, Frank; GIVARGIS, Tony. Platform tuning for embedded systems design. **IEEE Computer**, California, USA, p.112–114, 2001.

[VAS99] VASILKO, Milan. Dynasty: a temporal floorplanning based cad framework for dynamically reconfigurable logic systems. In: PROCEEDINGS OF FIELD PROGRAMMABLE LOGIC AND APPLICATIONS WORKSHOP, 1999, Glasgow, Escócia. **Anais...** 1999. p.124–133.

[VIL97] VILLASENOR, John; MANGIONE-SMITH, William. Configurable computing. **Scientific American**, v.19, p.54–58, 1997.

[WAI97] WAINGOLD, Elliot; TAYLOR, Michael; SRIKRISHNA, Devabhaktuni; SARKAR, Vivek; LEE, Walter; LEE, Victor; KIM, Jang; FRANK, Matthew; FINCH, Peter; BARUA, Rajeev; BABB, Jonathan; AMARASINGHE, Saman; AGARWAL, Anant. Baring it all to software: Raw Machines. **IEEE Computer**, New York, USA, p.86–93, 1997.

[WIR95] WIRTHLIN, Michael J.; HUTCHINGS, Brad L. A dynamic struction set computer. In: PROCEEDINGS OF THE IEEE WORKSHOP ON FPGAS FOR CUSTOM COMPUTING MACHINES, 1995, Los Alamitos, California. **Anais...** IEEE Computer Society Press, 1995. p.99–107.

[WIR98] WIRTHLIN, Michael J.; HUTCHINGS, Brad L. Improving functional density using run-time circuit reconfiguration. Nova Iorque, EUA: [s.n.], 1998. v.6, n.2, p.247–256.

[XIL99] XILINX. **Satus and control semaphore registers using partial reconfiguration**. Disponível por WWW em http://www.xilinx.com/xapp/xapp138.pdf (Junho 1999).

[XIL99a] XILINX. **XC6200 datasheet**. Disponível por WWW em http://dec.bournemouth.ac.uk/drhw_lib/archive/xc6200.pdf (Junho 1999).

[XIL00] XILINX. **Frequently asked questions about partial reconfiguration**. Disponível por WWW em http://www.xilinx.com/xilinxonline/partreconfaq.htm (2000).

[XIL00a] XILINX. **Virtex series configuration architecture user guide**. Disponível por WWW em http://www.xilinx.com/xapp/xapp151.pdf (Setembro 2000).

[XIL01] XILINX. **The JBits 2.7 SDK for Virtex**. Disponível por FTP em ftp://customer:xilinx@ftp.xilinx.com/download/JBits2_7.exe (Outubro 2001).